On Non-Intrusive Workload-Aware Database Replication

Doctoral Dissertation submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Vaidė Narváez

under the supervision of
Fernando Pedone

June 2009

Dissertation Committee

M. Jazayeri	University of Lugano, Switzerland
A. Carzaniga	University of Lugano, Switzerland
R. Jiménez-Peris	Technical University of Madrid (UPM), Spain
B. Kemme	McGill University, Quebec, Canada
R. Oliveira	University of Minho (UMinho), Portugal

Dissertation accepted on xx June 2009

Research Advisor	PhD Program Director
Fernando Pedone	**Fabio Crestani**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Vaidė Narváez
Lugano, xx June 2009

to Christie

Everything should be made as
simple as possible, but not simpler

Albert Einstein

Abstract

Performance and high-availability are the crucial factors in the development of nowadays distributed database systems. Both of these challenges are commonly addressed by means of the same technique, database replication. The overall throughput of the system is increased by leveraging parallel computation on different sites, and in case of replica failures, availability is improved by redirecting requests to operational replicas. However, providing transparent database replication is not an easy task. Although database replicas should be as independent of each other as possible for performance and availability reasons, some synchronization is required to provide data consistency.

This thesis is about non-intrusive (or middleware) database replication protocols. More specifically, this thesis focuses on the development of practical replication protocols that use off-the-shelf database engines, take advantage of group communication primitives, cope with failures of system components, behave correctly, and, by exploiting the specific characteristics of the application, achieve high performance.

In the first part of this thesis we address the following problem: non-intrusive database replication protocols cannot obtain fine-grained information about transactions due to limited access to the database engine internals. We make three contributions in this part. The first contribution is the *Multiversion Database State Machine*, a middleware extension of the Database State Machine, a kernel-based replication approach. The Multiversion Database State Machine assumes predefined, parameterized transactions. The particular data items accessed by a transaction depend on the transaction's type and the parameters provided by the application program when the transaction is instantiated. The second contribution of this thesis is a technique to bypass the extraction and propagation of readsets and writesets in non-intrusive replication protocols. We present the *SQL Inspector*, a tool capable to automatically identify conflicting transactions before their actual execution by partially parsing them. The performance of the Multiversion Database State Machine can be further improved if transactions execute at carefully chosen database sites. Thus, the third contribution of this thesis is

the *conflict-aware load-balancing techniques*. To keep the abort rate low despite the coarse granularity of non-intrusive replication protocols, conflict-aware load-balancing techniques attempt to reduce the number of conflicting transactions executing on distinct database sites and seek to increase the parallelism among replicas.

In the second part of this thesis we investigate correctness criteria for replicated databases from the client's perspective. We study the performance cost of ensuring stronger consistency degrees in the context of three middleware replication protocols: primary-backup, optimistic update-everywhere and *BaseCON*, a non-intrusive replication protocol that takes advantage of workload characterization techniques to increase the parallelism in the system. BaseCON makes use of total-order broadcast primitives to provide strong consistency and fault-tolerance. A lightweight scheduler interposed between clients and the database replicas allows the system to adapt easily to the correctness criterion required and serves as a load-balancer for read-only transactions.

In the last part of this thesis we address the problem of partial replication. In particular, we investigate the effects of distributed transactions on the abort rate of such systems. Our contribution in this area is a *probabilistic model of transaction abort rates* for two different concurrency control mechanisms: lock- and version-based. The former models the behavior of a replication protocol providing one-copy serializability; the latter models snapshot isolation.

Acknowledgements

This thesis is a result of several years of effort and there is a number of people who in a way or another contributed to the success of this work. First and foremost, I would like to thank my advisor Fernando Pedone. Four years of guidance taught me much about the art of research. His confidence and continuous encouragements maintained my motivation up and engrained in me the curiosity and persistence to continue. Much of this dissertation is a result of his endless support. I thank you.

Next I wish to thank Josep M. Bernabé-Gisbert. The collaboration with him taught me more than he might think. The many discussions we had influenced significantly the research presented in this thesis. Gracias, Josep!

Part of the work of this thesis has been done in the context of the EU GORDA project. I wish to thank all the partners for making this work possible. I would also like to express my gratitude to the members of my dissertation committee for their valuable feedback and the time spent examining my dissertation proposal and this thesis.

All the people I shared my endeavor and made friends with: Lásaro, Paolo, Rodrigo, Marcin, Nicolas, Marija, Jochen, Amir – I am thankful for your company.

Finally, I wish to thank my dearest companion in life, Christian, to whom I dedicate this thesis. None of this work would have been possible without his patience and support.

Preface

This thesis concerns my PhD work done under the supervision of Prof. Fernando Pedone at the University of Lugano, from 2004 to 2009. During this period I was also involved in the EU project "Open Replication of Databases" (GORDA). The main goal of the GORDA project was to promote the interoperability of DBMSs and replication protocols by defining a generic architecture and interfaces that can be standardized. Within this project, the objective of this thesis has been to provide non-intrusive database replication solutions.

Most of the results presented in this thesis appear in previously published articles and technical reports:

Zuikevičiūtė, V. and Pedone, F. [2005]. Revisiting the Database State Machine Approach, *WDIDDR'05: Proceedings of VLDB Workshop on Design, Implementation, and Deployment of Database Replication*, pp. 1–7.

Zuikevičiūtė, V. and Pedone, F. [2006]. Conflict-Aware Load-Balancing Techniques for Database Replication, *Technical Report 2006/01*, University of Lugano.

Zuikevičiūtė, V. and Pedone, F. [2008a]. Conflict-Aware Load-Balancing Techniques for Database Replication, *SAC'08: Proceedings of ACM Symposium on Applied Computing, Dependable and Adaptive Distributed Systems Track*, ACM Press, pp. 2168–2173.

Zuikevičiūtė, V. and Pedone, F. [2008b]. Correctness Criteria for Database Replication: Theoretical and Practical Aspects, *DOA'08: Proceedings of 10th International Symposium on Distributed Objects, Middleware, and Applications*, Springer Verlag 2008.

Bernabé-Gisbert, J. M., Zuikevičiūtė, V., Muñoz-Escoí, F. D. and Pedone, F. [2008]. A probabilistic analysis of snapshot isolation with partial replication, *SRDS'08: Proceedings of 27th IEEE International Symposium on Reliable Distributed Systems*, IEEE, pp. 249–258.

Contents

Figures

Tables

Chapter 1

Introduction

Over the last decades the amount of data captured by enterprises has exploded. This has been fueled by high speed networks, the decreasing cost of powerful hardware, and the invasion of digital devices into everyday life and every part of the commercial world. Internet-based services have become the new standard of today's businesses where massive amounts of data are being collected and manipulated each day. Google alone processes multiple exabytes of data per year; Facebook claims to be one of the largest MySQL installations running thousands of databases serving millions of queries a day, MySpace records 7 billion user events per day; and the latest physics particle accelerator is expected to produce 10-15 petabytes of data each year. By no surprise, high-availability and performance are crucial in such systems. Both of these challenges are commonly addressed by means of the same technique, namely replication.

1.1 Motivation

Replication is an area of interest to both distributed systems and databases: in database systems replication is done mainly for performance and availability, while in distributed systems mainly for fault tolerance. The synergy between these two disciplines offers an opportunity for the emergence of database replication protocols based on group communication primitives. To overcome the performance limitations of classical replication solutions these protocols take advantage of message ordering and atomicity guarantees provided by group-communication systems (e.g., the former Isis, Horus, Totem, Transis and the current Appia, Spread, Ensemble, JGroups, Cactus).

In spite of thirty years of active research in database replication, there is still room for improvement, mainly concerning the practicality of the existing

solutions. Traditional replication approaches implement replica control within the database kernel. Although kernel-based solutions can benefit from optimizations in the database engine, the approach has a number of shortcomings. First, it requires access to the source code of the database management system limiting the implementation of the replication to database vendors or open source solutions only. Even if the source code is available, modifying it is not an easy task. Second, such protocols are tightly integrated with the implementation of the regular database functionality, and therefore are difficult to maintain in a continuously evolving software. For the sake of portability and heterogeneity, replication protocols should be independent of the underlying database management system. As a consequence, non-intrusive (or middleware) database replication has received a considerable amount of attention in the last years (e.g., [Cecchet et al., 2004; Correia et al., 2005; Lin et al., 2005; Muñoz-Escoí et al., 2006; Patiño-Martínez et al., 2005; Pedone and Frølund, 2008; Plattner and Alonso, 2004; Rodrigues et al., 2002]). Such solutions can be maintained independently of the database engine, and can even be used in heterogeneous settings. Non-intrusive protocols match the semantics of standard database access interfaces (e.g., ODBC or JDBC), which makes it straightforward to migrate from centralized to replicated environments. The downside of the approach is that the protocols usually have limited information about the data accessed by the transactions, and may end up duplicating back-end database logic into the middleware, resulting in reduced concurrency, or increased abort rate, or both. Standard database interfaces do not provide fine-grained information from the database engine, such as, accessed data items per transaction, when a transaction begins its execution, when different SQL statements take place, and when a transaction finishes. Such level of detail is essential in order to guarantee strong consistency and achieve high concurrency, and consequently better performance of the replicated system.

Not all database sites in the replicated system need to keep a full database copy — databases can be replicated partially only and thus can improve the scalability of the system. Most work on database replication using group communication concentrates on full replication strategies. However, scalability of such protocols is limited under update-intensive workloads: Each replica added to the system allows to submit more transactions; if these transactions modify the database, they will add load to every individual database. Unfortunately, it is not obvious how to extend many of the protocols developed for full replication to systems with partial replication. The majority of existing partial replication protocols (e.g., [Cecchet et al., 2004; Coulon et al., 2005; Schiper et al., 2006; Sousa et al., 2001]) build on the strong assumption that transactions can al-

ways execute locally at one database site. Such an assumption requires prior knowledge of the workload and a very precise data distribution over the replicas or at least a single replica that holds the whole database. Lack of support for distributed execution of transactions is a serious constraint, which limits the applicability of partial replication solutions to particular workloads only.

To circumvent the limitations of traditional middleware solutions in the context of both full and partial replication, it is necessary to rethink the way replica correctness is handled. In [Stonebraker et al., 2007] the authors criticize the one-size-fits-all paradigm of databases and argue for a redesign, which would take into account application-specific needs. Motivated by these observations, this thesis focuses on the development of practical protocols for database replication that use off-the-shelf database engines, cope with failures of system components, behave correctly, and, by exploiting the specific characteristics of the application, achieve high performance.

1.2 Research Contributions

This thesis provides the following contributions.

Multiversion Database State Machine. The Multiversion Database State Machine is a middleware extension of the Database State Machine replication, a kernel-based replication technique. The Multiversion Database State Machine assumes predefined, parameterized transactions. The particular data items accessed by a transaction depend on the transaction's type and the parameters provided by the application program when the transaction is instantiated. By estimating the data items accessed by transactions before their execution, even if conservatively, the replication protocol is spared from extracting readsets and writesets during the execution. In the case of the Multiversion Database State Machine, this has also resulted in a certification test simpler than the one used by the original Database State Machine, although both techniques guarantee the same strong consistency.

Workload Characterization Techniques. Transaction readsets and writesets information is an essential component of database replication protocols, in particular when strong system consistency is targeted. The workload characterization techniques allow to bypass the extraction and propagation of readsets and writesets in non-intrusive replication protocols. We present the SQL Inspector, a tool capable to automatically identify conflicting transactions by partially

parsing them. The SQL inspector is a conservative technique and may over-approximate the set of data items accessed by the transaction that could cause conflicts. However, the evaluation of the TPC-C benchmark reveals that the SQL Inspector achieves finer than just table-level granularity. To the best of our knowledge this is the first attempt to automate the extraction of transaction readsets and writesets outside the database engine.

Conflict-Aware Load-Balancing techniques. Optimistic database replication protocols, especially non-intrusive ones, may suffer from excessive synchronization aborts. A key property of the Multiversion Database State Machine is that if transactions with similar access patterns execute at the same database replica, then the local replica's scheduler will serialize them and both can commit reducing the abort rate. Based on the information obtained by the SQL Inspector, we can carefully assign transactions to database replicas avoiding conflicts as much as possible. The proposed conflict-aware load-balancing techniques schedule transactions to replicas so that the number of conflicting transactions executing on distinct sites is reduced and the load over the replicas is equitably distributed. Experimental results show that exploring specific workload information while assigning transactions to the replicas is a promising technique to improve the performance of non-intrusive replication protocols.

BaseCON and the cost of correctness criteria for non-intrusive database replication. It has been generally believed that additional constraints on correctness degrades the performance of a replicated system. To verify this statement we investigate the performance cost of implementing different consistency degrees in BaseCON and two other non-intrusive replication protocols. BaseCON is a simple yet fault-tolerant non-intrusive replication protocol that takes advantage of workload characterization techniques to increase the parallelism in the system. The experimental evaluation reveals that stronger consistency does not necessarily imply worse performance in the context of middleware-based replication. On the contrary, two of the three protocols evaluated are able to provide different consistency guarantees without penalizing system's performance. Even though the implementation of strong serializability requires ordering read-only transactions in all protocols studied, the overhead introduced by total-order primitives is insignificant in non-intrusive replication.

Probabilistic model of transaction abort rates for partial replication. Not all the database sites in a replicated system need or are able to keep a full copy

of the database — databases can be replicated partially only. If each replica keeps only part of the database, a transaction may require access to data stored on remote replicas and thus, a distributed execution involving more than one replica becomes necessary. The problems introduced by distributed transactions in partially-replicated systems differ depending on the concurrency control mechanism used. We introduce a probabilistic model for abort rates of partially replicated systems when lock- and version-based concurrency control mechanisms are used. The former models the behavior of a replication protocol providing one-copy-serializability; the latter models snapshot isolation.

The analytical evaluation shows that in the version-based system the number of data versions available decreases the execution abort rate exponentially. Furthermore, in the version-based system even if the workload over the partially replicated system is dominated by read-only transactions, but the few update transactions perform a lot of updates, distributed read-only transactions can still cause a noticeable number of aborts, as opposed to typical full replication protocols, in which the number of versions available is (in principle) unbounded, and thus, read-only transactions executing under snapshot isolation are never aborted.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the system model details, terminology, assumptions, and the definitions on which the replication protocols suggested in this thesis are based. In Chapter 3 we first recall the concepts of deferred-update and the Database State Machine replication; then, we show how the original Database State Machine can be implemented in the middleware layer and introduce the Multiversion Database State Machine replication. In Chapter 4 we present the SQL Inspector, a tool capable to automatically identify conflicting transactions by partially parsing them. We explain the idea, the actual implementation of the tool and the case study of the two workloads chosen. Chapter 5 introduces conflict-aware load-balancing techniques that can be used to further improve the performance of the Multiversion Database State Machine. We illustrate the behavior of the algorithms with the chosen benchmarks and present a thorough experimental evaluation of the load-balancing techniques. In Chapter 6 we investigate the performance cost of different consistency degrees in the context of non-intrusive replication protocols and introduce BaseCON, a simple, yet fault-tolerant, middleware-based replication protocol. Chapter 7 addresses the issue of partial replication. In particular,

a simple probabilistic analysis of transaction abort rates is introduced. Finally, Chapter 8 concludes the thesis and outlines some future research directions.

Chapter 2

System Model and Definitions

This chapter presents the main definitions and assumptions on which the replication protocols suggested in this thesis are based. All the protocols proposed hereafter rely on three building blocks: database sites, state machine replication and total-order broadcast. In the following we detail each of them.

2.1 Model

Distributed systems can be classified according to the way the system components exchange information and the way they fail and recover. In this thesis we assume a system in which individual components may crash and recover. The distributed system considered is composed of database clients, $\mathscr{C} = \{c_1, c_2, ..., c_m\}$, and a set of database sites (or database replicas), $\mathscr{S} = \{s_1, s_2, ..., s_n\}$. First we require that each database site store a full copy of the database, then in Chapter 7 we relax the requirement of full replication and assume partial replication. Database sites fail independently and only by crashing and may eventually recover during execution but never behave maliciously. Even though sites may recover, they are not obliged to do so once they have failed.

In the crash-recovery model, sites are classified as *good* and *bad* according to their behavior concerning failures [Aguilera et al., 1998]:

Good: if site s never crashes or if site s crashes at least once, but there is time after which s is permanently up.

Bad: if there is time after which site s is permanently down or if site s crashes and recovers infinitely many times.

All sites have access to stable storage which they can use to keep their state in between failures. State not kept in stable storage is lost during the crash.

Communication among sites is by message passing and there is no access to a shared memory or a global clock. We further assume the existence of total-order broadcast and total-order multicast primitives defined in Section 2.5.

The system is asynchronous, that is, we do not make any assumptions about the time it takes for clients and database sites to execute and for messages to be transmitted. We do assume however that the system is augmented with unreliable failure detectors [Chandra and Toueg, 1996]. Such oracles can make an unbounded number of mistakes: sites may be incorrectly suspected to have crashed even if they are just slow to respond. We rely on failure detectors targeted specifically for crash-recovery models and defined by Aguilera et al. [1998]. Informally, to ensure progress such failure detectors satisfy the following properties[1]: for every bad site s, at every good site there is a time after which s is never trusted (*completeness*); and some good site is eventually trusted forever by all good sites (*accuracy*).

2.2 Database and Transactions

A database $\mathscr{D} = \{d_1, d_2, ...\}$ is a set of data items. For each database site s_i, $Items(s_i) \subseteq \mathscr{D}$ is defined as the set of data items stored at s_i. If full database replication is considered $Items(s_i) = \mathscr{D}$ for all database sites s_i.

A transaction T_i starts with a begin operation b_i and a sequence of read and write operations followed by a commit c_i or an abort a_i. $start(T_i)$ and $commit(T_i)$ are b_i and c_i timestamps. We denote $T_i's$ write operation on data item x by $w_i[x_i]$, and read operation by $r_i[x_j]$, where x_j is the value of x written by T_j. A transaction is called *read-only* if it does not contain any write operations; otherwise it is called an *update* transaction. The transaction's *readset* and *writeset* identify the data items read and written by the transaction, denoted as $rs(T_i)$ and $ws(T_i)$ for transaction T_i respectively. The transaction's *updates* contain the values written and can be redo logs or the rows it modified and created. The *workload* submitted to the database is composed of a set of transactions $\mathscr{T} = \{T_1, T_2, ...\}$.

We target enterprise environments in which transactions are embedded in application programs to enforce the business logic. Thus, clients interact with the system by triggering application-oriented functions. Each of these functions is a predefined, parameterized transaction. We refer to application-oriented

[1]Formal definitions of the properties are defined in [Aguilera et al., 1998], where authors also introduce the epoch number of a site. The epoch number is an estimate of the number of times the site has crashed and recovered in the past.

functions as simply transactions. Each transaction is identified by its type and the parameters provided by the application program when the transaction is instantiated. We say that two transactions T_i and T_j conflict, denoted as $T_i \sim T_j$, if they access some common data item, and at least one transaction writes it.

2.3 Consistency Criteria

We are mainly interested in two database consistency criteria: *serializability* (SR) [Bernstein et al., 1987] and *snapshot isolation* (SI) [Berenson et al., 1995].

2.3.1 Serializability

We recall some basic definitions of serialization theory introduced by Bernstein et al. [1987]. Transactions executing in a database are formalized by histories. A history H over a set of transactions $\mathscr{T}$ is a partial order with ordering relation $<_H$, where

a) H contains all the operations op of each transaction $T_i \in \mathscr{T}$;

b) $\forall T_i \in \mathscr{T}$ and $\forall op_i \in T_i$: if op_i precedes op_i' in T_i, then $op_i <_H op_i'$; and

c) if T_i reads data item x from T_j, then $w_j[x_j] <_H r_i[x_j]$.

A history H is *serial* if, for every two transactions T_i and T_j in H, either all operations of T_i happen before all operations of T_j or vice versa. Two histories H, H' are *view equivalent* ($\equiv$) if

a) they are over the same set of transactions;

b) for any T_i, T_j and data item x: if $w_j[x_j] <_H r_i[x_j]$, then $w_j[x_j] <_{H'} r_i[x_j]$; and

c) for each x, if $w_i[x_i]$ is the final write on x in H, then it is also the final write of x in H'.

A typical correctness criterion for replicated databases is *one-copy serializability* (1SR)[Bernstein et al., 1987].

Definition 1. *History H is* one-copy serializable *iff there is some serial history H_s such that $H \equiv H_s$.*

Informally, 1SR requires the execution of concurrent transactions on different replicas to appear as if transactions were executed in some sequential order on a single replica. To guarantee serializability most DBMSs implement *two-phase locking* (2PL) or *strict* 2PL concurrency control [Bernstein et al., 1987], where locks on data items are handled by a transaction in two consecutive phases during its execution.

2.3.2 Snapshot Isolation

In snapshot-isolated databases transactions read data from a committed snapshot of the database taken at the time the transaction starts. We denote the time when transaction's T_i snapshot is taken as $snapshot(T_i)$. All transactions execute without interfering with each other, however, transaction T_i can only successfully commit if there exists no other transaction T_j that committed after T_i started and updated the same data items (*first-committer-wins* rule). If no such a transaction exists, then T_i can commit and its updates will be visible to all the transactions that start after T_i's commit.

Snapshot isolation is less abstract and more implementation related when compared to serializability. As a result, several extensions of conventional SI can be found in the literature. In [Lin et al., 2005] the authors develop a formalism for SI in replicated systems and introduce *one-copy snapshot isolation* (1CSI). Elnikety et al. [2005] further extend snapshot isolation to better fit replicated databases and define *generalized snapshot isolation* (GSI). GSI is based on the observation that a transaction need not necessarily observe the latest snapshot. More formally, GSI was defined as follows:

Definition 2. *For any history H created by GSI, the following two properties hold:*

- **GSI Read Rule**
  $\forall T_i, x_j$ *such that* $r_i[x_j] \in H$*:*
  1. $w_j[x_j] \in H$ **and** $c_j \in H$*;*
  2. $commit(T_j) < snapshot(T_i)$*;*
  3. $\forall T_k$ *such that* $w_k[x_k], c_k \in H$*:*
     $$[commit(T_k) < commit(T_j) \textbf{ or } snapshot(T_i) < commit(T_k)].$$
- **GSI Commit Rule**
  $\forall T_i, T_j$ *such that* $c_i, c_j \in H$*:*

4. $\neg(ws(T_i) \cap ws(T_j) \neq \emptyset$ **and**
   $snapshot(T_i) < commit(T_j) < commit(T_i))$.

The GSI Read Rule regulates read operations and ensures that only committed data is read. The GSI Commit Rule guarantees *first-committer-wins* behavior.

Unfortunately snapshot isolation does not forbid all the serialization anomalies [Berenson et al., 1995]. For instance, it allows the *write-skew* anomaly:

$$r_i[x_k], r_i[y_l] \ldots r_j[x_k], r_j[y_l], w_j[x_j], c_j \ldots w_i[y_i], c_i$$

In the above example transactions T_i and T_j are executed concurrently at two different sites, T_i reads data items x_k and y_l, T_j also reads x_k and y_l, writes x_j and tries to commit. Then transaction T_i writes y_i and tries to commit. Both transactions pass the certification test because their writesets do not intersect, however the execution is not serializable (i.e., no serial execution is equivalent to it).

2.4 State Machine Replication

The state machine approach is a non-centralized replication technique [Schneider, 1990]. Its key concept is that all database sites receive and process the same sequence of requests in the same order. Consistency is guaranteed if replicas behave deterministically, that is, when provided with the same input (e.g., a request) each replica will produce the same output (e.g., state change).

The way requests are disseminated among replicas can be decomposed into two requirements:

1. **Agreement.** Every good site receives every request.

2. **Order.** Every good site processes the requests it receives in the same relative order.

2.5 Total-Order Broadcast and Multicast

In order to satisfy the aforementioned state machine requirements, database sites interact by means of atomic or total-order broadcast. Total-order broadcast allows to send messages to all the database sites in a system with the guarantee that all sites agree on both either to deliver or not to deliver the message, and

the order according to which the messages are delivered. More formally, total-order broadcast is defined by the primitives to-broadcast(m) and to-deliver(m), and guarantee the following properties:

1. **Integrity.** For any message m, every site delivers m at most once, and only if some site broadcast m.

2. **Agreement.** If a site delivers a message m, then every good site eventually delivers m.

3. **Validity.** If a site broadcasts message m and does not fail, then every good site eventually delivers m.

4. **Total Order.** No two sites deliver any two messages in different orders.

The notion of uniform delivery captures the concept of durability, that is, a database site must not forget that it has delivered a message after it recovers from a crash. After recovery the site delivers first all the messages it missed during the crashed period.

Contrary to the broadcast, multicast allows messages to be addressed to a subset of the database sites in the system. Similarly to total-order broadcast, total-order multicast ensures that the addressees of every message agree either to deliver or to not deliver the message, and no two sites deliver any two messages in different order. Although total-order multicast is the preferred abstraction for partial replication, existing total-order multicast protocols are costlier to ensure than the broadcast ones [Delporte-Gallet and Fauconnier, 2000].

Chapter 3

Replication Made Simple

This chapter introduces the first contribution of this thesis, the Multiversion Database State Machine. The Multiversion Database State Machine (vDBSM) is a middleware extension of the Database State Machine (DBSM) [Pedone et al., 2003], a kernel-based replication protocol. The vDBSM is a non-intrusive full replication protocol that ensures strong consistency (i.e., one-copy serializability), can possibly be used with heterogeneous off-the-shelf database management systems and can be implemented on cheap commodity hardware.

In this chapter, we first recall the original Database State Machine and the concept of deferred-update replication that DBSM is built upon; then we show how to solve the problem of readsets extraction essential to guarantee correctness in DBSM. We present two solutions: we begin with an approach which is more of a theoretical interest and complete with a practical protocol.

3.1 Revisiting the Database State Machine Approach

3.1.1 Deferred-Update Replication

The main idea of deferred-update replication [Bernstein et al., 1987] consists in executing all operations of a transaction on a single database site, so that transactions are synchronised according to some local concurrency control mechanism. Read-only transactions can commit immediately at the replica they executed; update transactions must be globally certified and, if committed, have their update operations reflected at all database sites. The certification ensures that committing transactions do not lead the database to an inconsistent state; otherwise the transaction is aborted.

Since most practical workloads have a majority of read-only transactions,

deferred-update replication allows for a good load balance among the replicas that execute such transactions completely independent of each other. Because of its good performance deferred-update replication lay basis for a number of database replication protocols (e.g., [Amir and Tutu, 2002; Patiño-Martínez et al., 2000; Lin et al., 2005]; Plattner and Alonso [2004]; Irún-Briz et al. [2005]).

In this thesis we focus on the Database State Machine (DBSM) approach for handling full database replication [Pedone et al., 2003] .

3.1.2 DBSM

The Database State Machine is an optimistic kernel-based replication protocol. In DBSM read-only transactions are processed locally at some database replica; update transactions do not require any synchronization between replicas until commit time. When an update transaction is ready to be committed, its updates, readsets, and writesets are total-order broadcast to all replicas. All sites receive the same sequence of requests in the same order and certify them deterministically. The certification procedure ensures that committing transactions do not conflict with concurrent already committed transactions.

During processing, transactions pass through some well-defined states:

1. *Executing state*. In this state transaction T_i is locally executed at site s_i according to *strict* 2PL.

2. *Committing state*. Read-only transactions commit immediately upon request. If T_i is an update transaction, it enters the committing state and s_i starts the termination protocol for T_i: T_i's updates, readsets, and writesets are broadcast to all replicas. Upon delivering this message, each database site s_i certifies T_i.

   Transaction T_i is allowed to commit at database site s_i only if there is no conflicting transaction T_j that executed concurrently and has been certified to commit already. Only *write-read* conflicts are considered: *write-write* conflicts are solved by total-order broadcast which guarantees that updates are applied at all replicas in the same order; *read-write* conflicts are not relevant since only committed transactions take part in T_i's certification test. More formally, transaction T_i passes the certification test at s_i if the following condition holds:

$$\left[\begin{array}{l} \forall T_j \text{ committed at } s_i : \\ T_j \rightarrow T_i \vee (ws(T_j) \cap rs(T_i) = \emptyset) \end{array}\right],$$

where $T_j \rightarrow T_i$ denotes the *precedence relation*, and is defined as follows:

- If T_i and T_j execute on the same replica s_i, then $T_j \rightarrow T_i$ only if T_j enters the committing state at s_i before T_i enters the committing state at s_i.
- If T_i and T_j execute on different replicas s_i and s_j, respectivelly, then $T_j \rightarrow T_i$ only if T_j is committed at s_i before T_i enters the committing state at s_i.

Total-order broadcast ensures that the sequence of transactions certified by each replica is the same; together with a deterministic certification test it guarantees one-copy serializability.

3. *Committed/Aborted state*. If T_i passes the certification test, its updates are applied to the database and T_i passes to the committed state. Transactions in the executing state at s_j holding locks on data items updated by T_i are aborted.

The main weakness of the DBSM lies in its dependency on transaction readsets. In the original DBSM, readsets of update transactions need to be broadcast to all sites for certification. Although storing and transmitting readsets are sources of overhead, extracting them from transactions is a more serious problem since it usually implies accessing the internals of the database engine or parsing SQL statements outside the database. On the other hand, obtaining writesets is less of a problem: writesets tend to be much smaller than readsets and can be extracted during transaction processing (e.g., using triggers or log sniffing). Thus, for the sake of portability, simplicity, and efficiency, certification in non-intrusive replication solutions should be "readsets-free".

3.2 DBSM* or Readsets-free Certification

In order to solve the problem of readsets extraction in middleware replication some protocols require all the operations of the transaction to be known in advance [Pacitti et al., 2005]; others assume coarser granularity: the tables accessed by the transaction and the type of the access (i.e., read or write) are required beforehand [Amza et al., 2003; Patiño-Martínez et al., 2005]. Coarser granularity information is easier to obtain, however, it decreases parallelism among transactions and leads to spurious aborts. Other approaches [Plattner and Alonso, 2004; Elnikety et al., 2006; Lin et al., 2005] guarantee

snapshot isolation, which does not rely on transactions readsets to ensure correctness.

The DBSM has an interesting property: if all transaction requests are submitted to the same replica, the DBSM will behave as primary-backup replication. Since all transactions would then be executed under *strict* 2PL at the primary site and serialized accordingly, and the updates would be applied at all sites in the same order, 1SR would be ensured. In particular, if all transactions execute at the same site, neither readsets nor writesets information is required for the certification test: due to 2PL scheduler for all transactions T_j committed at the database site s_i, the relation $T_j \rightarrow T_i$ is always true. Therefore, another way to ensure 1SR in the DBSM is by carefully scheduling update transactions to some selected database site; read-only transactions could still be executed at any replica. However, for load-balancing and availability reasons, localizing the execution of update transactions in the same site may not be such a good idea.

Following these ideas, we describe next the DBSM*, a readsets-free DBSM, which guarantees 1SR with no communication overhead w.r.t. the original DBSM. The basic idea of the DBSM remains the same: transactions are executed locally according to *strict* 2PL. In contrast to the original DBSM, in DBSM* when an update transaction requests a commit, only its updates and writesets are broadcast to the other sites. Briefly, the mechanism works as follows:

1. The database is logically divided into a number of disjoint sets (according to tables, rows, etc), each one under the responsibility of a different replica, and extended with a control table containing one dummy row per logical set. This control table is used for conflict materialization [Fekete et al., 2005]. Note that each replica still stores a full copy of the database.

2. Each replica is responsible for processing update transactions that access data items in its assigned logical set. Transactions that only access data items in a single logical set and execute at the corresponding replica (we call them "local") are serialized with other transactions of the same type by the 2PL scheduler on the replica where they execute.

3. Update transactions that access data items in more than one logical set should execute on a site responsible for one of these logical sets. We call such transactions "complex". Total-order broadcast ensures that complex transactions are serialized with other transactions updating data items in intersecting logical sets in the same order at all database sites. However total-order broadcast is not sufficient to serialize them with interfering transactions executing at different replicas. Two transactions interfere if

one reads what is written by the other (notice that this is precisely what the original DBSM certification test guarantees).

4. To ensure 1SR update transactions that read data items in a logical set belonging to the remote replica are extended with update statements for dummy rows corresponding to each remote logical set read. This can be done when the application requests the transaction commit. Dummy rows are constructed in such a way to materialize write-write conflicts between complex or local transactions that access data items in the same logical set. Therefore, if transaction T_i executes at s_i and one of T_i's operations reads a data item that belongs to s_j's logical set, a dummy write for s_j logical set is added to T_i. This ensures that if T_i executes concurrently with some transaction in s_j, complex or not, only one transaction will pass the certification test of DBSM*:

$$\left[\begin{array}{l} \forall T_j \text{ committed at } s_i : \\ T_j \rightarrow T_i \vee (ws(T_j) \cap ws(T_i) = \emptyset) \end{array}\right]$$

5. Read-only transactions can be executed at any database site independently of the data items accessed.

Although DBSM* does not rely on exact transactions readsets information, the division of the database into logical sets and transaction scheduling must be precise to avoid spurious aborts. Abort rates can be further reduced if the division of the database into logical sets takes the workload into account. For example, a criterion for defining logical sets could be the tables accessed by the transactions. We do not have to know exactly which data items are accessed by a transaction to schedule it to its correct replica; only its logical sets have to be known (e.g., which tables are accessed by the transaction). Furthermore, any transaction can be executed at any database site as long as a corresponding dummy writes for remote logical sets read are added to materialize the conflict.

Row level knowledge of transaction readset and writeset would allow to eliminate spurious aborts. In the following sections we show how such data can be obtained from the middleware layer and without modifying DBMS internals.

3.3 Multiversion Database State Machine

In this section we introduce the Multiversion Database State Machine (vDBSM), which together with the SQL Inspector (see Chapter 4) allows for efficient mid-

dleware implementation of the DBSM. Like the DBSM, the vDBSM guarantees one-copy serializability; we prove its correctness in Section 3.3.1.

The vDBSM assumes predefined, parameterized transactions. As we will show in Chapter 4, the readset and writeset of a predefined transaction can be estimated, even if conservatively, before the transaction is executed. Because of that the certification test of vDBSM is much simpler than the DBSM's certification test.

We further recall, that the transaction's T_i readset and writeset are denoted as $rs(T_i)$ and $ws(T_i)$, respectively. The new protocol works as follows:

1. We assign to every table and to each data item in the database a version number. Thus, besides storing a full copy of the database, each replica s_k also has a vector V_k of version numbers. The current versions of the table $table$ and the data item d_x at site s_k are denoted by $V_k[table]$ and $V_k[x]$ respectively.

2. Both read-only and update transactions can execute at any replica. Just like in the DBSM, read-only transactions are local to the replica and are committed immediately upon request.

3. During the execution of an update transaction, the versions of the data items read by the transaction are collected. If transaction readset cannot be obtained at the row level, table version is taken instead.

   We denote by $V(T_i)[x]$ the version of each data item d_x read by T_i. Similarly, the versions of the tables accessed by T_i are denoted as $V(T_i)[tables]$. The versions of the data read by T_i are broadcast to all replicas together with its readset, writeset, and updates at commit time.

4. Upon delivery update transactions are certified. Transaction T_i passes certification if all data items it read during its execution are still up-to-date at certification time. If row-level granularity is not available, table versions are compared.

   For example, two conflicting transactions T_i and T_j execute concurrently on different database sites. T_j updates data item x and commits. T_i reads x concurrently but is delivered after T_j. Initially, $V_k[x] = 0$, once T_j has committed the version of data item x is incremented at all database sites. The version of data item x collected during execution of transaction T_i is $V(T_i)[x] = 0$; since T_i is certified after T_j, $V_k[x] \neq V(T_i)[x]$, and thus T_i is aborted. If T_i and T_j execute at the same replica, due to 2PL concurrency

control, T_i will obtain a read lock on data item x only after T_j has released its locks, i.e., only after T_j is committed. Consequently, $V_k[x] = V(T_i)[x]$ and thus T_i is allowed to commit too. Notice that such a certification test implements the certification test of the original DBSM (see Section 3.1.2).

More formally, T_i passes certification on replica s_k if the following condition holds:

$$\left[\begin{array}{l} \forall tables \in rs(T_i):\ V_k[table] = V(T_i)[table] \\ \textbf{and} \\ \forall d_x \in rs(T_i):\ V_k[x] = V(T_i)[x] \end{array}\right]$$

5. If T_i passes certification, its updates are applied to the database, and the version numbers of the tables and data items it wrote are incremented. The versions of the tables to which the modified rows belong are also incremented. All replicas must ensure that transactions that pass certification are committed in the same order.

In the vDBSM as well as in the original DBSM, if two transactions T_i and T_j conflict and are executed concurrently on different database sites, certification may abort one of them. If they execute on the same replica, however, the local scheduler of the replica will serialize conflicting transactions appropriately, and thus, both T_i and T_j can commit. When two conflicting transactions execute on distinct replicas, one transaction may read data items concurrently being updated by another transaction, if the latter is certified to commit first, the former has read inconsistent data and thus must abort to ensure strong consistency. If such transactions execute at the same database site, the local concurrency control mechanism will forbid reading intermediate data, that is, data items modified by an uncommitted transaction are locked until that transaction commits or aborts.

In optimistic replication protocols that usually suffer from aborts due to lack of synchronization and especially in non-intrusive solutions where the abort rate is further increased due to coarse grain conflict information, careful scheduling of transactions may considerably improve the performance of the system. We explore this idea further and present a thorough evaluation of the vDBSM in Chapter 5.

3.3.1 Proof of Algorithm Correctness

Proposition 1. *The vDBSM ensures one-copy serializability.*

Proof (Sketch). We show that every history H produced by vDBSM has an acyclic multiversion serialization graph (MVSG). From [Bernstein et al., 1987], if $MVSG(H)$ is acyclic, then H is one-copy serializable. MVSG is a directed graph with the nodes representing committed transactions and edges representing read-from and version order relationships between them. From the protocol, the version order on every data object is defined by the commit order of transactions. If $\ll$ is a version order and T_i and T_j update data object x, then $x_i \ll x_j \Leftrightarrow commit(T_i) < commit(T_j)$.

To prove that $MVSG(H)$ is acyclic we show that for every edge $T_i \rightarrow T_j \in MVSG$, it follows that $commit(T_i) < commit(T_j)$. In the following we consider each edge type of $MVSG(H)$.

- *Read-from edge.* If T_j reads data object x from T_i, i.e., $r_j[x_i]$, then $T_i \rightarrow T_j \in MVSG(H)$. We have to show that $commit(T_i) < commit(T_j)$.

  If both T_i and T_j are update transactions, T_j can read the updates of T_i only if T_i has been already certified and committed. If T_j is a read-only transaction, due to 2PL, T_j can read only committed data. Thus, $commit(T_i) < commit(T_j)$

- *Version order edge.*

  - If both T_i and T_j update x, such that $x_i \ll x_j$, then $T_i \rightarrow T_j \in MVSG(H)$.

    From the definition of version order we have that $x_i \ll x_j \Leftrightarrow commit(T_i) < commit(T_j)$.

  - If T_i reads x_k and both T_k and T_j write x such that $x_k \ll x_j$, then $T_i \rightarrow T_j \in MVSG(H)$.

    For a contradiction, assume that $commit(T_j) < commit(T_i)$. To commit at some site S_l in the vDBSM, T_i must pass the certification test. Therefore, it follows that for every data item y and for every $table$ read by T_i, $V_l[y] = V(T_i)[y]$ and $V_l[table] = V(T_i)[table]$.

    First, let's assume that T_i and T_j execute in parallel. During the execution of T_i, the current versions of data items read are collected and stored in $V(T_i)[y]$ and $V(T_i)[table]$. Since T_i reads data item x, depending on the granularity achieved, $V(T_i)[x]$ or $V(T_i)[table]$ is collected also. From the definition of version order edge T_j updates data item x. When T_j commits, versions $V_l[x]$ and $V_l[table]$ are incremented. Since T_j commits before T_i is certified, it cannot be

that $V_l[x] \neq V(T_i)[x]$ and $V_l[table] \neq V(T_i)[table]$. Thus, since T_j writes data items read by T_i, T_i and T_j cannot execute in parallel.

Now, let's assume that transaction T_i started after T_j has committed. T_i reads x_k from transaction T_k, so $commit(T_k) < commit(T_i)$, and this can only happen if T_j updates x and commits before T_k, that is $x_j \ll x_k$. This contradicts that $x_k \ll x_j$ and consequently proves that $commit(T_i) < commit(T_j)$.

In case of failures, uniform total order broadcast guarantees that recovering sites first deliver all the messages missed during the crashed period. Since delivery and certification of transactions is in total order, even the recovering sites will reach the same decision on transactions fate. □

3.4 Related Work and Final Remarks

Gray et al. [1996] classify recent database replication protocols according to two parameters. The first parameter determines *where* transaction updates take place and thus distinguishes primary-backup and update everywhere replication approaches. Primary-backup replication requires update transactions to be submitted to the same dedicated replica, while in the update everywhere approach any replica can execute any transaction. Clearly, both solutions require changes introduced by a transaction to be applied at all the database sites. Thus the second parameter determines *when* transactions updates are applied at the replicas. In eager replication a transaction synchronizes with all the replicas before it commits at the local replica. Such protocols guarantee strong consistency at the price of increased response times due to communication overhead. Lazy replication schemes allow changes introduced by a transaction to be propagated to other replicas after the transaction has committed. Lazy replication is a standard feature of most mainstream database management systems (Microsoft SQL Server, Oracle, Sybase, IBM DB2, PostgreSQL, MySQL). Such solutions scale very well, but provide weaker consistency guarantees and require complex reconciliation techniques.

As an alternative, group communication-based replication protocols have emerged. In contrast to replication based on distributed locking and atomic commit protocols, group communication-based protocols minimize the interaction between replicas and the resulting synchronization overhead. Proposed solutions vary in the execution mode—transactions can be executed conservatively [Kemme et al., 1999; Patiño-Martínez et al., 2000, 2005] or optimistically

[Kemme and Alonso, 2000; Lin et al., 2005; Pedone et al., 2003; Wu and Kemme, 2005], and in the database correctness criteria they provide—one-copy serializability [Amir and Tutu, 2002; Patiño-Martínez et al., 2000, 2005; Pedone and Frølund, 2008; Pedone et al., 1998, 2003; Rodrigues et al., 2002] or snapshot isolation [Elnikety et al., 2006; Lin et al., 2005; Wu and Kemme, 2005]. In the following we summarize recent group communication-based replication protocols and compare them with our approach. In particular we are interested in the solutions that are closely related to ours, specially with regards to non-intrusive aspect.

Agrawal et al. [1997] present a family of replica management protocols that exploit the properties of total-order broadcast and are based on state machine replication. Their first protocol suffers from excessive broadcast overhead since every transaction's operation is broadcast to every database replica; in the second protocol read operations are localized; and finally, in the third protocol, transactions are executed locally and total-order broadcast is used only for transactions termination. Deferred updates replication based on group communication was also proposed in [Pedone et al., 1997]. However, both approaches assume preexisting knowledge on data accessed by the transactions.

Patiño-Martínez et al. [2005] present three conservative protocols (DISCOR, NODO and REORDERING) which use conflict classes for concurrency control of update transactions. A conflict class represents a partition of the data and is used for transaction synchronization. Transactions are sent to all the replicas using optimistic total-order broadcast [Kemme et al., 2003] but the outcome of transactions is only decided at the master site of the respective conflict class. Hence, remote sites do not even have to wait for the definitive total order to execute a transaction. The authors require the data partitions accessed by the transaction to be known in advance; no discussion on how conflict classes could be obtained automatically is provided.

The Pronto protocol presented by Pedone and Frølund [2008] was among the first to consider building database replication without requiring modifications to the database engine, allowing it to be deployed in a heterogeneous environment. The protocol is based on the primary-backup replication model and total-order broadcast primitives. Although the solution does not require readsets and writesets information, the throughput of update transactions is limited by the performance of a single primary replica handling all update load. Whereas in vDBSM update transactions can execute at any replica.

Clustered JDBC (C-JDBC) [Cecchet et al., 2004] is an open-source middleware solution for database clustering on a shared-nothing architecture built with commodity hardware. C-JDBC supports both partial and full replication. The

approach consists in hiding a lot of database complexity in the C-JDBC layer, outside the database.

Snapshot Isolation is especially attractive for middleware-based concurrency control. In Lin et al. [2005] the authors propose SI-Rep, a pure middleware replication solution providing a replicated form of snapshot isolation. Transactions are first executed at a local database replica providing SI. At the end of the execution the updated records are extracted. After retrieving the writesets, SI-Rep performs a validation to check for write/write conflicts with transactions that executed at other replicas and have been already validated. If validation succeeds, the transaction commits at the local replica and the writeset is applied at the remote replicas in a lazy fashion. Elnikety et al. Elnikety et al. [2006] also take advantage of snapshot isolation concurrency control and presents Tashkent, which further improves system performance by uniting transactions ordering and durability. Although snapshot isolation is an appealing alternative to one-copy serializability, certain anomalies forbidden in serializable executions may occur and may be unacceptable for some applications.

Irún-Briz et al. [2005] present MADIS, a platform that supports a number of different replication protocols and implements an extension to the database schema (called report tables) in order to retrieve information accounting the execution of various transactions at each replica. In Muñoz-Escoí et al. [2006] the authors suggest using database system's locking tables to determine conflicting transactions in middleware-based database replication architectures. Even if all DBMSs would have such tables, their handling is DBMS dependent, what rules out one of the goals of non-intrusive replication to be independent of the underlying database engine.

Pacitti et al. [2005] present preventive replication that supports multi-master and partial configurations. The consistency is guaranteed by annotating transactions with a chronological timestamp value and ensuring FIFO order among messages sent by a replica. The transaction is allowed to execute on a database site only when the upper bound of the time needed to multicast a message has exceeded. Further optimizations of the protocol allow concurrent transactions execution at the replicas and eliminate the delay time. The proposed protocol is also studied in the context of partial database replication. However, the authors assume that transaction access patterns (its readsets and writesets) are known.

The DBSM has several advantages when compared to existing replication schemes. In contrast to lazy replication techniques, the DBSM provides strong consistency (i.e., one-copy serializability) and fault tolerance. When compared with primary-backup replication, it allows transaction execution to be done in parallel on several replicas, which is ideal for read-only transactions and work-

loads populated by a large number of non-conflicting update transactions. By avoiding distributed locking used in synchronous replication, the DBSM scales to a larger number of database sites. Finally, when compared to active replication, it allows better usage of resources because each transaction is completely executed by a single replica only. In this chapter we have addressed one of the main weaknesses of the DBSM: its dependency on transaction readsets for certification. The conflict materialization technique adopted for DBSM* does that without sacrificing strong consistency or increasing communication overhead. Furthermore, we have introduced vDBSM, a practical non-intrusive replication protocol. To provide strong consistency and reduce spurious aborts the vDBSM takes advantage of workload characterization techniques introduced in the upcoming chapter.

Chapter 4

Workload Characterization Techniques

Transaction readsets and writesets information is an essential component of database replication protocols, in particular when strong system consistency is targeted. Such knowledge may be useful even before actually executing the transactions at the database engine. Conservative replication protocols require transaction's access patterns data beforehand to guarantee replicated system's consistency; optimistic protocols may use such information to reduce the synchronization aborts induced by conflicting transactions executing concurrently at distinct database replicas.

Non-intrusive replication solutions have no access to fine grain concurrency control information at the database engine and often suffer from duplicating the back-end database logic into the middleware or from reduced concurrency among transactions and increased number of aborts. However, data items accessed by predefined transactions can be detected automatically before their actual execution by partially parsing SQL statements. By estimating the data items accessed by transactions before their execution, even if conservatively, the replication protocols are spared from extracting readsets and writesets during the execution.

In this chapter we introduce the second contribution of this thesis, the SQL Inspector, a tool capable to automatically identify conflicting transactions by partially parsing them. In the following sections we explain the idea, the actual implementation of the tool, and the case study of the two workloads chosen.

4.1 SQL Statements Inspection

Transactions' writeset can be obtained during transaction execution by using triggers or log sniffing. A database trigger is a procedure that is automatically executed in response to a certain event on a particular table in a database. Most modern DBMS (e.g., Oracle, MySQL, PostgreSQL, Microsoft SQL Server) support triggers (also known as Data Manipulation Language (DML) triggers) that fire on data modification events, such as INSERT, UPDATE and DELETE. Thus, the identifiers of rows modified by the transaction can be obtained by triggers and stored in a temporary table. Some DBMSs (e.g., Microsoft SQL Server, Oracle) also support Data Definition Language (DDL) triggers, which can fire in reaction to a very wide range of events, including: DROP, CREATE or ALTER *table*.

Unfortunately, triggers cannot be used to audit data retrieval via SELECT statements. One may be mistaken by the fact that the identifiers of data items collected while executing some SELECT statement actually define the transaction readset. Primary keys of the records returned after executing SQL statements do not necessary indicate the records physically read by the database. There are many situations when the database scans the whole table to retrieve a set of particular tuples and such cases differ from DBMS to DBMS depending on the concurrency control mechanism employed and the way the database engine handles phantoms [Gray and Reuter, 1993]. To simplify the explanation of the tool hereafter we assume MySQL InnoDB storage engine [*MySQL 5.0 Reference Manual*, 2008], however, most of the general ideas are still applicable to other DBMSs. We outline how locking is implemented in MySQL in Secion 4.1.1.

4.1.1 Locking in MySQL InnoDB

InnoDB implements standard row level locking without lock escalation. There are two types of locks: a shared lock and an exclusive lock. Shared locks are required to read database records and exclusive locks must be obtained when modifying the data. When the database engine searches or scans an index of a table, it sets shared or exclusive locks on the index records it encounters. Thus, the row level locks are actually index record locks. If no suitable index exists, MySQL has to scan the whole table to process the statement, every row of the table becomes locked, which in turn blocks all inserts to the table.

To ensure that no *phantom phenomenon* occurs InnoDB uses *next-key locking*. Phantoms may arise when a transaction accesses tuples via an index: in such cases the transaction typically does not lock the entire table, just the records in the table that are accessed by the index. In the absence of a table level lock,

other transactions may insert or delete records from the same table. If these newly inserted or deleted records satisfy a predicate of another transaction, they will appear is subsequent accesses via that predicate. Although the two transactions conflict logically, there is no record in common: the record that causes the conflict is a *phantom record* that has suddenly appeared/disappeared from the database.

In conventional next-key locking [Gray and Reuter, 1993] an insertion of a tuple with index key k allocates an exclusive lock on the next-key tuple that exists in the index: the tuple with the lowest key greater than k. Thus the subsequent insertions cannot appear in between two tuples that were returned previously by an active transaction; furthermore tuples cannot be inserted just below the lowest-keyed tuple previously returned. The next-key locking protocol requires read transactions to be modified as well, so that they must get a shared lock on next-key tuple in the index as well — the minimum-keyed tuple that does not satisfy the query predicate.

Thus in InnoDB next-key locks affect not only the index record accessed but also the "gap" before it. If there is a shared or an exclusive lock on the record in the index, new index record cannot be inserted immediately before it in the index order. InnoDB sets exclusive or shared next-key locks when executing most of SQL statements, except an INSERT SQL statement — only a simple exclusive lock is set on the inserted row. If a duplicate key error occurs, a shared lock on the duplicate index record is set.

4.1.2 DML

If the workload is known in advance a transaction's readset and writeset can be estimated before their actual execution. In the following we explain how such information can be extracted from SQL statements of Data Manipulation Language.

Readsets analysis

A SELECT query in SQL retrieves a set of tuples from one or more tables. It can consist of up to six clauses, but only two, SELECT and FROM, are mandatory:

```
SELECT <attribute list> FROM <table list> [WHERE <condition>]
                                          [GROUP BY <grouping attributes>]
                                          [HAVING <group condition>]
                                          [ORDER BY <attribute list>]
```

The SELECT clause lists the attributes to be retrieved, the FROM clause specifies all tables required by the query, the WHERE clause defines the conditions for selection of tuples from those tables. GROUP BY specifies grouping attributes, whereas HAVING defines a condition on the groups being selected rather than individual tuples. ORDER BY specifies an order for displaying the result of a query. Omitting the WHERE clause indicates that all tuples of the table are read. If the WHERE keyword is present it is followed by a logical expression, also known as a predicate, which identifies the tuples to be retrieved. If the predicate is an equality comparison on a unique indexed attribute, the exact rows scanned by the database can be estimated. For example, consider the query where the predicate is based on a primary key attribute:

```
SELECT clientName FROM clientsTable WHERE clientId = 10
```

The database engine will lock only the record that is identified by the client's id equal to 10. Thus, the record read by such a query is simply identified by the client's id.

A unique index of the table can be also composed of several attributes. In both cases the database locks only the records identified by their unique indices. However, as explained in Section 4.1.1, to prevent phantoms the database engine may lock part or even the whole table to retrieve a set of particular records. As another example, suppose one wants to obtain all the clients that are older than 30 years. Assume there is a non-unique index on the column `clientAge`:

```
SELECT clientSurname FROM clientsTable WHERE clientAge >30
```

The surnames of the clients obtained by the query are not necessarily the records physically locked by the database engine. The query scans the index starting from the first record where client's age is bigger than 30. If the locks set on the index records would not prevent inserts in the gaps, a new tuple might be meanwhile inserted to the table. If the same SELECT statement is executed again within the transaction, a new row would be returned in the resultset of the query. However, such execution violates the isolation property of transactions [Gray and Reuter, 1993]: even if transactions execute concurrently, it should appear that they were executed one after the other. When InnoDB scans an index, it can also lock the gap after the last record in the index. In the previous example, the locks set by InnoDB forbid inserting new clients older than 30.

Hence to ensure strong consistency we have two alternatives: choose table level granularity when queries retrieve records based on non-unique indices—an undesirable solution due to reduced transaction concurrency, or record level

granularity as if considering primary key indices. However, in the latter case, consistency still needs to be ensured. On the other hand, one might always decide to relax the consistency requirement assuming such phenomena do not arise frequently and they are not crucial for the application considered.

Besides equality, the WHERE clause may contain different comparison operators (<,>,<=,>=), complex expressions that consist of extended functions (LIKE, BETWEEN, NOT NULL) or another (nested/sub-) query. The records physically read by the database processing such queries are DBMS implementation dependent. The same holds when joins on non-unique indices over several tables are performed. Since we aim at non-intrusive replication, table level granularity is used in all the situations above and the cases not covered.
Notice that the OLTP workloads (e.g., TPC [2005, 2001]) are typically composed of SQL statements that mainly use primary indices to retrieve the particular records, and thus row level granularity can be achieved in the majority of the cases.

Writesets extraction

There are three SQL statements for data modification: INSERT, UPDATE and DELETE. INSERT is used to add rows to an existing table and has the following form:

```
INSERT INTO <table> (column1,[column2,...]) VALUES (val1,[val2,...])
```

INSERT sets an exclusive lock on the inserted tuples, thus the exact rows can be estimated by their primary keys.[1] For example, the writeset of the following statement is identified by the primary key attribute, i.e., only the row with client's id equal to 11 is actually locked by the database engine:

```
INSERT INTO clientsTable (clientId,clientName) VALUES (11, 'Jim')
```

UPDATE is used to modify the values of a set of existing table rows:

```
UPDATE <table> SET column1 = val1[, column2 = val2...][WHERE <condition>]
```

Entire rows can be deleted from a table using the DELETE clause:

```
DELETE FROM <table> [WHERE <condition>]
```

The writesets of both, DELETE and UPDATE statements similarly to SELECT, can be estimated at the record level only if the WHERE clause contains an equality comparison on unique identifiers. Otherwise, table level granularity is considered.

[1] Recall that we assume MySQL InnoDB engine. Other DBMSs may handle INSERT and DELETE statements differently (e.g., using table locks).

4.1.3 DDL

The workload we assume in this thesis is composed of predefined, parameterized transactions. Usually such workloads, as for example, TPC-C and TPC-W benchmarks, do not contain any Data Definition Language SQL statements. Although there is nothing fundamentally different about these SQL statements that would prevent our tool to detect the writesets (CREATE, ALTER, DROP, RENAME *table* statements would clearly require table level granularity), currently we do not consider DDL statements in our workload analysis.

4.1.4 The SQL Inspector

Following the aforementioned ideas, we have built the SQL Inspector, a tool to partially parse SQL statements and detect potential conflicts among transactions. In this section we outline the architecture of the tool and detail the steps required to analyze the workload.

Our tool is implemented entirely in Java and relies on the open source SQL parser *JSqlParser* [2008]. The SQL Inspector can be used as a standalone application for transaction analysis or it can be integrated into a replication protocol to simplify the concurrency control management of the replicated system.

Hereafter we assume the following general structure of predefined, parameterized transactions:

```
TransactionName(parameterName1, parameterName2, ...)
sql statements ...
```

Each transaction template is identified by its name (*TransactionName*) and a number of parameters (*parameterName1, parameterName2*). For example, transaction *UpdateOrderTransaction* has a single parameter *paramName* which is the name of the client. This parameter is later used within the SQL statements of the transaction:

```
UpdateOrderTransaction(paramName)
SELECT clientId FROM clientsTable WHERE clientName = paramName
UPDATE ordersTable SET orderAmount = 500 WHERE orderId = clientId
```

Not all SQL statements in a transaction template are executed on every invocation of the transaction. Loops and “if ... else” statements require special attention. Thus *the first step* of the analysis is to rewrite the transactions templates so that every significant path of control flow is presented as a separate

parameterized transaction. In the case when the condition of the "if ... else" statement is not known in advance, the transaction is considered as if all the SQL statements in both execution paths would execute. Being hard to automate, this step is done manually.

The second step of the analysis is the database schema (SQL DDL statements) parsing. Although we do not support DDL statements within transactions, our tool is able to parse SQL statements used to create tables in a database. The Inspector parses CREATE *table* statements to obtain primary keys and other unique indices information for each database table. Such information is necessary to be able to identify conflicts among transactions at the record level. If table level granularity is requested no schema parsing is required.

Finally, during *the third step* of the analysis transaction templates are parsed and a preliminary conflict relation is constructed. Defining a conflict relation for the whole workload is not trivial. First, the parameter space is large: we cannot check all permutations of transactions instantiated with all possible parameter values; and second, the decisions made by the SQL Inspector may vary depending on specific application requests. We identify the following parameters that can affect conflict relation:

1. The conflicts among transactions can be determined at different *granularities*:
   - table level,
   - record level.
2. Based on concurrency control provided by the database engine (2PL or snapshot isolation), different *conflicts types* may be considered:
   - read-write and write-write conflicts,
   - only write-write conflicts.
3. An application may demand different *degrees of consistency*:
   - strong, where no phantom phenomena are allowed, or
   - weaker isolation levels, where phantoms are tolerated.

All combinations of such requirements are possible. For example, the application might request row level granularity but permit weaker isolation; or strong consistency is required but table level granularity is acceptable. Such requirements can be identified by the application developer and given to the SQL Inspector as an input.

The SQL Inspector can be used for online and offline analysis of transactions. The online analysis will certainly introduce some overhead, but may result in more precise conflict information when compared to offline analysis. The online processing costs can be reduced if database schema is preprocessed beforehand. On the other hand, the overhead introduced by offline analysis is close or equal to zero. Depending on the nature of transactions, such offline workload analysis may still result in coarse conflict information. For example, if not all the parameters of the operations within the transaction are known in advance:

```
UpdateOrderTransaction(paramName)
SELECT clientId FROM clientsTable WHERE clientName = paramName
UPDATE ordersTable SET orderAmount = 500 WHERE orderId = clientId
```

Assume there is a unique index on client's name column in the clients table and a primary key index on order's id in the orders table. The client's name is known in advance (*paramName*), but the order's id is obtained only during transaction execution. The database engine would use row level locking for this transaction, however the SQL Inspector has no information about which row is locked in the orders table — the value of order's id is not known beforehand. If there is another transaction accessing the orders table, SQL Inspector would detect a conflict, even though the actual rows locked in the table would be different. A finer granularity of information can be achieved if the same SQL statements inspection strategy is used per operation and online (during real execution of transactions).

The presented SQL statements inspection allows for higher flexibility of replication protocols: ranging from table to record level granularity and providing different consistency guarantees based on application demands. It turns out that such simple SQL analysis covers all the transaction types in standard database benchmarks, i.e. TPC-C and TPC-W. We show next the analysis of these benchmarks.

Being a conservative technique the SQL Inspector uses table level granularity for unsupported statements, such as, explicit locking (e.g., LOCK TABLES, LOCK IN SHARE MODE) or SELECT ... FOR UPDATE, UNION statements.

4.2 Analysis of the Benchmarks

In this section we overview the TPC-W and TPC-C benchmarks, show how they can be mapped to our transactional model, and how our SQL statements inspection can be applied to them.

4.2.1 TPC-W

TPC-W [TPC, 2001] is an industry standard transactional web benchmark used to evaluate e-commerce systems. It implements an on-line bookstore and has three workload mixes that differ in the relative frequency of each of the transaction types. The *ordering* workload mix has 50% updates, the *shopping* workload mix has 20% updates, and the *browsing* workload mix has 5% updates. The bookstore (a total of 8 tables) size is chosen from among a set of given scale factors, which is the number of items in inventory and varies from 1,000 to 10,000,000 items.

TPC-W defines its workload as a mix of *web interactions*. Each web interaction is composed of one or more transactions. Therefore, to be able to use the SQL Inspector, we must identify the transaction templates within the web interactions. This has to be done manually. After careful analysis this resulted in 20 different transaction templates.

To illustrate how SQL Inspector works, consider two very simple transactions, *BuyRequest* and *CustomerRegistration*:

```
BUYREQUEST(c_id)
UPDATE customer SET c_login = ?, c_expiration = ? WHERE c_id = c_id
```

```
CUSTOMERREGISTRATION(c_id)
SELECT c_uname FROM customer WHERE c_id = c_id
```

Since both transactions use a primary key (*c_id*) to access data, readsets and writesets can be identified at the row level. The SQL Inspector determines that the two transactions conflict if they are issued by the same customer, that is, the value of *c_id* is the same in both transactions. The Inspector can also be used on already instantiated transactions, in that case, the result of the analysis is just true or false, i.e., transactions conflict or not.

4.2.2 TPC-C

TPC-C is another benchmark for online transaction processing (OLTP) from TPC [2005]. It represents a generic wholesale supplier workload. The benchmark's database consists of a number of warehouses, each one composed of 10 districts and maintaining a stock of 100000 items; each district serves 3000 customers. All the data is stored in a set of 9 relations: *Warehouse, District, Customer, Item, Stock, Orders, Order Line, New Order,* and *History*.

Being simpler than TPC-W, TPC-C defines only five transaction types: the *New Order* transaction places an order from a warehouse; the *Payment* transaction records a payment received from a customer; the *Delivery* transaction processes pending orders at the specified warehouse; the *Order Status* transaction returns the status of the customer's last order; and the *Stock Level* transaction examines the quantity of stock for the items ordered at the selected warehouse. *Order Status* and *Stock Level* are read-only transactions; the others are update transactions. Since we are interested in readsets and writesets of only update transactions, there are only three transaction types to consider: *Delivery*, *Payment*, and *New Order*. These three transaction types compose 92% of TPC-C workload.

The SQL Inspector can help to detect that, for example, both *New Order* and *Payment* transactions access *Customer*, *District* and *Warehouse* relations, but none ever writes on fields read by the other. However, since traditional databases consider row level granularity, *New Order* and *Payment* transactions might conflict if they operate on the same warehouse. *Delivery* and *Payment* transactions have conflicts only on the *Customer* relation: both update customer's balance information and there might also be a read-write conflict since *Payment* transactions also read this data. *New Order* and *Delivery* transactions conflict because of read and write operations on *New Order, Order Line* and *Orders* tables.

Since TPC-C workload is relatively simple (only 3 transaction templates to consider), careful analysis of the above-mentioned information allows us to define a conflict relation for the whole workload. Let's denote the workload of update transactions as:

$$\mathscr{T} = \{D_i, P_{ijkm}, NO_{ijS} \mid \begin{array}{l} i,k \in 1..\#\text{WH}; \\ j,m \in 1..10; \\ S \subseteq \{1,...,\#WH\}\} \end{array}$$

where #WH is the number of warehouses considered. D_i stands for a *Delivery* transaction accessing districts in warehouse i. P_{ijkm} relates to a *Payment* transaction which reflects the payment and sales statistics on district j and warehouse i and updates the customer's balance. In 15% of the cases, the customer is chosen from a remote warehouse k and district m. Thus, for 85% of transactions of type P_{ijkm}: $(k = i) \wedge (m = j)$. NO_{ijS} is a *New Order* transaction referring to a customer assigned to warehouse i and district j. For an order to complete, some items must be chosen: 99% of the time the item chosen is from the home warehouse i and 1% of the time from a remote warehouse. S represents a set of remote warehouses.

If read-write and write-write conflicts at the row level are considered the conflict relation $\sim$ between transaction types is as follows:

$$
\begin{aligned}
\sim \; = \; & \{(D_i, D_x) \mid (x = i)\} \cup \\
& \{(D_i, P_{xykm}) \mid (k = i)\} \cup \\
& \{(D_i, NO_{xyS}) \mid (x = i)\} \cup \\
& \{(P_{ijkm}, P_{xyzq}) \mid (x = i) \vee ((z = k) \wedge (q = m))\} \cup \\
& \{(NO_{ijS}, NO_{xyZ}) \mid ((x = i) \wedge (y = j)) \vee (S \cap Z \neq \emptyset)\} \cup \\
& \{(NO_{ijS}, P_{xyzq}) \mid (x = i) \vee ((z = i) \wedge (q = j))\}
\end{aligned}
$$

For instance, two *Delivery* transactions conflict if they access the same warehouse.

Notice that we do not have to consider every transaction that may happen in the workload in order to define the conflict relation between transactions. Only the transaction types and how they relate to each other should be taken into account.

4.2.3 Accuracy of the SQL Inspector

The SQL Inspector is a conservative technique and may over-approximate the set of data items accessed by the transaction that could cause conflicts. In other words, the offline workload analysis may result in false positives. To evaluate the accuracy of the SQL Inspector we have estimated the types of locks actually used by the database engine for each transaction in TPC-C benchmark and the granularity of locks achieved by the Inspector. We detect the type of locks used by the database from the transaction execution plan (in MySQL such information can be obtained with EXPLAIN command). The results are presented in Table 4.1. The numbers correspond to the amount of statements within each transaction using the specific kind of lock.[2] Row and table lock types indicate that row or table level locks are used, respectively. Range level lock indicates that part of the table is locked by the transaction.

Notice that there are no table locks introduced by the SQL Inspector. Some row locks are escalated to range locks mainly due to the fact that in offline analysis there are some intermediate values that are not known in advance. Even being a conservative technique, the SQL Inspector achieves finer granularity than just table names accessed by the transactions — an approach taken by some middleware replication solutions [Amza et al., 2003; Patiño-Martínez et al., 2005].

[2]The given number is an estimate and should not be taken as the actual number of statements within each transaction: some of the statements may be executed more than once.

Locks:	Row	Range	Table
NewOrder	10	0	0
Payment	8	1	0
Delivery	4	3	0
OrderStatus	1	3	0
StockLevel	1	1	0

(a)

Locks:	Row	Range	Table
NewOrder	6	4	0
Payment	4	5	0
Delivery	0	7	0
OrderStatus	0	4	0
StockLevel	1	1	0

(b)

Table 4.1. Accuracy of the SQL Inspector, TPC-C benchmark; (a) the locks used by the database, and (b) the granularity achieved by the SQL Inspector

4.3 Related Work and Final Remarks

Little research has considered workload analysis to increase the performance of replicated systems. To the best of our knowledge, only the authors of [Jorwekar et al., 2007] analyze SQL statements, although for a different purpose. The tool implemented, similarly to our SQL Inspector, parses SQL statements to extract syntactic readsets and writesets. Differently from us, the obtained information is used to detect anomalies in snapshot isolated executions. No replication is considered.

To enable transparent heterogeneous database replication C-JDBC [Cecchet et al., 2004] parses SQL statement at the middleware layer. However, it is a complete, conventional SQL parser, which, unlike our SQL Inspector, cannot be used as a standalone application. Readsets and writesets extraction outside the database engine by means of triggers and log mining was also discussed in [Salas et al., 2006]

In this chapter we have shown how to automatically obtain transactions data access pattern information. The proposed SQL inspector allows to estimate, even if conservatively, the conflicts between predefined, parameterized transactions. We have presented the idea and actual implementation of the tool. To the best of our knowledge this is the first attempt to automate the extraction of transactions readsets and writesets based on SQL statements analysis outside the database engine. In the next chapter we show how exploiting the same workload information the performance of database replication protocols can be further improved.

Chapter 5

Conflict-Aware Load-Balancing Techniques

Optimistic database replication protocols, especially non-intrusive ones, may suffer from excessive synchronization aborts. In such protocols, each transaction is first executed locally on some replica and during execution there is no synchronization between database sites. If two conflicting transactions execute concurrently on distinct sites, one of them is aborted during certification to ensure strong consistency (e.g., one-copy serializability). A key property of the vDBSM is that if transactions with similar access patterns execute at the same database replica, then the local replica's scheduler will serialize them and both can commit reducing the synchronization aborts. Based on the information obtained by the SQL Inspector, we can carefully assign transactions to database replicas avoiding conflicts as much as possible.

In this chapter we introduce conflict-aware load-balancing techniques that schedule transactions in vDBSM to *preferred database sites*. Our algorithms strive to assign transactions to replicas so that the number of conflicting transactions executing on distinct database sites is reduced and the load over the replicas is equitably distributed. In the following sections we present two greedy algorithms that prioritize different requirements when assigning transactions to preferred database sites: *Minimizing Conflits First (MCF)* and *Maximizing Parallelism First (MPF)*. We illustrate the behavior of the algorithms with the chosen benchmark: TPC-C and a micro-benchmark, and present a thorough evaluation of vDBSM with and without the load-balancing techniques.

5.1 Minimizing Conflicts and Maximizing Parallelism

The proposed conflict-aware load-balancing techniques build on two simple observations: (I) If conflicting transactions are submitted to the same database site, the replica's local scheduler serializes the conflicting operations appropriately, reducing aborts. (II) In the absence of conflicts, however, performance is improved if transactions execute concurrently on different replicas. Designing load-balancing techniques that exploit the given observations efficiently is not trivial. Concentrating conflicting transactions in a few replicas will reduce the abort rate, but may leave many replicas idle.

Ideally, we would like to both (a) *minimize* the number of conflicting transactions executing on distinct replicas and (b) *maximize* the parallelism between transactions. If the workload is composed of many conflicting transactions and the load over the system is high, then (a) and (b) become opposite requirements: While (a) can be satisfied by concentrating transactions on few database sites, (b) can be fulfilled by spreading transactions on multiple replicas. But if only few transactions conflict, then maximizing parallelism becomes the priority; likewise, if the load is low, few transactions will execute concurrently and minimizing the number of conflicting transactions executing at distinct replicas becomes less important.

We propose a hybrid load balancing technique which allows to give more or less significance to minimizing conflicts or maximizing parallelism. We call it *Maximizing Parallelism First*(MPF). MPF prioritizes parallelism between transactions. Consequently, it initially tries to assign transactions in order to keep the replicas' load even. If more than one option exists, the algorithm attempts to minimize conflicts.

To account for the computational resources needed to execute different transactions, each transaction T_i in the workload is assigned a *weight* w_i. For example, simple transactions have less weight than complex transactions. The load of a replica s_k is given by the aggregated weight of active transactions at some given time t, denoted as $w(s_k, t) = \sum_{T_j \in s_k^t} w_j$.
To compare the load of two database sites, we use factor $f, 0 < f \leq 1$. Sites s_i and s_j have similar load at time t if the following condition holds: $f \leq w(s_i, t)/w(s_j, t) \leq 1$ **or** $f \leq w(s_j, t)/w(s_i, t) \leq 1$. For example, MPF with $f = 0.5$ allows the difference in load between two replicas to be up to 50%. We denote MPF with a factor f as MPF f. MPF works as follows:

1. Consider replicas s_1, s_2, ..., s_n. To assign each transaction T_i in the workload to some site at time t execute steps 2–4, if T_i is an update transaction,

or step 5, if T_i is a read-only transaction.

2. Let $W(t) = \{s_k \mid w(s_k,t) * f \leq \min_{l \in 1..n} w(s_l,t)\}$ be the set of replicas with the lowest aggregated weight $w(s_l,t)$ at time t.

3. If $|W(t)| = 1$ then assign T_i to the replica in $W(t)$.

4. If $|W(t)| > 1$ then let $C_W(T_i,t)$ be the set of replicas containing conflicting transactions with T_i in $W(t)$: $C_W(T_i,t) = \{s_k \mid s_k \in W(t)$ and $\exists T_j \in s_k$ such that $T_i \sim T_j\}$.

   (a) If $|C_W(T_i,t)| = 0$, assign T_i to the s_k in $W(t)$ with the lowest aggregated weight $w(s_k,t)$.

   (b) If $|C_W(T_i,t)| = 1$, assign T_i to the replica in $C_W(T_i,t)$.

   (c) If $|C_W(T_i,t)| > 1$, assign T_i to the replica s_k in $C_W(T_i,t)$ with the highest aggregated weight of transactions conflicting with T_i; if several replicas in $C_W(T_i,t)$ satisfy this condition, assign T_i to any of these.

   More formally, let $C_{T_i}(s_k^t)$ be the subset of s_k^t containing conflicting transactions with T_i only: $C_{T_i}(s_k^t) = \{T_j \mid T_j \in s_k^t \wedge T_j \sim T_i\}$. Assign T_i to the replica s_k in $C_W(T_i,t)$ with the greatest aggregated weight $w(C_{T_i}(s_k^t)) = \sum_{T_j \in C_{T_i}(s_k^t)} w_j$.

5. Assign read-only transaction T_i to the replica s_k with the lowest aggregated weight $w(s_k,t)$ at time t.

The choice of f depends heavily on workload characteristics: the number of conflicting transactions, their complexity, and the load over the system.

We call *Minimizing Conflicts First*(MCF) a special case of MPF with a factor $f = 0$. MCF attempts to minimize the number of conflicting transactions assigned to different replicas, even though this may create an unbalance in the load of replicas. The algorithm initially tries to assign each transaction T_i in the workload to the replica containing conflicting transactions with T_i. If there are no conflicts, the algorithm tries to balance the load among the replicas.

Notice that regardless of the database site chosen for the execution of a transaction, the vDBSM always ensures strong consistency.

5.2 Static vs. Dynamic Load Balancing

The proposed conflict-aware load-balancing techniques can be implemented in a static or in a dynamic load balancer. A static load balancer executes MCF

and MPF offline, considering each transaction in the workload at a time in some order—for example, transactions can be considered in decreasing order of weight, or according to some time distribution, if available. Since the assignments are pre-computed, during the execution there is no need for the replicas to send feedback information to the load balancer. The main drawback of this approach is that it can potentially make poor assignment decisions.

Dynamic load balancing can potentially outperform static load balancing by taking into account information about the execution of transactions when making assignment choices. Moreover, the approach does not require any preprocessing since transactions are assigned to replicas on-the-fly, as they are submitted. As a disadvantage, a dynamic scheme requires feedback from the replicas with information about the execution of transactions. Receiving and analyzing this information may introduce overheads.

MCF and MPF can be implemented in a dynamic load balancer as follows: The load balancer keeps a local data structure $s[1..n]$ with information about the current assignment of transactions to each database site. Each transaction in the workload is considered at a time, when it is submitted by the client, and assigned to a replica according to MCF or MPF. When a database site s_k finishes the execution of a transaction T_i, committing or aborting it, s_k notifies the load balancer. Upon receiving the notification of termination from s_k, the load balancer removes T_i from $s[k]$.

A key difference between static and dynamic load balancing is that the former will only be effective if transactions are pre-processed in a way that resembles the real execution. For example, assume that a static assignment considers that all transactions are uniformly distributed over a period of time, but in reality some transaction types only occur in the first half of the period and the other types in the second half. Obviously, this is not an issue with dynamic load balancing.

Another aspect that distinguishes static and dynamic load balancing is membership changes, that is, a new replica joins the system or an existent one leaves the system (e.g., due to a crash). Membership changes invalidate the assignments of transactions to database sites. Until MCF and MPF are updated with the current membership, no transaction will be assigned to a new replica joining the system, for example. Therefore, with static load balancing, the assignment of preferred replicas has to be recalculated whenever the membership changes. Notice that adapting to a new membership is done for performance, and not consistency, since the certification test of the vDBSM does not rely on transaction assignment information to ensure one-copy serializability; the consistency of the system is always guaranteed, even though out of date transaction assignment

information is used.

Adjusting MCF and MPF to a new system membership using a dynamic load balancer is straightforward: as soon as the new membership is known by the load balancer, it can update the number of replicas in either MCF or MPF and start assigning transactions correctly. With static load balancing, a new membership requires executing MCF or MPF again for the complete workload, which may take some time. To speed up the calculation, transaction assignments for configurations with different number of "virtual replicas" can be done offline. Therefore, if one replica fails, the system switches to a pre-calculated assignment with one replica less. Only the mapping between virtual replicas to real ones has to be done online.

We use a static load balancer to analyze the example workloads (see Section 5.3), and a dynamic load balancer in the prototype evaluation (Section 5.4).

5.3 Analysis of the Benchmarks

We are interested in the effects of our conflict-aware load-balancing algorithms when applied to different workloads. In the following sections we first demonstrate the behavior of the proposed techniques with a simple example workload and then we provide a more realistic analysis using the TPC-C workload. For illustrative purposes in this section we present only a static analysis of the benchmarks. We analyze the behavior of our algorithms as if all transaction types were submitted to the system simultaneously. To keep our characterization simple, we will assume that the weights associated with the workload represent the frequency in which transactions of some type may occur in a run of the benchmark.

5.3.1 A Simple Example

Consider a workload with 10 transactions, $T_1, T_2, ..., T_{10}$, running in a system with 4 replicas. Transactions with odd index conflict with transactions with odd index; transactions with even index conflict with transactions with even index. Each transaction T_i has weight $w(T_i) = i$. All transactions are submitted concurrently to the system and the load balancer processes them in decreasing order of weight.

MCF will assign transactions T_{10}, T_8, T_6, T_4, and T_2 to s_1; T_9, T_7, T_5, T_3, and T_1 to s_2; and no transactions to s_3 and s_4. MPF 1 will assign T_{10}, T_3, and T_2 to s_1;

T_9, T_4, and T_1 to s_2; T_8 and T_5 to s_3; and T_7 and T_6 to s_4. MPF 0.8 will assign T_{10}, T_4, and T_2 to s_1; T_9 and T_3 to s_2; T_8 and T_6 to s_3; and T_7, T_5, and T_1 to s_4.

MPF 1 creates a balanced assignment of transactions. The resulting scheme is such that $w(s_1) = 15, w(s_2) = 14, w(s_3) = 13$, and $w(s_4) = 13$. Conflicting transactions are assigned to all database sites however. MCF completely concentrates conflicting transactions on distinct replicas, s_1 and s_2, but the aggregated weight distribution is poor: $w(s_1) = 30, w(s_2) = 25, w(s_3) = 0$, and $w(s_4) = 0$, that is, two replicas are idle. MPF 0.8 is a compromise between the previous schemes. Even transactions are assigned to s_1 and s_3, and odd transactions to s_2 and s_4. The aggregated weight is fairly balanced: $w(s_1) = 16, w(s_2) = 12, w(s_3) = 14$, and $w(s_4) = 13$.

5.3.2 Scheduling TPC-C

To analyze TPC-C, we studied the load distribution over the database sites and the number of conflicting transactions executing on different replicas. To measure the load, we use the aggregated weight of all transactions assigned to each replica. To measure the conflicts, we use the *overlapping ratio* $O_R(s_i, s_j)$ between database sites s_i and s_j, defined as the ratio between the aggregated weight of update transactions assigned to s_i that conflict with update transactions assigned to s_j, and the aggregated weight of all update transactions assigned to s_i. For example, consider that T_1, T_2, and T_3 are assigned to s_i, and T_4, T_5, T_6, and T_7 are assigned to s_j. T_1 conflicts with T_4, and T_2 conflicts with T_6. Then the overlapping ratio for these replicas is calculated as follows:

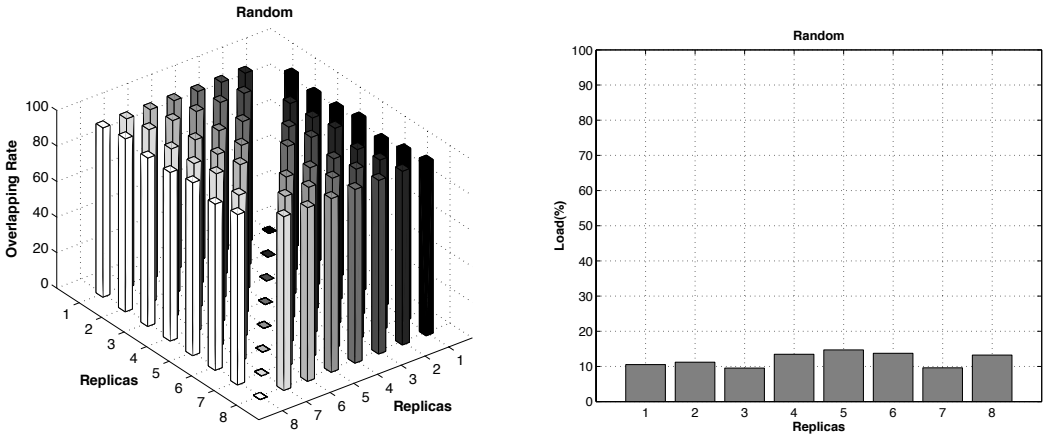
$$O_R(s_i, s_j) = \frac{w(T_1) + w(T_2)}{w(T_1) + w(T_2) + w(T_3)},$$

$$O_R(s_j, s_i) = \frac{w(T_4) + w(T_6)}{w(T_4) + w(T_5) + w(T_6) + w(T_7)}.$$

Notice that since our analysis here is static, the overlapping ratio gives a measure of "potential aborts"; real aborts will only happen if conflicting transactions are executed concurrently on different replicas. Clearly, a high risk of abort translates into more real aborts during the execution.

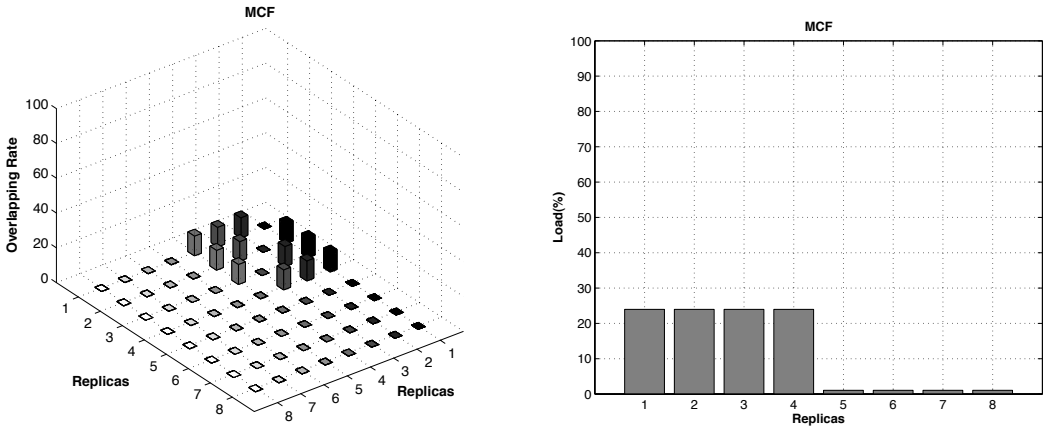
For the analysis we have considered 4 warehouses (i.e., #WH = 4) and 8 database replicas, resulting in total of 2244 transactions. The order in which transactions are considered by the load balancer matters to the final distribution obtained. Evaluating all possible permutations would take exponential time (with the input size). Instead the load balancer processes the requests sequen-

tially in some random order. The results shown correspond to the lowest overlapping rates found after executing each algorithm with 1000 different inputs.[1] We compared the behavior of MCF and MPF, with various f, with a random assignment of transactions to replicas (dubbed Random). The random assignment is what we would expect when no policy is used to schedule transactions. In such a case, transactions would be randomly assigned to database replicas. The results are presented in Figures 5.1 – 5.6.



(a) (b)

Figure 5.1. Random: overlapping rate and load distribution over the replicas



(a) (b)

Figure 5.2. MCF: overlapping rate and load distribution over the replicas

[1]We have also run longer analysis based on 200 000 executions, but found that the improvement in overlapping rates is less that 1% with respect to the best result found with 1000 inputs.

A random assignment of transactions to replicas results in a very high overlapping ratio although the load is distributed over all the replicas (Fig. 5.1). As expected, MCF (see Fig. 5.2) minimizes significantly the number of conflicts, but update transactions are distributed over 4 replicas only; the other 4 replicas execute just read-only transactions. This is a consequence of TPC-C and the 4 warehouses considered. Even if more replicas were available, MCF would still strive to minimize the overlapping ratio, assigning update transactions to only 4 replicas.

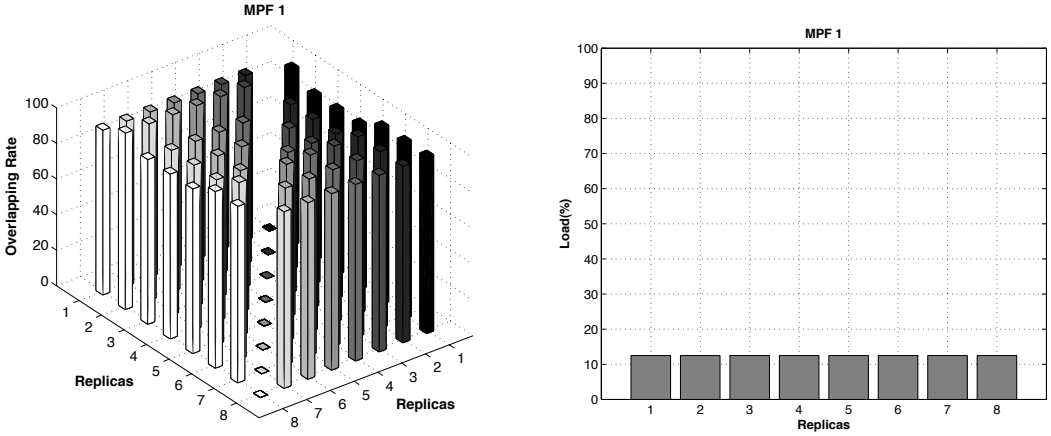


Figure 5.3. MPF 1: overlapping rate and load distribution over the replicas

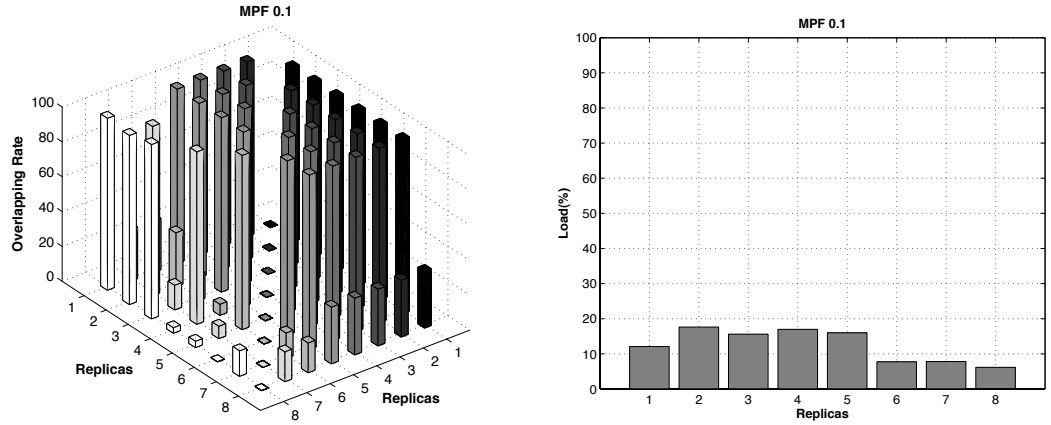


Figure 5.4. MPF 0.1: overlapping rate and load distribution over the replicas

While MPF 1 (see Fig. 5.3) distributes the load equitably over the replicas, similarly to Random, it has a very high overlapping ratio. A compromise between maximizing parallelism and minimizing conflicts can be achieved by varying the f factor of the MPF algorithm. Figures 5.4, 5.5 and 5.6 present the results

obtained with MPF 0.1, MPF 0.5 and MPF 0.8. MPF 0.1 allows up to 90% of load difference between the replicas, but, giving more importance to conflicts, results in much lower overlapping ratio than MPF 1 and Random. With $f = 0.5$ and $f = 0.8$ more significance is given to parallelism and thus the load among the replicas is distributed better but also the overlap is higher. MPF with different f allows to trade even load distribution for low transaction aborts. Finally, notice that TPC-C transactions have very sparse data access pattern (i.e., no "hot-spots") and a small database schema. The benchmark was conceived to avoid a "perfect distribution of transactions" (i.e., no conflicts and perfect load balance).

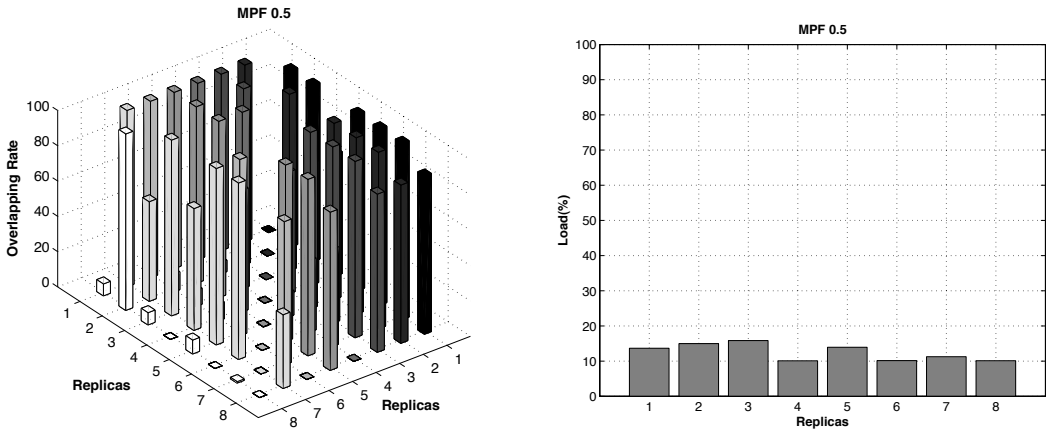


Figure 5.5. MPF 0.5: overlapping rate and load distribution over the replicas

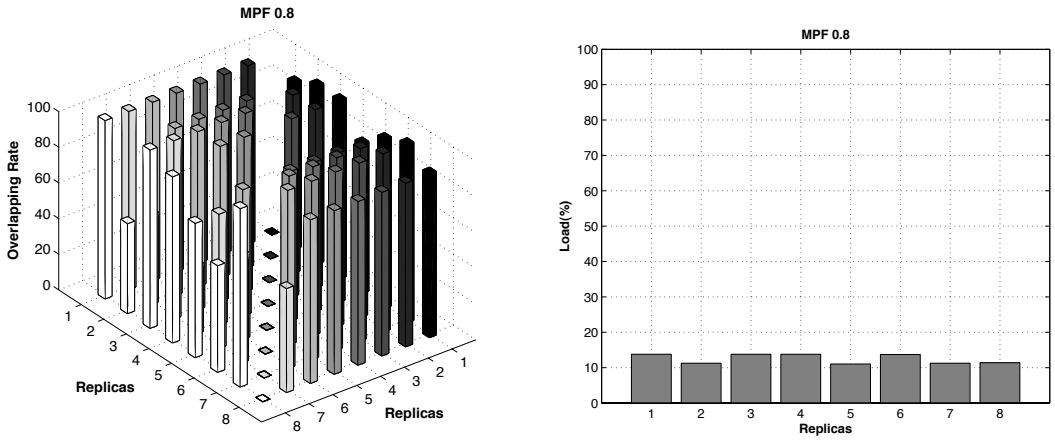


Figure 5.6. MPF 0.8: overlapping rate and load distribution over the replicas

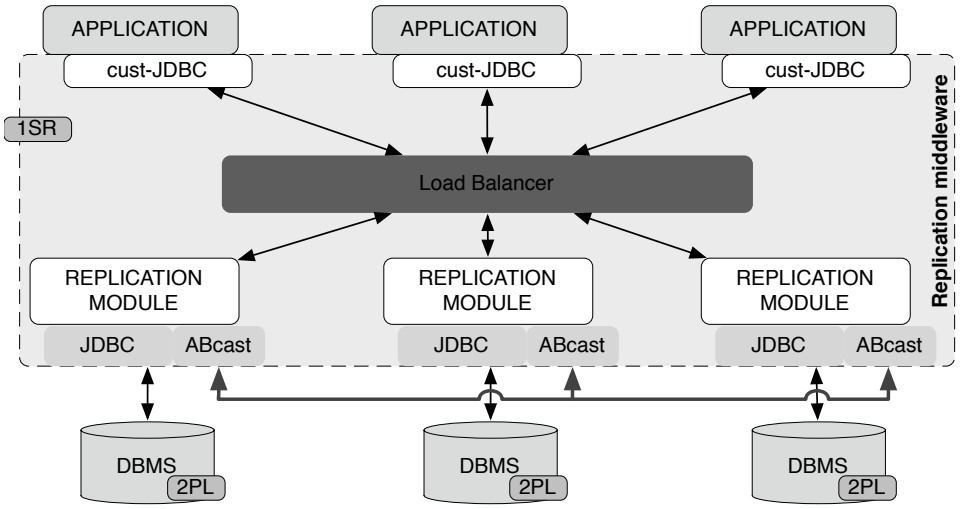


Figure 5.7. Prototype architecture

5.4 Evaluation

In order to evaluate the effects of the proposed conflict-aware load-balancing techniques on the performance of the replicated compound, we have built a prototype of the vDBSM and implemented MCF and MPF on top of it. This section describes the prototype developed and presents the experimental results obtained in a cluster of database sites running the TPC-C benchmark.

5.4.1 Prototype Overview

The prototype of the vDBSM has been implemented entirely in Java v.1.5.0 (Fig. 5.7). Client applications interact with the replicated compound by submitting SQL statements through a customized JDBC-like interface. Application requests are sent to the *load balancer* and then re-directed to a database site. A *replication module* at each site is responsible for executing transactions against the local database, and certifying and applying them in case of commit. Every transaction received by the replication module is submitted to the database through the standard JDBC interface. The communication between clients, replicas and the load balancer uses Unix sockets. Update transactions are broadcast to all replicas using a communication library implementing the Paxos algorithm [*The DaisyLib/Fractal project*, 2007].

On delivery, transactions are enqueued for certification. While transactions execute concurrently in the database, their certification and possible commit-

ment are sequential. The current versions of the data items are kept in main memory to speed up the certification process; however, for persistency, every row in the database is extended with a version number. If a transaction passes the certification test, its updates are applied to the database and the versions of the data items written are incremented both in the database, as part of the committing transaction, and in main memory. We group the updates of several remote transactions into a single transaction in order to reduce the number of disk writes.

To ensure that all replicas commit transactions in the same order, before applying T_i's updates, the database site aborts every locally executing conflicting transaction T_j. To see why this is done, assume that T_i and T_j write the same data item d_x, each one executes on different replicas s_i and s_j, respectively. T_i is delivered first, and both transactions pass the certification test. T_j already has a lock on d_x at replica s_j, where it executed, but T_i should update d_x first. We ensure correct commit order by aborting T_j at s_j and re-executing its updates later. If T_j keeps a read lock on d_x, it is a doomed transaction, and in any case it would be aborted by the certification test later.

The assignment of submitted transactions to database sites is computed on-the-fly based on currently executing transactions at the replicas. The load balancer keeps track of each transaction's execution and completion status at the replicas. No additional information exchange is needed for locally executing transactions. For better load estimation, the replication modules periodically inform the load balancer about the remote updates waiting to be applied at the replicas. Our load balancer is lightweight: CPU usage at the load balancer is less than 4% throughout all experiments.

5.4.2 Experimental Setup

The experiments were run in a cluster of Apple Xservers equipped with a dual 2.3 GHz PowerPC G5 (64-bit) processor, 2GB DDR SDRAM, and an 80GB 7200 rpm disk drive. Each server runs Mac OS X Server v.10.4.11. The servers are connected through a switched 1Gbps Ethernet LAN. We used MySQL v. 5.0.67 with InnoDB storage engine as our database server. The isolation level was set to serializable throughout all experiments.

Each server stores a TPC-C database, populated with data for 8 warehouses. In all experiments clients submit transactions as soon as the response of the previously issued transaction is received (i.e., we do not use TPC-C think times). Besides Random and MCF, we also report results of MPF with f values ranging from 0.1 to 1. We measure throughput in terms of the number of committed

transactions per minute (tpm). The response time reported represents the mean response time of committed transactions in milliseconds (msec). We further assume that all transactions in the workload have the same weight.[2]

5.4.3 Throughput and Response Time

We initially studied the effects of our techniques under various load conditions. Figure 5.8 shows the achieved throughput and response time of committed transactions on a system with 4 and 8 database replicas. MCF, which primarily takes conflicts into consideration, suffers from poor load distribution over the replicas and fails to improve the throughput. Even though the aborts due to lack of synchronization are reduced significantly, the response time grows fast. Response time increases as a consequence of all conflicting transactions executing on the same replica and competing for the locks on the same data items.

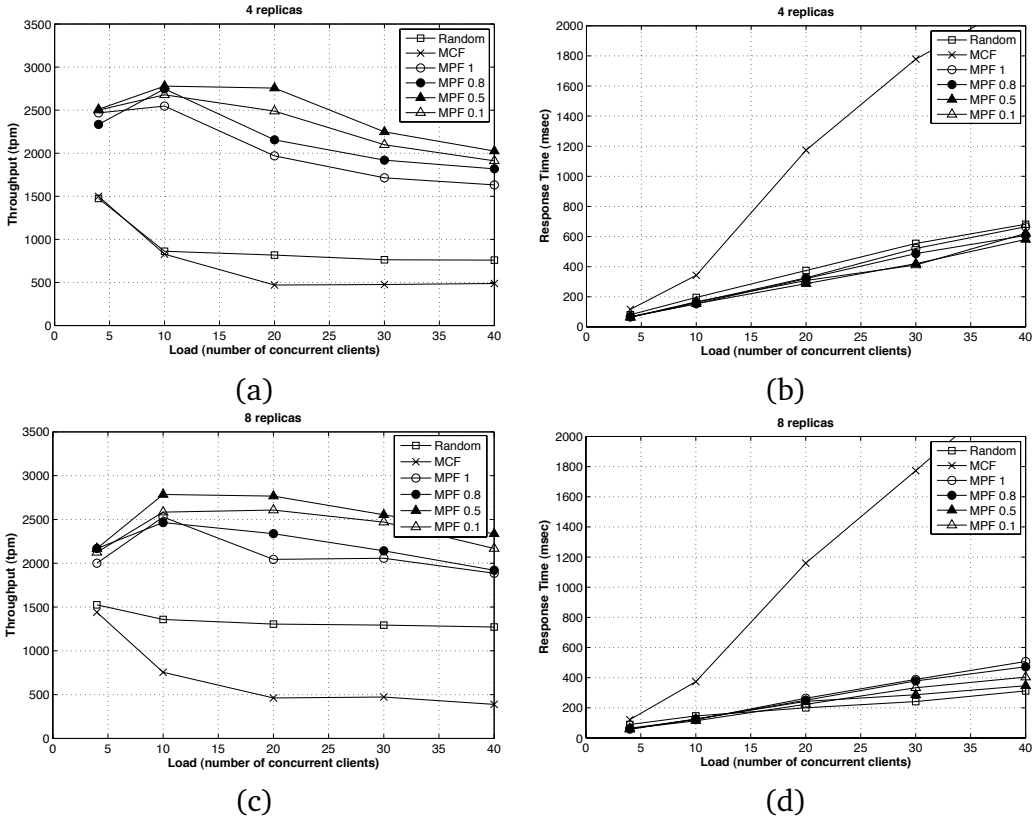
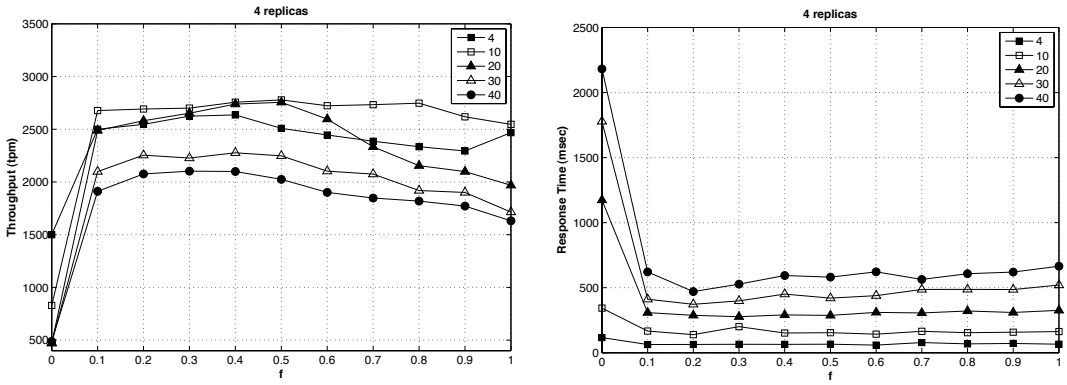


Figure 5.8. vDBSM: Throughput and response time

[2]A more realistic estimation of weight could be the average execution time of transaction.

Prioritizing parallelism (MPF 1) doubles the achieved throughput when compared to Random (Fig. 5.8(a)). Although Random assigns transactions equitably to all replicas, differently from MPF 1, it does not account for the various execution times of transactions. Under light loads MPF 1 and the hybrid load-balancing techniques, MPF 0.1, MPF 0.5 and MPF 0.8, which consider both conflicts between transactions and the load over the replicas, demonstrate the same benefits in performance. If the load is low, few transactions will execute concurrently and minimizing the number of conflicting transactions executing at distinct replicas becomes less effective. However, once the load is increased, the hybrid techniques outperform MPF 1. The same conclusions hold for both experimental setups, i.e., the systems with 4 (Fig. 5.8(a)(b)) and 8 (Fig. 5.8(c)(d)) database replicas.

To better understand the effects of the load difference allowed among the replicas we have varied the factor f from 0 to 1. The results obtained are depicted in Figure 5.9. Each curve corresponds to a different number of concurrently executing clients.



(a) (b)

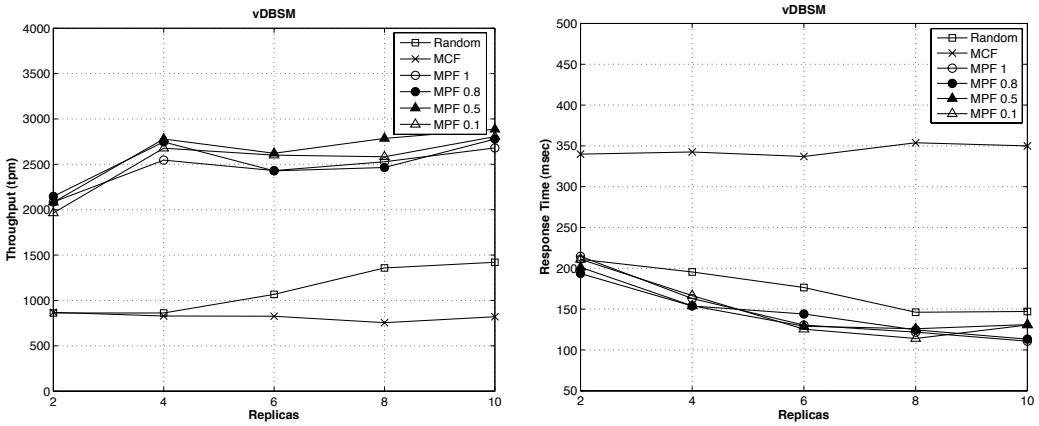
Figure 5.9. MPF, the effects of the parameter f

It appears that choosing the parameter f is not a trivial task. Except for $f = 0$, the response time of the transactions is nearly the same for all f values, slightly growing with the increasing load (Fig. 5.9(b)). Under the given experimental setup the higher throughput is achieved with MPF where f values range from 0.2 to 0.5 (Fig. 5.9(a)), that is, the TPC-C workload benefits more from the conflict-aware load-balancing techniques when more importance is given to conflicts than load distribution.

We then considered how the proposed algorithms react to a varying number of replicas. Notice that adding new replicas in an update intensive environment

(92% af all transactions in TPC-C are updates) will improve the availability of the system but may not increase its performance. In the vDBSM, transactions are executed and broadcast by one database site, then delivered, validated, and applied by all replicas. More replicas may help spreading the load of executing transactions, but in any case, their updates will be processed at every database site. Moreover, in some techniques adding more replicas increases the chance that conflicting transactions are executed on distinct replicas, consequently causing more aborts.

For the following experiments, we kept the load constant at 10 concurrently-executing clients while increasing the number of replicas from 2 to 10. Figure 5.10 shows that the effects of our load-balancing algorithms are sustainable. In fact, Random benefits from an increasing number of replicas. However neither Random nor MCF reach the performance of MPF, which behave similarly for all configurations considered. From these experiments, it turns out that increasing the availability of the system with MPF can be achieved without performance penalties.



(a) (b)

Figure 5.10. vDBSM, varied number of replicas: Throughput and response time

5.4.4 Abort Rate Breakdown

To analyze the effects of conflict-awareness we present a breakdown of abort rate. There are four main reasons for a transaction to abort: (i) it fails the certification test, (ii) it holds locks that conflict with a committing transaction (see Section 5.4.1), (iii) it times out after waiting for too long to obtain a lock, and (iv) it is aborted by the database engine to resolve a deadlock. Notice that

aborts due to conflicts are similar in nature to certification aborts, in that they both happen due to the lack of synchronization between transactions during the execution. Thus, a transaction will never be involved in aborts of type (i) or (ii) due to another transaction executing on the same replica.

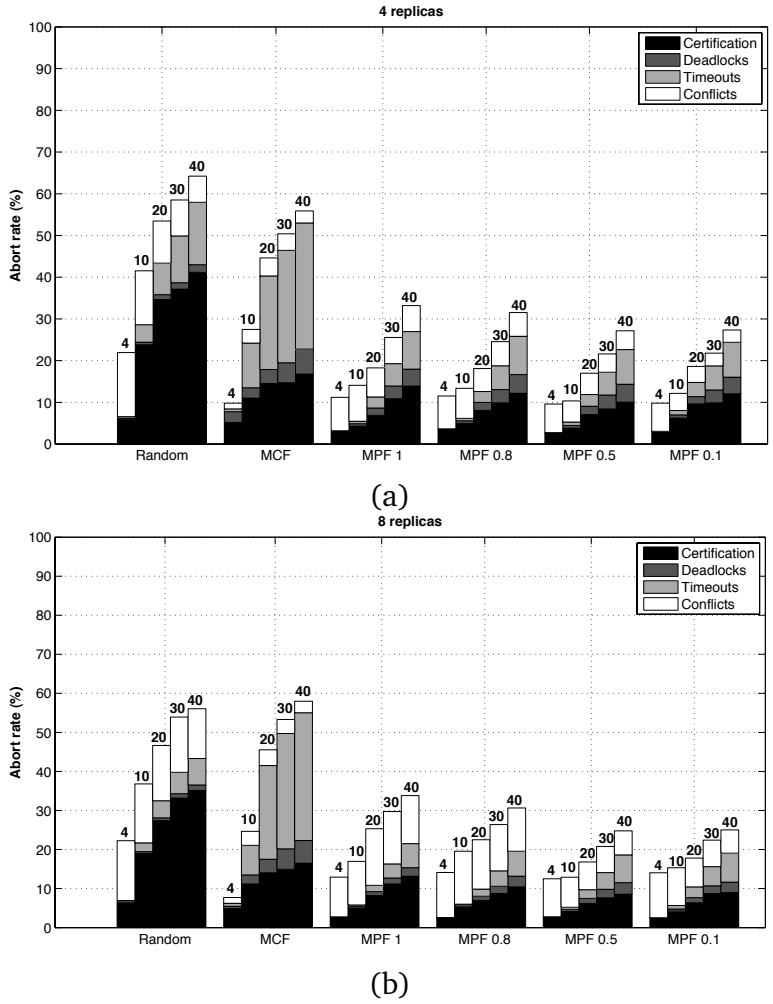


Figure 5.11. Abort rates, 4 and 8 replicas

Figure 5.11 shows the abort rate breakdown for each of the techniques run on the system with 4 and 8 replicas; each vertical bar per technique represents different submitted load. Random and MPF 1 lead to aborts mainly due to conflicts and certification, whereas aborts in MCF are primarily caused by timeouts. Due to better precision in load balancing and conflict-awareness, MPF 1 also results in lower abort rates when compared to Random. MCF successfully reduces the number of aborts due to lack of synchronization. However, increasing the

load results in many timeouts caused by conflicting transactions competing for locks. A hybrid approach, which tries to account for both conflicts and load, is able to reduce the number of aborts significantly. In particular, MPF 0.5 decreases the abort rate from ≈ 40% to ≈ 11% (data corresponds to 10 clients and 4 replicas setup).

5.5 Related Work and Final Remarks

We focus on related work in the area of database replication where some form of load balancing techniques are used.

In [Milán-Franco et al., 2004] the authors introduce a two-level dynamic adaptation technique for replicated databases. At the local level the algorithms strive to maximize performance of each individual replica by taking into account the load and the replica's throughput to find the optimum number of transactions that are allowed to run concurrently. At the global level the system tries to distribute the load over all the replicas considering the number of active transactions and their execution times. Differently from our approach, this work does not consider transaction conflicts for load balancing.

Elnikety et al. [2007] propose a load balancing technique that takes into account transactions memory usage. Transactions are assigned to replicas in such a way that memory contention is reduced. To lower the overhead of updates propagation in a replicated system, the authors also present a complementary optimization called update filtering. Replicas only apply the updates that are needed to serve the workload submitted, i.e., transaction groups are partitioned across replicas. Differently from ours, the load balancer in [Elnikety et al., 2007] doesn't consider conflicts among transactions. Further, if workload characteristics change, the assignment of transaction groups to replicas requires complex reconfiguration, which is limited if update filtering is used. On the contrary, our load-balancing decisions are made per transaction.

Clustered JDBC (C-JDBC) [Cecchet et al., 2004] uses round-robin, or weighted round-robin or least pending requests first for transactions scheduling to the database replicas. Similarly, the Ganymed scheduler [Plattner and Alonso, 2004] assigns read-only transactions to the slave replicas according to least-pending-requests-first rule. However, none of these approaches exploit transaction conflicts information.

A thorough study of load balancing and scheduling strategies is performed in [Amza et al., 2005]. Conflict-aware scheduling [Amza et al., 2003] is the winning technique of all considered. The scheduler is extended to include con-

flict awareness in the sense that requests are scheduled to replicas that are up-to-date. Unlike in our load balancing techniques, conflict awareness is at a coarse granularity, i.e., table. Further, if the scheduler fails, the system needs to deal with a complicated recovery procedure to continue functioning correctly; whereas in our approach the load balancer is independent of the system's correctness—even if the load-balancer fails, transactions can execute at any replica without hampering consistency.

In summary, to keep low abort rate despite the coarse granularity of non-intrusive replication protocols, we introduced conflict-aware load-balancing techniques. The proposed algorithms attempt to reduce the number of conflicting transactions executing on distinct database sites and seek to increase the parallelism among replicas. MCF concentrates conflicting transactions on a few replicas reducing the abort rate, but leaves many replicas idle and overloads others; MPF with the sole goal of maximizing parallelism distributes the load over the replicas, but ignores conflicts among transactions. A hybrid approach, combining MCF and MPF, allows database administrators to trade even load distribution for low transaction aborts in order to increase throughput with no degradation in response time.

Chapter 6

The Cost of Correctness Criteria for Non-Intrusive Database Replication

Just like kernel-based protocols, non-intrusive database replication solutions can implement a large variety of consistency criteria, ensuring different degrees of guarantees to the system's clients. A classic example of consistency criterion in replicated systems is *one-copy serializability* [Bernstein et al., 1987].

It has been generally believed that additional constraints on correctness degrades the performance of a replicated system. To verify this statement, in this chapter we investigate the performance cost of implementing different consistency degrees in middleware protocols. In the following sections we analyze correctness criteria for replicated databases from the client's perspective and present a uniform characterization of them. We further study the effects of different consistency criteria in the context of three replication protocols, each representing a different protocol class. The first protocol we consider falls into the primary-backup replication category; the other two protocols belong to the update everywhere replication class, however they differ in the execution mode: one executes transactions optimistically, the other, conservatively.

The conservative protocol, named BaseCON, is a simple yet fault-tolerant middleware-based replication protocol that takes advantage of workload characterization techniques to increase the parallelism in the system. We present three variants of BaseCON, one for each correctness criterion discussed, and analyze their behavior in case of failures and false suspicions.

6.1 Correctness Criteria

A great number of replication protocols proposed (e.g., Amir and Tutu [2002]; Patiño-Martínez et al. [2000, 2005]; Pedone and Frølund [2008]; Pedone et al. [1998, 2003]; Rodrigues et al. [2002]) guarantee *one-copy serializability* [Bernstein et al., 1987]. Informally, 1SR requires the execution of concurrent transactions on different replicas to appear as if transactions were executed in some sequential order on a single replica. 1SR does not necessarily preserve the order in which transactions are submitted to the system. It allows the situation where transaction T_j may not see the effects of T_i, even though T_i commits before T_j started executing. Although some applications can accept this for performance, in most cases transactions expect to read the updates of preceded transactions. For example, the effects of an update transaction should be seen by a successive read-only transaction issued by the same client. In practice, transactions of the same client are executed within a session. Thus, at least transactions issued in one session should see the effects of each other.

In order to capture this additional requirement, *session consistency* (SC) was introduced [Daudjee and Salem, 2004].

Definition 3. *History H is session consistent iff there is some serial history H_s such that (a) $H \equiv H_s$ (i.e., H is 1SR) and (b) for any two transactions T_i and T_j that belong to the same session, if the commit of T_i precedes the submission of T_j in real time, then T_i commits before T_j is started in H_s.*

SC is stronger than one-copy serializability: it preserves the real-time ordering of transactions submitted in the same session only; as a consequence, clients always read their own updates.

Consider the following example. Two clients share the same account on a flight reservation system. The first clients reserves a flight and tells the other to check it. The second client connects to the system, but he does not find the reservation. Such situation is allowed by SC since only transactions within the same session, i.e., submitted by the same client, must maintain the real-time order.

Thus, even stronger properties must be defined if we require that all transactions in the workload preserve the real-time order, i.e., any transaction reads updates of previously committed transactions. Such real-time ordering of transactions is captured by *strong serializability* (strong 1SR) introduced in [Sethi, 1982].

Definition 4. *History H is* strongly serializable *iff there is some serial history H_s such that (a) $H \equiv H_s$ and (b) for any two transactions T_i and T_j, if the commit of*

T_i precedes the submission of T_j in real time, then T_i commits before T_j is started in H_s.

Both session consistency and strong serializability strengthen the original correctness criterion by restricting what transactions are allowed to read. Thus, the notion of these stronger properties is valid regardless of how the original correctness criterion handles the execution of transactions. SC and strong 1SR are not fundamentally related to 1SR and they could be applied to other correctness criteria such as *snapshot isolation* [Berenson et al., 1995].

6.2 Replication Protocols

In this section we first present our classification of the replication protocols and then discuss how each of the correctness criteria can be achieved in the representative solution of each protocol class.

There have been several attempts to classify the existing replication approaches (e.g., [Gray et al., 1996; Wiesmann et al., 2000a,b]). Gray et al. [1996] group database replication protocols according to where transaction updates take place and when updates are applied to other database replicas. We limit the protocols considered to group communication-based replication and focus on the first parameter protocols, i.e., primary-backup and update everywhere replication. Update everywhere approaches can further differ in transaction's execution mode: transactions can be executed conservatively (pessimistically) or optimistically. In what follows we discuss each of the protocol classes.

6.2.1 Primary-Backup Replication

Primary-backup replication requires all update transactions to be submitted to the same dedicated database site, the primary replica. The requests are executed at the primary and only the updates are sent to the backups. Because there is only one replica executing the transactions, there are no conflicts across the database site. The local concurrency control mechanism of the primary replica determines the serialization order of update transactions and the communication between the primary and the backups must guarantee that updates are processed in the same order at all replicas.

As a representative example of primary-backup replication we have chosen the *Pronto* replication protocol [Pedone and Frølund, 2008]. In a nutshell the protocol works as follows. Clients submit update transactions to the primary

replica. Once the request is executed and ready to commit, the primary broadcasts update SQL statements to all backups. To ensure consistency despite multiple primaries resulting from incorrect failure suspicions, a total-order broadcast is used to propagate updates. Upon delivery of updates each site executes a deterministic validation test. The validation test ensures that only one primary can successfully execute a transaction. If the replica decides to commit the transaction, the update statements are applied to the database. The response is given to the client as soon as any replica commits the transaction. Read-only transactions are submitted and executed at random replica.

The original Pronto protocol guarantees one-copy serializability. The easiest way to provide session consistency in Pronto is to require clients to submit their read-only transactions to the replica which was the first to commit the previous transaction of the same client. To guarantee strong serializability the protocol needs to ensure that the effects of committed transactions are visible to all following transactions. To provide this in Pronto, read-only transactions are total-order broadcast to all replicas. However, there is no need to require all replicas to execute the query: one randomly chosen replica executes the transaction and replies to the client.

6.2.2 Optimistic Update Everywhere Replication

The main difference between primary-backup and update everywhere replication is that in the update everywhere approach any replica can execute any transaction. However, it is more difficult to achieve data consistency than in primary approaches. In update everywhere approach conflicting update transactions can run concurrently on distinct database sites, and thus, an additional synchronization phase between database sites is required to provide correctness.

In optimistic protocols each transaction is first executed locally at some database site. At commit time update transactions are total-order broadcast to all replicas. An update transaction is allowed to commit only if there were no conflicting transactions executing concurrently at remote database sites. To ensure that all replicas agree on the same outcome of the transaction a final validation or voting phase is executed.

An instance of optimistic update everywhere replication is the *Multiversion Database State Machine* replication protocol (see Chapter 3). The vDBSM protocol guarantees one-copy serializability. Session consistency can be trivially attained enforcing that the client always contacts the same replica for executing its transactions. As in Pronto strong serializability is ensured if read-only transactions are total-order broadcast to all replicas. The delivery of such a transac-

tion is ignored by all but one replica: the read-only transaction is executed at a selected database site.

6.2.3 Pessimistic Update Everywhere Replication

In conservative protocols an a priori coordination among the replicas ensures that during transaction execution there are no concurrent conflicting requests being executed at the distinct replicas and therefore transaction's success depends entirely on the local database engine. Throughout the transaction execution there is no synchronization between replicas.

BaseCON is a simple update everywhere replication protocol that executes transactions conservatively. We discuss the protocol in detail and show how BaseCON achieves each of the discussed correctness criteria in the following section.

6.3 BaseCON

BaseCON is a conservative replication protocol which takes advantage of total-order broadcast primitives to provide strong consistency and fault-tolerance. False suspicions are tolerated and never lead to incorrect behavior. In this section we first describe the behavior of the algorithm that guarantees 1SR in a failure-free scenario and in case of failures; then we discuss two BaseCON variants, each of which ensures different consistency guarantees: SC and strong 1SR. Correctness proofs of all BaseCON variants can be found in Section 6.3.4.

6.3.1 One-Copy Serializability

BaseCON assumes a lightweight scheduler interposed between the clients and the cluster of database sites. Such scheduler serves as a load-balancer for read-only transactions and implements different consistency properties. The main challenge is to guarantee that despite failures and false suspicions of the scheduler, the required consistency degree is still provided.

Algorithm 1 presents the complete BaseCON when no failures or false suspicions occur. The algorithm is composed of five concurrent tasks and several instances of $executeTask$. A new instance of $executeTask$ is created for each update or read-only transaction. Each line of the algorithm is executed atomically.

Algorithm 1 The BaseCON Algorithm: 1SR

1: Client c_k:
2: **if** $T.isReadOnly$
3: send(READONLY, T) to D_k
4: **else**
5: to_broadcast(T)
6: Scheduler D_k:
7: $\forall S_k \in \mathscr{S} : Load[S_k] \leftarrow 0$
8: $p \leftarrow 0$
9: **upon** receive(READONLY, T) {$T1$}
10: **if** $p = D_k$
11: $S_k^{min} \leftarrow min(Load[S_k], S_k \in \mathscr{S})$
12: $Load[S_k^{min}] \leftarrow Load[S_k^{min}] + T.weight$
13: $T.repId \leftarrow S_k^{min}$
14: send(READONLY, T) to S_k^{min}
15: **else**
16: send(READONLY, T) to p
17: **upon** to_deliver(T) {$T2$}
18: **if** $p = D_k$
19: $\forall S_k \in \mathscr{S} : Load[S_k] \leftarrow Load[S_k] + T.weight$
20: **upon** receive(RESULT, T) {$T3$}
21: **if** $p = D_k$
22: $Load[T.repId] \leftarrow Load[T.repId] - T.weight$
23: **if** $T.repId$ is the first replica to execute T
24: send(RESULT, $T.result$) to c_k
25: Replica S_k:
26: $txnQ \leftarrow \epsilon$
27: $p \leftarrow 0$
28: **function** conflict(T, T')
29: **return** $T.rs \cap T'.ws \neq \emptyset$
30: **upon** to_deliver(T) {$T4$}
31: $T.repId \leftarrow S_k$
32: $prTxn \leftarrow txnQ$
33: $enqueue(txnQ, T)$
34: **fork task** executeTask($T, prTxn$)
35: **upon** receive(READONLY, T) {$T5$}
36: **fork task** executeTask($T, \emptyset$)
37: **task** executeTask($T, txnSet$)
38: **if** $T.isReadOnly$
39: submit(T)
40: $T.result \leftarrow$ commit(T)
41: **else**
42: **wait until** $\nexists T' \in txnSet$: conflict $(T, T') \wedge T' \in txnQ$
43: submit(T)
44: **wait until** $T = head(txnQ)$
45: $T.result \leftarrow$ commit(T)
46: $dequeue(txnQ, T)$
47: send(RESULT, T) to p

Clients. Transactions are issued by one or more concurrent clients. Read-only transactions are sent only to the scheduler D_k(line 3). Update transactions are total-order broadcast to all database replicas and the scheduler (line 5).

Replicas. The algorithm uses two global variables shared among all the tasks at each replica: a queue of update transactions to be executed, $txnQ$, and an identifier of the scheduler, p. Access to these variables is mutually exclusive. Upon delivery of an update transaction the database site enqueues the transaction (line 33) and creates a new instance of $executeTask$ to process the transaction (line 34). Similarly, a new $executeTask$ is created once a read-only transaction is received (lines 35-36). Different instances of $executeTask$ can execute concurrently as long as the $txnQ$ data structure is thread-safe. If a transaction is read-only, it can be submitted to the database for execution and committed straightaway (lines 39-40). If a transaction is an update, the site checks whether there are any conflicts with previously received but not yet completed transactions (stored in $txnSet$). If there are no conflicts, the transaction is submitted to the database (line 43); if there are some conflicts, the transaction has to wait until conflicting transactions commit (lines 42). To ensure that all replicas converge to the same database state, conflicting update transactions must commit in the order they were delivered.

However, if non-conflicting update transactions can commit in different orders at the replicas and read-only transactions are allowed to execute at any database site, serializability guarantees may be violated. Consider four transactions: T_1:$w_1[x]$, T_2:$w_2[y]$, T_3:$r_3[x], r_3[y]$ and T_4:$r_4[y], r_4[x]$. Since T_1 and T_2 do not conflict they can execute and commit at database sites in different orders. Let's assume that T_1 commits at S_1 and T_2 commits at S_2 first. Transaction T_3 is scheduled for execution at S_1 and T_4 at S_2; then S_1 commits T_2 and S_2 commits T_1. As a consequence, transaction T_3 sees the updates of T_1, but not those of T_2, while T_4 sees the updates performed by T_2 but not by T_1, and thus, violates serializability. To avoid situations like this, we require all update transactions to commit in their delivery order. Therefore, a commit is sent to the database site only after the update transaction has reached the head of $txnQ$, i.e., all previously delivered update transactions have completed already (lines 44-45). As soon as the transaction commits, the result is communicated to the scheduler (line 47).

Scheduler. We consider a primary-backup model to tolerate scheduler failures. There is only one scheduler at a time serving transactions, the *primary*. If the primary scheduler fails or is suspected to have failed, a *backup* scheduler takes over. Since our scheduler is lightweight, any replica can play the role of a backup scheduler.

To ensure 1SR the scheduler can forward read-only transaction to any replica, however, in order to balance the load we send the transaction to the least-loaded replica. Any load balancing strategy can be applied. The scheduler maintains current load information about each database site in $Load[]$. Upon delivery of an update transaction at the primary scheduler the load over all sites is increased by the weight of the transaction (line 19). Once a read-only transaction is received by the primary scheduler, it is redirected to the replica with the lowest aggregated weight (lines 9-14). The result of the transaction is communicated to the client as soon as the corresponding response for the read-only transaction is received; or the first response of the update transaction is received from any replica (lines 23-24). Load information is updated with every reply from the database sites (line 22).

Dealing with failures. If a replica suspects the scheduler has failed (see Algorithm 2), a special NEWSCHEDULER message is total-order broadcast (lines 8-9). Upon delivery of this message, a new primary scheduler is selected from the backups (lines 11). If the scheduler was suspected incorrectly, it will also deliver the NEWSCHEDULER message and will update its state to a backup scheduler, thus, will stop serving read-only transactions (line 3). If a read-only transaction is received, it is immediately forwarded to the primary scheduler. A failover scheduler does not have any load information on the database sites. Therefore, replicas respond to the new scheduler with their load estimates required to handle read-only transactions (lines 12-13). Rarely, but it may happen that transaction results are lost during scheduler failover. Hence client application should be ready to resubmit transactions and ensure exactly-once semantics.

Since we do not make any assumptions on how long it takes for messages to be transmitted and failure detectors can make mistakes, it is possible that for a certain time period two schedulers may be able to process transactions simultaneously. Such scenario is depicted in Fig. 6.1. Client c_1 submits an update transaction T_1 which is total-order broadcast to all members of the system. Database site s_1 executes transaction T_1 and forwards the result to the scheduler p_1. Since it is the first reply for this transaction, the result is communicated to the client. Shortly after that, database site s_2 suspects the primary scheduler p_1 to have failed and broadcasts a NEWSCHEDULER message. Upon delivery of this message both database sites send their load information to the newly selected primary scheduler p_2. Database site s_2 executes transaction T_1 and since it has already delivered NEWSCHEDULER message, the result of T_1 is forwarded to the scheduler p_2. The old scheduler was suspected incorrectly: it is still fully functional, but since it hasn't yet delivered a scheduler change message it is unaware of the presence of the new scheduler. Consequently, it is unaware of transaction's

Algorithm 2 The BaseCON Algorithm: Scheduler failover

```
1:  Scheduler D_k:
2:   upon to_deliver(NEWSCHEDULER)
3:     p ← (p + 1) mod |𝒟|
4:   upon receive(STATE, load, S_k)
5:     if p = D_k
6:       Load[S_k] ← load

7:  Replica S_k:
8:   upon scheduler suspected
9:     to_broadcast(NEWSCHEDULER)
10:  upon to_deliver(NEWSCHEDULER)
11:    p ← (p + 1) mod |𝒟|
12:    load ← Σ_{T∈txnQ} T.weight
13:    send(STATE, load, S_k) to p
```

T_1 commitment at replica s_2. If client c_2 submits read-only transaction T_2 to the old scheduler p_1, the transaction will be scheduled based on erroneous load information at the replicas. However, to ensure 1SR read-only transactions can execute at any replica, thus even if a scheduler makes decisions based on incomplete information, the consistency is still guaranteed, and only load-balancing may be affected. We further discuss the implications of failures on consistency in the context of SC and strong 1SR.

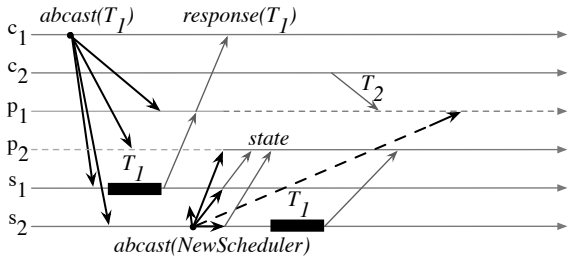


Figure 6.1. Dealing with failures and false suspicions

We use total-order broadcast primitives to replace the suspected scheduler. It is not required to ensure consistency. Since update transactions are also totally ordered, if a scheduler change occurs, all subsequent update transactions will be processed by the new scheduler, i.e., all replicas will directly contact the new scheduler with results of processed transactions. If a scheduler change was not totally ordered with respect to update transactions, different schedulers would

be contacted by different replicas after processing the same update transactions, and thus, more communication messages might be wasted for the client to receive the result of its transaction.

6.3.2 Session Consistency

To achieve session consistency the scheduler must forward read-only transactions to the replica which has committed previous update transactions of the same client.

We have modified BaseCON so that session consistency is ensured. The updated part of the algorithm is presented in Algorithm 3. In addition to the load

Algorithm 3 The BaseCON Algorithm: SC

1: Client c_k:
2: **if** $T.isReadOnly$
3: send(READONLY, T, id, rep) to D_k
4: Scheduler D_k:
5: $p \leftarrow 0$
6: $\forall S_k \in \mathscr{S} : Load[S_k] \leftarrow 0, Committed[S_k] \leftarrow 0$
7: **upon** receive(READONLY, T,id, rep)
8: **if** $p = D_k$
9: $\forall S_k \in \mathscr{S}$:
10: **if** $id \leq Committed[S_k]$
11: $\mathscr{I} \leftarrow \mathscr{I} \cup S_k$
12: **if** $\mathscr{I} \neq \emptyset$
13: $S_k^{min} \leftarrow min(Load[S_k], S_k \in \mathscr{I})$
14: $Load[S_k^{min}] \leftarrow Load[S_k^{min}] + T.weight$
15: $\mathscr{I} \leftarrow \emptyset$
16: $T.repId \leftarrow S_k^{min}$
17: send(READONLY, T) to S_k^{min}
18: **else**
19: send(READONLY, T) to rep
20: **else**
21: send(READONLY, T) to p
22: **upon** receive(RESULT, T)
23: **if** $p = D_k$
24: $Load[T.repId] \leftarrow Load[T.repId] - T.weight$
25: **if** $\neg T.isReadOnly$
26: $Committed[T.repId] \leftarrow T.id$
27: **if** $T.repId$ is the first replica to execute T
28: send(RESULT, $T.result$, $T.id$, $T.repId$) to c_k

information the scheduler also stores the identifier of the last update transaction committed per database site in $Committed[]$ (line 6). The identifier of an update transaction is a unique sequence number that follows the delivery and consequently the commit order. Since some replicas might have fallen behind with the application of update transactions, for read-only transactions the scheduler first determines the set of replicas $\mathscr{S}$ where previous transactions of the same client have been completed. From this set the scheduler then selects the least-loaded replica to execute the read-only transaction (lines 13-17).

To guarantee session consistency in case of false suspicions, when submitting a read-only transaction T the client also sends information about its last committed update transaction T': the identifier id of T' and the replica rep which executed T' first (line 3). On receiving T the scheduler checks whether it has information about T''s commitment (line 10). If so, it is safe to schedule T based on data available at the scheduler; if not, the result of T' was given to the client by a newly introduced primary scheduler. Thus, the scheduler has no other choice than sending T where T' has executed (line 19) — no load-balancing is performed in this case.

6.3.3 Strong Serializability

To implement strong serializability we must ensure that all transactions see all updates performed by previously committed transactions. We present BaseCON that guarantees strong serializability in Algorithm 4.

First let's see how strong 1SR can be guaranteed in a failure-free environment. As before, update transactions are total-order broadcast to all replicas. Since such transactions commit in their delivery order, they always observe the effects of previously committed transactions. Read-only transactions are sent to the scheduler (line 4). Upon reception of the transaction the scheduler first determines the set of replicas where preceding update transactions of any client have already been executed and committed. Then the scheduler chooses the least-loaded replica from that set and forwards the transaction for execution (lines 10-15).

Unfortunately, that is not enough for consistency if failures and false suspicions are tolerated: due to mistakes made by failure detectors two schedulers may simultaneously schedule read-only transactions. Therefore, to avoid inconsistent scheduler decisions, read-only transactions need to be total-order broadcast to all replicas as well (line 2). The scheduler still serves as a load-balancer: read-only transactions are executed at a single database site.

Algorithm 4 The BaseCON Algorithm: strong 1SR

1: Client c_k:
2: to_broadcast(T)
3: **if** $T.isReadOnly$
4: send(READONLY, T) to D_k
5: Scheduler D_k:
6: $p \leftarrow 0$
7: $\forall S_k \in \mathscr{S} : Load[S_k] \leftarrow 0, Committed[S_k] \leftarrow 0$
8: **upon** receive(READONLY, T)
9: **if** $p = D_k$
10: $\mathscr{I} \leftarrow max(Committed[S_k], S_k \in \mathscr{S})$
11: $S_k^{min} \leftarrow min(Load[S_k], S_k \in \mathscr{I})$
12: $Load[S_k^{min}] \leftarrow Load[S_k^{min}] + T.weight$
13: $T.repId \leftarrow S_k^{min}$
14: $T.scheduler \leftarrow p$
15: send(READONLY, T) to S_k^{min}
16: **else**
17: send(READONLY, T) to p
18: Replica S_k:
19: $txnQ \leftarrow \epsilon$
20: $p \leftarrow 0$
21: **upon** to_deliver(T)
22: $T.repId \leftarrow S_k$
23: $prTxn \leftarrow txnQ$
24: $enqueue(txnQ, T)$
25: **if**$\neg T.isReadOnly$
26: **fork task** executeTask($T, prTxn$)

27: **task** executeTask($T, txnSet$)
28: **if** $T.isReadOnly$
29: submit(T)
30: **wait until** $T \in txnQ$:
31: **if** $T.scheduler \neq p$
32: rollback(T)
33: submit(T)
34: $T.result \leftarrow$ commit(T)
35: **else**
36: **wait until** $\nexists T' \in txnSet$: conflict(T, T') $\wedge$ $T' \in txnQ$
37: submit(T)
38: **wait until** $T = head(txnQ)$
39: $T.result \leftarrow$ commit(T)
40: $dequeue(txnQ, T)$
41: send(RESULT, T) to p

In order to optimize the response time of read-only transactions in the absence of failures, we overlap the scheduling and the actual execution of the transaction with the time it takes for a replica to deliver it. Assuming that it takes less time to transmit a simple TCP message than an total-order broadcast message, when the chosen replica delivers the transaction, its execution has started already. A transaction can commit only if it passes the *validation* test at the time of the delivery. The validation test checks whether the transaction was executed and delivered during the operation of the same scheduler (line 31). If there were scheduler changes, the transaction is re-executed (lines 32-33), otherwise the transaction commits and its result is forwarded to the client (line 41).

6.3.4 Proofs of Correctness

In the following we show that each variant of BaseCON guarantees its corresponding correctness criteria.

Proposition 2. *The BaseCON-1SR ensures one-copy serializability*

Proof (Sketch). We show that every history H produced by BaseCON-1SR has an acyclic multiversion serialization graph (MVSG). To prove that $MVSG(H)$ is acyclic we show that for every edge $T_i \rightarrow T_j \in MVSG$, it follows that $commit(T_i) < commit(T_j)$. In the following we consider each edge type of $MVSG(H)$.

- *Read-from edge.* If T_j reads data object x from T_i, then $T_i \rightarrow T_j \in MVSG(H)$. We have to show that $commit(T_i) < commit(T_j)$.

  BaseCON executes conflicting update transactions sequentially. Thus, if both T_i and T_j are update transactions, they will be executed sequentially, and T_j can read the updates of T_i only if T_i has already committed. If T_j is a read-only transaction, due to 2PL, T_j can read only committed data. Thus, $commit(T_i) < commit(T_j)$.

- *Version order edge.*

  - If both T_i and T_j update x, such that $x_i \ll x_j$, then $T_i \rightarrow T_j \in MVSG(H)$.

    From the definition of version order we have that $x_i \ll x_j \Leftrightarrow commit(T_i) < commit(T_j)$.

– If T_i reads x_k and both T_k and T_j write x such that $x_k \ll x_j$, then $T_i \rightarrow T_j \in MVSG(H)$.

For a contradiction, assume $commit(T_j) < commit(T_i)$. If T_i is an update transaction, it is total-order broadcast to all database sites. Conflicting update transactions execute sequentially in their delivery order in BaseCON, thus if T_i reads x_k, then $commit(T_k) < commit(T_i)$. T_i can read x_k only if T_j updates x and commits before T_k executes, i.e., $commit(T_j) < commit(T_k)$ and $x_j \ll x_k$. This contradicts that $x_k \ll x_j$ and consequently proves that $commit(T_i) < commit(T_j)$.

If T_i is a read-only transaction, it is sent only to the scheduler D_k and forwarded to the least-loaded database site. Due to 2PL T_i can only read committed data, thus, $commit(T_k) < commit(T_i)$. The only way T_i can read x_k is if $commit(T_j) < commit(T_k)$. But this contradicts that $x_k \ll x_j$.

Let's consider the case with scheduler failures. If the scheduler D_k fails or is suspected to have failed, a special message is total-order broadcast and a new primary scheduler D_{k+1} is selected. We rely on client applications to resubmit transactions and ensure exactly-once semantics. Thus, if D_k receives transaction T_i and fails, the client will resubmit the transaction eventually (e.g., on delivery of the special NEWSCHEDULER message). If D_k fails after sending T_i to the database site, the result of the transaction will be forwarded to the newly selected scheduler D_{k+1}. If D_k fails after receiving the result of T_i but before sending it to the client, the client application will resubmit the transaction eventually and ensure exactly-once semantics.

□

Proposition 3. *The BaseCON-SC ensures session consistency.*

Proof (Sketch). To prove that every history H produced by BaseCON-SC is session consistent, we have to show that (a) H is 1SR and that (b) for any two transactions T_i and T_j that belong to the same session, if the commit of T_i precedes the submission of T_j in real time, then $commit(T_i) < commit(T_j)$.

a) Similarly to BaseCON-1SR (see Proposition 2) BaseCON-SC ensures one-copy serializability.

b) In the case where both T_i and T_j are update transactions, if the client received a commit for T_i, then all replicas will commit T_i and T_j in the

same order (ensured by total-order broadcast primitives): $commit(T_i) < commit(T_j)$.

If T_j is a read-only transaction, it is scheduled to the replica which has already executed T_i. Thus, $commit(T_i) < commit(T_j)$. Let's consider the case with scheduler failures. The scheduler D_k sends the response of T_i to the client and fails. The new scheduler D_{k+1} receives T_j, but has no information about previous T_i. Since T_j carries the information where T_i has executed, the scheduler forwards transaction to the specified replica. Hence, $commit(T_i) < commit(T_j)$.

□

Proposition 4. *The BaseCON-strong 1SR ensures strong serializability.*

Proof (Sketch). We show that (a) every history H produced by BaseCON-strong 1SR is 1SR and (b) for any two transactions T_i and T_j, if the commit of T_i precedes the submission of T_j in real time, then $commit(T_i) < commit(T_j)$.

a) Analogously to BaseCON-1SR (see Proposition 2) BaseCON-strong 1SR ensures one-copy serializability.

b) If both T_i and T_j are update transactions and the client receives commit of T_i, then at all replicas T_j will be ordered after T_i by total-order broadcast and thus, $commit(T_i) < commit(T_j)$.

   If T_j is a read-only transaction the scheduler forwards it for execution to the replica which has committed T_i already. Let's consider the situation where the scheduler was suspected incorrectly and, based on incomplete state information, schedules T_j to the replica where T_i has not been executed. When T_j is delivered, the NEWSCHEDULER message must have been delivered before and thus, T_j will fail the validation test and must be rescheduled. Thus, $commit(T_i) < commit(T_j)$ is always enforced.

□

6.4 Evaluation

In this section we evaluate experimentally the performance of BaseCON under different correctness criteria and compare it with primary-backup and optimistic update-everywhere replication solutions.

6.4.1 Experimental Environment

All experiments were run in the same setup as described in Section 5.4.2. We used a TPC-C database, populated with data for 8 warehouses, resulting in a database of approximately 800MB in MySQL, that fits in main memory of the server.

Since providing stronger correctness criterion has greater impact on read-only transactions, we have increased the percentage of read-only transactions in the TPC-C workload mix. In particular, we present results for two workloads: TPC-C 20, which contains only 20% of update transactions; and TPC-C 50, which represents TPC-C workload with balanced mix of update and read-only transactions. We measure throughput in terms of the number of transactions committed per minute (tpm). The response time reported represents the mean response time of committed transactions in milliseconds (msec). Both throughput and response time are reported separately for update and read-only transactions.

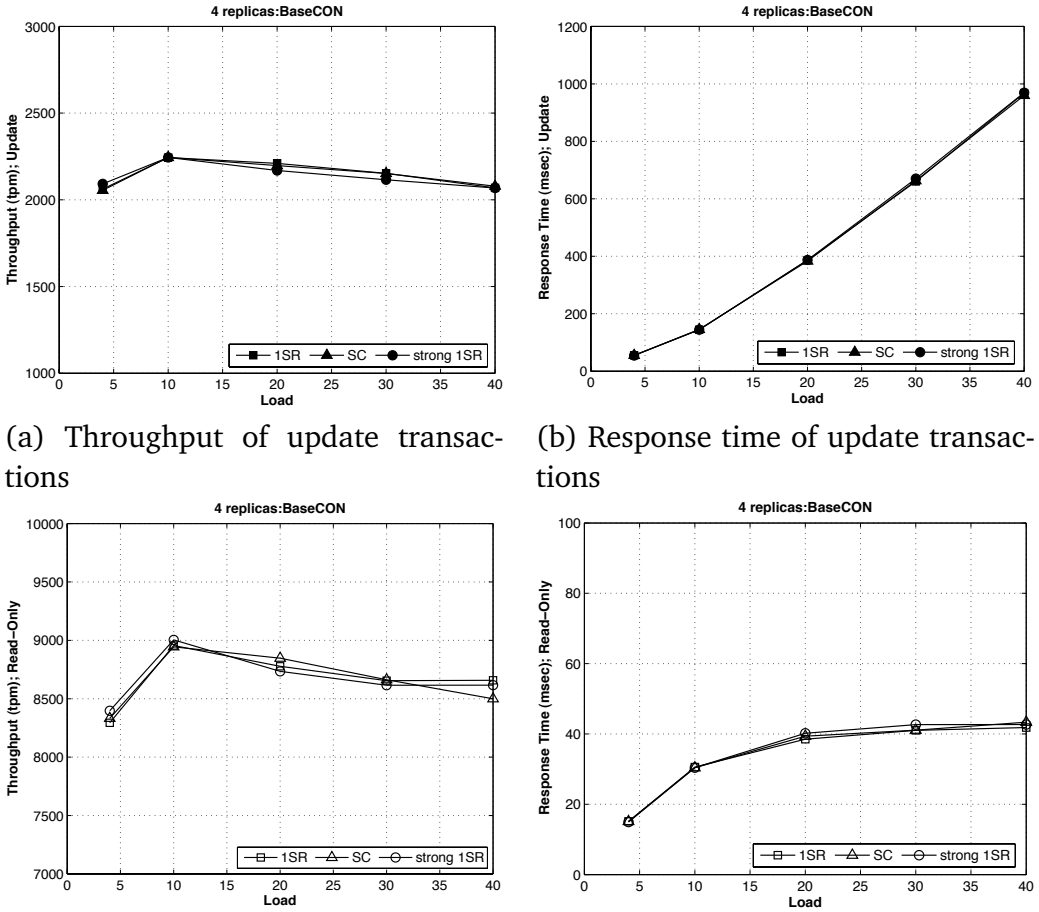
6.4.2 Performance Results

BaseCON

Figures 6.2 and 6.3 show the achieved throughput and response time of read-only and update transactions on a system with 4 database replicas for TPC-C 20 and TPC-C 50, respectively.

There is no significant performance difference among distinct variants of BaseCON: neither throughput nor response time suffer from stronger correctness requirements. The BaseCON scheduler assigns the read-only transaction to replicas so that there is no waiting involved: there is always at least one replica where read-only transactions can execute. Thus, the scheduler does not introduce any notable overhead to assign read-only transaction to a specific replica instead of randomly chosen. Response time of update and read-only transactions (Figs. 6.2(b) and (d)) is the same independently of the correctness criterion considered. This holds for all load conditions and different workloads considered (see also Fig. 6.3).

Since BaseCON implements conservative transaction execution there are no aborted transactions. On the other hand, due to conservative concurrency control response time of update transactions grows with increasing load, and thus throughput is not improved. Read-only transactions are significantly simpler and do not require additional synchronization, thus the growth in response time is lower. Differently from update transactions, which are fully executed at all replicas, read-only transactions execute only on a selected replica. The same



(a) Throughput of update transactions

(b) Response time of update transactions

(c) Throughput of read-only transactions

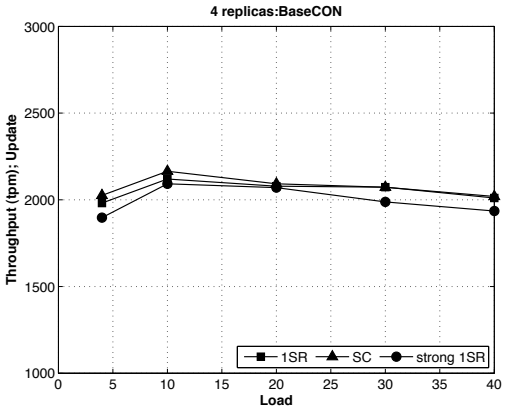
(d) Response time of read-only transactions

Figure 6.2. BaseCON, TPC-C 20: Throughput and response time

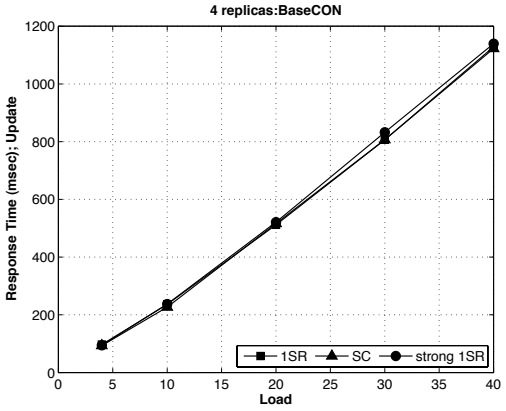
conclusions hold for bigger replicated systems as well: Figure 6.4 presents the results achieved with 8 database replicas and TPC-C 20 benchmark.

Figure 6.5 shows how adding more replicas to the system increases the total throughput of committed transactions. We have varied the number of replicas from 2 to 12, the number of concurrent clients is fixed to 20 for all experiments and the workload submitted is TPC-C 20. Adding more replicas to the system improves the total throughput from approximately 8500 transactions per minute to 12000 transactions per minute.

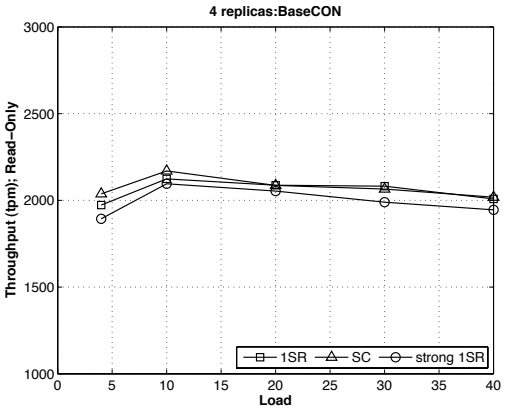
Response time of transactions (see Fig. 6.5(b)) is reduced with increasing number of replicas: since the load is fixed and is composed of a lot of read-only transactions adding more replicas decreases the load over a single database site



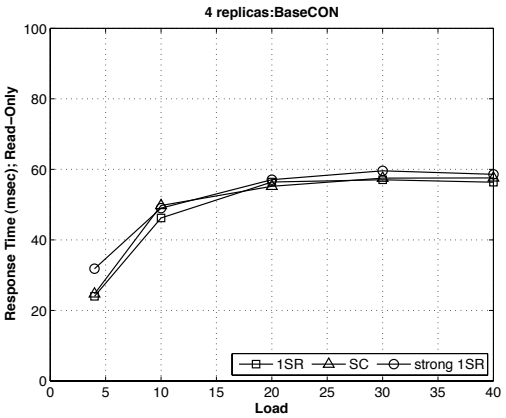
(a) Throughput of update transactions



(b) Response time of update transactions



(c) Throughput of read-only transactions



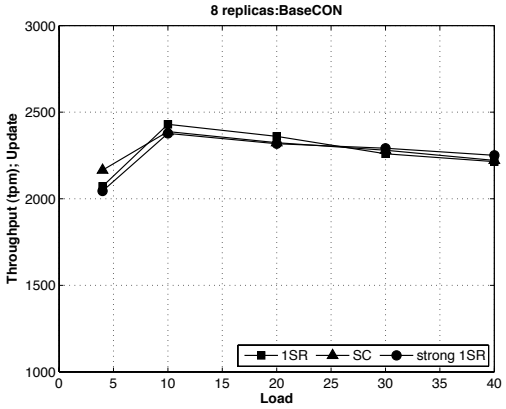
(d) Response time of read-only transactions

Figure 6.3. BaseCON, TPC-C 50: Throughput and response time

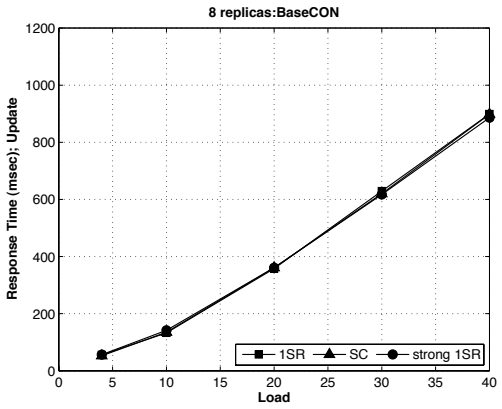
and thus improves the response time of transactions.

Bottleneck analysis

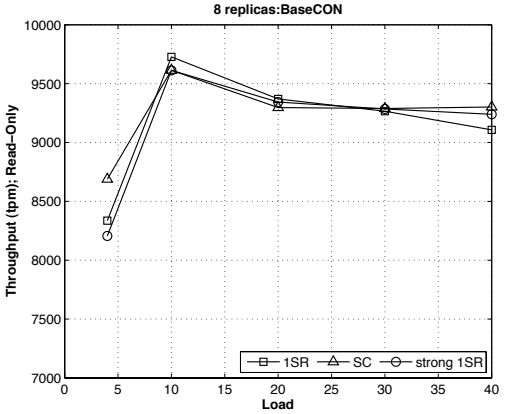
To better understand the performance results obtained hereafter we present a bottleneck analysis of the system. First we analyze hardware resources such as CPU and disk usage during our experiments and then we present the breakdown of transactions response time. The data was collected during the experiments with 4 database replicas for both benchmarks (TPC-C 20 and TPC-C 50) with 10 and 40 concurrent clients, unless specified differently.



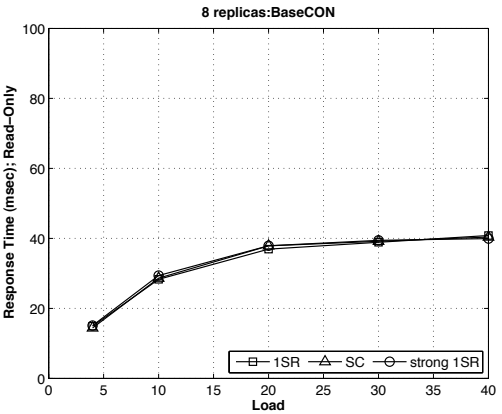
(a) Throughput of update transactions



(b) Response time of update transactions



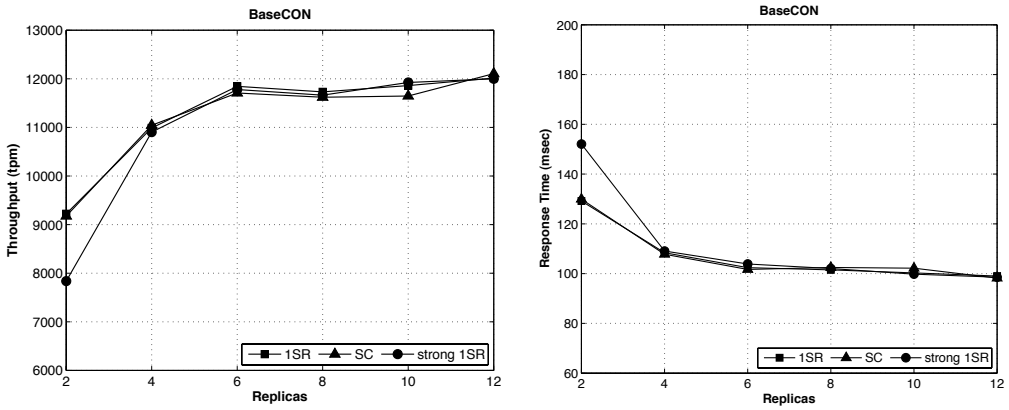
(c) Throughput of read-only transactions



(d) Response time of read-only transactions

Figure 6.4. BaseCON, TPC-C 20, 8 replicas: Throughput and response time

CPU and disk usage. Table 6.1 summarizes the use of CPU (in percentage) at the scheduler and database replicas for each variant of BaseCON. We did not observe a significant increase in CPU usage when providing stronger correctness guarantees. The CPU usage at the scheduler is approximately 1% for the TPC-C 20 workload independently of the number of concurrent clients and the correctness criterion ensured. The workload of TPC-C 50 is more CPU intense, nevertheless, the CPU usage is comparable for all BaseCON variants. As expected the use of CPU at the replicas is much higher: $\approx$ 50% for TPC-C 20 and $\approx$ 80% for TPC-C 50 workloads.



(a) Total throughput (b) Response time

Figure 6.5. BaseCON, TPC-C 20, varied number of replicas: Throughput and response time

		TPC-C 20			TPC-C 50		
		1SR	SC	strong 1SR	1SR	SC	strong 1SR
Scheduler	10	1.02	0.94	0.91	7.02	7.53	8.10
	40	1.06	1.02	1.00	9.62	9.59	9.55
Replicas	10	48.81	50.13	49.20	80.54	75.50	74.21
	40	49.30	49.55	50.31	80.24	79.18	77.29

Table 6.1. CPU usage (%)

Table 6.2 presents the average disk usage at the database replicas.[1] We report the number of disk reads and writes per second, denoted as rps and wps, respectively, and the total amount of data read and written per second in kilobytes (Kps). Almost no data is read from the disk ($\approx$ 1.5Kps), which is a consequence of the fact that the TPC-C database fits in main memory of the server. We observed approximately 200 disk writes per second ($\approx$ 4900 Kps) for the read-intensive workload and approximately 534 disk writes per second ($\approx$ 16312 Kps) for TPC-C 50 benchmark. The disk usage at the replicas is not affected by stronger concurrency criterion provided: none of BaseCON variants resulted in

[1]Since the scheduler does not read or write to the disk, we do not present disk usage at the scheduler here.

		TPC-C 20		TPC-C 50	
		rps / Kps	**wps / Kps**	**rps / Kps**	**wps/Kps**
1SR	10	0.33/1.63	178.10/4421.3	0.01/0.78	473.73/14434
	40	0.15/0.85	202.15/5118.6	0.04/0.25	577.00/17798
SC	10	0.33/1.50	199.98/5055.9	0.03/0.53	526.03/16125
	40	0.43/1.88	207.08/5201.1	0.03/0.53	471.52/14383
strong 1SR	10	0.35/1.75	181.63/4532.3	0.02/0.75	611.25/18344
	40	0.01/0.98	192.20/4847.5	0.04/0.50	548.75/16793

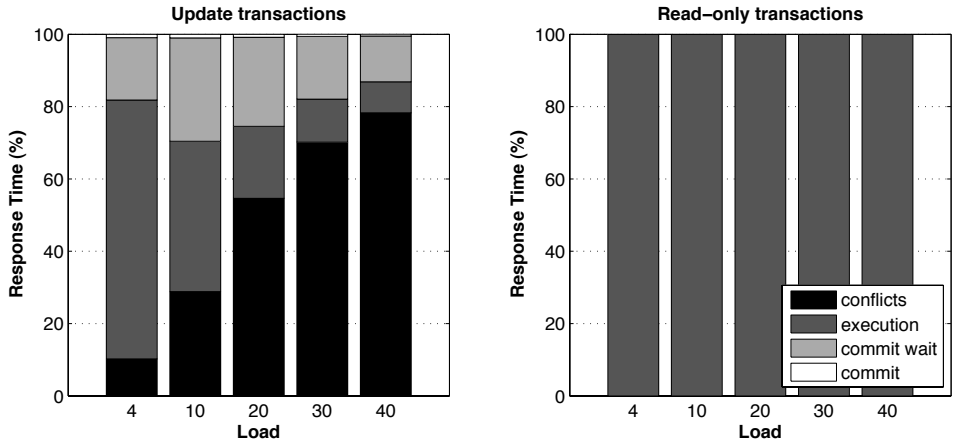
Table 6.2. Disk usage at the replicas

significantly different disk access patterns at the replicas.

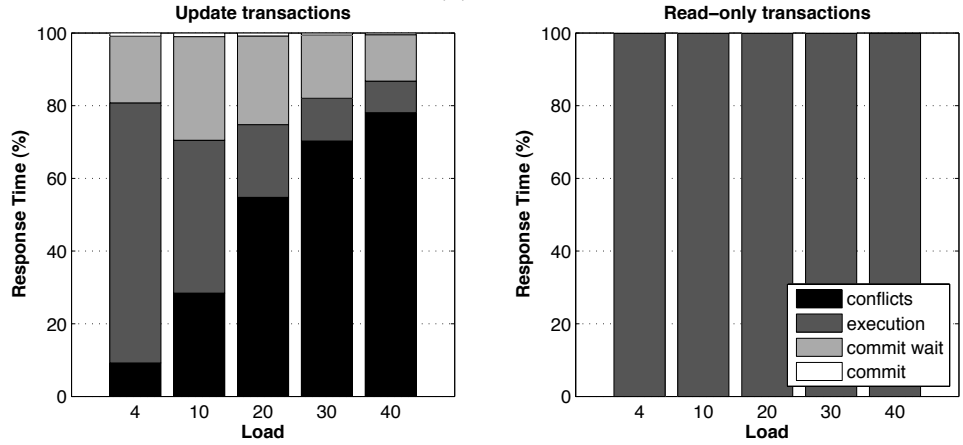
Response time breakdown. The response time of a transaction in BaseCON is composed of the following: (i) the time spent waiting until conflicting transactions in execution commit, (ii) the time of actual execution of operations of the transaction, (iii) the time spent waiting to ensure the same commit order at all database replicas, and (iv) the time it takes for the transaction to commit. In addition to that to guarantee strong serializability read-only transactions may need to wait for total-order delivery before committing (see Section 6.3.3).

Figure 6.6 shows the breakdown of response times for the TPC-C 20 benchmark run on the system with 4 database replicas. Since BaseCON handles read-only and update transactions differently, the results are reported separately. Each vertical bar represents different submitted load. Each different time spent while executing the transaction is color coded and represented as the percentage of the total response time.

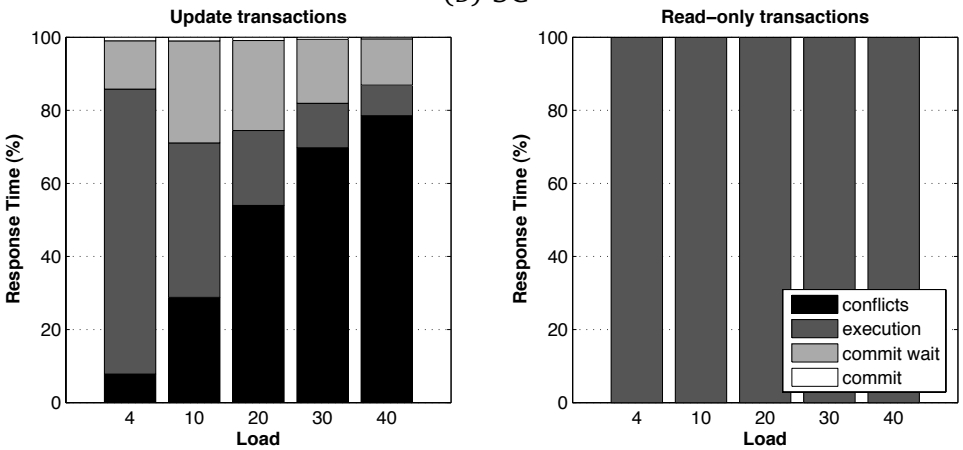
The response time of read-only transactions is 100% the actual execution of the operations of the transactions. This is also true for strong serializability: we did not observe any delays due to waiting for delivery of read-only transactions before their commit. On the other hand update transactions suffer from conservative concurrency control implemented at the middleware layer: the time spent waiting for conflicting transactions to commit constitutes more than 50% of total response time for an average load (20 concurrent clients), independently of the correctness criterion provided. Higher load further increases the waiting times. A significant part of time is also spent waiting for transactions to commit in the



(a) 1SR



(b) SC



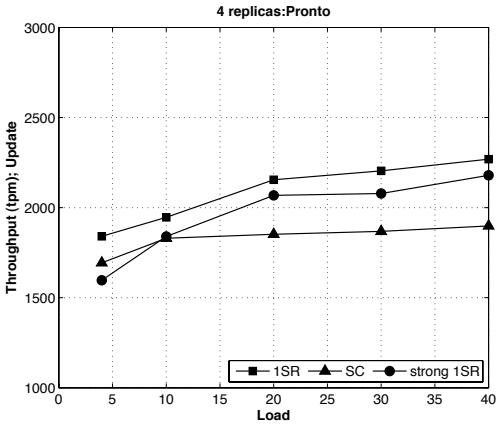
(c) strong 1SR

Figure 6.6. Response time breakdown, TPC-C 20

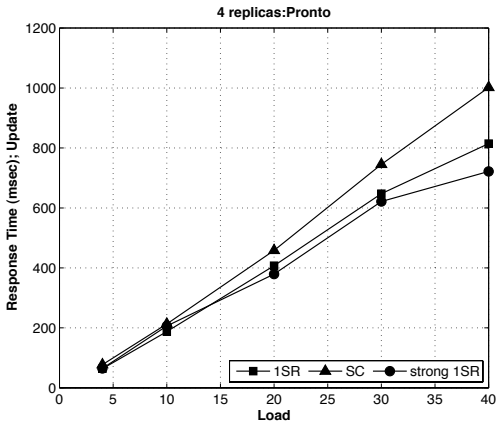
same order at all replicas.

Primary-backup replication

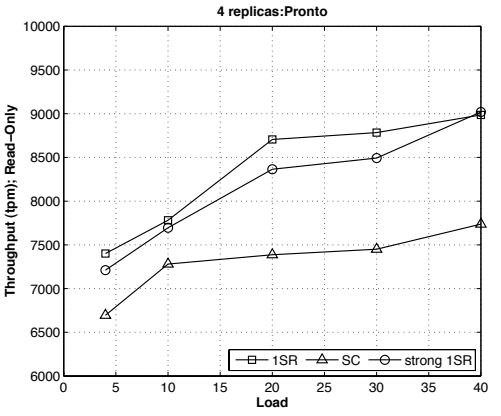
Figure 6.7 depicts the attained throughput and response time of Pronto running TPC-C 20 on the system with 4 database replicas. The throughput of update transactions is limited by a single primary replica handling most of the update transactions load (backup replicas apply only updates). On the contrary, read-



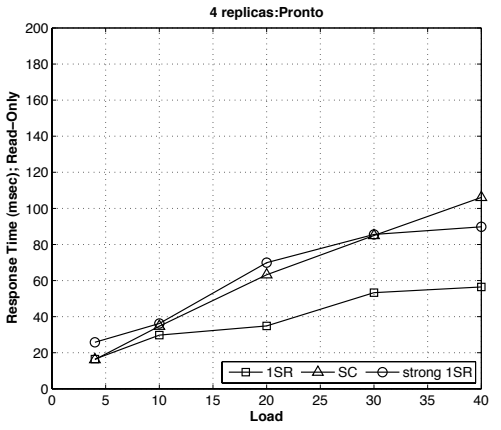
(a) Throughput of update transactions



(b) Response time of update transactions



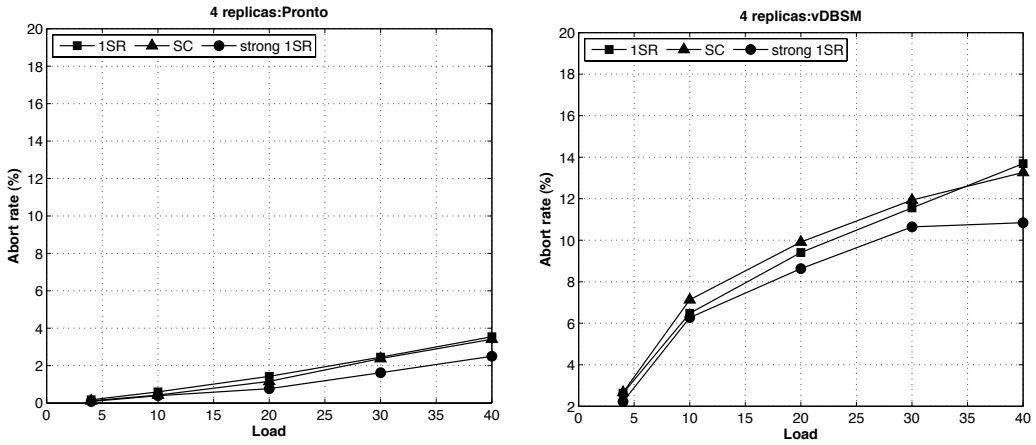
(c) Throughput of read-only transactions



(d) Response time of read-only transactions

Figure 6.7. Pronto, TPC-C 20: Throughput and response time

only transactions are distributed over all replicas and consequently higher transactions load results in higher throughput for both 1SR and strong 1SR. However,



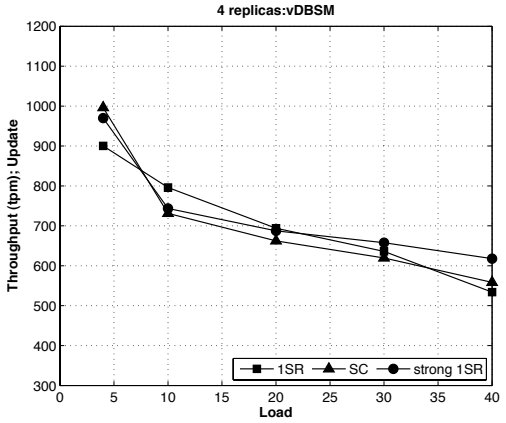
(a) Pronto (b) Update everywhere

Figure 6.8. TPC-C 20: Abort rates

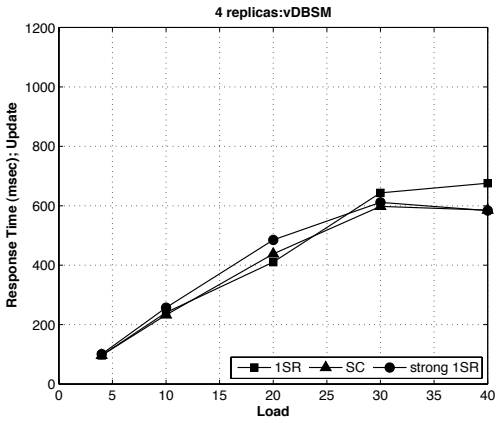
the performance of Pronto implementing SC is considerably worse. To guarantee SC read-only transactions must execute on a replica which was the first to commit previous transactions of the same client. Since upon delivery of updates the primary replica just performs the certification test and can commit the transaction, while backups still need to apply the received SQL statements, the primary replica is the fastest one to respond. As a result, both read-only and update transactions execute locally at the primary replica overloading it. To guarantee strong 1SR Pronto totally orders read-only transactions with respect to all other transactions but executes them only on selected replicas. In this way the load of read-only transactions is distributed over the replicas. Furthermore, such transactions execute in isolation, as opposite to SC, where read-only transactions execute concurrently with all the load submitted to the primary replica. Further, differently from BaseCON, Pronto aborts some transactions due to local timeouts and deadlocks (Fig. 6.8(a)).

Update everywhre replication

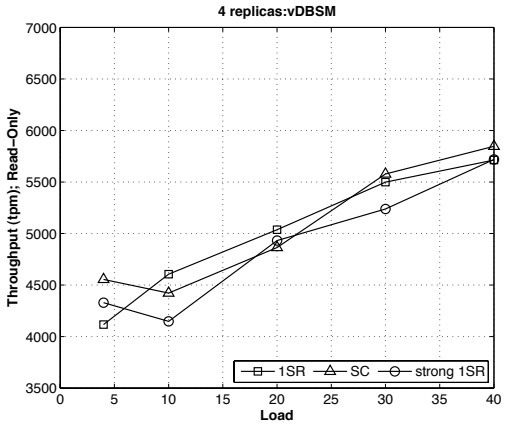
Figure 6.9 depicts performance graphs of vDBSM running TPC-C 20 on the system with 4 database replicas. As in the case of BaseCON, implementing different correctness criteria with vDBSM does not introduce any notable overhead and thus has no significant influence on system's performance. Even though the response time of committed transactions is comparable to BaseCON, such results are achieved at the price of high abort rate (see Fig. 6.8(b)). With in-



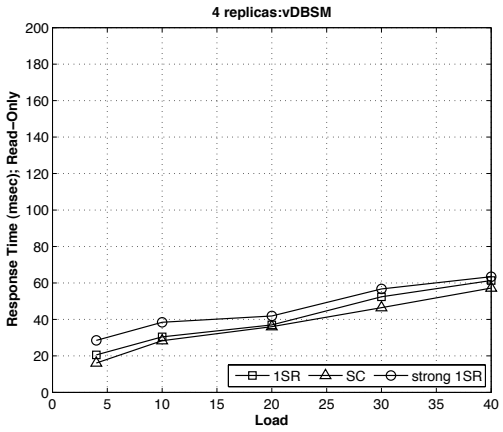
(a) Throughput of update transactions



(b) Response time of update transactions



(c) Throughput of read-only transactions



(d) Response time of read-only transactions

Figure 6.9. vDBSM, TPC-C 20: Throughput and response time

creasing number of concurrent clients more update transactions will execute in parallel without synchronization and consequently more will be aborted by the certification test to ensure strong consistency. Thus the throughput of update transactions degrades leaving more room for executing read-only transactions.

To summarize, we show experimentally that stronger consistency does not necessarily imply worse performance in the context of middleware-based replication. On the contrary, two of the three protocols evaluated are able to provide different consistency guarantees without penalizing system's performance. Even though the implementation of strong serializability requires ordering read-only transactions in all protocols studied, the overhead introduced by total order

primitives is insignificant in middleware-based replication. Moreover, implementation of session consistency in the primary-backup protocol exhibits worse performance than strong serializability.

6.5 Related Work and Final Remarks

In this chapter we are interested in the correctness criteria used in non-intrusive database replication systems. The majority of database replication protocols proposed in the literature ensure either one-copy serializability (e.g., [Patiño-Martínez et al., 2005; Pedone and Frølund, 2008; Pedone et al., 2003; Rodrigues et al., 2002]) or snapshot isolation (e.g., [Lin et al., 2005; Wu and Kemme, 2005; Elnikety et al., 2005]). However, some applications may require stronger guarantees.

Daudjee and Salem [2004] address the problem of transaction inversions in lazily replicated systems that ensure one-copy serializability . They introduce the notion of strong session serializability, which is equivalent to our session consistency, and show how it can be implemented. However, only a simulation study is presented. An analogous study is conducted for snapshot isolation in [Daudjee and Salem, 2006].

Causality expected by the clients is studied in [Oliveira et al., 2006] as well. Differently from us, Oliveira et al. [2006] consider only the DBSM and present two ways (optimistic and conservative) to provide expected causality. The conservative way is equivalent to the one chosen by us, since the optimistic technique results in unacceptable abort rate of read-only transactions. The results reported in [Oliveira et al., 2006] confirm the results we have attained.

Ganymed, a lightweight scheduler that routes transactions to a set of snapshot isolation based replicas, was introduced in [Plattner and Alonso, 2004]. The main idea behind Ganymed is a separation between updates and read-only transactions: updates are handled by a master replica and lazily propagated to the slaves, where read-only transactions are processed. The scheduler assigns read-only transactions to the replica which contains the latest version of the database, thus ensures strong serializability, but might introduce delays if such a replica is not available. If clients are not willing to accept such delays, the correctness criterion can be weakened: read-only transactions are scheduled to replicas so that some staleness threshold is satisfied or clients are guaranteed to read at least their own updates.

Besides uniform characterization of consistency degrees, we also show how each of them can be achieved in the context of BaseCON and two other repli-

cation solutions. BaseCON was originally inspired by conflict-aware scheduling, a replication technique by Amza et al. [2003]. The authors use transaction scheduling to design a lazy replication scheme. To avoid aborting conflicting transactions the scheduler is augmented with a sequence numbering scheme and conflict-awareness to provide strong consistency. Differently from BaseCON, failures in [Amza et al., 2003] are handled in an ad hoc manner. Moreover, the correctness of the protocol relies on stronger assumptions than ours.

Chapter 7

Partial Database Replication

Most work on database replication using group communication concentrates on full replication strategies. However, scalability of full replication protocols is limited under update-intensive workloads: Each replica added to the system allows to submit more transactions; if such transactions modify the database, they will add load to every individual database. To improve the scalability of the system, databases can be replicated partially only.

Partial replication usually favors large systems exhibiting strong access locality (e.g., geographically dispersed information systems). Each database site may not have enough disk or memory resources to fully replicate large amounts of data, thus partitioning allows to support a bigger database. If access locality is observed and each transaction needs only a small subset of all sites to execute and commit, the processing and communication overhead associated with replication can be reduced significantly. Unfortunately, it is not obvious how to extend many of the protocols developed for full replication to systems with partial replication. In full database replication, if all transactions are delivered and processed in the same total order at all replicas and the replicas guarantee consistency locally, global correctness of the system is ensured. However, that is not true if partial replication is allowed [Alonso, 1997]. In a partial replication scenario where each replica only holds a subset of the database, even when using total-order broadcast, replicas may compromise the commit atomicity of the transaction. An additional agreement phase is required to ensure a consistent decision on the transaction's output (e.g., [Sousa et al., 2001]).

Partial replication is a complex subject and this chapter addresses just one aspect of it: distributed transactions execution. More specifically, we are interested in the effects of distributed transactions on the abort rate of partially replicated systems. In the following sections we introduce a simple probabilistic analysis of transaction abort rates for two different concurrency control mechanisms: lock-

and version-based. The former models the behavior of a replication protocol providing one-copy-serializability; the latter models snapshot isolation.

7.1 Distributed Transactions in Partial Replication

If the database is replicated partially, a transaction may require access to data stored on remote replicas and thus, a distributed execution involving more than one replica becomes necessary. The problems introduced by distributed transactions in partially-replicated systems differ depending on the concurrency control mechanism used. In lock-based systems, ensuring one-copy serializability, transactions executing over multiple replicas will acquire locks on remote data items, which may increase the likelihood of distributed deadlocks [Gray et al., 1996], a problem that group communication protocols can mitigate [Agrawal et al., 1997]. In version-based systems, guaranteeing snapshot isolation, both local and remote read operations of each transaction must execute in a consistent global database snapshot. Obtaining the requested snapshot for remote reads may be a challenge in some contexts (e.g., middleware approaches based on standard off-the-shelf databases). If such snapshot is not available, the transaction is aborted.

Snapshot isolation has received a considerable amount of attention in the context of full database replication. There are two main reasons for such a trend. First, replication protocols ensuring SI are spared from readsets extraction. Second, read-only transactions executing under snapshot isolation are never blocked or aborted. Unfortunately, in partial replication distributed transactions may compromise these advantages: (a) inability to obtain the requested snapshot for remote reads may cause aborts even for read-only transactions; and (b) transaction readset information may be necessary in order to ensure correctness of the system [Schenkel et al., 1999]. That being said, is snapshot isolation still the preferred correctness criterion when partial replication is considered?

Both distributed deadlocks in lock-based systems and failed remote read operations in version-based systems result in aborted transactions. Hence, we introduce a probabilistic model for abort rates of partially replicated systems when lock- and version-based concurrency control mechanisms are used. First, we briefly describe the chosen replication strategy and then we present our analytical model. We show how the number of data versions available affects the abort rate of the version-based system and try to identify the settings under which snapshot isolation can be safely used with partial replication.

7.2 Simple Probabilistic Analysis

7.2.1 Replication Model

We assume a partial replication model where the original database is partitioned and replicated over the database sites. No database site is expected to store the whole set of items, although that is not forbidden. We call *local* the replica to which the transaction is submitted, and *remote* the replica which contains data items accessed by the transaction and not stored at the local site. Similarly, an operation is called *local* if it is executed on a local replica, and *remote* otherwise. Transactions that access data at more than one database site during execution are called *distributed*.

We distinguish two phases through which transactions pass during processing:

1. *Execution phase.* Transactions are initially submitted to one database site. However, in partial replication, where each replica only holds a subset of the database, a transactions may require access to data items not available locally. In such a case, the transaction operation is forwarded to one of the database sites holding the required item and executed remotely. If the database engine adopts lock-based concurrency control, such distributed transactions inevitably introduce the possibility of distributed deadlocks; if the replicas implement snapshot isolation, the key problem is to obtain a consistent global snapshot of the database composed of individual snapshots taken at each replica involved in the execution of the distributed transaction. We assume that every remote request includes the required snapshot version. Upon processing such a request the remote site demands the correct snapshot from the database. If such a version of the data is not available, the transaction is aborted.

2. *Termination phase.* Read-only transactions commit immediately upon request. Update transactions are forwarded to all (or a subset of) database sites using a total-order multicast primitive. We assume that all database sites involved in the execution of the transaction eventually reach a consistent decision on the transaction's fate: commit or abort. Depending on the replication protocol, this may require a voting phase as part of the transaction's termination (e.g., [Sousa et al., 2001]). For different consistency criteria, different certification tests are used. Two concurrent update transactions T_i and T_j are allowed to commit only if:

- (to ensure 1SR) T_i and T_j executed at distinct replicas, T_i has been delivered first and $ws(T_i) \cap rs(T_j) = \emptyset$. If T_i and T_j execute at the same replica, the local database scheduler guarantees a serializable execution.
- (to ensure SI) The writesets of T_i and T_j do not intersect, that is, $ws(T_i) \cap ws(T_j) = \emptyset$. We assume that SI is implemented using strict *first-committer-wins* rule, i.e., transactions are never aborted during the execution phase because of a write-write conflict.

If the transaction passes the certification test, its updates are applied on all copies of modified data items.

7.2.2 Analytical Model

In partial replication settings where each replica holds only a subset of the database, support for execution of distributed transactions is inevitable, unless "perfect data partitioning" is assumed.[1] In lock-based systems distributed transactions may get involved in distributed deadlocks, while in version-based systems remote read operations may be unable to obtain the requested database snapshot at remote replicas. Both distributed deadlocks and failed remote read operations result in aborted transactions. Hence, the goal of our probabilistic analysis is twofold: (a) to quantify the abort rate of transactions due to distributed execution; and (b) to estimate the abort rate of transactions at the termination phase.

The replicated system is modeled as a number of database sites, $sites$, and a fixed-size database composed of DB_SIZE items. Every database item has a number of $copies$ uniformly distributed over the replicas. Thus, the entire system consists of $DB\_SIZE \cdot copies$ resources. All transactions submitted to the database have the same number of operations op and all operations take the same op_time to execute. An operation is defined as a read or a write on one data item; as a consequence, a single SQL statement may consist of many operations. Each data item has the same probability of being accessed (there are no hotspots). We model neither communication between database sites — both local and remote accesses to data items have the same cost — nor failures of the replicas.

[1] If the database is partitioned so that every transaction can execute at a single site, support for distributed transactions is not needed. However, such an approach requires prior knowledge of the workload and a very particular data distribution over the replicas or at least a single site that holds the whole database.

Every database site receives TPS transactions per second, so the total load over the replicated system is $TotalTPS = TPS \cdot sites$ and there are always $txn = TotalTPS \cdot op \cdot op\_time$ concurrent transactions executing. Every transaction is read-only with the probability of L. Each operation within the update transaction has the probability k to be a read operation. The number of concurrent read-only transactions in the system is $r\_txn = L \cdot txn$, each with op read operations. The number of update transactions is given by $w\_txn = (1-L) \cdot txn$ with $r\_op = k \cdot op$ read and $w\_op = (1-k) \cdot op$ write operations. We also require that the average number of data items accessed by concurrent transactions do not exceed the database size. The main parameters of the model are listed in Table 7.1.

DB_SIZE	database size
TPS	number of transactions per second submitted to a database site
L	fraction of read-only transactions
op	number of operations in a transaction
k	fraction of read operations in update transactions
op_time	time to execute an operation in seconds
sites	number of replicas in the system
copies	number of copies of each data item

Table 7.1. Model parameters

In the following two sections we introduce our probabilistic analysis for evaluating the abort rate of partial replication when lock- and version-based concurrency control mechanisms are used. We assume that the lock-based model ensures 1SR, while the version-based model provides GSI.

Lock-based system

Our model has been strongly influenced by the analytical model introduced by Gray et al. [1996], where the authors analyze the deadlock rate of fully replicated database systems based on locking only. Besides the assumptions considered throughout our probabilistic modelling, the work in [Gray et al., 1996]

does not account for read operations — all transactions are composed of updates only. Hereafter we model read operations within update transactions as well as read-only transactions. To calculate the abort rate at the termination phase, we have followed the ideas introduced in [Pedone, 1999].

Execution phase. As in [Gray et al., 1996], we suppose that, in average, each transaction is about half way complete, thus the number of resources locked by executing transactions is at most

$$res\_locked = ro\_read\_locks + u\_locks, \tag{7.2.1}$$

where

$$ro\_read\_locks = \frac{r\_txn \cdot op}{2}, \tag{7.2.2}$$

$$\begin{aligned} u\_locks &= u\_write\_locks + u\_read\_locks \\ &= \frac{w\_txn \cdot w\_op}{2} + \frac{w\_txn \cdot r\_op}{2} = \frac{w\_txn \cdot op}{2}. \end{aligned} \tag{7.2.3}$$

From Eq. 7.2.3, the probability that a read operation waits because of update transactions is

$$p\_r\_op\_waits\_u = \frac{u\_write\_locks}{DB\_SIZE \cdot copies} = \frac{w\_txn \cdot w\_op}{2 \cdot DB\_SIZE \cdot copies}. \tag{7.2.4}$$

Similarly, $p\_w\_op\_waits\_u$ and $p\_w\_op\_waits\_r$ are the probabilities that a write operation waits for resources locked by update and read-only transactions:

$$p\_w\_op\_waits\_u = \frac{u\_locks}{DB\_SIZE \cdot copies} = \frac{w\_txn \cdot op}{2 \cdot DB\_SIZE \cdot copies}, \tag{7.2.5}$$

$$p\_w\_op\_waits\_r = \frac{ro\_read\_locks}{DB\_SIZE \cdot copies} = \frac{r\_txn \cdot op}{2 \cdot DB\_SIZE \cdot copies}. \tag{7.2.6}$$

Now we can calculate the probability that a read-only transaction waits for resources held by update transactions,

$$p\_r\_tran\_waits\_u = 1 - (1 - p\_r\_op\_waits\_u)^{op}, \tag{7.2.7}$$

and the probability that an update transaction waits because of other update transactions,

$$\begin{aligned} p\_u\_tran\_waits\_u = 1 &- (1 - p\_r\_op\_waits\_u)^{r\_op} \\ &\times (1 - p\_w\_op\_waits\_u)^{w\_op}, \end{aligned} \tag{7.2.8}$$

and because of read-only transactions,

$$p\_u\_tran\_waits\_r = 1 - (1 - p\_w\_op\_waits\_r)^{w\_op}. \tag{7.2.9}$$

A deadlock is created if transactions form a cycle waiting for each other. We do not consider deadlocks that involve more than two transactions: deadlocks composed of cycles of three or more transactions are very unlikely to occur [Gray et al., 1996]. So the probability for a read-only transaction to deadlock is

$$p\_r\_deadlock \approx \frac{p\_r\_tran\_waits\_u \cdot p\_u\_tran\_waits\_r}{r\_txn}, \tag{7.2.10}$$

and the probability that an update transaction deadlocks is

$$\begin{aligned} p\_w\_deadlock \approx & \frac{p\_u\_tran\_waits\_u^2}{w\_txn} \\ & + \frac{p\_u\_tran\_waits\_r \cdot p\_r\_tran\_waits\_u}{w\_txn}. \end{aligned} \tag{7.2.11}$$

From Eq. 7.2.10 and 7.2.11, read-only and update transactions deadlock rates are:

$$r\_deadlock\_rate = \frac{p\_r\_deadlock}{op \cdot op\_time}, \tag{7.2.12}$$

$$w\_deadlock\_rate = \frac{p\_w\_deadlock}{op \cdot op\_time}. \tag{7.2.13}$$

Finally, we can estimate the total number of deadlocks of the system (in transactions per second) as

$$\begin{aligned} aborts\_deadlock = & r\_deadlock\_rate \cdot r\_txn \\ & + w\_deadlock\_rate \cdot w\_txn. \end{aligned} \tag{7.2.14}$$

Termination phase. If there is only one copy of each data item (i.e., there is no replication), *strict* 2PL ensures serializability and thus transactions are not aborted during the termination phase. For more than one copy, two conflicting transactions executing concurrently at distinct database sites may violate 1SR. As mentioned in Section 7.2.1, to ensure 1SR, each committing transaction has to pass the certification test which checks that there is no transaction that executed concurrently and updated data items read by the committing transaction. Notice that conflicts appear only if transactions access different copies of the same item.

We consider only those transactions that were not aborted during execution. Thus, $TotalTPS$, the number of read-only and update transactions are:

$$TotalTPS' = TotalTPS - aborts\_deadlock, \tag{7.2.15}$$

$$r\_txn' = r\_txn \cdot (1 - p\_r\_abort), \tag{7.2.16}$$

$$w\_txn' = w\_txn \cdot (1 - p\_w\_abort), \tag{7.2.17}$$

$$txn' = TotalTPS' \cdot op \cdot op\_time. \tag{7.2.18}$$

If there are only two concurrent transactions in the system, the probability that an update transaction passes the certification test is $\left(1 - w\_op/DB\_SIZE\right)^{r\_op}$. Then the probability that the i-th transaction passes the certification test after the commit of $(i-1)$ transactions is

$$p\_i\_txn\_pass = \left(1 - \frac{(i-1) \cdot w\_op}{DB\_SIZE}\right)^{r\_op}. \tag{7.2.19}$$

On average, the probability that a transaction does not pass the certification test is

$$p\_txn\_no\_pass = 1 - \frac{1}{N} \cdot \sum_{i=1}^{N} p\_i\_txn\_pass, \tag{7.2.20}$$

where N is the number of concurrent update transactions, excluding those that execute at the same replica and do not cause certification aborts:

$$N = w\_txn' \cdot \frac{sites - 1}{sites}. \tag{7.2.21}$$

Consequently, the abort rate of update transactions that do not pass the certification test is defined as follows:

$$u\_abort\_rate = \frac{p\_txn\_no\_pass}{op \cdot op\_time}. \tag{7.2.22}$$

And at last, the total number of aborts due to the certification test is

$$aborts\_sr\_cert = u\_abort\_rate \cdot w\_txn'. \tag{7.2.23}$$

Version-based system

During the execution, transactions are aborted if the requested versions of the data items are not available. We assume that all database sites are able to maintain up to V versions per data item, e.g. with $V = 1$, transactions can only obtain the current version of the data item. Notice that we assume a strict *first-committer-wins* rule, i.e., transactions are never aborted during the execution phase due to write-write conflict; such conflicts are resolved at termination.

Execution phase. In the same way as Eq. 7.2.1, during its execution, a transaction updates at most w_op resources. Therefore, at any time there are $res\_upated\_exec = (w\_txn \cdot w\_op)/2$ resources updated because of the transactions in the execution phase. Some of these transactions will be successfully certified and their updates will be propagated to all the copies of the data items accessed. These remote updates will influence the total number of resources updated.[2] Therefore, during termination there are

$$res\_updated\_term = \frac{(copies-1)\cdot w\_txn' \cdot w\_op}{2} \times p\_commit \qquad (7.2.24)$$

resources updated, where w_txn' is defined in Eq. 7.2.34. p_commit is the probability for an update transaction to pass the certification test and is equal to $1 - p\_w\_abort\_term$ (see Eq. 7.2.38). Hence, the total number of resources updated is $res\_updated = res\_updated\_exec + res\_updated\_term$ and, consequently, the probability for an item to be updated V times by concurrent transactions is:

$$p\_item\_v\_updated = \left(\frac{res\_updated}{DB\_SIZE \cdot copies}\right)^V. \qquad (7.2.25)$$

The probability for a read operation to abort is the same as the probability of waiting for V locks, i.e., the probability of V concurrent transactions to update the same item:

$$p\_r\_op\_abort = p\_item\_v\_updated. \qquad (7.2.26)$$

Since each read-only transaction has op operations, the probability for a read-only transaction to abort is

$$p\_r\_abort = 1 - (1 - p\_r\_op\_abort)^{op}, \qquad (7.2.27)$$

and the probability of abort of an update transaction is

$$p\_w\_abort = 1 - (1 - p\_r\_op\_abort)^{r\_op}. \qquad (7.2.28)$$

From Eq. 7.2.27 and 7.2.28, the abort rates for read-only and update transactions are as follows:

$$r\_abort\_rate = \frac{p\_r\_abort}{op \cdot op\_time}, \qquad (7.2.29)$$

[2] We do not account for remote updates in the lock-based model since in general the deadlock rate is very small and some remote updates will not affect significantly the final deadlock rate.

and

$$w\_abort\_rate = \frac{p\_w\_abort}{op \cdot op\_time}. \tag{7.2.30}$$

Therefore, the total number of aborts during the execution phase of transactions is

$$aborts\_exec = r\_abort\_rate \cdot r\_txn + w\_abort\_rate \cdot w\_txn. \tag{7.2.31}$$

Termination phase. Similarly to Eqs.7.2.15–7.2.18, we have to recalculate the number of concurrent transactions that reach the termination phase:

$$TotalTPS' = TotalTPS - aborts\_exec, \tag{7.2.32}$$

$$r\_txn' = r\_txn \cdot (1 - p\_r\_abort), \tag{7.2.33}$$

$$w\_txn' = w\_txn \cdot (1 - p\_w\_abort), \tag{7.2.34}$$

$$txn' = TotalTPS' \cdot op \cdot op\_time. \tag{7.2.35}$$

Furthermore, transactions aborted during the execution phase also affect the fraction of read-only and update transactions present at the termination phase:

$$L' = \frac{r\_txn'}{txn'}. \tag{7.2.36}$$

Thus, the probability that a write operation conflicts with another write operation is

$$p\_w\_op\_con = \frac{w\_txn' \cdot w\_op}{2 \cdot DB\_SIZE}. \tag{7.2.37}$$

The probability that an update transaction aborts is

$$p\_w\_abort\_term = 1 - (1 - p\_w\_op\_con)^{w\_op}. \tag{7.2.38}$$

Update transactions abort rate due to write-write conflicts is determined as

$$w\_abort\_rate\_term = \frac{p\_w\_abort\_term}{op \cdot op\_time}. \tag{7.2.39}$$

Finally, the total number of aborts at the termination phase is

$$aborts\_si\_cert = w\_abort\_rate\_term \cdot w\_txn'. \tag{7.2.40}$$

7.3 Analytical Evaluation

7.3.1 Objectives

We have analytically estimated the transaction abort rate of a partially replicated system to answer the following questions:

- What is the impact of distributed transactions on the abort rate of 1SR and SI systems?
- How do data versions affect the abort rate of SI systems in the context of partial replication?
- Under which environments are SI systems comparable to 1SR lock-based systems ?

In the following we present the parameters used throughout the evaluation.

7.3.2 Parameter Values

As a base scenario we consider a system composed of 8 database sites and 2 copies of 2.500.000 items database. The system processes 250 transactions per second. Each transaction takes 0.170 seconds to execute and is composed of 200 operations. 90% of the transactions in the workload are update transactions.[3] All the parameters used are summarized in Table 7.2; parameters of the base scenario are highlighted in bold.

In all figures we report the percentage (%) of transactions aborted during execution and termination or just the total system abort rate. In the execution phase the lock-based system is denoted as LB; VB V represents the version-based system, where V is the number of versions available per data item (e.g. VB 1 indicates a scenario where only the current data version is obtainable). In the termination phase we denote the different systems as 1SR and SI V.

[3] We used the TPC-C benchmark [TPC, 2005] as a reference for our base case parameters. Our implementation of the benchmark for 5 warehouses results in a database of 2.595.055 items and an average transaction response time of 0.170 seconds. In TPC-C update transactions account for 92% of the workload.

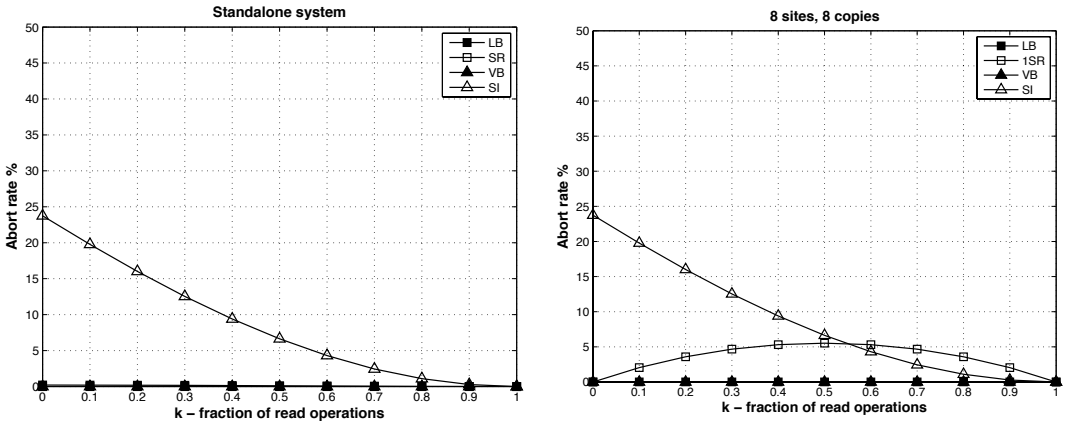
Parameter	Values considered
DB_SIZE	500.000, **2.500.000**
TPS	(100, **250**, ..., 800)/sites
op	**200**, 1000
op_time	**0.170**/*op*
L	0, **0.1**...1
V	1, 2, 3, 4
sites	1, 2, 4, **8**, 12, 16, 20
copies	1, **2**, 8

Table 7.2. Model parameter values

7.3.3 Standalone vs. Fully Replicated System

Figure 7.1 compares the execution and termination abort rates for standalone and fully replicated systems. The standalone lock-based system has very low deadlock rate and there are no aborts due to the certification test (see Fig. 7.1(a)): if two conflicting transactions execute concurrently at the same replica, the local database scheduler will serialize them, and thus both transactions can commit. Adding replicas increases the number of transactions executing at distinct sites, thus the aborts due to lack of synchronization grow accordingly (see Fig. 7.1(b)). Differently from the model introduced by Gray et al. [1996], where the deadlock rate rises as the third power of the number of replicas, in our replication model all commits of update transactions are ordered and thus replication does not increase the deadlock rate of the 1SR system — the aborts in Figure 7.1(b) are due to the certification test. We further use the lock-based 1SR system as a baseline for analyzing the aborts of partially replicated SI systems.

In a version-based system, even if two conflicting transactions execute at the same database site, only one is allowed to commit — notice that in SI two concurrent transactions conflict if they update the same data item. Therefore, replication does not affect the system abort rate. Moreover, there are no aborts due to unavailable consistent snapshots in the standalone and the fully replicated systems; in both cases the number of data versions available is unbounded.



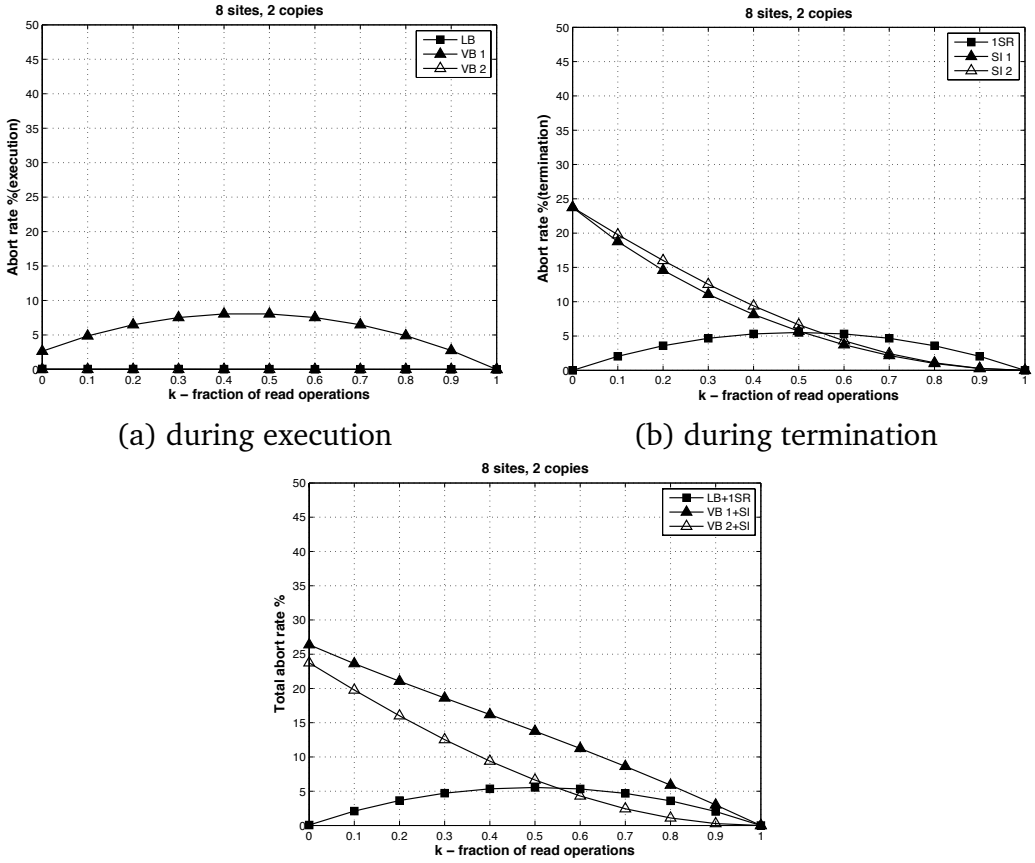
(a) standalone system (b) fully replicated system

Figure 7.1. Standalone vs. fully replicated system, $TotalTPS = 250$

7.3.4 Two Data Versions are Sufficient to Eliminate Execution Aborts

Figure 7.2 presents the transaction abort rate during (a) execution and (b) termination, and (c) the total system abort rate for the base scenario configuration. If only a single data version is available during the execution of transactions under version-based concurrency control, up to $\approx 8\%$ of transactions may abort due to failed reads (see Fig. 7.2(a), VB 1 curve). The abort rate depends on the number of write operations in update transactions. With 100% of write operations in update transactions, only read-only transactions can abort due to not obtaining the requested database snapshot. On the other hand, with 100% of read operations there are no updates and, hence, no failed reads.

The availability of at least one additional data version is sufficient to almost completely eliminate the aborts during execution (Fig. 7.2(a), VB 2 curve)! This is because the number of data versions available reduces the abort rate exponentially (see Eq. 7.2.25). We investigate this phenomenon further in Figure 7.3, which depicts the system abort rate at the execution phase when increasing the number of available versions. Two versions of each data item reduce the execution aborts so that they become insignificant (0.008% in the worst case). Therefore, increasing further the number of versions available will not affect remarkably the abort rate.



(a) during execution (b) during termination

(c) total abort rate

Figure 7.2. The effects of distributed transactions; base scenario

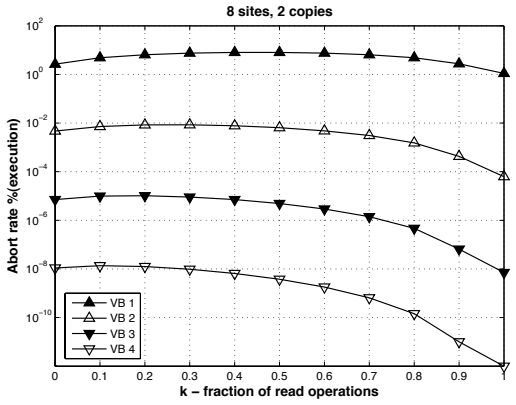


Figure 7.3. The effects of versions available; base scenario, y-axis in logarithmic scale

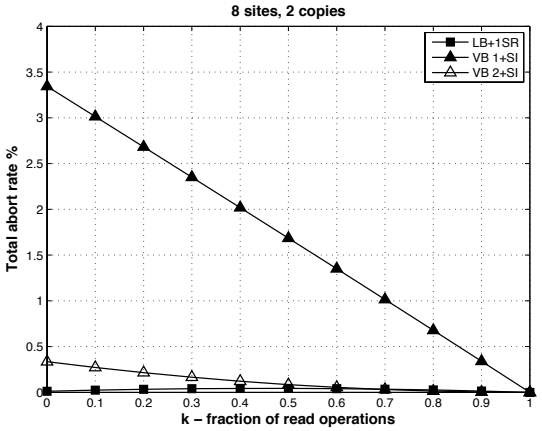


Figure 7.4. The effects of increasing read-only transactions in the workload; $L = 0.9$

If update transactions contain a high number of read operations, VB 1+SI can be as good as LB+1SR, while VB 2+SI may even outperform LB+1SR (see Fig. 7.2(c), when 60% or more of read operations in update transactions). However, if the workload is dominated by update operations, version-based systems abort more transactions, regardless of the number of versions per data item available. This is due to the differences in the certification test. As presented in Section 7.2.1, to ensure SI the certification test of a version-based system checks write-write conflicts between concurrent transactions, while to ensure 1SR the certification test of a lock-based system checks read-write conflicts. With a lot of update operations the probability of write-write conflicts increases and thus, the version-based system exhibits higher abort rate. During the termination phase, the abort rate of SI 1 is lower than SI 2 (see Fig. 7.2(b)). This is because in the termination phase our model accounts for aborts that happen during execution. Since in VB 1 a lot of transactions are aborted during execution, fewer transactions reach the termination phase, and consequently, fewer transactions are aborted.

7.3.5 The Impact of Read-Only Transactions

To evaluate the impact of read-only transactions in the workload, we have varied the L parameter. Figure 7.4 illustrates the total system abort rate when $L = 0.9$. The abort rate of both LB+1SR and VB 2+SI is very low since with very few updates the termination abort rate is small and it is unlikely that transactions deadlock during the execution. However, for VB 1+SI, if the fraction of write

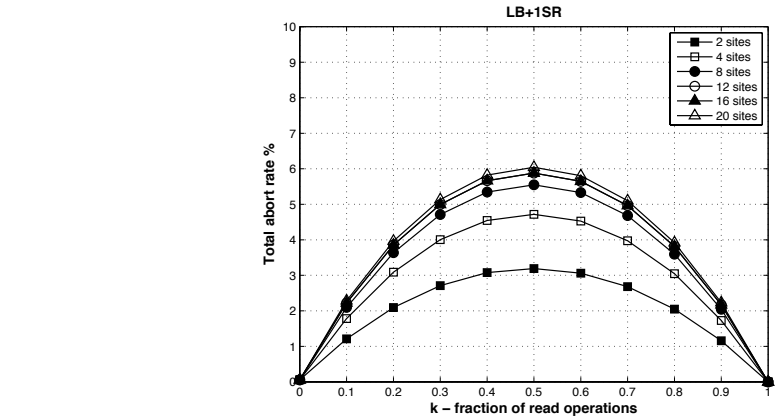
operations in the transactions is high, the execution aborts dominate the total system abort rate. If all update transactions perform a lot of write operations, read-only transactions still have a significant probability of aborting due to not obtaining the requested version of the data item. For example, for VB 1+SI, $L = 0.9$ and $k = 0$, the probability for a read operation to abort due to not obtaining the correct version is $0.169 \cdot 10^{-3}$, but the probability for a read-only transaction to abort during execution is 0.033 (see Eq. 7.2.26 and 7.2.27, respectively). Thus, even if the workload over the partially replicated system is dominated by read-only transactions, but the few update transactions perform a lot of updates, read-only distributed transactions can still cause a noticeable number of aborts. This is in contrast to typical fully replicated SI systems, in which the number of versions available in each replica is unbounded, and thus, read-only transactions never abort.[4]

7.3.6 Adding Database Sites to the System

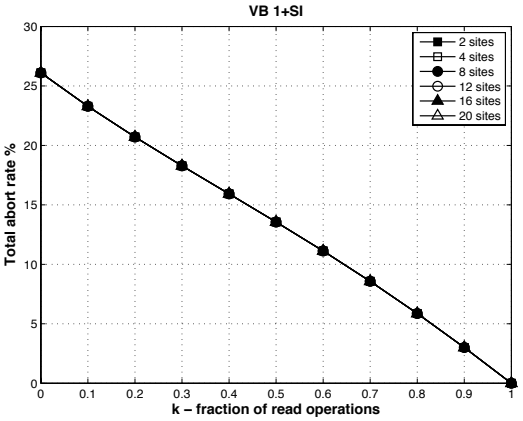
Figure 7.5 illustrates how the number of database sites affects the total system abort rate. We use 2 database copies; the number of transactions executing concurrently over the replicated system is kept constant for all configurations considered, i.e., $TotalTPS = 250$.

Adding replicas to the system has a different impact on lock- and version-based systems. Although varied number of replicas has no influence on the abort rate during the execution phase in both systems, the difference comes from the distinct certification tests. In a lock-based system, if two conflicting transactions execute concurrently at the same replica, the local database scheduler will ensure serializable execution, and thus both transactions can commit. Increasing the number of replicas in the system, increases the number of transactions executing at distinct replicas, thus the aborts due to lack of synchronization grow accordingly (Fig. 7.5(a)). In a version-based model, even if two conflicting transactions execute at the same database site, only one is allowed to commit. Therefore, the number of database replicas in the system and so the amount of parallelism among conflicting transactions does not affect the abort rate of the system (Figs. 7.5(b) and (c)).

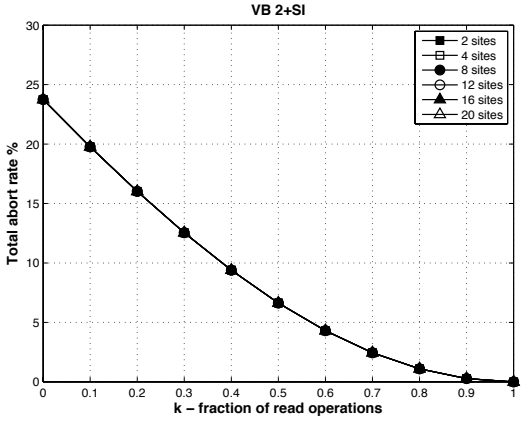
[4]Efficient implementations of SI, such as Oracle or PostgreSQL, limit the amount of space used to record data versions. Thus, if the workload is update intensive, even read-only transactions can abort.



(a) lock-based system



(b) version-based system, $V = 1$

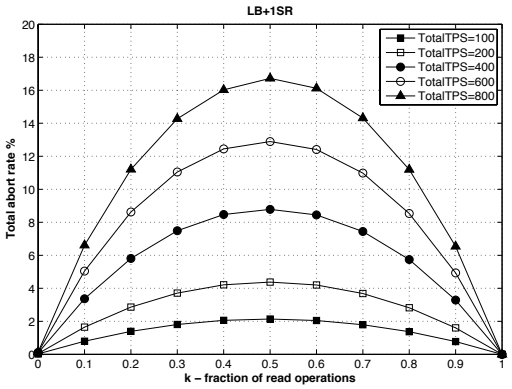


(c) version-based system, $V = 2$

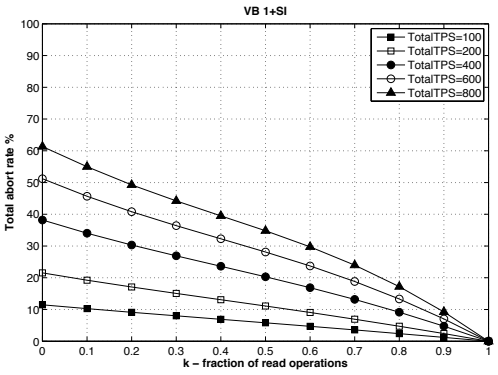
Figure 7.5. The effects of the number of database sites, base scenario

7.3.7 The Effects of the Load, the Number of Operations and the Database Size

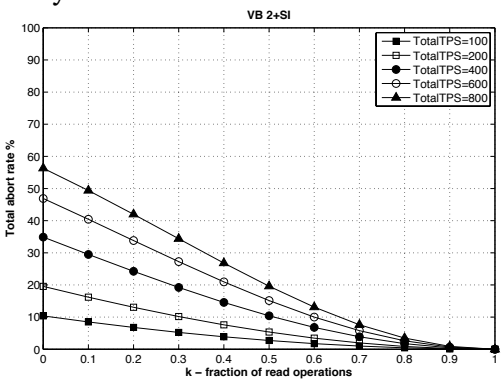
The throughput over the system has a linear impact on the total system abort rate (see Fig. 7.6), while the number of operations has an exponential effect on the aborts for both lock- and version-based systems (Fig. 7.7(a)).
Augmenting the number of operations increases the number of concurrent transactions in the system, and thus has a quadratic impact on the number of resources updated or locked (see Eq. 7.2.1). However, once we calculate various probabilities (probability that transaction waits), the number of operations appears in the exponent of the formula (e.g., see Eq. 7.2.7).



(a) lock-based system

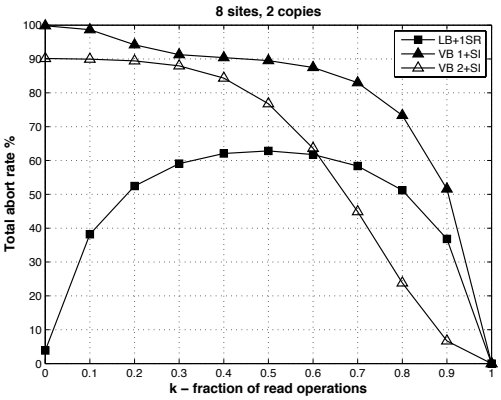


(b) version-based system, $V = 1$

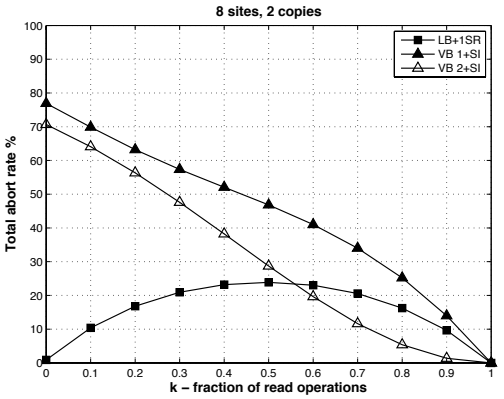


(c) version-based system, $V = 2$

Figure 7.6. The impact of the load over the system; base scenario



(a) increased number of operations



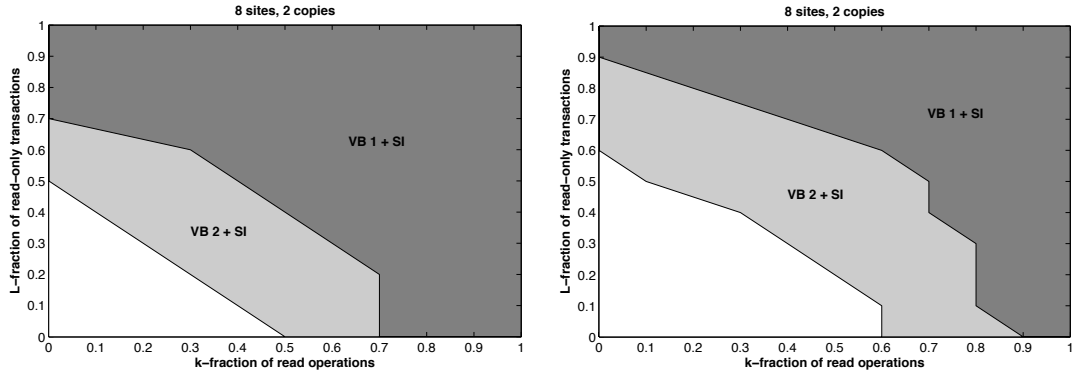
(b) reduced database size

Figure 7.7. The impact of the number of operations, $op = 1000$ (a) and the database size, $DB\_SIZE = 500.000$ (b); base scenario

The impact of the database size is smaller: being just at the denominator of the Eq. 7.2.1 formula, reducing the database size, increases the total system aborts (see Fig. 7.7(b)).

7.3.8 The Environments Beneficial to SI Version-Based Systems

Figure 7.8 depicts the environments under which SI can be safely used with partial replication. We have varied the L and k parameters and report the results where the total system abort rate is equal to or below 10% with $TotalTPS = 250$ (Fig. 7.8(a)) and $TotalTPS = 400$ (Fig. 7.8(b)). The dark and light gray areas represent configurations of version-based SI systems with one (i.e., VB 1+SI) and two data versions available (i.e., VB 2+SI). Workloads composed of a lot of read-only transactions and workloads where update transactions contain many read operations represent environments beneficial to SI systems.



(a) $TotalTPS = 250$ (b) $TotalTPS = 400$

Figure 7.8. Configurations where abort rate of SI systems is $\leq$ 10%; base scenario

7.4 Related Work and Final Remarks

The majority of partial replication solutions in the literature guarantee one-copy serializability (e.g., [Camargos et al., 2007; Cecchet et al., 2004; Coulon et al., 2005; Fritzke and Ingels, 2001; Holliday et al., 2002; Kemme, 2000; Schiper et al., 2006; Sousa et al., 2001]) and assume that database replicas adopt lock-based concurrency control. Several of these protocols (e.g., [Cecchet et al., 2004; Coulon et al., 2005; Schiper et al., 2006; Sousa et al., 2001]) build on the

strong assumption that transactions can always execute locally at one database site. Such an assumption requires prior knowledge of the workload and a very precise data distribution over the replicas (e.g., [Elnikety et al., 2007]) or at least a single replica that holds the whole database (e.g., [Cecchet et al., 2004]). The protocols in [Armendáriz-Íñigo et al., 2008; Serrano et al., 2007] allow distributed transactions by ensuring that all remote accesses are able to find the required data version. The work in [Armendáriz-Íñigo et al., 2008] orders the beginnings of distributed transactions at all the replicas involved and, thus, prevents local executions. Serrano et al. [2007] proposes to use special "dummy" transactions, created at all database sites every time an update transaction commits. All remote operations must execute within the dummy transaction associated with the required database snapshot. Maintaining a high number of dummy transactions may affect the performance of the system.

This chapter presents a simple probabilistic analysis of abort rates in partially replicated systems. Two concurrency control mechanisms are considered: lock- and version-based. The lock-based system models the behavior of a replication protocol providing one-copy serializability, while the version-based system ensures snapshot isolation.

The presented analytical evaluation revealed that in the version-based system the number of data versions available decreases the execution abort rate exponentially. As a consequence, in all cases considered, two versions of each data item were sufficient to eliminate the aborts due to distributed transactions. Furthermore, in the version-based system even if the workload over the partially replicated system is dominated by read-only transactions, but the few update transactions perform a lot of updates, distributed read-only transactions can still cause a noticeable number of aborts, as opposed to typical full replication protocols, in which the number of versions available is (in principle) unbounded, and thus, read-only transactions executing under snapshot isolation are never aborted.

Chapter 8

Conclusions

8.1 Contributions

Performance, high-availability, and correctness are the ultimate goals in the development of distributed database systems and have been the subject of a large amount of research in the last three decades. In spite of the many replication protocols proposed, their application in real scenarios is limited. Although kernel-based replication solutions can benefit substantially from the optimizations in the database engine, such protocols are difficult to maintain in practice. Databases usually take part in complex IT infrastructures where replication should ideally be transparent to both the end users and the database sites, and should therefore be implemented in a middleware layer. Thus, the main focus of this thesis is on non-intrusive database replication protocols. Such protocols are independent of the underlying database engine and consequently can be easily ported to different systems. The protocols proposed are workload-aware: the performance of the system is improved by exploiting specific knowledge of the workload. The thesis research has led to five contributions summarized below.

Multiversion Database State Machine replication. This thesis has presented the Multiversion Database State Machine, an instance of the Database State Machine replication placed at the middleware layer. The Multiversion Database State Machine assumes predefined, parameterized transactions. The particular data items accessed by a transaction depend on the transaction's type and the parameters provided by the application program when the transaction is instantiated. By estimating the data items accessed by transactions before their execution, even if conservatively, the replication protocol is spared from extracting readsets and writesets during the execution. In the case of the Multiversion

Database State Machine, this has also resulted in a certification test simpler than the one used by the original Database State Machine, although both techniques guarantee the same strong consistency.

Workload Characterization Techniques. This thesis has presented workload characterization techniques that allow to identify transaction access patterns outside the database engine. Such knowledge may be useful even before actually executing the transactions at the database engine. Conservative replication protocols require transaction's access patterns data beforehand to guarantee replicated system's consistency; optimistic protocols may use such information to reduce the synchronization aborts induced by conflicting transactions executing concurrently at distinct database replicas. We have introduced the SQL Inspector, a tool capable to automatically identify conflicting transactions by partially parsing them. To the best of our knowledge this is the first attempt to automate the extraction of transactions readsets and writesets outside the database engine.

Conflict-Aware Load-Balancing Techniques. A key property of the Multiversion Database State Machine is that if transactions with similar access patterns execute at the same database replica, then the local replica's scheduler will serialize them and both can commit, reducing the abort rate. However, performance is improved if transactions execute concurrently on different replicas. We have proposed a hybrid load balancing technique which allows to give more or less significance to minimizing conflicts or maximizing parallelism: based on the information obtained by the SQL Inspector, we schedule transactions to replicas so that the number of conflicting transactions executing on distinct replicas is reduced and the load over the replicas is equitably distributed. Experimental results showed that exploring specific workload information while assigning transactions to the replicas is a promising technique to improve the performance of non-intrusive replication protocols.

BaseCON and the cost of correctness criteria for non-intrusive database replication. We have investigated the performance cost of implementing different consistency degrees in BaseCON and two other non-intrusive replication protocols. The experimental evaluation revealed that stronger consistency does not necessarily imply worse performance in the context of middleware-based replication. On the contrary, two of the three protocols evaluated are able to provide different consistency guarantees without penalizing system's performance. Fur-

thermore, even being conservative BaseCON results in higher throughput than both primary-backup and optimistic update-everywhere replication protocols.

Probabilistic model for partial replication. In this thesis we have introduced a probabilistic model for abort rates of partially replicated systems. Two correctness criteria are considered: the lock-based system models the behavior of a replication protocol providing one-copy-serializability; the version-based system models snapshot isolation. The analytical evaluation has revealed that in the version-based system the number of data versions available decreases the execution abort rate exponentially. Furthermore, even if the workload over the partially replicated system is dominated by read-only transactions, but the few update transactions perform a lot of updates, distributed read-only transactions can still cause a noticeable number of aborts, as opposed to typical full replication protocols, in which the number of versions available is (in principle) unbounded, and thus, read-only transactions executing under snapshot isolation are never aborted.

8.2 Future Directions

In extension to the work presented in this thesis, we believe that several points are worth further investigation.

Application-specific replication. In [Stonebraker et al., 2007] the authors criticize the one-size-fits-all paradigm of databases and argue for a redesign, which would take into account application-specific needs. Likewise, we believe that database replication protocols can achieve better performance if application knowledge is taken into account. Both protocols presented in this thesis and the conflict-aware load-balancing techniques are workload-aware, i.e., the SQL Inspector introduced in Chapter 4 allows to exploit application specific information to improve the overall performance of the system. A more thorough evaluation of load-balancing techniques, for example, with more emphasis on transaction weight and with different workloads, would allow to take further conclusions about the approach.

Recovery. The majority of replication protocols rely on the state transfer mechanism of group-communication to cope with recovery of failed sites, merging of partitions, or joining of new sites. Therefore the database sites update and

resynchronize their database copies by copying the entire database state to another site. This consumes time and bandwidth, as normal update operations of the database system are interrupted. To speed up the recovery process specific knowledge about transaction profiles may be necessary [Camargos et al., 2009]. It may be interesting to explore further how the SQL Inspector presented in this thesis can be used to automatically obtain such information.

Adaptability and reconfiguration of the replication protocols. The distribution of transactions in the workload over time may vary significantly. If the protocol is tailored for executing one type of transactions, but the mix changes considerably, the performance of the system may drop dramatically. Replica failures and recovery may have similar effects. There have been only few works on self-reconfiguration and adaptability in the area of database replication [Milán-Franco et al., 2004; Correia et al., 2008; Serrano et al., 2008]. Design and evaluation of the protocols able to react to different changes in the system (e.g., reconfiguration, workload) seems to be an interesting and relevant problem.

Practical partial replication. Both vDBSM and BaseCON were designed to cope with full replication only. Although the original DBSM has been previously extended to support partial replication [Sousa et al., 2001; Schiper et al., 2006], the proposed solutions build on the strong assumption that transactions can always execute locally at one database site. It would be worth investigating if our workload characterization techniques can help porting BaseCON and vDBSM to partial replication environments, where no restrictions on data placements are applied.

Snapshot Isolation and partial replication. In Chapter 7 we have presented a probabilistic model of transaction abort rates in partially replicated systems, reasoning about the usage of snapshot isolation in such environments. Snapshot isolation was originally introduced as a correctness criterion for centralized databases [Berenson et al., 1995], later formalized for replicated environments by Lin et al. [2005], and extended further to better fit replication in [Elnikety et al., 2005]. We believe that partial replication requires yet another extension of SI: existing definitions may forbid some execution histories which may be acceptable, e.g., from the client point of view. In partial replication setting, it may be worth recalling and investigating further the Forward Consistent View introduced by Adya [1999].

Bibliography

Adya, A. [1999]. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*, PhD thesis, Massachusetts Institute of Technology. 106

Agrawal, D., G.Alonso, Abbadi, A. and I.Stanoi [1997]. Exploiting atomic broadcast in replicated databases, *Euro-Par'97 : Proceedings of the 3th International Euro-Par Conference on Parallel Processing*. 22, 84

Aguilera, M. K., Chen, W. and Toueg, S. [1998]. Failure detection and consensus in the crash-recovery model, *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*, Springer-Verlag, pp. 231–245. 7, 8

Alonso, G. [1997]. Partial database replication and group communication primitives, *ERSADS'97: Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems*. 83

Amir, Y. and Tutu, C. [2002]. From total order to database replication, *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, IEEE Computer Society, Washington, DC, USA, pp. 494 – 506. 14, 22, 56

Amza, C., Cox, A. L. and Zwaenepoel, W. [2005]. A comparative evaluation of transparent scaling techniques for dynamic content servers, *ICDE'05: Proceedings of the 21st International Conference on Data Engineering*, pp. 230 – 241. 52

Amza, C., Cox, A. and Zwaenepoel, W. [2003]. Conflict-Aware Scheduling for Dynamic Content Applications, *USITS'03: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pp. 6–12. 15, 35, 52, 81

Armendáriz-Íñigo, J. E., Mauch-Goya, A., de Mendívil, J. R. G. and Muñoz-Escoí, F. D. [2008]. SIPRe: A partial database replication protocol with SI repli-

cas, *SAC'08: Proceedings of the 23rd ACM Symposium on Applied computing*, pp. 2181–2185. 102

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P. [1995]. A critique of ANSI SQL isolation levels, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 1 – 10. 9, 11, 57, 106

Bernabé-Gisbert, J. M., Zuikevičiūtė, V., Muñoz-Escoí, F. D. and Pedone, F. [2008]. A probabilistic analysis of snapshot isolation with partial replication, *SRDS'08: Proceedings of 27th IEEE International Symposium on Reliable Distributed Systems*, IEEE, pp. 249–258.

Bernstein, P., Hadzilacos, V. and Goodman, N. [1987]. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley. 9, 10, 13, 20, 55, 56

Camargos, L. J., Pedone, F. and Wieloch, M. [2007]. Sprint: a middleware for high-performance transaction processing, *Proceeding of EuroSys*, pp. 385–398. 101

Camargos, L., Pedone, F., Pilchin, A. and Wieloch, M. [2009]. Optimistic recovery in middleware storage systems, *Technical Report 2009/03*, University of Lugano. 106

Cecchet, E., Marguerite, J. and Zwaenepoel, W. [2004]. C-JDBC: Flexible database clustering middleware, *USENIX'04: Proceedings of USENIX Annual Technical Conference, Freenix track*, pp. 9–18. 2, 22, 36, 52, 101, 102

Chandra, T. D. and Toueg, S. [1996]. Unreliable failure detectors for reliable distributed systems, *Journal of the ACM* **43**(2). 8

Correia, A. J., Pereira, J. and Oliveira, R. [2008]. Akara: A flexible clustering protocol for demanding transactional workloads, *OTM'08: On the Move to Meaningful Internet Systems, Adaptive Distributed Systems*, pp. 691–708. 106

Correia, A. J., Sousa, A., Soares, L., J.Pereira, Moura, F. and Oliveira, R. [2005]. Group-based replication of on-line transaction processing servers, *LADC'05: Proceedings of Second Latin American Symposium on Dependable Computing*, pp. 245–260. 2

Coulon, C., Pacitti, E. and Valduriez, P. [2005]. Consistency management for partial replication in a high performance database cluster, *ICPADS'05: Proceeding of 11th International Conference on Parallel and Distributed Systems*, pp. 809–815. 2, 101

Daudjee, K. and Salem, K. [2004]. Lazy database replication with ordering guarantees, *ICDE'04: Proceedings of the 20th International Conference on Data Engineering*, pp. 424–430. 56, 80

Daudjee, K. and Salem, K. [2006]. Lazy database replication with snapshot isolation, *VLDB'06: Proceedings of the 32nd International Conference on Very Large Databases*, pp. 715 – 726. 80

Delporte-Gallet, C. and Fauconnier, H. [2000]. Fault-tolerant genuine atomic multicast to multiple groups, *OPODIS'00: Proceedings of the 4th International Conference on Principles of Distributed Systems*, Suger, Saint-Denis, rue Catulienne, France, pp. 107–122. 12

Elnikety, S., Dropsho, S. and Pedone, F. [2006]. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication, *Proceedings of EuroSys*, pp. 117 – 130. 15, 22, 23

Elnikety, S., Dropsho, S. and Zwaenepoel, W. [2007]. Tashkent+: Memory-aware load balancing and update filtering in replicated databases, *Proceeding of EuroSys*, pp. 399 – 412. 52, 102

Elnikety, S., Zwaenepoel, W. and Pedone, F. [2005]. Database replication using generalized snapshot isolation, *SRDS'05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pp. 73– 84. 10, 80, 106

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P. and Shasha, D. [2005]. Making snapshot isolation serializable, *ACM Transactions on Database Systems* . 16

Fritzke, U. J. and Ingels, P. [2001]. Transactions on partially replicated data based on reliable and atomic multicasts, *ICDCS'01: Proceeding of 21st International Conference on Distributed Computing Systems*, pp. 284–291. 101

Gray, J., Helland, P., O'Neil, P. and Shasha, D. [1996]. The dangers of replication and a solution, *SIGMOD Rec.* **25**(2): 173–182. 21, 57, 84, 87, 88, 89, 94

Gray, J. and Reuter, A. [1993]. *Transaction Processing: concepts and techniques*, Morgan Kaufmann Publishers. 26, 27, 28

Holliday, J., Agrawal, D. and Abbadi, A. E. [2002]. Partial database replication using epidemic communication, *ICDCS'02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pp. 485–493. 101

Irún-Briz, L., Decker, H., de Juan, R., Castro-Company, F., Armendáriz-Íñigo, J. E. and Muñoz-Escoí, F. D. [2005]. MADIS: A slim middleware for database replication, *Euro-Par'05: Proceedings of the 11th International Euro-Par Conference*, pp. 349–359. 14, 23

Jorwekar, S., Fekete, A., Ramamritham, K. and Sudarshan, S. [2007]. Automating the detection of snapshot isolation anomalies, *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases*, pp. 1263–1274. 36

JSqlParser [2008].
URL: *http://sourceforge.net/projects/jsqlparser* 30

Kemme, B. [2000]. *Database Replication for Clusters of Workstations*, PhD thesis, Swiss Federal Institute of Technology Zürich, Switzerland. 101

Kemme, B. and Alonso, G. [2000]. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication, *VLDB'00: Proceedings of the 26th International Conference on Very Large Databases*, pp. 134 – 143. 22

Kemme, B., Pedone, F., Alonso, G. and Schiper, A. [1999]. Processing transactions over optimistic atomic broadcast protocols, *ICDCS'99: Proceedings of 19th International Conference on Distributed Computing Systems*, pp. 424–431. 21

Kemme, B., Pedone, F., Alonso, G., Schiper, A. and Wiesmann, M. [2003]. Using optimistic atomic broadcast in transaction processing systems, *IEEE Transactions on Knowledge and Data Engineering* **15**(4): 1018–1032. 22

Lin, Y., Kemme, B., Patiño-Martínez, M. and Jiménez-Peris, R. [2005]. Middleware based data replication providing snapshot isolation, *Proceedings of the 2005 ACM SIGMOD International Conference on Management of data*, pp. 419–430. 2, 10, 14, 15, 22, 23, 80, 106

Milán-Franco, J. M., Jiménez-Peris, R., Patiño-Martínez, M. and Kemme, B. [2004]. Adaptive middleware for data replication, *Middleware'04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pp. 175–194. 52, 106

Muñoz-Escoí, F. D., Pla-Civera, J., Ruiz-Fuertes, M. I., Irún-Briz, L., Decker, H., Armendáriz-Íñigo, J. E. and de Mendívil, J. R. G. [2006]. Managing transaction conflicts in middleware-based database replication architectures, *SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pp. 401–410. 2, 23

MySQL 5.0 Reference Manual [2008].
URL: *http://dev.mysql.com/doc/refman/5.0/en/index.html* 26

Oliveira, R., Pereira, J., Correia, A. and Archibald, E. [2006]. Revisiting 1-copy equivalence in clustered databases, *SAC'06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pp. 728 – 732. 80

Pacitti, E., Coulon, C., Valduriez, P. and Özsu, M. T. [2005]. Preventive replication in a database cluster, *Distributed and Parallel Databases* **18**: 223 – 251. 15, 23

Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B. and Alonso, G. [2000]. Scalable replication in database clusters, *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pp. 315–329. 14, 21, 22, 56

Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B. and Alonso, G. [2005]. Consistent Database Replication at the Middleware Level, *ACM Transactions on Computer Systems* . 2, 15, 21, 22, 35, 56, 80

Pedone, F. [1999]. *The Database State Machine and Group Communication Issues*, PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland. 88

Pedone, F. and Frølund, S. [2008]. Pronto: High availability for standard off-the-shelf databases, *Journal of Parallel and Distributed Computing* **68**(2): 150–164. 2, 22, 56, 57, 80

Pedone, F., Guerraoui, R. and Schiper, A. [1997]. Transaction reordering in replicated databases, *SRDS'97: Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*. 22

Pedone, F., Guerraoui, R. and Schiper, A. [1998]. Exploiting atomic broadcast in replicated databases, *Euro-Par'98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*. 22, 56

Pedone, F., Guerraoui, R. and Schiper, A. [2003]. The database state machine approach, *Journal of Distributed and Parallel Databases and Technology* **14**: 71–98. 13, 14, 22, 56, 80

Plattner, C. and Alonso, G. [2004]. Ganymed: scalable replication for transactional web applications, *Middleware'04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pp. 155 – 174. 2, 14, 15, 52, 80

Rodrigues, L., Miranda, H., Almeida, R., Martins, J. and Vicente, P. [2002]. The GlobData fault-tolerant replicated distributed object database, *Proceedings of the 1st Eurasian Conference on Advances in Information and Communication Technology*. 2, 22, 56, 80

Salas, J., Jiménez-Peris, R., Patiño-Martínez, M. and Kemme, B. [2006]. Lightweight reflection for middleware-based database replication, *SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pp. 377 – 390. 36

Schenkel, R., Weikum, G., Weißenberg, N. and Wu, X. [1999]. Federated transaction management with snapshot isolation, *Proceedings of the 8th International Workshop on Foundations of Models and Languages for Data and Objects - Transactions and Database Dynamics '99*, pp. 1–25. 84

Schiper, N., Schmidt, R. and Pedone, F. [2006]. Optimistic algorithms for partial database replication, *OPODIS'06: Proceedings of the 10th International Conference on Principles of Distributed Systems*, pp. 81–93. 2, 101, 106

Schneider, F. B. [1990]. Implementing fault-tolerant services using the state machine approach: A tutorial, *ACM Computing Surveys* **22**(4): 299–319. 11

Serrano, D., Patiño-Martínez, M., Jiménez-Peris, R. and Kemme, B. [2007]. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation, *PRDC'07: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pp. 290–297. 102

Serrano, D., Patiño-Martínez, M., Jiménez-Peris, R. and Kemme, B. [2008]. An autonomic approach for replication of internet-based services, *SRDS'08: Proceedings of 27th IEEE International Symposium on Reliable Distributed Systems*, pp. 127–136. 106

Sethi, R. [1982]. Useless actions make a difference: Strict serializability of database updates, *Journal of the ACM* **29**(2): 394–403. 56

Sousa, A., Pedone, F., Moura, F. and Oliveira, R. [2001]. Partial replication in the database state machine, *NCA'01: Proceedings of the IEEE International Symposium on Network Computing and Applications*, pp. 298–309. 2, 83, 85, 101, 106

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N. and Helland, P. [2007]. The end of an architectural era: (it's time for a complete

rewrite), *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pp. 1150–1160. 3, 105

The DaisyLib/Fractal project [2007]. http://daisylib.sourceforge.net/. 46

TPC [2001]. TPC benchmark W. Standard Specification. http://www.tpc.org/tpcw/spec/. 29, 33

TPC [2005]. TPC benchmark C. Standard Specification. http://www.tpc.org/tpcc/spec/. 29, 33, 93

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. and Alonso, G. [2000a]. Database replication techniques: a three parameter classification, *SRDS'00:Proceedings the 19th IEEE Symposium on Reliable Distributed Systems*, pp. 206–215. 57

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. and Alonso, G. [2000b]. Understanding replication in databases and distributed systems, *ICDCS'00: Proceedings of the 20th International Conference on Distributed Computing Systems*, pp. 464–474. 57

Wu, S. and Kemme, B. [2005]. Postgres-R(SI):combining replica control with concurrency control based on snapshot isolation, *ICDE'05: Proceedings of the IEEE International Conference on Data Engineering*, pp. 134 – 143. 22, 80

Zuikevičiūtė, V. and Pedone, F. [2005]. Revisiting the Database State Machine Approach, *WDIDDR'05: Proceedings of VLDB Workshop on Design, Implementation, and Deployment of Database Replication*, pp. 1–7.

Zuikevičiūtė, V. and Pedone, F. [2006]. Conflict-Aware Load-Balancing Techniques for Database Replication, *Technical Report 2006/01*, University of Lugano.

Zuikevičiūtė, V. and Pedone, F. [2008a]. Conflict-Aware Load-Balancing Techniques for Database Replication, *SAC'08: Proceedings of ACM Symposium on Applied Computing, Dependable and Adaptive Distributed Systems Track*, ACM Press, pp. 2168–2173.

Zuikevičiūtė, V. and Pedone, F. [2008b]. Correctness Criteria for Database Replication: Theoretical and Practical Aspects, *DOA'08: Proceedings of 10th International Symposium on Distributed Objects, Middleware, and Applications*, Springer Verlag 2008.