Detecting and Tolerating Byzantine Faults in Database Systems

by

Benjamin Mead Vandiver

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by .....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Detecting and Tolerating Byzantine Faults in Database Systems

by

Benjamin Mead Vandiver

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

This thesis describes the design, implementation, and evaluation of a replication scheme to handle Byzantine faults in transaction processing database systems. The scheme compares answers from queries and updates on multiple replicas which are off-the-shelf database systems, to provide a single database that is Byzantine fault tolerant. The scheme works when the replicas are homogeneous, but it also allows heterogeneous replication in which replicas come from different vendors. Heterogeneous replicas reduce the impact of bugs and security compromises because they are implemented independently and are thus less likely to suffer correlated failures. A final component of the scheme is a repair mechanism that can correct the state of a faulty replica, ensuring the longevity of the scheme.

The main challenge in designing a replication scheme for transaction processing systems is ensuring that the replicas state does not diverge while allowing a high degree of concurrency. We have developed two novel concurrency control protocols, *commit barrier scheduling* (CBS) and *snapshot epoch scheduling* (SES) that provide strong consistency and good performance. The two protocols provide different types of consistency: CBS provides single-copy serializability and SES provides single-copy snapshot isolation. We have implemented both protocols in the context of a replicated SQL database. Our implementation has been tested with production versions of several commercial and open source databases as replicas. Our experiments show a configuration that can tolerate one faulty replica has only a modest performance overhead (about 10-20% for the TPC-C benchmark). Our implementation successfully masks several Byzantine faults observed in practice and we have used it to find a new bug in MySQL.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

Many people have helped me along the way. My adviser Barbara Liskov has been an invaluable source of guidance. She has helped me to refine my thinking, ensuring that I fully understand my work.

I also wish to thank my fellow co-authors and thesis committee members: Sam Madden, Hari Balakrishnan, and Mike Stonebraker. Working with them has been insightful and fun; I feel like I have learned a lot.

My research group, the Programming Methodology Group, has been a continual source of intellectual and not-so-intellectual conversation. One of the more enjoyable parts of graduate school is talking about research with other smart people. Thanks to Winnie Cheng, James Cowling, Dorothy Curtis, Dan Myers, Dan Ports, David Schultz, Ben Leong, Rodrigo Rodrigues, and Sameer Ajmani.

In addition, I wish to thank all of the people I have taught with over the years. I love teaching and you were my partners-in-crime. Thanks to Eric Grimson, Duane Boning, Joel Moses, Franz Kaashoek, Gill Pratt, Lynn Stein, Seth Teller, Michael Collins, Randy Davis, many other professors and still more graduate TAs.

My family and friends have also always been there for me. Mom and Dad always had words of encouragement. Alex and Amy attended numerous *thesis lunches.* Jenny Dorn also joined me for many lunches on the subject of the graduate student condition.

Finally, I wish to thank my wife Jen. She has been a truly incredible source of support and encouragement. I could go on, but I have not the space to thank her enough.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Transaction processing database systems are complex, sophisticated software systems involving millions of lines of code. They need to reliably implement ACID semantics, while achieving high transactional throughput and availability. As is usual with systems of this magnitude, we can expect them to contain thousands of fault-inducing bugs in spite of the effort in testing and quality assurance on the part of vendors and developers.

A bug in a transaction processing system may immediately cause a crash; if that happens, the system can take advantage of its transactional semantics to recover from the write-ahead log and the only impact on clients is some downtime during recovery. However, bugs may also cause *Byzantine faults* in which the execution of a query is incorrect, yet the transaction commits, causing wrong answers to be returned to the client or wrong data items to be stored in the database. A Byzantine fault is one where the faulty entity may perform arbitrary incorrect operations. Examples of such faults include concurrency control errors, incorrect query execution, database table or index corruption, and so on. In fact, even if a bug eventually results in a crash, the system could have performed erroneous operations (and exhibited Byzantine behavior) between the original occurrence of the bug and the eventual crash.

Byzantine faulty behavior is hard to mask because it is difficult to tell if any given operation was executed correctly or not. Existing database systems offer no protection against such faults. The field of Byzantine fault tolerance is well known

in the distributed systems research community, along with the solution: replication. This thesis describes the design, implementation, and evaluation of new replication schemes to mask and recover from both Byzantine and crash faults in transaction processing systems.

In this chapter, we first describe the goals of our system: what we set out to accomplish. Next, we discuss the approach we used to address the goals. We then present an overview of the three major contributions of the work. Finally, we conclude with an outline of the rest of the thesis.

1.1 Goals

Our aim is to develop a replication system that tolerates Byzantine faults in databases while providing clients with good performance. This section describes the goals of the system.

Correctness. The primary goal of our system is correctness: the system must behave correctly in the presence of Byzantine-faulty database replicas. The system is parametrized by f, the maximum number of simultaneously faulty replicas. If no more than f replicas are faulty, then clients must receive only correct answers for transactions that commit and non-faulty database replicas must have equivalent logical state.

The system must appear to the clients as if it were a non-replicated system (*e.g.*, *single-copy* consistency). Furthermore, it must provide a strong consistency model such as serializable or snapshot isolation. With strong consistency, clients may ignore much of the additional complexity introduced by replication.

Good Performance. We require that the performance of our system must not be substantially worse than that of a single, non-replicated database. In the transactional database context, good performance implies support for high concurrency. The problem is that, if presented with a workload consisting of operations from a set of concurrent transactions, different replicas may execute them

in different orders, each of which constitutes a correct execution at that replica. However, these local orders may not all be consistent with each other, which can lead to divergence in the state of non-faulty replicas. Ordering problems can be avoided by running one transaction at a time, but this approach eliminates all concurrency and performs poorly.

Support for Heterogeneity. The system must support heterogeneous database replicas to ensure failure independence. Heterogeneous replicas are unlikely to share the same set of bugs since their implementations were produced independently. Heterogeneity can also reduce the probability that security vulnerabilities will affect the correct behavior of the replicated system, because a single vulnerability will likely affect only one of the replicas. Clearly all replicas in a deployment must be similar enough that the system can cause all non-faulty replicas to process queries "identically."

1.2 Approach

The goals listed above are individually straightforward to implement, but taken together they describe a significant design challenge. Our system satisfies the goals through the use of the following four techniques:

Voting. By running each client operation on multiple replicas and voting on the result, the system can guarantee correct answers as long as less than some threshold of the replicas are faulty. However, voting only works if replica execute operations in equivalent orders and the operation results are comparable across heterogeneous replicas.

Middleware. Our system interposes an intermediary between the clients and the database replicas. This middleware runs the replication protocol, ensuring correct answers and strong consistency for the clients and consistent state for the database replicas. Middleware simplifies client code by hiding the replication mechanism from the clients. To clients, middleware acts like a single database

with a standard interface. The middleware interacts with database replicas via the standard client interface, thus requiring no (or minimal) modifications to database replica software. In some cases, it does not even require any additional software to run on the machine hosting a replica database. Treating each replica as a mostly "shrink-wrapped" subsystem eases the deployment and operation of our system and allows it to work with commercial offerings.

Primary/Secondary Scheme. To achieve good performance, our system selects one replica to be the primary and make ordering decisions about transactions for the rest of the replicas (the secondaries). As long as the primary is non-faulty, it is highly efficient at determining the available concurrency in the workload. If the primary becomes faulty, the system replaces it with a secondary. Since the expectation is that databases are infrequently faulty, we trade off performance loss in an unusual case for a faster transaction ordering mechanism in the common case. Obviously, a Byzantine-faulty primary must be prevented from impacting the correctness of the system.

Repair. Once a database replica has suffered a Byzantine fault, its state may be incorrect. A repair operation corrects the state of the faulty replica so that it can once again contribute positively to the voting process. Without an efficient repair mechanism, the likelihood of exceeding f faults increases over time, limiting the lifetime of the system.

1.3 Contributions

This thesis presents and evaluates three major new contributions to the area of Byzantine fault tolerance of databases. The first two contributions are novel concurrency control protocols that provide good performance while tolerating Byzantine faults in databases. The third contribution is a repair mechanism that can be used to correct the state of database replicas that have become faulty.

Commit Barrier Scheduling. A key contribution of this thesis is a new concurrency control protocol, called *commit barrier scheduling* (CBS), that allows our system to guarantee correct behavior and single-copy serializable execution while achieving high concurrency. CBS constrains the order in which queries are sent to replicas just enough to prevent conflicting schedules, while preserving most of the concurrency in the workload. Additionally CBS ensures that users see only correct responses for transactions that commit, even when some of the replicas are Byzantine faulty. CBS requires that each replica implement concurrency control using rigorous two-phase locking, but this is not onerous since rigorous two-phase locking is used in many production databases. CBS does not require any modification to the database nor any co-resident software. Though unnecessary for correctness, such co-resident software can improve the performance of the protocol.

We have implemented Commit Barrier Scheduling as part of *HRDB (Heterogeneous Replicated DataBase)*, which uses SQL databases as replicas. HRDB supports replicas from different vendors (we have used IBM DB2, MySQL, Microsoft SQLServer, and Derby). Our experiments with the HRDB prototype show that it can provide fault-tolerance by masking bugs that exist in some but not all replicas. HRDB is capable of masking deterministic bugs using replicas from heterogeneous vendors and non-deterministic bugs using different versions from the same vendor. In addition, using HRDB we discovered a serious new non-deterministic bug in MySQL; a patch for this bug has been included in a recent MySQL release. We found HRDB to have reasonable overhead of 10-20% (compared to a non-replicated database) on TPC-C, a database industry standard benchmark that models a high-concurrency transaction processing workload.

Snapshot Epoch Scheduling. The second major contribution is another new concurrency control control protocol, called *snapshot epoch scheduling* (SES), which has the same correctness and performance properties as CBS, but provides

single-copy snapshot isolation instead. Snapshot isolation is weaker than serializability, but is supported by many popular databases (*e.g.,* Oracle, PostgreSQL, and Microsoft SQLServer) due to its desirable performance characteristics. Rather than requiring that replicas provide serializable isolation using rigorous two-phase locking, SES instead requires that replicas implement snapshot isolation. Due to the nature of snapshot isolation, SES requires additional functionality from component databases: writeset extraction. Writeset extraction is required to retrieve data necessary for efficient response to faults.

We implemented SES as a module for HRDB and tested it with PostgreSQL, a popular open-source database that implements snapshot isolation and has an extension that implements writeset extraction. We found HRDB running SES to perform well: replication only introduces 18% overhead.

Repair Algorithm. The third contribution is two database repair mechanisms for correcting the state of a faulty replica, one for CBS and one for SES. These mechanisms use only the SQL interface, thus they work on heterogeneous replicas. In addition to these mechanisms, we also analyze two compare-and-repair algorithms, one we developed and one from the literature. The algorithms efficiently correct faulty state, achieving an order of magnitude performance improvement over a naive algorithm.

Summing up the individual contributions presented above, this thesis demonstrates a practical application of Byzantine fault tolerance protocols. Databases suffer from Byzantine faults and clients care about receiving correct answers to their queries. Moreover, the database community provides various implementations with similar interfaces, allowing us to achieve the replica failure independence assumed present, but often overlooked, in other applications of Byzantine fault tolerance.

1.4 Outline

The thesis starts off with a discussion of related work in Chapter 2, then describes our system model and assumptions in Chapter 3. Chapter 4 presents Commit Barrier Scheduling and Chapter 5 details Snapshot Epoch Scheduling. Chapter 6 evaluates the overhead and fault tolerance properties of both CBS and SES. The repair mechanisms are described and evaluated in Chapter 7. Finally, we conclude and discuss future work in Chapter 8.

Chapter 2

Related Work

To the best of our knowledge, HRDB is the first practical Byzantine fault-tolerant transaction processing system that is able to execute transactions concurrently. We begin by discussing research on systems that tolerate Byzantine faults. Then we discuss work on the use of replication to allow database systems to survive crashes. We conclude with a brief discussion of database repair tools which could be used for replica state repair.

2.1 Tolerating Byzantine Faults

The BFT library [5] provides efficient Byzantine fault-tolerance through state machine replication; it requires that all operations be deterministic. One might ask whether this library, previously used for NFS, can be easily adapted to transactional databases. BFT ensures that all the replicas process operation requests in the same order; in our system operations are queries, COMMITs, and ABORTs. As originally proposed, BFT requires that operations complete in order; with this constraint, the first query to be blocked by the concurrency control mechanism of the database would cause the entire system to block. Kotla and Dahlin propose [19] a way to loosen this constraint by partitioning operations into non-conflicting sets and running each set in parallel. To do this, however, they require the ability to determine in advance of execution whether two operations (or transactions) conflict, which is possible in

some systems (such as NFS) and difficult in databases [36]. In database systems, the statements of the transaction are often not available at the time the transaction begins. Furthermore, just determining the objects that a single database statement will update involves evaluating predicates over the contents of the database (an operation that must itself be properly serialized.) To see why, consider an update in an employee database that gives a raise to all employees who make less than $50,000; clearly, it is not possible to determine the records that will be modified without (partially) evaluating the query. In contrast, our scheme does not require the ability to determine the transaction conflict graph up front but still preserves concurrency while ensuring that state machine replication works properly.

BASE [6] is an extension of BFT that allows the use of heterogeneous replicas by implementing stateful conformance wrappers for each replica. Typically, the conformance wrapper executes the operations while maintaining additional state that allows it to translate between the local state of the replica and the global system state. Our system interacts with replicas via SQL, which is a more expressive interface than the BASE examples. Rather than maintaining state in the wrapper, our scheme depends on rewriting the SQL operations to keep the replicas' logical state the same. A more full featured implementation might maintain wrapper state to circumvent some vagaries of SQL.

Any system that provides agreement in the presence of Byzantine faults requires that no more than 1/3 of the replicas are faulty. Recent work [47] shows that, provided with a mechanism to agree on the order of operations, execution under a Byzantine fault model requires only that no more than 1/2 of the execution replicas are faulty. Thus a two-tier system can use a set of lightweight nodes to perform agreement and a smaller set of nodes to execute the workload. We use this property in our design.

2.1.1 Byzantine faults and Databases

A scheme for Byzantine fault-tolerant database replication was proposed two decades ago [25], but this work does not appear to have been implemented and the proposal did not exhibit any concurrency, implying that it would have performed quite poorly.

When this paper was written, Byzantine fault tolerance was not as well understood, particularly the lower bound of $3f + 1$ replicas necessary to perform agreement in the presence of up to f Byzantine faulty replicas. Their algorithm uses only $2f + 1$ replicas to perform agreement, making it impossible that the scheme worked.

C-JDBC [9] defines a RAIDb level called RAIDb-1ec that uses voting to tolerate Byzantine failures in databases. Unlike the other RAIDb levels, no performance results for RAIDb-1ec exist, and the feature has been removed from subsequent versions of C-JDBC (or Sequoia as it is now called). It is unlikely that RAIDb-1ec was ever fully implemented. It seems unlikely that RAIDb-1ec supports concurrent write operations because the other modes of C-JDBC do not.

Goldengate Veridata [12] uses a log extraction mechanism to compare the update streams of a pair of databases, looking for discrepancies. Veridata is a commercial product devoted to data integrity, showing that there is real demand for this service. Since Veridata does not observe interactions with clients, it can only discover eventual inconsistencies and cannot verify answers that are actually sent to clients. Because it compares streams from two replicas, Veridata can flag inconsistencies, but it lacks the required 3 replicas to decide what the correct answer should be.

Gashi *et al.* [10] discuss the problem of tolerating Byzantine faults in databases and document a number of Byzantine failures in bug reports from real databases (see the discussion in Section 3.3.1). They propose a middleware-based solution where transactions are issued to replicas and the results are voted on to detect faults. They do not, however, present a protocol that preserves concurrency or discuss the associated problem of ensuring equivalent serial schedules on the replicas.

Castro *et al.* successfully used the BASE [6] library to construct a Byzantine fault tolerant version of an object-oriented database called Thor [23]. While Thor supports transactions with strong consistency, it is an object store, not a modern relational database system. The specific techniques used to make Thor work with BASE are not applicable to generic databases.

With the outsourcing of data storage, certifying that untrusted databases have correctly executed queries has become a research topic called data publishing. A data

owner provides data for the service provider to serve to clients, and the owner would like the clients to have confidence that their queries are being executed correctly on the data supplied by the owner. One common mechanism is for the owner to compute a Merkle [24] tree of signed hashes [27, 28], while another method is to seed the data with marker tuples [46]. All of these mechanisms require a *correct owner* and support a restricted set of queries on the data. Many of them also require modifications to database software.

2.2 Database Replication for Crash Faults

Database replication is at the core of modern high availability and disaster recovery mechanisms. None of the database replication mechanisms mentioned below tolerate Byzantine faults. Many of the schemes can not be modified to support tolerating Byzantine faults because they use asymmetric execution: transactions are executed on a single replica and the results propagated to the other replicas. Asymmetric execution provides good performance by avoiding duplication of work. However, if the executing replica is faulty, then it can propagate incorrect values into the state of correct replicas. Symmetric execution introduces additional challenges with regards to concurrency control: many replicas must execute transactions in the same order.

Replication has been well researched and implemented by industry [15, 41]. Many of the points in the space have been examined [13, 44], along a number of dimensions, such as consistency model, agent or entity performing replication, what operations the agent has available, replica operating environment, and types of faults tolerated. This thesis focuses on providing strong consistency with middleware using heterogeneous, well-connected replicas. Weak consistency models [30, 13] can be unsatisfying to users and introduce new challenges for Byzantine fault tolerance. Supporting heterogeneous replicas is key to fault independence and middleware reduces the cost of supporting heterogeneity. While some work has addressed database replication with adverse network conditions [37, 1], we do not: we focus on replication with well-provisioned, low latency, and highly reliable interconnect.

A typical strong consistency model is single-copy serializable [18, 4, 3], where the system behaves as if it is a single, non-replicated database that executes concurrent transactions as if in some serial order. Providing single-copy serializable behavior with a middleware-based system poses a challenge because the middleware does not have direct control over the concurrency manager in the database replicas. One resolution is to implement a concurrency manager in the middleware, usually using a course-grained mechanism like partitioning [17] or table-level locking [2]. The performance of such solutions is dependent on the workload matching the granularity of the mechanism.

The Pronto [29] system uses a mechanism similar to ours: have one replica decide the serial schedule for the rest. In Pronto, the primary replica decides the ordering and the secondary replicas execute transactions sequentially in the order they complete executing on the primary. Performance is limited to that of sequential execution, but Pronto can guarantee that the secondary replicas are hot spares: a crash of the primary does not result in loss of any committed transactions, nor does any recovery need occur to promote a secondary to be the new primary.

Recent research has focused on providing single-copy snapshot isolation [22]. Snapshot isolation is a popular concurrency control mechanism that is slightly weaker than serializable but performs well on common workloads. Postgres-R(SI) [45] modifies the base level PostgreSQL implementation to efficiently support replicated snapshot isolation. The Ganymede [31] and SI-Rep [22] provide single-copy snapshot isolation with middleware. All of the above systems require writeset extraction. They use writesets to achieve good performance by having only one replica execute a transaction and extracting the writeset for application on the other replicas. Furthermore, since transactions conflict under snapshot isolation when their writesets intersect, writesets enable middleware to perform conflict detection and resolution. Writeset extraction is not available as a standard database operation; all the implementations depend on modified versions of the PostgreSQL database.

A number of systems support heterogeneous replication. Probably the closest to our system is C-JDBC [7, 9], which uses JDBC middleware to provide scalability and

crash fault tolerance. C-JDBC supports partitioning and partial replication, allowing scale-up by sending queries to the appropriate replica. C-JDBC is optimized for workloads with few write operations: only one write operation may be in progress at a time. For high-contention workloads like TPC-C [42], C-JDBC is unlikely to perform well.

Goldengate [11] provides high performance heterogeneous replication using log shipping. The Goldengate engine reads the database log and converts its entries to a database independent format for transfer. The log entries can then be shipped over the network and applied to another database. Goldengate can extract data from many databases (Oracle, DB2, SQL Server, MySQL, Sybase, etc) and can import into any ODBC compatible database. Since database log format is implementation specific, adding support to Goldengate for additional databases requires significant effort. Sybase [41] appears to use a similar mechanism for heterogeneous replication.

While not heterogeneous replication *per se*, federated databases face a number of similar challenges [35]. A federated database is comprised of many autonomous sub-databases that each contain distinct datasets and may be running software from different vendors. Such systems provide only eventual consistency due to replica autonomy, but they do face challenges of SQL and schema compatibility. Oracle Heterogeneous Connectivity [26] provides a mechanism to run queries against Oracle and other heterogeneous databases and claims to resolve many SQL translation issues.

2.3 Database Repair

Should a database's state become faulty, a repair operation is necessary to rectify the problem. Central to efficient repair is detecting which sections differ between the faulty and correct state. A number of commercial programs [33] for database comparison exist, mostly developed to aid database administrators who are comparing databases by hand to determine what went wrong with their replication systems. The companies do not detail their mechanisms, merely stating that they are "efficient" and "scale to large databases."

When extracting data from transactional systems for storage in data warehouses, the extractor must also detect new or changed data [20, 32]. These mechanisms rely on data structure, triggers, or temporal locality to isolate updates. However, such mechanisms may not efficiently reveal corrupted state.

Database repair bears some resemblance to synchronizing files between file-systems, a task that rsync [43] accomplishes well. Rsync works by computing rolling hashes of file blocks, effectively summarizing file content while isolating which blocks of the file differ. Blocks in files are ordered and named by their offset from the beginning of the file. A block inserted at the beginning of the file changes the offsets of all subsequent blocks, necessitating the rolling hash so data can be matched, regardless of its new position. Database rows are unordered and named by primary key. Inserting a row does not change the primary key of other rows, obviating the need for the rolling hashes used in rsync. An optimization of rsync uses multiple rounds of hashes to further reduce bandwidth consumed [21].

Coelho describes an efficient database comparison algorithm used in PostgreSQL's `pg_comparator` tool [8]. The algorithm works by building a tree of hashes (similar to a Merkle tree [24]), which is then used to isolate rows that differ. We analyze the performance of this algorithm in Chapter 7.

Chapter 3

Architectural Model

The system presented in the following chapters uses a middleware-based architecture. Interposed between the actual clients and the database, middleware acts like a database to the clients and a client to the databases. This chapter starts with the middleware architecture that is at the core of the thesis. Next, we describe the client/database interface, which details both what clients expect to interact with and what operations the middleware has available to it. The final section introduces more carefully the fault model and its consequences, along with a brief survey of faults in databases.

3.1 Middleware Model

In our system, clients do not interact directly with the database replicas. Instead they communicate with a *shepherd*, which acts as a front-end to the replicas and coordinates them. The simplest shepherd design is a single machine, however the shepherd could be replicated for increased fault tolerance. Figure 3-1 shows two possible shepherd system architectures. The architecture requires $2f + 1$ database replicas, where f is the maximum number of simultaneously faulty replicas the system can tolerate. It requires only $2f + 1$ replicas because the database replicas do not carry out agreement; instead they simply execute statements sent to them by the shepherd. The figure illustrates a system in which $f = 1$.

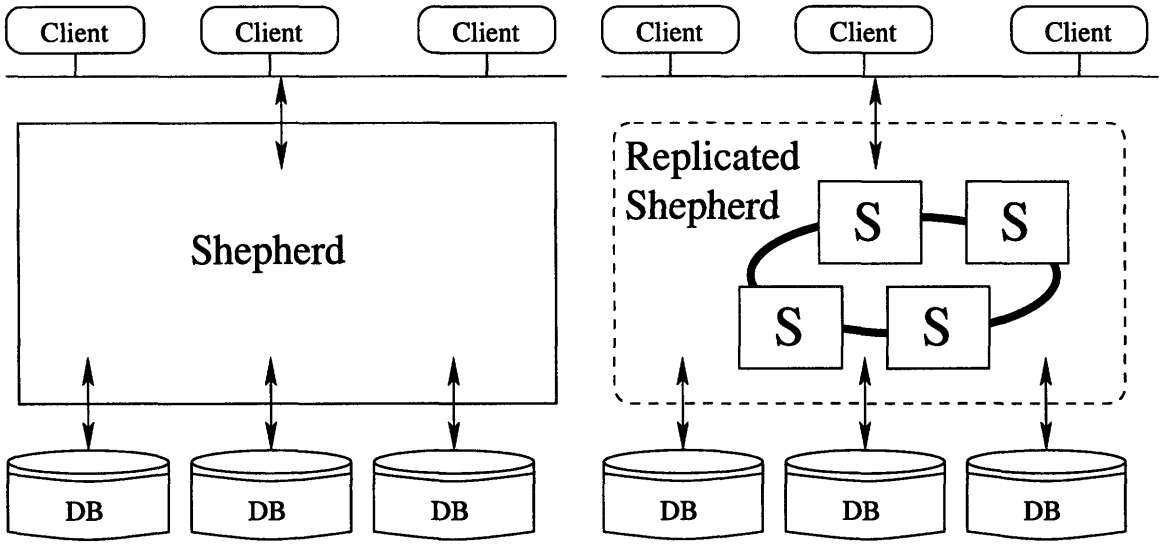


Figure 3-1: Middleware system architecture. Left-hand architecture uses a single shepherd, the right hand, a replicated shepherd.

We assume the shepherd itself is trusted and does not have Byzantine faults, though it might crash (*e.g.*, because the machine it runs on crashes). Since the complexity and amount of code in the shepherd is orders of magnitude smaller than in the replicas, we believe that assuming non-Byzantine behavior is reasonable. Furthermore, we expect that a trusted shepherd is the most likely way the system would be deployed in practice. The shepherd can also be replicated for Byzantine fault tolerance; this is a subject for future work.

We assume that all transactions that run on the replicas are sent via the shepherd.

3.2 Database Interface

Clients interact with a database by connecting to it, likely over a network connection, and issuing a sequence of transactions. A database connection has a number of properties on which clients and databases depend. First, each connection executes transactions serially. Second, clients expect that if two transactions are submitted on the same connection that the later one will see the effects of the previous one. Third, the connection is established with a certain set of credentials, which govern which operations will be allowed by the database. Fourth, some objects are only accessible

from the connection that created them, such as prepared statements, connection variables, and temporary tables. Finally, transactions are tied to the connection that created them: breaking the connection aborts the transaction in progress.

There is a set of common connection protocols, like JDBC or ODBC, that client applications use to communicate with a database. Each database vendor provides a library that implements one (or more) of these protocols and client applications link against and call into the library to interact with the database. All of these libraries provide a blocking interface to the database: control is not returned to the client application until the database operation has completed. Finally, these protocols support authenticated and encrypted connections (typically SSL).

A transaction consists of an ordered list of statements followed by a *commit* or an *abort*. When executing a transaction, the database ensures that the statements are executed in order, with the effects of the previous statements visible to the later ones. A transaction *commits* by atomically and durably updating the state of the database to reflect the operations of the transaction. A transaction that aborts has no effect on the database state. In addition to atomicity and durability, the database also provides consistency and isolation in the presence of concurrent transactions (ACID semantics [14]).

3.2.1 Transaction Isolation Level

While each connection can only submit one transaction at a time, the database may have many clients connected to it that are issuing transactions concurrently. The transaction isolation level determines how the database handles multiple concurrent transactions, particularly when two transactions access the same object.

> Two transactions (or statements) *conflict* when they both access the same object and at least one of the accesses is a write.

Databases provide a number of different levels of isolation between conflicting transactions. The highest level of isolation is SERIALIZABLE, where transactions are executed as if in some serial order. Two execution orders are serially equivalent

if, for each pair of conflicting transactions T_A and T_B, if T_A appears before T_B in one order then T_A appears before T_B in the other order. See Figure 3-2 for a comparison of serial transaction schedules. For all transactions whose order matters (*e.g.*, conflicting transactions), serializable isolation assigns a strict ordering; thus allowing an exterior system such as ours to reason about operation ordering on a per transaction basis instead of a per statement basis.

Transaction Schedule	Equivalent Schedule	Different Schedule
$B \leftarrow A$	$C \leftarrow A$	$\mathbf{A \leftarrow B}$
$C \leftarrow A$	$B \leftarrow A$	$B \leftarrow A$
$\mathbf{A \leftarrow B}$	$\mathbf{A \leftarrow B}$	$C \leftarrow A$

Figure 3-2: Serializable schedules and conflicting transactions. Transaction $\mathbf{A \leftarrow B}$ conflicts with the other two transactions.

Databases typically implement isolation using either optimistic or pessimistic concurrency control. An optimistic scheme keeps track of the read and write sets of a transaction, and at commit time decides whether the isolation property has been upheld, aborting the transaction if isolation is violated.

In a pessimistic system, the database acquires shared or exclusive locks on the data items that it touches. Locking is conservative: databases may lock more items than are actually necessary (*e.g.*, pages instead of rows). Additionally, use of locking can result in deadlocks, forcing the database's deadlock detector to abort a transaction. Two-phase locking is a pessimistic mechanism that guarantees serializability but can lead to cascading aborts so is never used in practice.

Rigorous two-phase locking (R2PL) is a stricter locking mechanism that prevents cascading aborts. In R2PL, the database acquires shared or exclusive locks for each operation and holds them until the commit point. The implication of R2PL is that if a transaction A attempts an operation that conflicts with a concurrently running transaction B, then A's operation will block until B either *commits* or *aborts*. Similarly, if a transaction has successfully executed and gotten answers for a set of statements, but has not yet committed, it holds locks for all of the statements executed so far. R2PL ensures that the order in which transactions commit is the same as the execution order. Since commit order is visible to the client, serializable isolation with rigorous

two-phase locking allows an external system to observe (and enforce) a particular execution order.

3.2.2 SQL

Transaction statements are usually presented in SQL (Structured Query Language), a relatively standard declarative query language that most widely used databases support. A SQL statement declares the objective of the read or write operation, not the manner in which it should be performed. The database's query optimizer and execution engine determine how to implement the query. Even for a given database architecture and table layout, many query plans are possible.

SQL statements fall into several broad categories. While the following list is by no means complete, it does cover the basic SQL statements:

- *Reads* consist of `SELECT` statements.
- *Writes* include `INSERT`, `UPDATE`, and `DELETE` statements.
- *Transaction control* statements manage transactions (`BEGIN`, `COMMIT`, and `ROLLBACK` (*abort*)).
- *Data definition* statements deal with updates to the table schemas (`CREATE`, `DROP`, or `ALTER`).

We support all of the above except for statements that fall into the data definition category. Many database implementations do not respect transactional semantics when executing data definition statements. Thus, we provide no consistency guarantees for workloads that contain them. However, actual client workloads rarely contain data definition statements.

Databases provide two abstraction mechanisms to simplify client SQL: views and stored procedures. Views are virtual tables that result from running a `SELECT` query. Reading from a view is equivalent to reading from the result of the query associated with the view. As such, the results from complicated queries can be simply retrieved

and the complicated queries fully optimized beforehand. Stored procedures are client-authored programs stored in the database and executed using a `CALL` SQL statement. Since the text of the procedure is only sent and compiled once, using stored procedures reduces bandwidth and processing time. Each database implementation typically has its own language for expressing stored procedures. Both views and stored procedures hide implementation details from the SQL. Mechanisms that depend on SQL analysis to determine the effect of SQL statements have trouble dealing with views and stored procedures.

Finally, clients can submit parametrized statements (called *prepared statements*) to the database ahead of time. The database can then parse and do preliminary execution planning for the statement. When the client wishes to execute the prepared statement, it provides values for the parameters. For example, a client inserting many rows into the same table could use a prepared statement to send the `INSERT` SQL to the database only once. Prepared statements are typically tied to the connection that created them. Objects tied to connections can pose problems for middleware that may use many actual database connections to service a single client connection.

3.2.3 SQL Compatibility

Many different relational database implementations exist: Oracle, IBM DB2, Microsoft SQLServer, MySQL, PostgreSQL, Sybase, Apache Derby, etc. While heterogeneity of implementation is key to increased failure independence, differences in interface present a challenge. A number of SQL standards exist (SQL-92, SQL-99, SQL:2003, ...), which specify both syntax and semantic meaning of SQL statements. While all of the above databases claim to support most of SQL-99 [16], they all have their own proprietary extensions to the language to support additional features. In addition to differences in syntax, each database supports a different set of available functions, handles NULL values and dates slightly differently, etc. A summary of some of these incompatibilities is shown in Figure 3-3.

To address this SQL compatibility issue, we present two possibilities: an online solution and an offline alternative. The online solution is for the system to translate

Item	Example
Ordering of NULLs in result sets	Oracle & DB2 consider them higher, MySQL & SQLServer consider them lower.
Limiting result sets	Oracle and DB2 support a standard `ROW_NUMBER` construct, MySQL & SQLServer each have non-standard SQL like `LIMIT n`.
Multi-row inserts	Optional standard implemented by DB2 and MySQL, but not Oracle and SQLServer.
BOOLEAN type	Only Postgres implements it, DB2 suggests CHAR, SQLServer suggest BIT, MySQL aliases BOOLEAN to TINYINT.
CHAR type	MySQL silently truncates over length strings instead of returning errors.
TIMESTAMP type	MySQL's TIMESTAMP is a magic self-updating value; MySQL's DATETIME is like TIMESTAMP standard except it doesn't store fractional seconds.
SELECT without table source	DB2 and Oracle require dummy table (`SYSIBM.SYSDUMMY1` and `DUAL`, respectively)

Figure 3-3: Examples of SQL Compatibility Issues

from client issued SQL into database-native SQL for each database replica to execute. There are existing software packages [39, 40] that accomplish this complex task. To improve performance, the system could also maintain a cache of translated statements to avoid re-translating common statements. A database administrator could then look over the repository of cached translations and optimize them for better performance.

The offline solution is to hide non-standard SQL from the system by burying it in views and stored procedures. Both views and stored procedures are often written by database administrators offline to improve performance. By independently implementing the same view or stored procedure on each heterogeneous database, the implementation details of the operation are removed from the SQL sent by the client application to the system. Of course, using views or stored procedures requires users of the system to implement their queries in several different database systems.

In general, either of these solutions may require some effort by users to port their SQL queries to several different database systems. Some effort may also be required to set up identical tables on different systems and optimize performance via creation of

indices and other optimizations. Our system provides a basic translation mechanism to make writing compatible SQL simpler. We believe that these overheads are not overly onerous, especially given the increased robustness and fault tolerance that the system offers.

3.2.4 Non-determinism

In order to build a replicated system, we require that each query presented by a client be processed identically by every replica. However, some SQL queries can be non-deterministic—different replicas legitimately produce different results for the same operation. While some differences do not alter the meaning of the operation, others do. For reasons of simplicity, we require that all SQL queries be deterministic. The system can assume the burden of ensuring that SQL queries are deterministic by using a translation engine to rewrite queries into an appropriate form. The rest of this section presents potential sources of non-determinism and mechanisms for resolving them.

Since relations are unordered sets, different replicas may end up returning differently ordered responses to the same SQL query. For example, a query asking for people living in Cambridge, MA could return the residents in any order. An `ORDER BY` clause specifies one or more attributes with which to order the results. If the attribute has unique values, the row order is completely specified and deterministic. Thus, adding an `ORDER BY` with a primary key column to a query is both correct and resolves the issue.

One situation where result set ordering is relevant is when the system is comparing `SELECT` query results. From a practical standpoint, a set-compare routine could determine that two differently-ordered results were the same. However, such a routine would likely be less efficient than the database sorting routine. Additionally, if a query has an `ORDER BY` on a non-unique column, then some parts of the ordering are important for correctness, while others are not. For example, a query asking for people living in Massachusetts ordered by town specifies nothing about the ordering of people from the same town. Comparison complexity can be entirely avoided by

adding the appropriate `ORDER BY` clause.

Another potential source of non-determinism is database-assigned row identifiers. The database maintains a counter and assigns a number to each row as it is inserted. The mechanism is often used to generate unique primary key values for rows. In MySQL, it is called `auto_incrementing` columns; in Oracle, `sequences`. While the mechanism is deterministic in the usual case, problems crop up around crashes. The database may skip some values when it recovers, as it is unable to prove that it did not use them already.

The system could duplicate the sequence mechanism and rewrite queries to include the assigned ids. However, bulk inserts done with `INSERT ... SELECT ...` queries do not provide an opportunity for the system to provide ids. For our system, the client is required to provide their own row ids and avoid using the database functionality.

Other sources of non-determinism include non-deterministic functions like `timestamp` and `rand`. Once again, for simple queries the system could produce an output for the function and rewrite the query to use that value. However, more complicated queries are not amenable to rewriting. Handling non-deterministic functions in arbitrary SQL queries is an unsolved problem for all front-end based replication systems. Hence, our system requires that clients not use non-deterministic functions in their SQL queries.

3.2.5 Operating Environment

Databases running in a production environment are heavyweight pieces of software, typically run as the sole application on high performance hardware. The database instance is carefully tuned to take maximal advantage of the hardware. Since the data stored within is valuable, database machines usually reside in a data center with good physical security and power failure resilience. The common database execution environment is reliable and well-provisioned.

While the system could be run over a wide area network, the additional latency would have a significant impact on performance. We expect that the system is typically used to tolerate bugs more than disasters. As such, the database replicas are

likely located in the same data center. The data center provides a well provisioned network connection with high bandwidth and low latency.

3.3 Faults

As described in Chapter 2, previous research in database fault tolerance has focused on *fail-stop* fault tolerance. A fail-stop failure results in the failed replica ceasing participation in the system. A replicated system is parametrized by f, the number of simultaneously faulty replicas it can tolerate while still providing service. In the case of fail-stop failures in a synchronous environment, at least $f+1$ replicas are necessary to survive f failures. An example of this is a primary/failover system, which contains 2 replicas and can survive 1 crash. However, such a system depends on synchrony: the failover mechanism must be able to determine (quickly) that the primary has failed. In an asynchronous environment where timeouts are not a reliable mechanism for determining if a replica is faulty, at least $2f + 1$ replicas are necessary. In such a system operations are performed by a *quorum* of $f + 1$ replicas, and safety requires that every quorum intersect in at least 1 non-faulty node.

However, this thesis focuses on tolerating Byzantine faults in databases. As discussed in Chapter 1, a Byzantine faulty replica may perform arbitrary operations, including but not limited to: returning incorrect answers, lying about what operations it has seen or performed, acting like a non-faulty replica, or colluding with other faulty replicas. Tolerating f Byzantine faults in an asynchronous environment requires at least $3f + 1$ replicas, due to the requirement that each quorum must intersect in at least $f + 1$ non-faulty replicas. Work by Dahlin [47] has shown that using middleware to perform agreement reduces number of replicas needed to execute operations to $2f + 1$.

The Byzantine fault model is fundamentally different from the fail-stop fault model. In the context of databases, a Byzantine-faulty replica could, but is not limited to:

1. Return the incorrect answer to a query

Bug category	DB2 2/03-8/06	Oracle 7/06-11/06	MySQL 8/06-11/06
DBMS crash	**120**	**21**	**60**
Non-crash faults	**131**	**28**	**64**
Incorrect answers	81	24	63
DB corruption	40	4	(inc. above)
Unauth. access	10	unknown	1

Table 3.1: Summary of bugs reported in different systems. In all cases, over 50% of the reported bugs cause non-crash faults resulting in incorrect answers to be returned to the client, database corruption, or unauthorized accesses. Current techniques only handle crash faults. The numbers for the different systems are reports over different time durations, so it is meaningless to compare them across systems.

2. Incorrectly update or delete rows
3. Fail to adhere to transaction isolation requirements
4. Stall a transaction arbitrarily
5. Spuriously abort a transaction claiming deadlock
6. Fail to execute an operation and raise any error
7. Send messages outside the connection protocol specification

We will assume that the library implementations of the connection protocol correctly handle malformed responses. Our system masks and recovers from all of the other types of faults on the list.

Many of the techniques that are used in existing systems for handling fail-stop faults do not transfer to a system that handles Byzantine faults. One of the most common techniques is to have only one replica execute an operation and distribute the results of the operation to all the other replicas. Obviously, if the executing replica is faulty, it could return incorrect answers to be incorporated into the state of correct replicas.

3.3.1 Bugs in Databases

Bugs lead to both fail-stop and Byzantine faults in databases. We performed a simple analysis of bug reports in three database systems: DB2, Oracle and MySQL. Our goal was to understand to what extent reported bugs return incorrect answers to clients

rather than cause crashes. Because it is hard to get access to authoritative information about bugs in practice, we looked at information available on vendor web sites about bug fixes. In all cases, we found thousands of bug reports, a rate consistent with other large software systems.

We divided the bugs into two broad categories: those that caused wrong results to be reported or the database to be corrupted and those that caused a crash. It is possible that crash-inducing bugs also caused wrong results to be reported; we count these as crashes. Table 3.1 summarizes the results.

DB2: We looked at the set of bugs (called "Authorized Program Analysis Reports", or APARs) reported as being fixed in DB2 UDB 8 Fixpaks 1 through 13; these were released between February 2003 and August 2006. There are several thousand such reports—many of the reports are related to tools associated with DB2, or with the optimizer generating inefficient query plans. However, a number of them result in database corruption or incorrect answers that neither the user nor the database system itself would detect. Examples of such bugs include reports titled "incorrect results returned when a query joins on char columns of different lengths and uses like predicate with string constant", "query compiler incorrectly removes a correlated join predicate leading to an incorrect result set", and "in some rare cases, reorg operations may result in corrupted index".

Of the 251 bugs reported, the majority (131) caused non-crash faults.

Oracle: We studied bugs that were fixed between July and November 2006 for an upcoming release (labeled version 11.1 at the time) of the Oracle Server Enterprise Edition by searching the Oracle Bug Database on the Oracle Metalink Server. We did not characterize security bugs, as these are reported separately from the database engine. The number of our sampled bugs that cause wrong results or database corruption (28) exceeds the number that cause crashes (21).

MySQL: We analyzed 90 days of bug reports from MySQL between August 15 and November 13, 2006. Of the 900 issues reported, about 131 of them are verified issues inside the database server. As with DB2 and Oracle, more than half of the

verified MySQL database system engine bugs result in wrong answers.

Because current recovery and replication techniques only handle bugs that immediately cause crashes, they apply to less than half of the bugs seen in this informal study. We also note that our results are consistent with those reported by Gashi *et al.* in a recent and more detailed study [10] of 181 bugs in PostgreSQL 7.0 and 7.2, Oracle 8.0.5, and SQL Server 7. They tested these systems and found that 80 of the reported bugs result in "non-self-evident" faults—*i.e.*, in incorrect answers without crashes or error messages. They also report that for all but five bugs, the bug occurred in only one of the four systems they tested, and in no case did any bug manifest itself in more than two systems.

Chapter 4

Commit Barrier Scheduling

In this chapter we present a novel replication scheme for extracting good performance while tolerating Byzantine faults. Furthermore, the scheme provides strong consistency: client applications do not know they are talking to a replicated system because the shepherd provides a single-copy serializable view of the replicated database. To provide strong consistency, we require that all database replicas use R2PL to provide *serializable* isolation.

Our Byzantine fault tolerant replication scheme must meet the following objectives:

1. Ensure that all non-faulty replicas have equivalent logical state such that they will return the same answer to any given query.

2. Ensure that the client always gets correct answers to queries belonging to transactions that commit, even when up to f replicas are faulty.

3. Detect faulty replicas and flag them for repair.

The first objective can be met by coordinating the replicas to guarantee that they all execute equivalent serial schedules. The naive approach of running transactions one at a time to effect equivalent serial schedules on the replicas has poor performance (see section 6.2.2). At the other extreme, simply sending the transactions concurrently to a set of replicas will usually lead to different serial schedules, because each replica

can pick a different, yet "correct", serial ordering. Our solution to this problem is *commit barrier scheduling (CBS)*. CBS ensures that all non-faulty replicas commit transactions in equivalent serial orders, while at the same time preserving much of the concurrency in the workload.

The second objective can be met by ensuring that each query response sent to the client is attested to by at least $f + 1$ replicas. If $f + 1$ replicas agree on the answer, then at least one non-faulty replica agrees that the answer is correct. We optimize for the common case where database replicas return correct answers: the system can return an unverified answer to the client provided it subsequently forces the transaction to abort if the answer turns out to be incorrect.

The third objective can be difficult to achieve because Byzantine-faulty replicas are hard to detect. Replicas that return answers that do not match the answer agreed upon by $f + 1$ other replicas are obviously faulty. However, a faulty replica can claim to execute an operation correctly, when it actually did something completely different. Additionally, a faulty replica can fail to make progress for a number of different reasons. Our system flags replicas that return incorrect answers and uses a set of heuristics to suspect replicas of other problems.

The chapter presents the basic design of commit barrier scheduling, followed by the details of the protocol. Subsequently, we address how the scheme handles failures (fail-stop and Byzantine) of the replicas and crashes of the shepherd. Following the discussion of failures, we present a brief proof sketch for correctness and liveness. We conclude by discussing the practical side of running CBS, along with a couple of optimizations to the protocol.

4.1 Basic Design

As shown in Figure 4-1, the shepherd runs a single *coordinator* and one *replica manager* for each back-end replica. The coordinator receives statements from clients and forwards them to the replica managers. Replica managers execute statements on their replicas, and send answers back to the coordinator. The coordinator sends re-

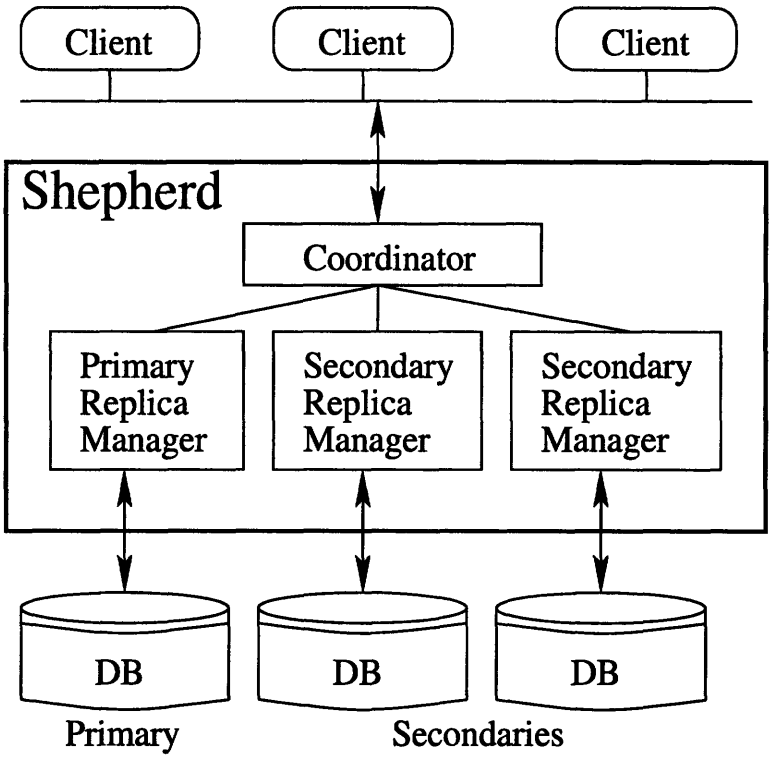


Figure 4-1: Commit Barrier Scheduling Shepherd Architecture.

sults back to the clients, compares query answers for agreement, and determines when it is safe for transactions to commit. The coordinator may also decide to abort and retry transactions or initiate repair of faulty replicas.

The correctness of CBS depends on our assumption that the replicas use R2PL. The pessimistic nature of R2PL guarantees that if a transaction has executed to completion and produced some answers, the shepherd can be confident that the transaction could actually commit with those answers. Under R2PL, a transaction holds locks until the end of the transaction: a transaction that has executed all of its queries holds all the locks it needs to be able to commit. By contrast, an optimistic concurrency control mechanism provides no guarantees about isolation until the transaction actually commits. CBS must be able to compare definitive answers *before* deciding whether to commit the transaction.

In CBS, one replica is designated to be the *primary*, and runs statements of transactions slightly in advance of the other *secondary* replicas. The order in which transactions complete on the primary determines a serial order. CBS ensures that all

the non-faulty secondaries commit transactions in an order equivalent to that at the primary. Furthermore, the shepherd achieves good performance by allowing queries to execute concurrently on the secondaries where it observes the queries execute concurrently on the primary. CBS optimizes for the common case where the primary is not faulty: when the primary is non-faulty the system performs well. A faulty primary can cause performance to degrade but CBS maintains correctness.

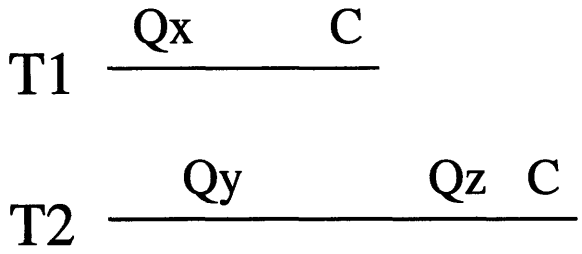


Figure 4-2: Possible transaction execution schedule on the primary as observed by the shepherd. Primary indicates that Q_x and Q_y do not conflict because they both complete without an intervening commit.

A non-faulty primary running R2PL exposes information about concurrency available in the workload, which we can use to achieve good performance. R2PL ensures that a transaction T will hold all read and write locks acquired by any query in T until T commits (or aborts). With two-phase locking, if a transaction T_2 is able to complete a query Q_y after a query Q_x from transaction T_1 has run but before T_1 commits (or aborts), then Q_x and Q_y do not conflict and can be executed in any order. Such a scenario is illustrated in Figure 4-2. Furthermore, Q_y does not conflict with any statement of T_1 before Q_x. The essence can be summed up in the following property:

> If a non-faulty database replica completes executing two statements without an intervening commit, then the two statements do not conflict.

The coordinator uses this property to extract concurrency information by observing the primary execute transactions: if the (non-faulty) primary allows a set of queries to run without conflicting, then the secondaries can run these queries concurrently (*e.g.*, in parallel), or in any order, without yielding a different equivalent serial ordering from the primary.

Obviously, a faulty primary can produce execution schedules in which conflicting statements appear to execute concurrently. However, database implementations use different mechanisms to detect conflicting transactions, which can result in non-faulty database replicas disagreeing about which transactions conflict. For example, if one replica locks individual database rows, while another replica locks whole database pages, the second replica might block when the first one does not. Our approach of using the primary to determine if queries can run in parallel performs well only when the primary's locking mechanism is an accurate predictor of the secondary's mechanism. Differences in mechanism that lead to secondary replicas blocking is *not* a correctness issue, as the protocol works with an arbitrarily faulty primary. For good performance, we require that concurrency control at the replica selected to be the primary is *sufficiently blocking*:

> A replica is *sufficiently blocking* if, whenever it allows two queries to run in parallel, so do all other non-faulty replicas.

For good performance CBS requires $f+1$ sufficiently blocking replicas so that we can freely use any of them as a primary

4.2 Protocol

CBS does not limit concurrency for processing queries at the primary in any way. When the coordinator receives a query from a client, it immediately sends it to the primary replica manager, which forwards it to the primary replica. Hence, the primary replica can process queries from many transactions simultaneously using its internal concurrency control mechanism (R2PL). As soon as it returns a response to a query, the coordinator sends that query to each secondary replica manager. Each of them adds the query to a pool of statements that will eventually be executed at the corresponding secondary.

In addition to sending the query to the secondary replica managers, the coordinator also send the primary's response to the client. Since only the primary has

executed the query at this point, the answer is still unverified and could be incorrect[1]. However, as discussed earlier, incorrect answers sent to the client are resolved at transaction commit. We get performance benefits from returning the primary's answer to the client immediately, as it reduces statement latency as observed by the client. It also allows the processing of transactions to be pipelined: the primary executes the next statement while the secondaries execute the previous one. Sending the primary's response to the client as soon as it arrives also improves overall system performance because transactions complete more quickly and therefore hold database locks for shorter periods.

The coordinator decides the transaction ordering. It greedily commits a transaction and assigns it a space in the global order when (a) the client has issued a COMMIT for the transaction, and (b) $f + 1$ replicas (including the primary) are *ready to commit* it.

> A replica is *ready to commit* transaction T if it has processed all queries of T and committed every transaction the coordinator has committed.

Should two transactions reach this point simultaneously, the coordinator chooses an ordering for them.

The coordinator will allow the commit only if the query results sent to the client are each agreed to by f secondaries that are ready to commit T; otherwise it aborts the transaction. Thus we ensure that the client receives correct answers for all transactions that commit. The coordinator waits for agreement before assigning a transaction a space in the global order to avoid gaps in the order that would result from transactions that had to be aborted because incorrect answers were sent to the client.

The job of the secondary replica manager is to send transaction statements to the secondary, with as much concurrency as possible, while ensuring that the secondary, in the absence of faults, will execute the statements in a serial order equivalent to that selected by the coordinator. To achieve this property, the secondary manager *delays* sending statements to the secondary replica, using three ordering rules:

[1]Section 4.2.2 shows that a faulty primary can return incorrect answers to the client even if the system verifies the answer by waiting for $f + 1$ votes

- *Query-ordering rule.* A query or COMMIT of transaction T can be sent to the secondary only after the secondary has processed all earlier queries of T.

- *Commit-ordering rule.* A COMMIT for transaction T can be sent to a secondary only after the secondary has committed all transactions ordered before T.

- *Transaction-ordering rule.* A query from transaction T_2 that was executed by the primary after the COMMIT of transaction T_1 can be sent to a secondary only after it has processed all queries of T_1.

These rules are the only constraints on delaying statement execution on the secondaries; they permit considerable concurrency at the secondaries.

The first two rules are needed for correctness. The query-ordering rule ensures that each individual transaction is executed properly. The commit-ordering rule ensures that secondaries serialize transactions in the order chosen by the coordinator.

The transaction-ordering rule ensures good performance when the primary is non-faulty because it avoids deadlocks at secondaries. In the example shown in Figure 4-2, query Q_z of T_2 ran at the primary after the primary committed T_1. If a secondary ran Q_z before running query Q_x from T_1, Q_z might acquire a lock needed to run Q_x, thus preventing T_1 from committing at the secondary. The transaction-ordering rule prevents this reordering. However, it allows queries from T_2 that ran before T_1 committed on the primary (Q_y in the example) to be run concurrently with queries from T_1 on the secondaries (such queries are guaranteed not to conflict because R2PL would not have allowed Q_y to complete on the primary if it had any conflicts with Q_x.) Hence, CBS preserves concurrency on the secondaries.

At least f secondaries are likely "current" with the primary: they have committed every transaction the primary has committed. Neither the commit-ordering rule nor the transaction-ordering rule will delay queries from issuing on a "current" secondary. In the common case of a non-faulty primary, a query that completes executing on the primary can be immediately and successfully executed on a non-faulty secondary. Thus, commit barrier scheduling provides good performance in the common case.

4.2.1 Commit Barriers

We implement the commit-ordering and transaction-ordering rules using *commit barriers*. The coordinator maintains a global commit barrier counter, B. When the coordinator gets a response to query Q from the primary, it sets Q's barrier, $Q.b$, to B. The coordinator commits a transaction T by sitting T's barrier, $T.b$, to B, incrementing B, and sending the COMMIT to the replica managers. Each secondary replica manager maintains a barrier for the replica, $R.b$. The secondary replica manager waits to send query Q to the secondary until $Q.b \geq R.b$. The secondary replica manager can commit transaction T when $R.b = T.b$, after which it increments $R.b$. Finally, a secondary replica is *ready to commit* transaction T when it has executed all queries of T and $R.b = B$.

The use of commit barriers is conservative: it may delay a query unnecessarily (*e.g.,* when the query doesn't conflict with the transaction whose COMMIT is about to be processed). It is correct, however, because delaying the running of a query isn't wrong; all it does is cause the processing of that transaction to occur more slowly.

The pseudo-code for the coordinator is given in Figure 4-3. The code is written in an event-driven form for clarity, although our implementation is multi-threaded. The code for the primary replica manager is not shown; this manager is very simple since it sends each statement to its replica as soon as it receives it from the coordinator, and returns the result to the coordinator. Figure 4-4 shows the code for a secondary replica manager.

Figure 4-5 shows an example schedule of three transactions, T1, T2, and T3, as executed by the primary (assumed non-faulty for this example). Each COMMIT ("C") causes the barrier, B, to be incremented. With CBS, the secondary replicas can execute the statements from different transactions in the same barrier (*i.e.,* between two COMMITs) in whatever order they choose; the result will be equivalent to the primary's serial ordering. Of course, two statements from the same transaction must be executed in the order in which they appear. Note that CBS does not extract all available concurrency. For example, a secondary manager delays sending T3's $W(z)$

- Event: Receive query Q from client.
  Action: Send Q to primary replica manager.

- Event: Receive response for query Q from the primary replica manager.
  Actions:
  1. Send the response to the client.
  2. $Q.b \leftarrow B$.
  3. Record response as $Q.ans$.
  4. Send Q to secondary replica managers.

- Event: Receive response from a secondary replica manager for query Q.
  Action: Add response to $votes(Q)$.

- Event: Receive ABORT from client.
  Actions:
  1. Send ABORT to replica managers.
  2. Send acknowledgment to client.

- Event: Receive COMMIT for transaction T from client.
  Actions: Delay processing the COMMIT until $f+1$ replicas are ready to commit T. Then:
  1. If the response $Q.ans$ sent to the client for some query Q in T is not backed up by f votes in $votes(Q)$ from replicas that are ready to commit T, send ABORT to the replica managers and inform the client of the ABORT.
  2. Otherwise:
     (a) $T.b \leftarrow B; B \leftarrow B + 1$.
     (b) Send acknowledgment to client.
     (c) Send COMMIT to replica managers.

Figure 4-3: Pseudo-code for the coordinator.

- For each query Q in the pool, determine whether it is ready as follows:
  1. All earlier queries from Q's transaction have completed processing.
  2. $Q.b \geq R.b$

  Execute each query Q that is ready on the replica and send the result to the coordinator.

- The replica manager can issue a COMMIT of transaction T to the replica if all queries of T have completed processing at the secondary replica and $R.b = T.b$. When the COMMIT completes processing, increment $R.b$.

- For each ABORT statement for a transaction T in the pool, discard from the pool any queries of T that have not yet been sent to the replica and send the ABORT to the replica.

Figure 4-4: Pseudo-code for secondary managers.

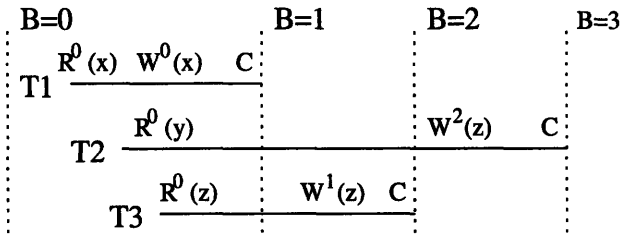


Figure 4-5: Example schedule of three transactions, as executed by the primary. Note that the coordinator does not know the identities of x, y, and z (or even that they are distinct). Each query is super-scripted with the barrier CBS would assign to it.

to the secondary until after the secondary has committed T1. This is because we assume, conservatively, that T3's $W(z)$ *might* conflict with queries of T1. The same rule prevents sending T2's $W(z)$ before T3's $W(z)$ has been processed and in this case the delay is needed since there is a conflict.

4.2.2 Handling a Faulty Primary

The pseudo-code in Figures 4-3 and 4-4 provides correct behavior even when the primary is Byzantine faulty. A primary that fails by returning a wrong answer is handled by detecting that the client received an incorrect answer at the transaction commit point and aborting the transaction. However a faulty primary might have a concurrency control bug, so that it allows transactions that conflict to run in parallel.

A concurrency error is most likely to lead to a liveness problem, where transactions are unable to make progress because they block at secondaries. Some additional mechanisms are necessary to ensure forward progress.

Ensuring Progress

There are a number of ways a Byzantine faulty primary can attempt to prevent the system from making progress. The simplest ones involve executing transactions very slowly or spuriously signaling deadlocks or other error conditions. All of these occurrences arise during non-faulty execution; the faulty primary merely adjusts their frequency. If the primary aborts a transaction due to deadlock or error, it provides no ordering information about how to execute the transaction relative to the rest of the workload. Thus, doing anything but aborting the transaction on the secondaries may result in replicas blocked and unable to make progress. The primary replica manager must use historical data or heuristics for gaging when a primary is obstructing progress.

However, the most interesting failures are those that involve incorrectly serializing transactions. The primary can claim to have executed two conflicting statements or transactions concurrently, thus indicating to the coordinator that they do not conflict. Each correct secondary will then select one of the conflicting items to execute and one to block. Two possible scenarios arise: there are enough matching responses from secondaries to commit a transaction, or there aren't. In the first case, the system overall continues to make progress, but some secondaries may have executed (but not committed) transactions in an order that is not consistent with the coordinator. In the second, the entire system can deadlock.

Secondaries may find themselves unable to make progress due to executing transactions in an order that differs from the order selected by the coordinator. As shown in Figure 4-6, transaction T_1 waits on transaction T_2 in the database, while T_2 waits on T_1 in the replica manager. This situation can arise due to faults or an insufficiently blocking primary. The secondary must rollback T_2, allowing T_1 to proceed. CBS uses a timeout to detect stalled secondary replicas, and rolls back transactions

- Transactions T_1 and T_2 both consist of a single query, $W(x)$, and a commit.
- The Coordinator commits transaction T_1 followed by T_2.
- Secondary Replica S1 does the following:
  1. issues T_1's $W(x)$ to the database
  2. issues T_2's $W(x)$ to the database
  3. receives the result from T_2's $W(x)$, finishing the execution of T_2
- Replica S1 gets stuck:
  - S1 cannot commit T_1 because S1 hasn't finished executing T_1 due to T_2 holding a lock on x that T_1 needs. The database's concurrency control mechanism blocks T_1 from executing.
  - S1 cannot commit T_2 because S1 must commit T_1 before it commits T_2. The *Commit ordering rule* blocks T_2 from committing.

Figure 4-6: Secondary replica gets stuck because it executes transactions in an order that differs from the coordinator's ordering.

on the replica to allow it to make progress.

However, there may be many concurrently executing transactions and the secondary replica manager cannot easily determine which transaction to abort to allow T_1 to proceed. The simple approach is to abort all transactions other than the one which the replica is supposed to commit next. A more gradual back off would result in fewer aborted transactions but higher delays before finding the appropriate transaction to abort. Transactions aborted in this manner can be safety re-executed once the replica has completed executing the previously blocked transaction.

The case where the whole system deadlocks only arises when $f > 1$, and is illustrated in Table 4.1. A faulty primary indicates that conflicting transactions T_1, T_2, and T_3 do not conflict. Each of the first 3 secondaries selects a different transaction to execute and blocks the others. The fourth secondary has crashed and does not return any answers. Each transaction lacks $f + 1 = 3$ matching votes and thus none of them can commit. Note that while none of these transactions will make progress, the system as a whole can continue processing transactions that do not conflict with any of the deadlocked transactions. Since the dependency cycle is in the coordinator, the database level deadlock detector does not help. Nor does the coordinator have

Transaction	Status by Replica				
	Primary *faulty*	S1	S2	S3	S4 *crashed*
T1	exec	exec	block	block	
T2	exec	block	exec	block	
T3	exec	block	block	exec	

Table 4.1: System with $f = 2$ where a faulty primary causes a system-wide deadlock: no transaction is executed by $f + 1$ replicas

information about the dependencies of the various transactions; thus it is limited to using timeouts to detect deadlocks of this nature.

The symptom of this condition is that a set of transactions complete executing on the primary and some secondaries, but never commit, despite the commit being received from the client. There are two possible locations to detect and resolve the problem: the coordinator or the secondary replica managers. The coordinator could use a global timer to detect when some transactions appear stuck and can instruct all the secondary replica managers to abort the same transaction. Alternatively, the secondary replica managers could timeout locally and abort transactions. While this raises the possibility of livelock (each secondary aborts different transactions and the system blocks again), it does not require global coordination and keeps all timeouts local to the secondary replica managers. With a single shepherd, a global timeout is the simplest mechanism; we use a global timeout to abort stuck transactions.

The introduction of deadlock resolution mechanisms ensures that individual replicas and the system as a whole continue to make progress in the presence of a faulty primary. Since the mechanisms require timeouts, aborting, and re-executing transactions, performance suffers when a primary behaves badly. The impact on performance is dependent on the timeout lengths, which are in turn dependent on the workload. Since the system essentially drops to sequential execution after a timeout, the performance is worse than sequential execution. As mentioned previously, we assume that the faulty or insufficiently blocking primary necessary to arrive at the situation is the uncommon case, and therefore poor performance is acceptable when handling the scenario.

Faulty primary	Replica R1	Replica R2
$T_1 : A = 1$	$T_1 : A = 1$	T_1: Waiting for T_2
$T_2 : A = 1$	T_2: Waiting for T_1	$T_2 : A = 1$

Table 4.2: A faulty primary's reply could be incorrect, yet match a non-faulty secondary's reply and be the majority. The variable A is initially 0 at all replicas, and each transaction increments the value of A by 1.

Ensuring Correctness

Not all concurrency faults of the primary lead to liveness issues. The example in Table 4.2 illustrates the point. Consider two transactions, T_1 and T_2, each of which increments a variable A and returns its value. If A were 0 to start with, then the correct value after executing both transactions should be 2. However if the primary is faulty the following might occur: The primary runs T_1 and T_2 concurrently, incorrectly returning $A = 1$ as the result of both increment statements. Each secondary is correct; however, replica R1 runs T_1 before T_2, acquiring the lock for A and blocking T_2. Similarly, replica R2 runs T_2 before T_1, blocking T_1. Furthermore, replica R1 endorses the primary's response for T_1 and replica R2 endorses the primary's response for T_2. Thus waiting for $f + 1$ replicas to agree on an answer before sending it to the client does not ensure that the answer is correct!

The key realization from Table 4.2 is that each replica is voting on transaction answers given some serial schedule. Non-faulty replicas R1 and R2 disagree about the serial schedule, which is why they arrive at mutually inconsistent answers for the two transactions. Correct behavior requires committing one transaction and aborting the other, as the primary has sent two mutually inconsistent results to the client: one must be incorrect. If T_1 is first in the serial order, then the answer for T_2 is incorrect, and vice versa.

Our protocol ensures correct behavior because it requires matching query responses from replicas that are *ready to commit* the transaction. Recall that being *ready to commit* transaction T requires that the replica has committed all transactions the coordinator has committed. If the coordinator decides to commit T_1 first, it issues commits to the primary and R1. The coordinator cannot then commit T_2

because R2's vote for T_2 does not count because it isn't *ready to commit* T_2 (it has not also committed T_1). When R1 executes T_2 it produces the answer 2, which doesn't match the primary's answer. Thus, T_2 is prevented from committing.

T_2 will eventually get aborted when R2 times out and rolls back T_2. At this point, it can execute and commit T_1. When R2 re-executes T2 it produces a different answer than it did previously, namely 2. The coordinator considers only the last vote provided by the replica in deciding whether a transaction should commit or abort. The coordinator now has $f+1$ matching votes from replicas that are ready to commit T_2, and since the answer does not match the one sent to the client, the coordinator aborts the transaction.

4.2.3 View Changes

If the primary is faulty, CBS may be unable to make progress efficiently. The way to handle a faulty primary is to do a *view change* in which a new primary is selected by the coordinator and the old one is demoted to being a secondary. While any replica can be selected as the new primary, the best performance results from selecting one that is sufficiently blocking. A correct view change must ensure that all transactions that committed prior to the view change continue to be committed afterwords. This condition is easy to satisfy: retain these transactions and abort all the others.

It is always safe to do a view change, but view changes are undesirable since they can cause transactions to abort and interfere with the system making progress. The coordinator tries to decide when to do view changes intelligently but it doesn't always know when the primary is faulty. Clearly it knows the primary is faulty if it must abort a transaction due to incorrect results sent to a client. But if the primary doesn't respond, it might not be faulty; instead there might just be a network problem. Additionally, it may not be obvious which replica is faulty; for example, if the primary allows two statements to run in parallel, and a secondary blocks on one of them, this could mean that the secondary is faulty, or it could mean that the primary is faulty! Instead, the coordinator relies on heuristics: timers to limit waiting for the primary to respond, or counts of the number of times secondaries are blocking.

These heuristics need to ensure forward progress by guaranteeing the system will eventually choose a good replica as primary and will give it sufficient time to do its work. We use techniques similar to those that control view changes in BFT [5], *e.g.*, choosing a new primary in a round-robin fashion from all possible primaries and, when a series of view changes happen one after another, increasing timeouts exponentially between changes.

4.3 Fault Recovery

We now discuss how our system deals with recovery of failed nodes, *i.e.*, how it brings them back into a state where they can correctly participate in the protocol. We begin by discussing recovery of replicas that have suffered a crash failure. Then we discuss recovery of the shepherd. Obviously, some mechanism is needed to detect and repair a Byzantine-faulty replica. We defer the discussion of the techniques required to restore the state of a Byzantine-faulty replica to Chapter 7.

4.3.1 Recovery of a Crashed Replica

When a replica goes offline and then recovers, it rejoins the system in a consistent, but stale, state: all transactions that it did not commit prior to its failure have aborted locally. To become up-to-date, the replica must re-run any transactions that it aborted, but that were committed or are in progress in the system. All statements needed to recover the replica are in the pool of its manager.

A replica manager considers a replica to have crashed when the database connection to the replica breaks. If the manager had sent a COMMIT to the replica prior to learning that the connection had broken, but did not receive a reply, it cannot know whether that COMMIT had been processed before the replica went down. Furthermore, it cannot re-run the transaction if it already committed.

To allow the replica manager to determine what happened, we add a transaction log table to each replica; the table includes a row for each committed transaction, containing the commit barrier for that transaction ($T.b$). We also add a query to

the end of each transaction to insert such a row in the table. If the transaction committed prior to the failure, the transaction log table will contain an entry for it. When the connection is re-established, the replica manager reads the transaction log table, compares the list of committed transactions with information maintained on the shepherd, and replays any missing committed transactions. After all the committed transactions have been processed, the manager starts running statements of in-progress transactions.

Adding this extra statement to each transaction will not result in deadlocks because each transaction only touches the log table on its last statement, after the coordinator has cleared it to commit. Also, the table need not be large. We can truncate a replica's log table periodically by determining the barrier, $T.b$, of the oldest transaction, T, not committed at this replica, and storing this information on the shepherd. Then we can overwrite the log table on that replica to contain only entries from $T.b$ on.

Of course, the database connection between the replica manager and a replica can break even when the replica has not failed. When such a network failure occurs, both parties will detect the failure (and each will assume that the other has failed). The replica will simply abort all pending transactions. When the shepherd re-initiates the connection to the replica after the failure heals, it re-runs transactions that have not yet committed, using information in the log table to avoid running transactions more than once. Hence, there is essentially no difference between how the shepherd handles a failure of a replica and how it handles a failure of the network between it and the replica.

Note finally that we can be smart about how we run read-only operations. Slow replicas, including those that recover from crashes, need not execute read-only transactions and read-only queries. By executing only state-changing queries, we can bring slow replicas up to date more quickly. Additionally, read-only transactions need not write an entry into the transaction log table, which also allows them to remain read-only from the database viewpoint as well.

4.3.2 Recovery from a Shepherd Crash

To survive crashes the shepherd maintains a write-ahead log. When the coordinator determines that it can commit a transaction, it writes the transaction queries (and their barriers), along with the COMMIT to the log. The coordinator forces the log to disk before replying to the client. The log is also forced to disk after writing a replica's transaction log table information prior to truncating the table. The cost of flushing the coordinator's log to disk can be amortized with group commit, as described in Section 4.6.2.

To recover from a crash, the shepherd reads the log, identifies all committed transactions, and initializes the statement pools at the managers to contain the statements of these transactions. The coordinator knows which transactions to include in a replica manager's pool by examining the replica's transaction log table, just as if the replica had crashed. The shepherd can start accepting new client transactions when the replica selected as primary has completed executing all the statements in its pool.

The shepherd's log can be truncated by removing information about transactions that have committed at all the replicas. We can expect that most of the time all replicas are running properly and therefore the log need not be very large.

4.4 Correctness

In this section we present an informal discussion of the safety and liveness of our system. We assume here that no more than f replicas are faulty simultaneously.

4.4.1 Safety

CBS is safe because it guarantees that correct replicas have equivalent logical state, and that clients always get correct answers to transactions that commit.

Ensuring equivalent logical state depends on the following assumption about databases: if correct replicas start in the same state and execute the same set of

transactions in equivalent serial orders, they will end up in equivalent states and will produce identical answers to all queries in those transactions. This condition is satisfied by databases that ensure serializability via R2PL. The job of our system is to present transactions requested by clients to the databases in a way that ensures they will produce equivalent serial orders. The query-ordering rule ensures that each transaction runs at each replica as specified by the client; thus the transactions seen by the replicas are identical.

The commit-ordering rule ensures that COMMITs are sent to the replicas in a way that causes them to select equivalent serial orders. The coordinator assigns each committed transaction a place in the global serial order. Per the commit-ordering rule, each replica commits transactions sequentially in the order specified by the coordinator. Since commit order is equivalent to serial order, all replicas implement equivalent serial orders. Thus we can be sure that correct replicas will have equivalent states after they execute the same set of transactions.

Additionally, clients receive only correct results for queries of transactions that commit because we vote on the answers: $f + 1$ replicas must agree to each query response. The vote ensures that at least one correct replica agrees with the result, which implies that the result is correct. However, it is crucial that we delay the vote until each contributing replica is ready to commit the transaction, since only at that point can we be sure it is producing the answer that happens by executing that query at that place in the serial order. R2PL ensures that, barring faults, a transaction that is ready to commit can be successfully committed with the answers that it has produced.

We continue to provide correct behavior even in the face of the various recovery techniques (recovery of a crashed replica, recovery from a database disconnection, recovery of the shepherd). The shepherd records complete information about committed transactions, allowing it to recover from replica crashes and network disconnections; the log tables at the replicas are crucial to this recovery since they prevent transactions from executing multiple times. The write-ahead log at the shepherd ensures that the information about committed transactions isn't lost if the shepherd crashes.

View changes do not interfere with correct behavior because we retain information about committed transactions.

Finally note that the transaction-ordering rule is not needed for correctness. Instead, this rule is important for performance, because it ensures (assuming the primary is non-faulty and sufficiently blocking) that the secondaries execute queries in an order that avoids spurious deadlocks.

4.4.2 Liveness

Given a non-faulty primary, CBS is live assuming that messages are delivered eventually and correct replicas eventually process all messages they receive.

However, the primary may be faulty. We handle this case through the view change mechanism, which allows us to switch from a faulty primary to a non-faulty primary. We rotate the primary in a round-robin fashion among the secondaries, so as to prevent an adversary from forcing the primary to switch back and forth between faulty replicas.

The problem with view changes is that there are cases where we can't be sure that the primary is faulty, and yet we do the view change anyway, *e.g.*, when the primary is merely slow to respond or secondaries are incorrectly ordering transactions. An adversary could cause us to do a view change in which a non-faulty primary is replaced by a faulty replica. However, the adversary cannot cause us to continuously do view changes, without making progress in between, because we exponentially increase the timeouts that govern when the next view change happens, and we select primaries in a way that will eventually lead to a non-faulty one. Eventually these timeouts will become large enough so that a non-faulty primary will be able to make progress, assuming that network delays cannot increase exponentially forever. Under this assumption (used for BFT [5]), our system is live.

4.5 Practical Issues Running CBS

With CBS, clients must be prepared to occasionally receive wrong answers for transactions that abort. Thus, all answers in CBS must be considered "tentative" until the commit point, after which they are guaranteed to be correct (assuming no more than f faulty replicas). Our semantics will not cause a problem for clients that work in "auto commit" mode (COMMIT sent implicitly after each query). Our semantics will also not be a problem for clients that respect transactional semantics and do not let values produced by a transaction "escape" to the application prior to commit — most database applications are programmed in this way. Finally, note that even if bad results did escape, the situation is still better than it is today without CBS, because CBS will abort the transaction and ensure that the replicas have the correct logical state, whereas in existing systems, wrong answers are simply returned without any warning.

Should the shepherd crash with an outstanding commit from a client, the client is not only unaware of whether the transaction committed, the client also does not know if the answers it received are correct. For read-only transactions, the client could submit the transaction again. For read/write transactions, the client could submit a new transaction to confirm whether the changes from the earlier transaction were made. The shepherd could also provide a mechanism whereby clients could query the status of transactions.

The operator of the system can use a light-weight view change mechanism to switch the primary. For example, when the operator schedules downtime for the replica acting as the primary, the light-weight view change can avoid interruption in service. A secondary that has completed executing every statement the primary has executed agrees with all ordering decisions made by the primary. The secondary can seamlessly be promoted to be the primary without aborting any transactions. To produce the situation, the coordinator merely needs to pause sending queries to the primary and wait for a secondary to catch up. Obviously, because the primary could be Byzantine faulty, there may be no secondary that is able to execute all statements

in its pool. In this case, the coordinator times out waiting for a secondary to become eligible. Then it reverts to the heavy-weight view change mechanism: it aborts all currently executing uncommitted transactions, and promotes a secondary that has committed all previous transactions.

Setting timeouts and detecting a more subtly faulty primary both require knowledge of the system workload. Information about typical query execution time, deadlock probability, malformed transaction submission rate, etc. can be collected by the shepherd over time. Such information can be sanity checked by the system administrator. When the system performance deviates from the expected norm, the system can notify the administrator and take preventative measures (like proactively switching the primary).

4.6 Optimizations

4.6.1 Bandwidth Reduction

Having each replica return the entire query result can be expensive when the result is large. A way to reduce this cost is to have secondaries return a hash of the result instead. For example, the coordinator could rewrite the query for secondary replicas to nest the query statement in an aggregate that computes a hash over the entire result set. Most databases implement a standard hash function, like MD5, which could be used for this purpose. This optimization would significantly reduce the bandwidth requirements at the cost of increased computation at the database replicas.

4.6.2 Group Commit

When a database commits a transaction that modified the database state, it must perform a disk write to ensure that the transaction is durable. Databases amortize the cost of this slow disk access by committing many transactions simultaneously; this is called group commit. So far we have processed transaction commits serially, which prevents the database replicas from applying group commit.

To support group commit on the database replicas, we must ensure that:

1. The coordinator can agree to commit a transaction before the previous transaction has completed committing on the replicas.

2. Any transactions for which the coordinator does this do not conflict.

3. Database replicas can issue commits concurrently where the coordinator does.

We can achieve the first and second objective by relaxing the definition of *ready to commit* as follows:

> A replica is *ready to commit* transaction T if it has processed all queries of T and all queries of all transactions the coordinator has committed.

Suppose two non-conflicting transactions, T_1 and T_2, complete executing simultaneously on $f + 1$ non-faulty replicas. If the coordinator commits T_1, it can then immediately commit T_2, as $f + 1$ replicas are still *ready to commit* T_2: they have executed all queries of T_1. Thus, the coordinator can commit two simultaneous non-conflicting transactions. The coordinator cannot commit two simultaneous conflicting transactions, as $f + 1$ replicas running R2PL cannot claim to have executed both to completion.

A modification to the commit-ordering rule accomplishes the third objective:

> *Commit-ordering rule.* A COMMIT for transaction T can be sent to a secondary only after the secondary has processed *all queries* of transactions ordered before T.

A non-faulty replica cannot have executed two conflicting transactions to completion because it uses R2PL, and thus the secondary replica manager never issues concurrent COMMITs for conflicting transactions.

Faults

Faults complicate group commit because the coordinator is agreeing to commit transactions before receiving acknowledgments that prior transactions have successfully

committed. There exists a race condition where a replica aborts a transaction that has executed to completion, and completes executing another conflicting transaction before the shepherd hears about the transaction abort. The race condition leads to the bad scenario shown in Table 4.3. Transactions T_1 and T_2 conflict and a faulty primary claims they don't. A good secondary replica executes T_1 to completion, then aborts T_1 and executes T_2 to completion before the coordinator notices that it aborted T_1. The coordinator would then believe these two transactions do not conflict and could issue a group commit for them. At this point, the system has failed because good secondary 1 can commit the transactions in a different order than good secondary 2.

Transaction	Status by Replica		
	Primary *faulty*	S1	S2 *slow*
T1	exec	exec, then abort	
T2	exec	exec	

Table 4.3: Race condition in group commit

A non-faulty secondary would not randomly abort a transaction, but our communication protocol allows a malicious network to do so. When a connection between the shepherd and the database breaks, the database aborts any transaction from the connection that may have been in progress. Normally, database connections break only when the shepherd or the database fails. However, a malicious network can cause a network connection to break by filtering packets or forging a TCP reset packet. Both of these methods work even when the connection content and authenticity is protected by a protocol like SSL. Thus, an adversary in the network can cause any single transaction to abort whenever it desires. Key to the fault in Table 4.3 is that only one of the transactions aborts; the rest are left undisturbed. Therefore, a malicious network can cause the race condition on S1 without S1 being faulty. Even worse, by causing a similar event to happen on the primary, the primary need not be faulty to appear to concurrently execute two conflicting transactions to completion.

One resolution is to modify the system assumptions to tolerate only Byzantine faults in the databases themselves; not faults in the network. For some operating

environments (*e.g.*, data centers), this may be a reasonable assumption.

The alternative is to place a shim on the databases that makes transaction abort explicit. The shim communicates with the database via a reliable channel (*e.g.*, a local socket), such that a failure of the channel constitutes a failure of the replica. Breaking the connection between the replica manager and the shim has no effect on system state. Should the shepherd crash, it must send an explicit abort message to replicas when it restarts. Since these abort messages are actually protected by SSL, an adversarial network cannot forge them. Similarly, should the replica crash and recover, it sends a message informing the shepherd of the crash. The message must be acknowledged before the shim will process any new operations from the shepherd.

Protocol

Assuming one of the two resolutions of the race condition detailed above, the CBS protocol must be altered slightly to allow group commit. The simple solution is to increment the replica's barrier, $R.b$, when the commit for a transaction issues instead of when it completes. After $f + 1$ replicas have issued the commit for a transaction, the coordinator can agree that the next transaction is allowed to commit.

Group commit and the transaction log table may interact oddly in the presence of crash failures because a replica's log table may contain gaps when the replica crashes with an outstanding commit for a transaction that did not conflict with subsequent transactions (thus allowing them to commit before it). When the replica comes back up, it is safe to re-execute these gap-transactions because they could not have conflicted with subsequently committed transactions. However, the gaps must be filled before transactions with barrier numbers higher than the maximum barrier in the log table may be executed.

Coordinator Group Commit

Once the coordinator can agree to commit a transaction before the previous transaction has completed committing on the replicas, then the coordinator itself can implement group commit to reduce synchronization and disk write overheads. When

the coordinator receives a commit for a transaction from the client, it places the transaction on the *ready-list*. Whenever the coordinator receives an event from a replica manager that might allow a transaction to commit, it scans the ready list, attempting to commit transactions. Once it has successfully committed a transaction off the ready list, it restricts which replicas are allowed to participate in the agreement process during the rest of the scan. Specifically, only replicas that have agreed with all transactions previously committed during the scan may participate in agreement to commit the next. At the end of the scan, the coordinator can write records for all the committed transactions to disk, increment the global barrier B by the number of transactions committed, send acknowledgments to the clients, and issue commits to the replicas.

4.6.3 Read-only Transactions

CBS is important when most of the transactions modify the database state, but workloads typically contain read-only transactions. Obviously an entirely read-only workload requires no concurrency management because no transactions conflict. If the workload contains read-only transactions, there are a number of optimizations to CBS that can improve performance. Note that the client communication protocol can indicate to the shepherd that a transaction will be read-only.

The first optimization is to ensure CBS never blocks a read-only transaction from *committing* due to another read-only transaction. You might think that CBS could block a read-only transaction from *executing* due to another read-only transaction, but this almost never affects performance. CBS never blocks "current" replicas (*i.e.*, those whose barrier is identical to the coordinator's) and reads are not executed on replicas that are behind. However, two read-only transaction are considered sequentially when the coordinator is deciding if they are able to commit. We can relax this restriction to reduce the latency associated with read-only transaction commit. To accomplish this, we split the barrier into two parts: a write part and a read sub-part. Two parts of the protocol change:

- When the *Coordinator* or a *Replica Manager* commits a transaction, instead of executing `B = B + 1`, it does one of two things:
  - Read-only Transaction: `B.R = B.R + 1`
  - Read/Write Transaction: `B.W = B.W + 1; B.R = 0`
- When the *Coordinator* is determining whether a replica is up-to-date, instead of checking that $R.b = B$, it uses one of two criteria:
  - Read-only Transaction: $R.b.W = B.W$
  - Read/Write Transaction: $R.b.W = B.W \ \& \ R.b.R \leq B.R$

The commit of each read/write transaction effectively starts a new "epoch" of read-only transactions. The two-tiered scheme is necessary to ensure that commits from read/write transactions wait for specific read-only transactions to complete. For example, suppose read/write transaction A conflicts with read-only transaction B, but not read-only transaction C. Transactions B and C should be able to commit concurrently, but transaction A should wait for B to complete before committing.

CBS still incurs a latency penalty due to first executing the statements on the primary before issuing them to the secondaries. The penalty is most obvious for workloads with single-statement read-only transactions or long-running read-only queries. In a read-only heavy workload, the primary isn't needed to schedule transactions except when read/write transactions arrive. One further optimization that we have not implemented would be to send transactions to all replicas simultaneously when the workload is entirely read-only. When a read/write transaction arrives, the shepherd would cease issuing new statements to replicas until the primary completes executing all statements. At this point, the shepherd would revert to CBS until the workload becomes entirely read-only again.

4.6.4 Early Primary Commit

Suppose the primary committed transactions as soon as it completed executing them. Obviously, the primary's state would diverge if the coordinator subsequently aborted

a transaction that the primary had already committed. However, the only time the coordinator is required to abort a transaction that successfully executed on the primary is when incorrect results got sent to the client. This only occurs when the primary is faulty. If the primary is faulty, we've only made it slightly more faulty.

From a performance standpoint, early commit increases available parallelism. Time spent waiting for agreement is extra time that transactions hold locks on the primary. Since the primary is the arbiter of which transactions may run in parallel, extending transaction duration reduces parallelism. By committing transactions before waiting for agreement, the primary makes locks available for acquisition by subsequent conflicting transactions, effectively overlapping agreement with execution of subsequent transactions. For high contention workloads, this could result in significant improvement in performance.

The change to the commit barrier scheduling protocol comes in two parts: runtime and recovery. During runtime, statements are no longer marked with the coordinator's barrier B, as the primary may be ahead of the coordinator. Instead, statements are first annotated with the id of the latest transaction to commit on the primary. The statements cannot be immediately annotated with the barrier of the transaction because the coordinator has not necessarily assigned it yet. When a transaction commits on the coordinator, all statements marked with its id are updated with the barrier the coordinator just assigned to the committing transaction. No secondary may issue a statement which does not have a barrier annotation, as the secondary cannot have committed a transaction before the coordinator.

When considering recovery, the commit point for a transaction becomes important. If the coordinator crashes while the primary is ahead, when the coordinator recovers it must attempt to commit all the transactions the primary had already committed. Thus, when the commit is sent to the primary becomes the conditional commit point. It is conditional because the transaction only actually commits if at least f secondaries agree with the answers. The coordinator must write a log entry with the transaction, the answers sent to the client, and a tentative barrier value to its write-ahead log before issuing the commit to the primary. If the coordinator crashes, it recovers by

reading the log and attempting to execute the conditional transaction queries on the secondaries. If the coordinator acquires f answers that match the answer sent to the client for each query in the transaction, it can declare the transaction fully committed and write a log entry with its actual barrier value.

One final piece of complexity is how the coordinator handles crashes with a commit outstanding on the primary. In this situation, when the coordinator recovers, it does not know whether the transaction actually committed on the primary. The previously mentioned solution of writing transaction barrier values to a `xactions` table does not work because the coordinator has not necessarily assigned the transaction a barrier yet. Instead, the primary replica manager inserts a statement to write the transaction's id to the `xactions` table. If the id is present in the coordinator's log and in the primary's `xactions` table, then the coordinator must execute it on the secondaries. If the transaction's id is missing from the table but in the log, then the transaction may be safely discarded.

The overhead of adding an additional disk write to the commit process may outweigh the benefit of early commit. However, if the shepherd is replicated, then the trade off becomes a fast agreement round between lightweight shepherd replicas versus a slow agreement round between heavyweight database replicas. In this situation, the benefit of the optimization becomes clear.

Chapter 5

Snapshot Epoch Scheduling

A number of database systems provide a type of multi-version concurrency control called *Snapshot Isolation*. Under this concurrency control mechanism, strict serializability is relaxed to ensure that readers never block for writers. Snapshot isolation allows for better database performance while providing equivalent isolation for common benchmarks (*e.g.,* TPC-C). However, snapshot isolation provides fundamentally different consistency guarantees than serializability: there exist schedules of transaction operations that are valid under snapshot isolation but invalid under serializability and vice versa. Single-copy serializability is not achievable in a practical manner from a set of snapshot isolation replicas.

HRDB uses Commit Barrier Scheduling (CBS) to provide a single-copy serializable view of a set of databases that support serializable isolation. Since snapshot isolation differs from serializability, the specific technique used in CBS does not apply. However, single-copy *snapshot isolation* [45] is possible, and the general character of the commit barrier idea is transferable. The key insight is that, while more optimistic, snapshot isolation still provides the necessary guarantee: *Any transaction that has successfully executed all of its operations must have acquired all the resources it needs to commit.* Unlike CBS however, providing single-copy snapshot isolation requires modifications to the database replicas in order to handle faults.

In this chapter, we first present the details of how snapshot isolation works, then proceed to address the three challenges of replication with snapshot isolation: keeping

replicas synchronized, resolving conflicting transactions, and handling failures. Next, we present our solution, which we call Snapshot Epoch Scheduling (SES), and give an informal sketch of its correctness. We conclude with descriptions of a number of optimizations to the basic protocol.

5.1 Snapshot Isolation

In snapshot isolation, a transaction first acquires a snapshot of the database state, then performs a sequence of operations on this snapshot, and finally either commits or aborts. The database creates a snapshot of its state and assigns it to the transaction when the transaction starts executing. Typically, a database that provides snapshot isolation uses a multi-version concurrency control mechanism which encodes a snapshot as version of the database.

A transaction can be ordered before, after, or simultaneously with another transaction. Transaction ordering is determined by the timing of transaction snapshots relative to transaction commits. The ordering rules (illustrated in Figure 5-1) are as follows:

- Transaction T1 is ordered *before* transaction T2 if T1 commits before T2 acquires its snapshot. T2 observes values written by T1.
- Transaction T1 is *simultaneous* with transaction T2 if T1 snapshots before T2 commits and T2 snapshots before T1 commits. Neither T1 nor T2 observe the values of each other's writes.

Though snapshot isolation allows simultaneous transactions, it is not correct to think of sets of simultaneous transactions. *Before* is a transitive relation; *simultaneous* is not. That is, if T1 is *before* T2 and T2 is *before* T3, then T1 is *before* T3. However, if T1 is *simultaneous* with T2 and T2 is *simultaneous* with T3, then T1 is not necessarily *simultaneous* with T3. Figure 5-2 illustrates a scenario where *simultaneous* is intransitive: T1 is *before* T3, yet both T1 and T3 are *simultaneous* with T2.

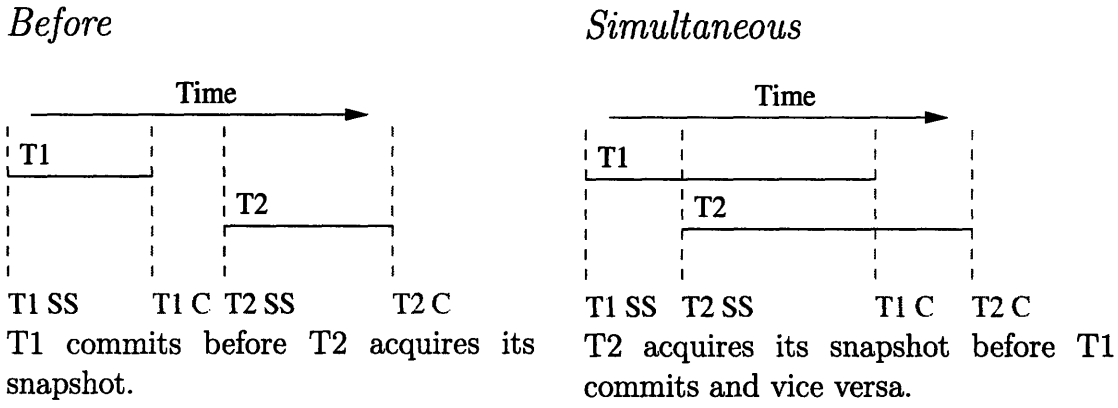


Figure 5-1: Illustration of transaction ordering rules

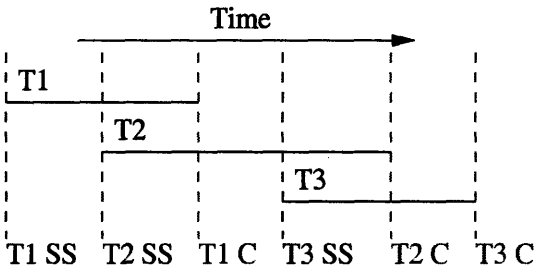


Figure 5-2: T1 is *simultaneous* with T2, and T2 is *simultaneous* with T3, yet T1 is not *simultaneous* with T3.

Snapshot isolation disallows concurrent writers: simultaneous transactions may not write the same data item. Should two simultaneous transactions attempt to write the same item, the first transaction to write will succeed and if it commits the later transaction will be forced to abort. To avoid unnecessary aborts, the later transaction is blocked until the earlier transaction commits or aborts. If the earlier transaction commits, the later transaction aborts. If the earlier transaction aborts, the later transaction's write succeeds and it continues executing. If the earlier transaction committed prior to the later transaction's write, the later transaction aborts immediately. Snapshot isolation places no restrictions on reads.

To make execution under snapshot isolation more concrete, we provide several examples in Table 5.1. Example 1 provides an example of two simultaneous transactions executing under snapshot isolation. Example 2 demonstrates a pair of transactions executing in a manner allowed by snapshot isolation, yet the schedule is not serializable. A serializable execution would result in both A and B having the same value (both have value 5 or both have value 7). As example 2 demonstrates, a third possi-

bility exists under snapshot isolation: the values have swapped. Finally, Example 3 shows a pair of transactions whose execution is serializable (T1 then T2), but one of them would be aborted by snapshot isolation due to a concurrent write.

Example 1:
Execution of simultaneous transactions under snapshot isolation

Time	Transaction 1	Transaction 2	Comment
1	Read A: 7		T1 acquires snapshot
2		Read A: 7	T2 acquires snapshot
3		Write A←12	
4	Read A: 7		T1 reads from its snapshot
5		COMMIT	
6	Read A: 7		T1's snapshot unchanged
7	Write A←8		Concurrent write
8	ABORTED		DB aborts T1 due to concurrent write

Example 2:
Schedule disallowed by serializability but allowed by snapshot isolation

Time	Transaction 1	Transaction 2	Comment
1	Read A: 5		
2		Read B: 7	
3	Write B←5		Under R2PL, T1 blocks here
4		Write A←7	Under R2PL, abort due to deadlock
5	COMMIT		
6		COMMIT	

Example 3:
Schedule disallowed by snapshot isolation but allowed by serializability

Time	Transaction 1	Transaction 2	Comment
1	Read A: 5		
2		Read A: 5	
3	Write B←5		
4	COMMIT		
5		Write B←7	Under SI, abort due to concurrent write
6		COMMIT	

Table 5.1: Snapshot Isolation Concurrency Examples

5.2 Key Issues

In this section, we present three key issues that snapshot epoch scheduling must address. First, it must keep replicas synchronized. Second, the protocol must resolve conflicts between transactions. Finally, the protocol must handle fail-stop and Byzantine faults in replicas.

5.2.1 Keeping Replicas Synchronized

To keep the database replicas synchronized, they must all execute transactions in the same order. As mentioned previously, under snapshot isolation, the ordering of snapshots and commits entirely determines transaction ordering. Thus, replicas will remain synchronized if they acquire snapshots and commit transactions in an identical order.

Ensuring identical order will result in unnecessary synchronization overhead; we only need equivalent order. The transaction ordering rules for snapshot isolation all compare one transaction's snapshot time to another transaction's commit time. The order of one transaction's snapshot time relative to another transaction's snapshot time does not matter. Figure 5-3 illustrates reorderings of snapshots and commits that do not affect the overall ordering of transactions: both transactions are still simultaneous. Given a set of snapshot acquisitions with no intervening commits, any ordering is acceptable. The equivalent property holds for sets of commits with no intervening snapshots. Thus, the system need only synchronize at the transition between snapshot and commit.

We ensure equivalent orders by dividing time into an alternating series of snapshot and commit epochs. A transaction is assigned to one snapshot epoch and one commit epoch. Each replica must acquire snapshots for all the transactions marked with a particular snapshot epoch before it can progress to the subsequent commit epoch. Similarly, a replica must commit all the transactions marked for a particular commit epoch before it can process snapshots in the following snapshot epoch. Within an epoch, all snapshots or commits may be issued concurrently as their relative order

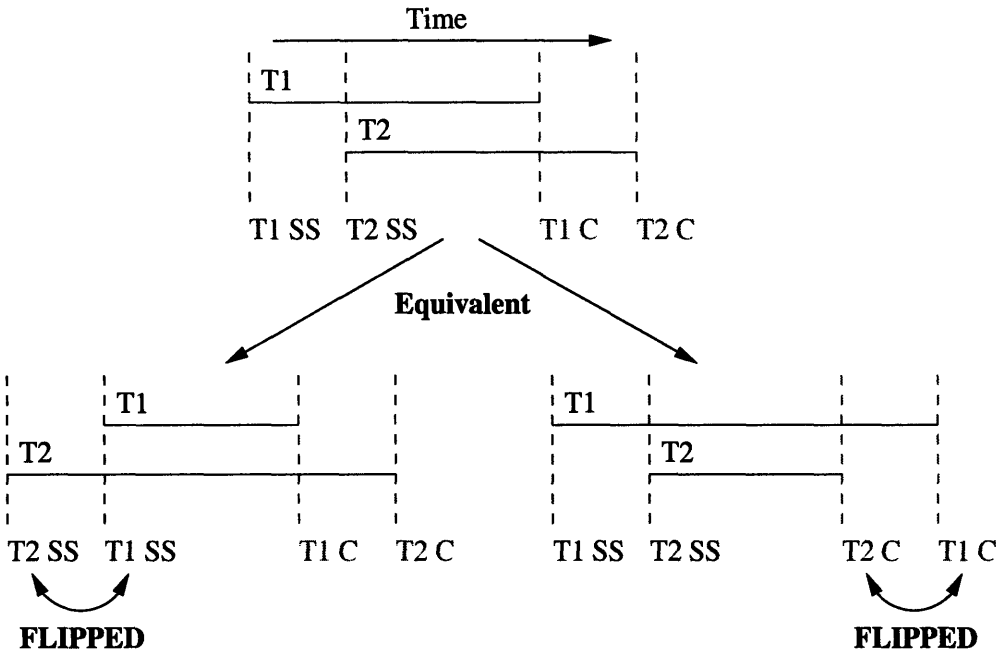


Figure 5-3: Equivalent orderings produced by reordering snapshots, reordering commits, but not reordering commits *and* snapshots.

does not matter.

While databases provide an explicit commit operation, they do not provide an explicit snapshot operation. Snapshot isolation does not specify when the snapshot is taken for a transaction. However, by the time a query has produced an answer, the snapshot must have been acquired. Thus, to force a transaction to acquire a snapshot, the system begins a transaction and issues a statement. When the statement returns, the transaction has acquired the snapshot it will use. Since the system cannot proceed to a subsequent commit epoch until after the statement returns, using a client-issued statement could impact performance if the statement runs for a long time. Therefore, the system issues a computationally trivial read operation (*e.g.*, reading the only row from a single row table) to acquire the snapshot for a transaction.

5.2.2 Resolving Conflicting Transactions

When the system is presented with a pair of simultaneous transactions that write to the same item, all of the replicas must select the same transaction to abort. The simplest solution is to only allow one outstanding write operation at a time, but such

serialization results in poor performance. Our solution is the same as in commit barrier scheduling: use a primary replica to decide which of the conflicting operations gets to proceed. Write operations are directed first to the primary, and are only sent to the secondary replicas when the operation completes executing on the primary. The primary's concurrency control mechanism will resolve conflicts between writes; either blocking or aborting the "loser" of any conflict. As long as the primary is not faulty, the secondary replicas are never exposed to concurrent writes and thus always select the same transactions to execute as the primary.

Unlike CBS, only writes need be sent first to the primary. Under snapshot isolation, reads cannot conflict with any other operation and thus do not need to be ordered by the primary. Thus, read-only operations can issue on all replicas immediately.

Also unlike CBS, all of the replica's concurrency control mechanisms must behave identically: given two transactions, all (non-faulty) replicas must agree on whether they conflict or not. CBS always had the option of executing transactions sequentially on a replica, which rendered the details of the replica's concurrency control mechanism moot. However, under snapshot isolation, if the coordinator schedules transactions simultaneously, then a replica must execute them simultaneously: dropping to serial execution is incorrect. CBS has the notion of a *sufficiently blocking* primary: the primary's concurrency control mechanism could be coarser grained than the secondaries. A replica with a coarser-grained mechanism makes no sense in SES, as the replica would be incapable of serving as a secondary (*i.e.*, it will abort transactions that the coordinator commits). We assume that fine-grained locking is the norm in databases that provide snapshot isolation.

5.2.3 Handling Faults

Under serializable execution, a replica that crashes will recover as a replica with consistent, but stale, state. However, as Table 5.2 demonstrates, the same is not true under snapshot isolation. The issue arises when a replica crashes after it has committed one of a pair of simultaneous transactions. When the replica recovers, it cannot re-execute the unfinished transaction because it has "lost" access to the snap-

shot epoch in which the transaction is supposed to snapshot. Consider the example in Table 5.2; transactions 1 and 2 are supposed to be *simultaneous*, but when the crashed replica recovers, it can only re-execute transaction 2 *after* transaction 1.

Time	Transaction 1 B ← A + 1	Transaction 2 A ← B + 1
1	Read A: 5	
2		Read B: 7
3	Write B←6	
4		Write A←8
5	COMMIT	
6		COMMIT

- Replica set commits both transactions, ending with A=8 and B=6.
- Faulty replica crashes after time 5 but before time 6, and recovers with A=5 and B=6.
- Faulty replica cannot re-execute Transaction 2 (A ← B + 1) to arrive at state A=8 and B=6.

Table 5.2: Re-execution cannot correctly update crashed replicas in Snapshot Isolation

We can update the database state as if the transaction were correctly executed by extracting from the non-faulty replicas the data items modified by the transaction, and applying the modifications directly to the crashed replica. With respect to the situation in Table 5.2, we would extract A←8 from transaction 2; applying this modification directly to the replica does correctly update its state. The set of modified data items is called a *writeset*, and it includes inserted, updated, and deleted items. *Writeset Extraction* is an operation run by a replica for a particular transaction; it returns the writeset of the transaction. Databases do not provide writeset extraction as a client operation; they must be modified to support it. By performing writeset extraction before committing each transaction, our system can tolerate replica crashes by replaying writesets to bring the replica up to date.

Tolerating Byzantine faulty replicas requires voting. As before, the system is parametrized by f, the maximum number of simultaneously faulty replicas. An answer with $f + 1$ matching votes is guaranteed to be correct, and we must run $2f + 1$ replicas to ensure that we always receive $f + 1$ responses.

A Byzantine faulty primary can allow conflicting transactions to reach the secondary replicas, presenting a potential correctness issue. Snapshot isolation is a pessimistic protocol with regard to concurrent writes: a transaction that has successfully

Transaction	Status by Replica		
	Primary *faulty*	S1	S2
T1	exec	exec	block
T2	exec	block	exec

- Transactions T1 and T2 conflict.
- $f = 1$; need $f + 1 = 2$ votes to commit

Table 5.3: Example of a situation where two conflicting transactions acquire $f + 1$ votes.

executed all of its write operations can commit successfully. Given a set of simultaneous conflicting transactions, a non-faulty replica will execute only one to completion, effectively voting for that transaction to be the one to commit. Table 5.3 illustrates a scenario where a pair of conflicting transactions both acquire $f + 1$ votes and could potentially commit. Committing both violates our correctness criteria for two reasons: S1 and S2's states diverge despite both being non-faulty and the clients observe two conflicting transactions that both commit. We resolve the issue by requiring that a replica must have executed all transactions previously committed by the system before voting on the next one. Table 5.4 shows what happens if the system commits T1. At this point, S2's vote for T2 will not count until it has executed T1. When S1 commits T1 it will abort T2 due to concurrent writes, disagreeing with the primary. Without S2, T2 lacks enough votes to commit and will stall. The mechanism prevents conflicting transactions from committing; the resulting liveness problem is discussed in section 5.3.1.

Transaction	Status by Replica		
	Primary *faulty*	S1	S2 *stale*
T1	commit	commit	block
T2	exec	abort	exec

- Transactions T1 and T2 conflict.
- S2's vote for T2 does not count because it has not executed T1.

Table 5.4: Situation from Figure 5.3 after coordinator commits T1.

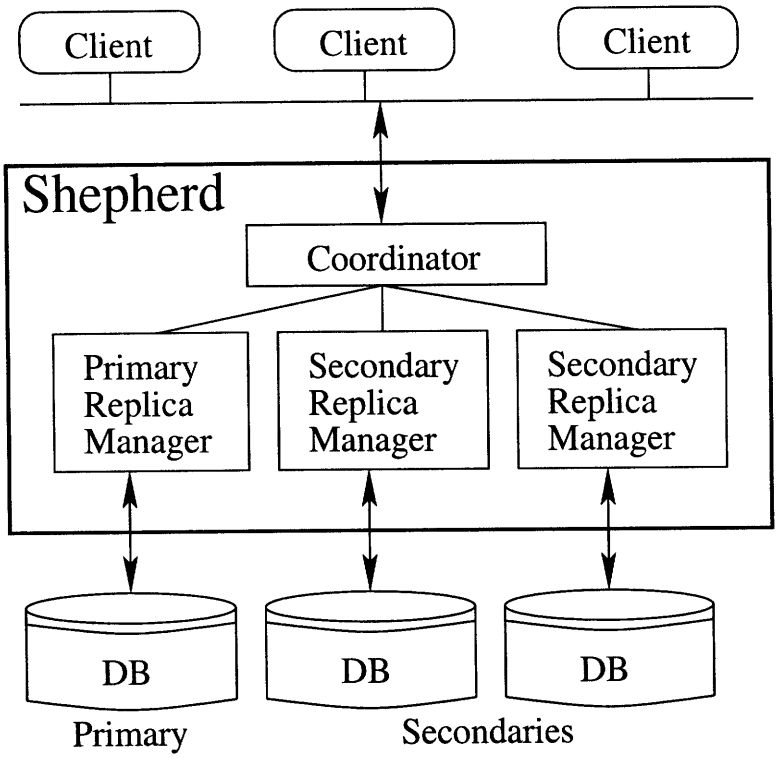


Figure 5-4: Snapshot Epoch Scheduling Shepherd Architecture.

5.3 Protocol

The architecture is the same as for CBS (as shown in Figure 5-4), namely, that clients interact with a *Shepherd*, which coordinates the database replicas. Clients issue transactions consisting of a sequence of queries, followed by a COMMIT or ABORT. These operations are not required to be issued in bulk. As before, $2f + 1$ replicas are required to tolerate f failures. All replicas are required to use snapshot isolation as their concurrency control mechanism. The shepherd runs a coordinator, and a replica manager for each replica. The shepherd maintains an on-disk write ahead log.

The shepherd divides time up into a series of alternating snapshot and commit epochs (as shown in Figure 5-5), which the replica managers use to ensure that each replica correctly orders transactions. For each transaction, the coordinator must schedule two events: a snapshot event and a commit event. The coordinator assigns transaction events to epochs using an epoch counter, *epoch*. Even numbered epochs are for snapshots, while odd numbered epochs are for commits. The shepherd moves

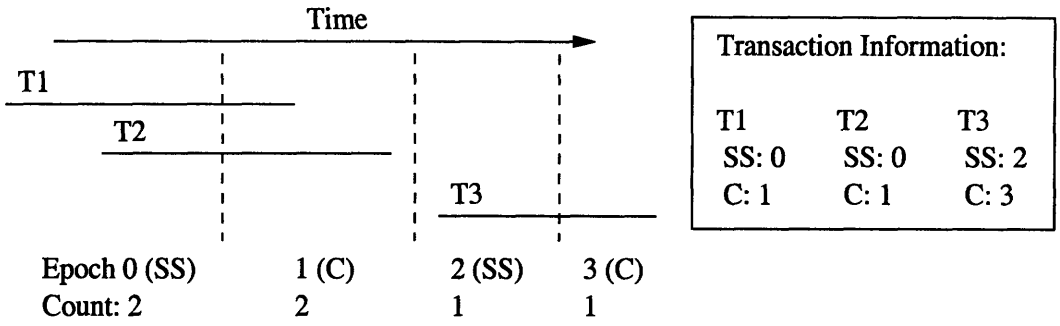


Figure 5-5: Transactions are scheduled into snapshot and commit epochs

to the next epoch when it must assign an event whose type (snapshot or commit) does not match the current epoch. For each epoch, the coordinator notes the number of events during the epoch in an array called *counts*. A replica manager can increment its epoch counter, *replica.epoch*, when *epoch* > *replica.epoch* and the replica has completed the processing of all events in *replica.epoch*, *i.e.*, it has processed *counts*[*replica.epoch*] events. The primary and secondary managers both follow these rules for incrementing their epochs. The snapshot and commit epoch mechanism guarantees that a transaction will see exactly the same snapshot on all replicas, thus ensuring that the replicas will stay synchronized and non-faulty replicas will produce the same answers.

When a new transaction arrives at the coordinator, the coordinator assigns the transaction to a snapshot epoch and issues the snapshot to the replicas. A replica manager is free to issue any query of a transaction it has seen to the replica, assuming that the transaction has already acquired its snapshot at the replica. As previously described, reads can be sent to all replica managers immediately, while writes must first be executed by the primary. The coordinator sends the primary's answers to the client.

When the client issues a commit for a transaction, the shepherd starts the commit process. The commit process verifies that the transaction has been executed correctly, then commits the transaction. To verify that a transaction has been executed correctly, the coordinator waits for $f + 1$ replicas to have executed the transaction to completion. Each of these replicas must have matching answers to those sent to the client. Should the coordinator discover $f+1$ matching answers to a query that do not

match the answer that was sent to the client, the coordinator aborts the transaction. Otherwise, the coordinator requests the transaction writeset from each replica, and commits the transaction upon receipt of $f + 1$ matching writesets.

All replicas participating in the commit process must be up to date: they must have executed, if not committed, every transaction committed on the coordinator. The coordinator commits transactions sequentially, incrementing a sequence number for each committed transaction. Each replica manager also maintains a sequence number, which the replica manager increments whenever it completes executing a committed transaction. An up to date replica is one whose replica manager's sequence number matches the coordinator's sequence number.

The commit point for a transaction is when the coordinator commits the transaction, not later when the replicas commit the transaction. When a transaction commits on the coordinator, the coordinator increments its sequence counter and assigns the transaction to a commit epoch. The coordinator also marks the transaction with the value of the sequence counter when the transaction commits; the transaction sequence number is used during recovery, as explained in section 5.4.1. The coordinator writes the transaction's agreed-upon writeset and commit epoch to the shepherd's log and sends an acknowledgment to the client. Then the coordinator issues a commit for the transaction to the replica managers. Replica managers may commit a transaction only when the replica manager's epoch matches the transaction's commit epoch.

The full protocol is presented in Figures 5-6 and 5-7. Unlike commit barrier scheduling, the primary's replica manager is no different than the secondary replica managers.

5.3.1 Handling a Faulty Primary

As mentioned in section 5.2.3, a faulty primary can cause a liveness problem by passing two conflicting transactions. Table 5.5 picks up the scenario where the section 5.2.3 left off; secondary S2 has executed T2 and blocked T1. Unfortunately, S2 now has a liveness issue: it cannot make progress on T1 since it executed T2. Replica S2 will be able to complete executing T1 if it aborts T2. However, aborting a transaction

- Event: Receive BEGIN TRANSACTION from client
  Action:
  1. If *epoch* is odd, $epoch \leftarrow epoch + 1$
  2. $counts[epoch] \leftarrow counts[epoch] + 1$
  3. $T.snapshotEpoch \leftarrow epoch$
  4. Issue BEGIN TRANSACTION for T to replica managers

- Event: Receive query Q from client.
  Action: If Q is read-only, send Q to all replica managers; else send Q to the primary replica manager.

- Event: Receive response for query Q from primary replica manager.
  Actions:
  1. Send the response to the client, noting it as $Q.ans$.
  2. If Q is an update query, send Q to secondary replica managers.
  3. Add response to $votes(Q)$, the set of responses received so far from the different replicas.

- Event: Receive response from secondary replica manager for query Q.
  Actions: Add response to $votes(Q)$.

- Event: Receive ABORT from client.
  Actions:
  1. Send ABORT to replica managers.
  2. Send acknowledgment to client.

- Event: Receive COMMIT for transaction T from client.
  Action: Request transaction writeset

- Event: Receive writeset from uncommitted transaction T from replica manager
  Actions:
  1. add writeset to $writesets(T)$
  2. if $f + 1$ matching writesets from replicas with $replica.seqno = B$ then:
     (a) If the response $Q.ans$ sent to the client for some query Q in T is not backed up by f votes in $votes(Q)$ from replicas that are ready to commit T, send ABORT to the replica managers and inform the client of the ABORT.
     (b) Otherwise:
        i. $T.seqno \leftarrow B; T.commitEpoch \leftarrow epoch; B \leftarrow B + 1$
        ii. If *epoch* is even, $epoch \leftarrow epoch + 1$
        iii. $counts[epoch] \leftarrow counts[epoch] + 1$
        iv. Send acknowledgment to client and COMMIT to replica managers.

Figure 5-6: SES Coordinator pseudo-code.

- Event: BEGIN TRANSACTION for transaction T arrives from coordinator
  Action: Issue snapshot when replica.*epoch* = T.*snapshotEpoch*

- Event: snapshot completes on replica
  Action:
  1. replica.*count* ← replica.*count* + 1
  2. if *epoch* > replica.*epoch* and *counts*[replica.*epoch*] = replica.*count* then
     (a) replica.*epoch* ← replica.*epoch* + 1
     (b) replica.*count* ← 0

- Event: query Q from transaction T arrives from coordinator
  Action: Issue Q when all previous queries of T have completed executing on the replica. Send response to Q to the coordinator.

- Event: coordinator requests writeset for transaction T
  Action: When all queries of T have completed executing, perform writeset extraction and send resulting writeset to coordinator.

- Event: COMMIT for transaction T arrives from coordinator
  Action:
  1. When writeset extraction for T has completed, replica.*seqno* ← replica.*seqno* + 1
  2. Then issue COMMIT when replica.*epoch* = T.*commitEpoch*.

- Event: COMMIT completes on replica
  Action:
  1. replica.*count* ← replica.*count* + 1
  2. if *epoch* > replica.*epoch* and *counts*[replica.*epoch*] = replica.*count* then
     (a) replica.*epoch* ← replica.*epoch* + 1
     (b) replica.*count* ← 0

- Event: ABORT arrives from coordinator
  Action: discard any unexecuted queries and abort transaction

- Event: coordinator advances *epoch*
  Action: if *counts*[replica.*epoch*] = replica.*count* then
  1. replica.*epoch* ← replica.*epoch* + 1
  2. replica.*count* ← 0

Figure 5-7: SES Replica Manager pseudo-code.

Transaction	Status by Replica		
	Primary *faulty*	S1	S2 *stale*
T1	commit	commit	block
T2	exec	abort	exec

- Transactions T1 and T2 conflict.
- S1 and S2 picked different transactions to execute.
- Coordinator committed T1.
- S2's vote for T2 does not count because it has not executed T1.

Table 5.5: Reprint of Table 5.4.

may prevent a replica from being able to re-execute the transaction. Thus, aborting a transaction must be a global decision: T2 must be aborted on the whole replica set. The coordinator uses the following rule to abort transactions:

> Any uncommitted transaction whose commit process has stalled beyond a given timeout is unilaterally aborted by the coordinator.

From a correctness standpoint, once the coordinator has committed T1 it must globally abort T2: given two simultaneous conflicting transactions, only one transaction may successfully commit. Thus, aborting T2 is the right decision, though it will occur after a timeout rather than immediately when T1 is committed. The coordinator could decide to abort T2 immediately if it compared writesets extracted from T1 and T2 and determined directly that they conflict. However, the situation only arises when the primary is faulty, which is assumed to be uncommon. Comparing writesets would slow down the common case to speed up the uncommon case: not usually a good trade-off.

In Table 5.5, the system could make progress even though replica S2 could not. However, the whole system can deadlock, in exactly the same scenario that afflicts commit barrier scheduling (see Table 5.6). The system-wide deadlock can only happen when $f > 1$ ($f = 2$ in the example). No transaction acquires $f + 1$ votes. The same resolution used above works for the system-wide deadlock: one of the transactions will be aborted globally, freeing one of the good secondaries to agree with another good replica.

Transaction	Status by Replica				
	Primary *faulty*	S1	S2	S3	S4 *crashed*
T1	exec	exec	block	block	
T2	exec	block	exec	block	
T3	exec	block	block	exec	

Table 5.6: Example of a situation where the entire system deadlocks.

Aborting stalled transactions resolves the liveness problem, but can result in inefficiency. In Table 5.7, T1, T2, and T3 all have $f+1$ votes; which is enough to commit. If the coordinator commits T1 first, then replicas S2 and S3's votes no longer count. The coordinator will be unable to commit any of T2, T3, and T4 until it aborts a transaction. The coordinator will not help the situation by aborting T3 or T4, but it has no way of knowing this information. The coordinator can abort both T3 and T4 on the way to aborting T2 (which actually resolves the situation). However, the coordinator only needed to abort one of T3 and T4, which demonstrates the aforementioned inefficiency. The coordinator can make the right choice by performing writeset comparison on all of the stalled transactions. Writeset comparison only need be done when sets of transactions stall past the timeout.

Xaction	Status by Replica				
	Primary *faulty*	S1	S2	S3	S4 *faulty*
T1	exec	exec	block	block	exec
T2	exec	block	exec	exec	
T3	exec	block	exec	exec	
T4	exec	exec	block	block	

- T1 and T2 conflict
- T3 and T4 conflict
- Coordinator commits T1 first, rendering S2 and S3 not up to date

Table 5.7: Example of a situation where aborting transaction can result in inefficiency.

One final concern is the effect of a Byzantine network on the system. By filtering packets or forging a TCP connection reset, the network can cause individual database connections to break. Under the default model, breaking a database connection aborts any transaction in progress from the connection. Since transactions are not restartable under snapshot isolation, this can result in good replicas being unable

to process the transaction. If enough good replicas are forced to abort a transaction, the shepherd will not be able to acquire $f + 1$ votes and must also abort the transaction. Additionally, since SES allows group commit, it suffers from the same issues that CBS does when transactions get aborted at commit. These issues can be avoided by running a shim on the database replica; the shim makes transaction abort explicit. Details of such a shim can be found in section 6.3.1.

5.3.2 View Changes

If the primary is faulty, SES may be unable to make progress efficiently. The way to handle a faulty primary is to do a *view change* in which a new primary is selected by the coordinator and the old one is demoted to being a secondary. SES performs view changes in exactly the same manner as CBS, namely, it aborts all currently executing transactions, selects a secondary to be the new primary, and starts executing transactions again. For details related to view changes, see Section 4.2.3.

5.4 Fault Recovery

We now discuss how our system deals with failed nodes, *i.e.,* how it brings them back into a state where they can correctly participate in the protocol. We begin by discussing recovery of replicas that have suffered a crash failure. Then we discuss recovery of the shepherd.

Obviously, some mechanism is needed to repair a Byzantine-faulty replica. As we did for CBS, we defer the discussion of the techniques required to restore the state of a Byzantine-faulty replica to chapter 7. However, with writeset extraction SES can catch some Byzantine faults before they affect the replica's state. If a replica's writeset disagrees with the overall agreed-upon writeset, then committing the transaction on the replica will result in incorrect values being written to the replica's state. However, by aborting the transaction on the replica and instead applying the agreed-upon writeset, the corruption of the replica's state can be avoided. We call this process *writeset correction*, and it occurs as each replica goes to commit a transaction.

5.4.1 Recovery of a Crashed Replica

When a replica goes offline and then recovers, it rejoins the system in a locally consistent state: all transactions that it did not commit prior to its failure have aborted locally. As discussed in section 5.2.3, the replica manager replays transaction writesets to bring the replica up to date. Recovery consists of three phases: determining where to begin replay, replaying operations, and rejoining the active set by starting to execute transactions that have not yet committed.

A replica manager considers a replica to have crashed when the database connection to the replica breaks. If the manager had sent a commit to the replica prior to learning that the connection had broken, but did not receive a reply, it cannot know whether that commit had been processed before the replica went down. Application of transaction writesets is idempotent, but replaying all committed transactions would be seriously inefficient.

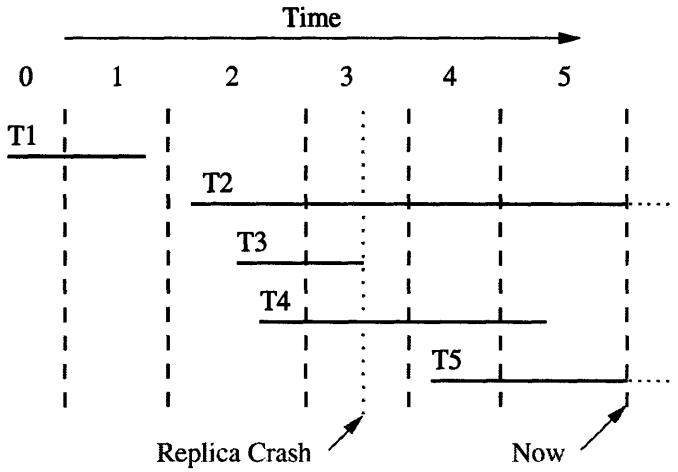
To allow the replica manager to bound transaction replay, we add a transaction log table to each replica; the table contains sequence numbers from committed transactions. We add to each transaction a query to insert a row into the table with the transaction's sequence number and commit epoch. If the transaction commits prior to a replica crash, the transaction log table will contain an entry for the transaction when the replica recovers.

When the replica recovers and the connection is re-established, the replica manager reads the transaction log table to determine where to begin replay. Replay starts in the most recent epoch with a transaction in the log table. Any transactions committed in this epoch yet not present in the table must be replayed, as well as all committed transactions in subsequent epochs. Writeset application is typically much faster than executing the transactions, ensuring that the recovering replica will eventually catch up to the rest of the replicas. The replica manager can insert rows into the transaction log table as it performs recovery to mark progress in case the replica fails amid recovery.

A recovering replica must also start executing transactions have not yet commit-

ted. Transactions currently executing on the rest of the replica set can be divided into two categories based on whether they acquired their snapshot before or after the replica crash. Currently executing transactions that acquired their snapshot before the crash require writeset application to commit on the recovering replica. They cannot be re-executed because the required snapshot is no longer accessible on the recovering replica. A further implication is that long-running transactions can have trouble if $f + 1$ nodes crash and recover during the lifetime of the transaction. This is a liveness issue and not a correctness issue because nodes that crash and recover will not vote on these transactions, which will result in the transactions timing out and getting aborted.

For executing transactions that acquired their snapshot after the replica crash, the necessary snapshots are available. The replica manager replays writesets, watching for transitions from one commit epoch to the next. Before committing any transactions in the next commit epoch, the replica manager checks to see if any executing transaction acquired its snapshot during the intervening snapshot epoch. If so, the replica manager begins executing the transaction and has it acquire its snapshot before continuing with transaction replay. When the recovering replica has caught up to the replica set, it will have correctly acquired the snapshots for all currently executing transactions and can once again contribute to the voting process.



Epoch Schedule

	Snapshot	Commit
T1	0	1
T2	2	-
T3	2	3
T4	2	5
T5	4	-

Replica `xactions` table

Transaction	Epoch
T1	1
T3	3

Figure 5-8: Crash Recovery Scenario.

Consider the situation in Figure 5-8, where a replica crashes during the commit

of transaction T3. When the replica recovers, the replica queries the transaction log table (called `xactions` in the figure) and discovers that T3 did in fact commit before the replica crashed. Replay will start in commit epoch 3, as this is the latest epoch mentioned in the `xactions` table. The replica must replay T4's writeset to bring it up to date with the rest of the replicas. Note that re-execution is not possible with T4 because the replica cannot start executing T4 with the correct snapshot: its state reflects the result of committing T3. Before replaying transactions, we note that two transactions are currently executing: T2 and T5. The recovering replica cannot execute T2, as it acquired its snapshot during epoch 2, which occurred before the replica crash. The replica will not be able to vote on T2; it will have to wait for a writeset if and when T2 commits. T5 does not have this issue as it acquired its snapshot after the replica crash. The replica manager starts T5, has it acquire a snapshot, then applies the writeset for T4, which completes the replica's recovery.

The replica uses an insert into the transaction log table instead of an update to avoid write-write conflicts. Inserting rows causes the table to grow in size, but the transaction log table can be truncated to a single row at any time. The replica manager injects a transaction into the workload to delete all rows less than the maximal epoch value. Since currently running transactions never insert values less than the current maximum, the delete operation will never result in a write-write conflict.

5.4.2 Recovery from a Shepherd Crash

As mentioned previously, the shepherd maintains a write-ahead log which it uses to survive crashes. When the coordinator determines that it can commit a transaction, it writes the transaction writeset and sequence number to the log. The coordinator forces the log to disk before replying to the client. Note that the coordinator can employ the group commit optimization, where by delaying the processing of a commit, the system can write a series of log records in one disk operation.

To recover from a crash, the shepherd reads the log and identifies the sequence number of the last committed transaction. The shepherd knows where to start writeset application for each replica by examining the replica's transaction log table, just

as if the replica had crashed. The shepherd can start accepting new client transactions when the primary replica manager's sequence number matches the coordinator's sequence number.

The shepherd's log can be truncated by removing information about transactions that have committed at all the replicas. We can expect that most of the time all replicas are running properly and therefore the log need not be very large.

5.5 Correctness

In this section we present an informal discussion of the safety and liveness of our system. We assume here that no more than f replicas are faulty simultaneously.

5.5.1 Safety

SES is safe because it guarantees that correct replicas have equivalent logical state, and that clients always get correct answers for transactions that commit.

Ensuring that correct replicas have equivalent logical state requires that they execute equivalent operations in equivalent orders. SES has the same SQL compatibility requirements as CBS with regard to SQL that executes identically on all replicas. Database SQL heterogeneity aside, within a transaction, statements are submitted in an identical order to each replica, thus ensuring that all replicas observe equivalent operations.

Additionally, a non-faulty replica must not commit a transaction that conflicts with any other transaction committed by a non-faulty replica. If the coordinator does not commit conflicting transactions then neither will the replicas. The coordinator cannot commit conflicting transactions because it requires $f + 1$ up to date replicas to commit a transaction and no non-faulty up to date replica will agree to commit a conflicting transaction.

Ensuring equivalent order requires that given two transactions, all non-faulty replicas order them identically. For example, if T1 is *before* (or *simultaneous* with) T2 on replica R1, then T1 must be *before* (or *simultaneous* with) T2 on all other replicas.

The transaction ordering rules from section 5.1 all compare the time one transaction commits to the time another transaction acquires a snapshot. SES breaks time up into discrete epochs, and fixes each transaction's snapshot time into an epoch and commit time into a later epoch. Since all correct replicas acquire transaction snapshots and commit transactions during the assigned epochs, if transaction T1 commits in epoch 5 and transaction T2 acquires its snapshot in epoch 6, then all replicas will order T1 *before* T2. Thus, replicas have equivalent state because they execute equivalent operations in equivalent orders.

During the commit process, the coordinator verifies that each transaction returned correct answers to the client. Each answer must be attested to by at least $f+1$ replicas, ensuring that at least one non-faulty replica agrees with the answer. Should the answer sent to the client differ from the answer that acquire $f+1$ votes, the transaction is aborted, as at least one non-faulty replica believes the answer is incorrect. If the coordinator waits long enough, all $f+1$ non-faulty replicas will participate, ensuring that the situation will eventually fit into one of the two aforementioned scenarios.

SES offers a trade-off between read-only query latency and correctness guarantees provided on query answers. Since reads never block, they are executed simultaneously on the whole replica set. Unlike CBS, SI replicas never change their answers during re-execution, thus the answer produced is always the final answer. By running the read operation on all replicas and reaching agreement *before* returning, the system can guarantee correct answers to reads before the commit point. Thus, read-only operations would run at the speed of the slowest of the $f+1$ fastest replicas and would be guaranteed correct answers. By contrast, the protocol as previously described returns answers when the primary produces them, which is likely faster than the slowest of the $f+1$ fastest replicas.

Unfortunately, the argument in the previous paragraph could be considered incorrect. While SI replicas will not change the content of their answer, they can change whether they return an answer at all. If a faulty primary passes two transactions that make concurrent modifications, one of the transactions should not return an answer for that statement (or any subsequent statements). At the commit point, one of the

transactions will get aborted, effectively withdrawing answers for a set of statements. Getting an answer when the system should not give one could be considered incorrect behavior.

On the other side of the trade-off would be to return the first answer produced, as opposed to the primary's answer. Latency is reduced to that of the fastest replica, but with the implication that a faulty replica can get its answer sent to the client with high probability (*e.g.,* lying is faster than actually computing the answer). Returning the first answer provides the same correctness guarantee as returning the primary's answer, however it is easier for an adversary to cause incorrect answers to be sent to the client.

5.5.2 Liveness

Given a non-faulty primary, SES is live assuming that messages are delivered eventually and correct replicas eventually process all messages they receive.

However, the primary may be faulty. A view change operation will replace a faulty primary with a new replica. However, the faulty primary could already have caused some transactions to stall and some replicas to be unable to make progress. The timeout mechanism will eventually abort these stalled transactions; a non-faulty primary would have merely aborted them quicker. A non-faulty replica can only stall if it executed a stalled transaction that will eventually abort. When the transaction aborts, the replica will once again be able to make progress.

Since the view change operation can involve aborting all currently running transactions, continuous view changes would result in the system being unable to make progress. An adversarial network cannot convince the shepherd that a primary returned incorrect answers (a sure sign it is Byzantine-faulty), it can merely make the primary appear very slow. By increasing the timeout used to decide when to do a view change because the primary is too slow, the shepherd can guarantee progress, given that messages are delivered eventually.

5.6 Practical Considerations Running SES

Writeset extraction must produce *logical* writesets for comparison by the shepherd. However, each database may store data in a database-specific layout and format. For example, one database could store data in a table, whereas another database could expose a view of the same name which computes the data from a collection of other tables. The writeset from one database will not directly match the other database. Thus, each database must also provide a bi-directional mapping function for converting between logical and physical writesets for that database. When producing writesets for comparison, the mapping function ensures that the shepherd compares logical writesets. Conversely, when the shepherd applies writesets to a database to bring it up to date, the mapping function converts logical writesets into physical writesets that can be applied to the database's data model. The mapping function is somewhat analogous to the SQL translation layer.

5.7 Optimizations

We conclude this chapter with a discussion of several optimizations to the basic SES protocol.

Scale-up

So far, replication has been used exclusively for increased fault tolerance, not for increased performance. Since the performance of the system is directly dependent on the performance of the primary, shifting load off the primary should improve performance. The primary does not have to execute reads to determine conflicts, thus the shepherd can avoid sending reads to the primary entirely. Reducing the load on the primary should allow it to scale up write-write conflict detection. If after some timeout a read still has not achieved $f + 1$ matching votes, the read can be sent to the primary for a tie-breaker.

We can run more than $2f+1$ replicas to provide performance scaling on a workload

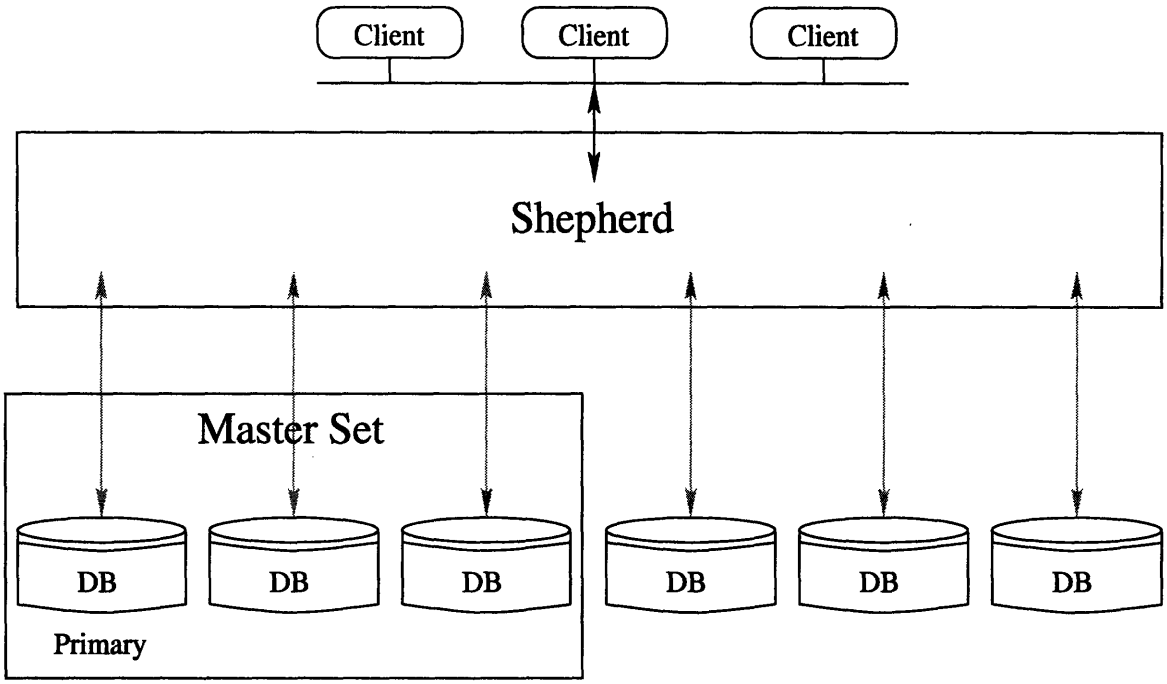


Figure 5-9: Replication for fault tolerance and performance ($f = 1$).

which consists mostly of read-only transactions. One such architecture, modeled on the Ganymede [31] system, appears in Figure 5-9. The system runs more than $2f+1$ replicas and the coordinator uses snapshot epoch scheduling to ensure that they remain synchronized. The coordinator schedules all transactions into snapshot and commit epochs as described above, but now it has a choice about which replicas will execute each transaction. The Master Set consists of $2f+1$ replicas that execute all of the update transactions. Reads are sent to any $2f+1$ replicas. Any replica that is not in the Master Set uses the writesets extracted from the update transactions, applying and committing them during the appropriate commit epoch. Writeset application is typically much faster than executing the update queries. The major change to the coordinator and replica manager code is that replica manager must maintain a replica.*counts*[*epoch*] to keep track of which transactions have been issued to the replica, rather than depending on a global *counts*[*epoch*].

Read-only Transactions

Read-only transactions are entirely ordered by their snapshot epoch. Thus, they need not be scheduled into a commit epoch, nor must they be assigned a sequence number. Furthermore, the coordinator can agree that a read-only transaction can commit (*i.e.*, all answers sent to the client are correct) without synchronizing relative to other transactions. The commit process for read-only transactions goes as follows:

- **Coordinator**
  - Event: Receive COMMIT for read-only transaction T from client
    Action: Wait for $f + 1$ replicas with matching answers for each query.
    * If the response $Q.ans$ sent to the client for some query Q in T is not backed up by $f + 1$ votes in $votes(Q)$ then ABORT
    * Else send acknowledgment to client and COMMIT to replica managers.
- **Replica Manager**
  - Event: Receive COMMIT for read-only transaction T from coordinator
    Action: Issue COMMIT to replica
  - Event: COMMIT for read-only transaction T completes on replica
    Action: Nothing

Scheduling Epochs

The coordinator can perform an optimization similar to group commit by delaying snapshots or commits to group them better into epochs. Epoch advancement on a replica is a synchronization point because all operations from the previous epoch must be completed before the replica can advance to the next epoch. When scheduling transactions into epochs, the coordinator can delay advancing to the next epoch slightly, hoping to be able to schedule another transaction into the current epoch. By doing so, the shepherd can increase epoch size, essentially allowing more concurrency. The potential downside of this optimization is that it increases the number

of simultaneous transactions, which would also increase the possibility of write-write conflicts.

Deferring transaction snapshot acquisition is more beneficial than deferring transaction commit. When a transaction is ready to commit, it holds locks on the items it has written. Deferring the commit extends the duration that the locks are held, potentially reducing throughput in high contention scenarios. However, snapshot acquisition occurs before any locks are acquired, thus there is more flexibility in snapshot scheduling than commit scheduling.

Chapter 6

Implementation and Performance

Heterogeneous Replicated DataBase (HRDB) is our prototype implementation of the shepherd. We have two implementations of the shepherd, one that runs Commit Barrier Scheduling (CBS) and one that runs Snapshot Epoch Scheduling (SES). In this chapter, we present some details of each implementation, followed by an analysis of the implementation's performance. We conclude with a discussion of bug tolerance and discovery with HRDB.

6.1 CBS Implementation

We implemented the HRDB prototype of CBS in Java and it comprises about 10,000 total lines of code. Clients interact with the shepherd using our JDBC driver. Implementing our own limited functionality JDBC driver was simpler than having the shepherd understand an existing JDBC wire protocol (such as the one used by MySQL's JDBC driver). Clients only need to load our driver JAR file and change the JDBC URL to connect to the shepherd, thus allowing HRDB to work with existing applications. The replica managers also use JDBC to interact with their replica databases. All major database implementations supply a JDBC driver.

JDBC access involves blocking operations: its calls do not return until the database returns an answer to the query. Thus, the shepherd must run each concurrent transaction in a separate thread on the replica manager. Overall, to support c concurrent

transactions, the shepherd must use $c(2f + 1)$ threads. Our implementation uses a thread pool containing a limited number of threads (typically 81 per replica), which allows thread-reuse but restricts the maximum number of clients. In practice, many databases also place restrictions on the maximum number of concurrent connections (*e.g.*, MySQL 5.1 defaults to 151).

We have successfully run HRDB with MySQL (4.1 and 5.1), IBM DB2 V9.1, Microsoft SQLServer 2005 Express, and Derby (10.1 and 10.3). Two other common options, Oracle and PostgreSQL, use Snapshot Isolation and are thus incompatible.

In the remainder of the section, we first present details of how our implementation uses heuristics to detect problems. Subsequently, we discuss how HRDB handles database heterogeneity. The final part of the implementation section describes a number of alternative mechanisms for handling concurrency that we implemented for comparison with CBS.

6.1.1 Heuristics

CBS requires that the implementation use a number of timeouts and heuristics to manage execution and handle faults. Our prototype does not police the primary's execution of transactions for execution time, deadlock count, or other subtle issues. It does track two timeouts to ensure forward progress:

- **Replica Stall Timeout** - Each replica manager tracks the last time it issued a statement to the replica. If the replica manager is lagging behind the coordinator (*i.e.*, $R.b < B$) and the last time it issued a statement is greater than the replica stall timeout (typically 1 second), then the replica reverts to sequential execution. All transactions other than the transaction that is supposed to commit next (the transaction with $T.B = R.b$) are rolled back and are not re-issued until the transaction that is supposed to commit next actually commits. At this point, normal execution resumes. The replica stall timeout ensures that a replica which executed transactions in an order that does not agree with the coordinator's order will eventually be able to make progress.

- **Transaction Stall Timeout** - Once a commit is received from the client, the transaction must commit on the coordinator before the transaction stall timeout expires. To receive a commit from the client, the primary must have completed executing the transaction, thus the transaction stall timeout represents how long the coordinator will wait for agreement. If the transaction stall timeout expires, then the secondaries likely disagree about transaction ordering (potentially a global deadlock), and aborting the transaction could help resolve the situation. Our implementation uses 1 second for the transaction stall timeout.

Obviously, the values chosen for our prototype reflect both our implementation and how TPC-C runs on the databases we test with. Our timeouts are static values, but an actual implementation would need to adjust them based on the workload and environment.

6.1.2 Handling Heterogeneity

HRDB has a basic translation layer for each database system that rewrites SQL queries using simple syntactic transformations (*e.g.*, translating function names, handling various data definition language expressions, etc.). However, we require that clients submit SQL that behaves deterministically. Ensuring deterministic queries was not onerous for us because the benchmarks we ran were mostly deterministic to begin with.

6.1.3 Handling Concurrency

HRDB supports a number of alternative schemes to enable us to evaluate the performance of Commit Barrier Scheduling. Each of the schemes detailed below is used by a replicated database system for concurrency control. By supporting other modes, we can compare the overhead of each scheme in the same environment. In addition to CBS, HRDB implements the following alternative schemes for concurrency control:

- **Serial** - Each transaction must acquire a global lock before executing, resulting in transactions being presented to the database replicas in an entirely sequential

manner. This scheme provides minimal performance, but obvious correctness. Serial execution is used by C-JDBC [7] for all write operations.

- **Table-level locking** - Before executing each SQL statement, the coordinator extracts the tables involved and acquires shared or exclusive locks depending on whether the statement is read-only (`SELECT`s typically require only shared locks). This coarse-grained locking scheme represents a simple SQL-parsing approach to determining conflicting transactions. The locking mechanism must also detect and resolve deadlocks that occur in the workload. A versioning scheme based on tables is used by [2].

- **Explicit-Partition locking** - By declaring categories of transactions that conflict (partitions), the client takes on responsibility for concurrency control. The client supplies special locking statements to the coordinator which explicitly indicate which resources each transaction depends on. The two primitives, `LOCKX id` and `LOCKS id`, request exclusive and shared access to the client-defined resource `id`. The locking mechanism itself is identical to the table-level locking scheme, except that the client chooses which locks to acquire. If the client does not send SQL that matches the partitions, the replica set can deadlock or diverge (as we proved while debugging the scheme). Client-specified partitions are used in [17].

- **Serial Secondaries** - Instead of using a barrier scheme to determine which statements can execute concurrently on the secondaries, secondaries execute transactions serially. Answers are sent to the client as they are produced by the primary, with the whole transaction shipped in batch to the secondary when it completes executing on the primary. The Pronto [29] system operates in a similar manner, though it does not attempt to tolerate Byzantine failures.

  The scheme is much simpler than CBS—there is no possibility of mis-ordering on the secondaries. However, the performance suffers from two issues: execution is not pipelined and transactions execute serially on the secondaries. When the commit arrives from the client, the secondaries must execute the complete

transaction, resulting in a longer wait before agreement is reached. System performance is fundamentally limited by sequential execution at the secondaries.

Since these alternative schemes ensure transaction isolation and consistency at the Shepherd level, the database replicas can execute at a lower isolation level. We run the database replicas using READ UNCOMMITTED, where the transactions do not acquire any locks during execution. Running at this lower isolation level improves sequential performance. The exception is that the Serial Secondaries scheme requires that the primary (only) run at the SERIALIZABLE isolation level.

6.2 CBS Performance Analysis

In this section, we characterize the performance of HRDB under different workloads and compare it to the alternative schemes mentioned above. Our tests were done with $2f + 1 = 3$ replicas, 1 primary and two secondaries, which can tolerate 1 faulty replica. Our implementation does not perform logging on the shepherd, nor does it use SSL to encrypt and authenticate JDBC connections.. The tests were run on a cluster of Dell Poweredge 1425 machines running Fedora Core 4 Linux, each with dual processor 2.8 GHz Xeons with 2GB of RAM and SATA 160 GByte disks, and all attached to the same gigabit Ethernet switch. Some tests also included a Dell OPTIPLEX GX620 3.8Ghz with 4GB of RAM running Windows XP. The overhead tests were run using MySQL 5.1.23 with InnoDB tables.

We use the TPC-C [42] query mix to test our system because it produces a high concurrency transaction processing workload with non-trivial transaction interaction. TPC-C simulates an order-entry environment where a set of clients issue transactions against a database. The TPC-C workload is heavily slanted towards read/write transactions, with read-only transactions comprising only 8% of the workload. Our tester implementation runs on a single machine, with one thread per client to submit requests. The test machine spends most of its time waiting for queries to complete, making it an unlikely performance bottleneck.

We use the same transaction types as TPC-C but do not attempt to model the

keying, wait, and think times. Instead, when a client finishes a transaction, it immediately submits another. Changes in wait time shift the number of clients required to reach saturation, but do not affect the relative performance of our system with respect to the database. Each test initializes the database for a specific number of warehouses, runs the system for 10 minutes with 20 clients to warm up the buffer cache, and then measures throughput for various client configurations.

The unit of concurrency in TPC-C is the warehouse: most transactions target a single warehouse and transactions targeting the same warehouse likely conflict. Each warehouse takes up approximately 100MB of space. Figure 6-1 shows the performance of a single MySQL5.1 database on TPC-C, for various numbers of warehouses. For the 1, 3, and 5 warehouse cases, the system is limited by low concurrency in the workload. For 10 and 30 warehouses, there is ample concurrency and the system is I/O limited. These 5 TPC-C scenarios are the ones we use when evaluating the performance of CBS.

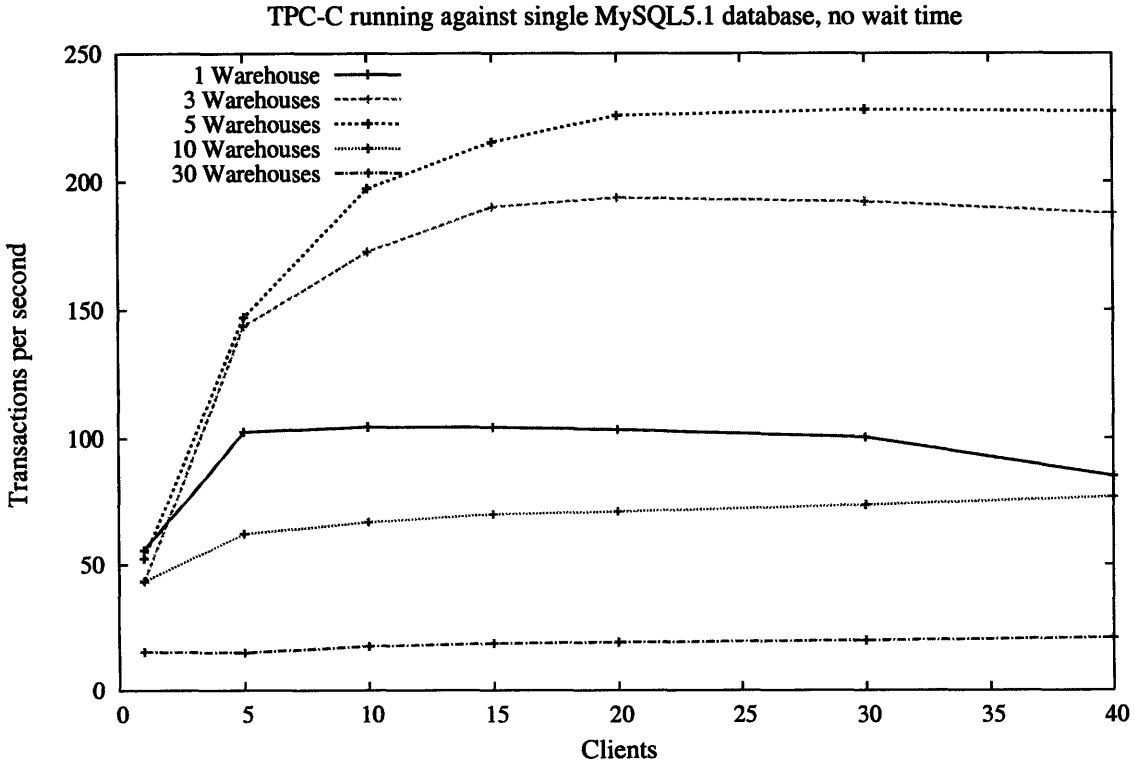


Figure 6-1: MySQL performance on TPC-C with various warehouse counts.

We begin by measuring the overhead of our middleware implementation, then progress to the overhead of commit barrier scheduling with a homogeneous replica set. Then we demonstrate that HRDB does work with a heterogeneous replica set and achieves good performance. We finish up the section with results that demonstrate that HRDB tolerates crash faults. We note that the HRDB implementation is an untuned Java prototype and the TPC-C implementation is also untuned.

6.2.1 Middleware Overhead

The middleware architecture interposes another server between the client and the database, creating a source of overhead independent of the concurrency control mechanism. Given ample network bandwidth and processing power on the middleware server, the additional latency per operation that it imposes can still reduce throughput. Specifically, increased operation latency lengthens transaction duration, which reduces throughput in high contention workloads. For example, if all transactions require the same lock, the maximum throughput is 1/(transaction duration). In addition to latency overhead, we also investigate the communication overhead of our JDBC driver to ensure that it does not adversely affect performance.

Latency Overhead

To measure the effect of middleware on system throughput, we compare the performance of running TPC-C directly on the database against running TPC-C through several different middleware implementations. Since we are evaluating the effect of the middleware's existence, each of the three implementations we benchmarked is as simple as possible: they perform no replication or concurrency control. Two implementations are merely TCP proxies which proxy the JDBC connections from the client to the database. The third is a JDBC proxy, which communicates with the client using our own JDBC protocol and communicates with the database using the database's JDBC protocol. Because JDBC is a blocking interface, the JDBC proxy uses many threads to support many concurrent clients. For comparison, one of the

TCP proxies was implemented using a multi-threaded, blocking I/O model, and one was implemented using a single-threaded, non-blocking I/O model.

Figures 6-2 and 6-3 show the results of running TPC-C through middleware that does no replication or concurrency control. The PassThrough is the JDBC proxy, where the clients communicate with it using our JDBC protocol and it communicates with the database using the MySQL JDBC protocol. None of the intermediaries perform any replication: there is only one database. In each configuration, there is a Java Virtual Machine with the full number of client threads using MySQL's JDBC libraries (on the tester machine for MySQL, Proxy, and ThreadProxy, and on the middleware machine for PassThrough).

When the database is limited by available concurrency instead of I/O (1, 3, or 5 warehouses, not 10 warehouses), introducing a TCP proxy (single or multi-threaded) results in significant performance overhead. For the 1 warehouse case, most transactions conflict and the proxies both show about 30% overhead. With 3 warehouses, the multi-threaded proxy (16% overhead) takes advantage of the quad-core CPU on the middleware machine and outperforms the single-threaded proxy (33% overhead). With 5 warehouses, thread scheduling starts interfering in the multi-threaded proxy (39% overhead) but not in the single-threaded proxy (31% overhead). With 10 warehouses the database is limited by I/O operations and thus additional operation latency has less of an effect on performance (both proxies exhibit only 14% overhead).

With enough available concurrency, the PassThrough has negligible overhead. Since it is JDBC-aware, it interacts with the client and the database on the operation level, never needing to block waiting on the wrong entity. The MultiThread proxy uses twice as many threads (one to wait on the client and one to wait on the database) because it does not understand the JDBC protocol. Finally, the PassThrough's JDBC driver for communicating with the client is slightly more bandwidth-efficient for TPC-C (about 20%).

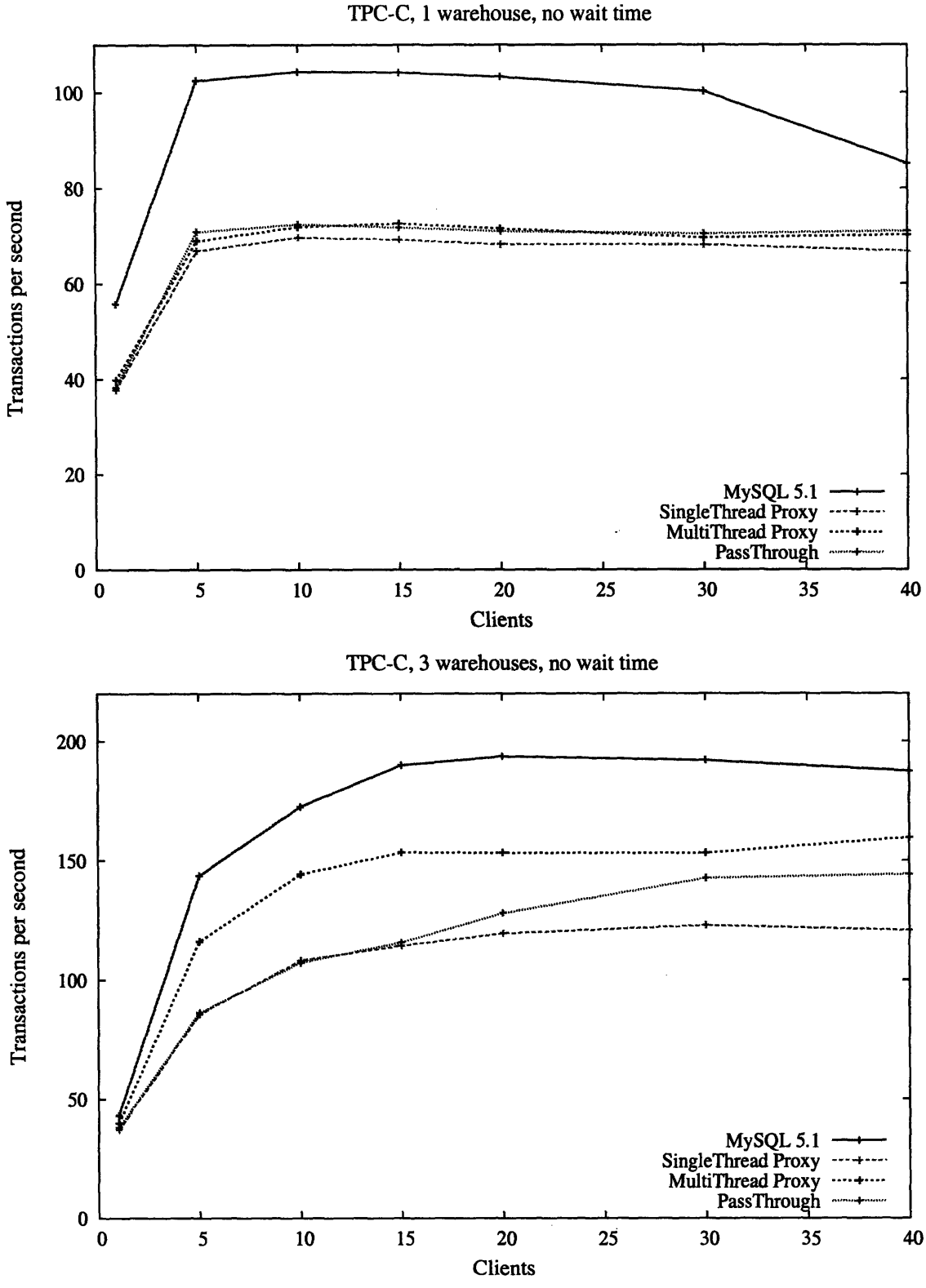


Figure 6-2: TPC-C with 1 and 3 Warehouses run through middleware (no replication)

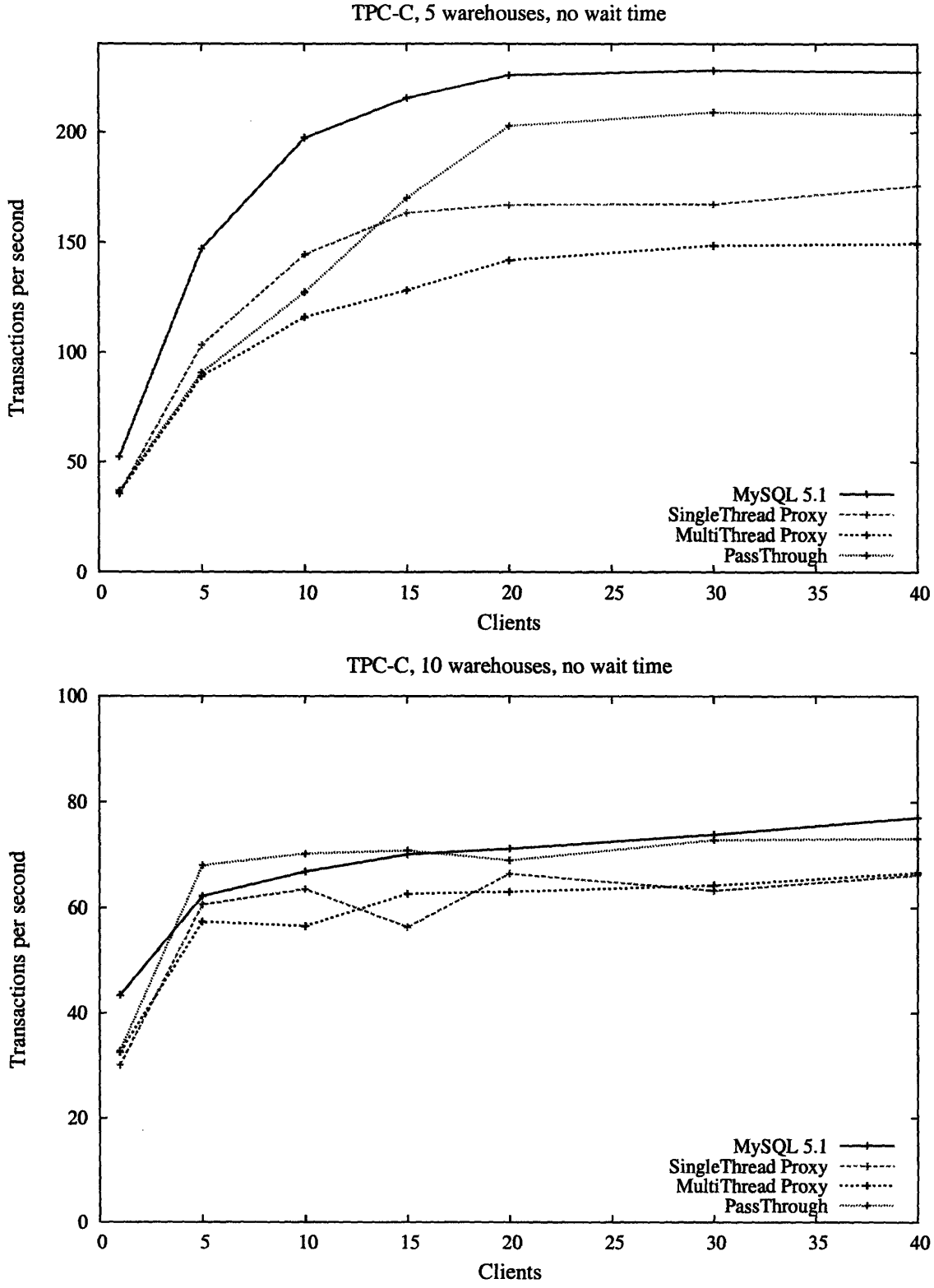


Figure 6-3: TPC-C with 5 and 10 Warehouses run through middleware (no replication)

Communication Overhead

Clients use our own JDBC driver to interact with HRDB or the PassThrough. Our JDBC wire format is independent of the component replica database interfaces. While each database provides a JDBC driver with a standard interface, the wire format used by the driver for query results is database-specific. Thus, query results must be unmarshaled in the shepherd and remarshaled to be sent to the client. This step is also necessary to store the result set from the primary for comparison with the result sets from the secondaries.

Our wire format compares favorably with the native MySQL JDBC driver. TPC-C run directly against MySQL averages 10.6KB I/O with the client per transaction (3.3KB sent, 7.3KB received). TPC-C run through the PassThrough averages 8.1KB I/O with the client per transaction (3.5KB sent, 4.6KB received). Our JDBC implementation sends less header information with query results, potentially due to our incomplete implementation of the JDBC specification. For this benchmark, our JDBC protocol averaged 20% less bytes exchanged with the client.

However, the above results are only half of the story: the Shepherd or Passthrough must also receive the results from the database via the MySQL JDBC driver. Thus, the overall bandwidth consumed by the PassThrough is 180% of running TPC-C directly against MySQL. Given the TPC-C consumption reported above, even 200 transactions per second through the PassThrough only consumes 4MB/second of the gigabit switch. The Shepherd running with 3 database replicas would consume 380% of the bandwidth, or a little less than 8MB/second, which is still a small fraction of the gigabit switch.

One obvious situation where 180% bandwidth overhead would cause issues is when a query generates much larger results. We measured the overhead for single operations with large results by running a `SELECT` query with a large result set directly against a MySQL5.1 database and also through the PassThrough. The query requests TPC-C customer data for a particular warehouse and by varying the number of districts we control how much data is returned by the query. Selecting one district worth of

customer data returns a result about 2 megabytes in size; selecting all 10 districts produces the expected 20 megabyte result. TPC-C's customer table is a representative sample because it has a good mix of data types.

System	Query Size	Time	Result Size	Total Bytes	MS per MB
MySQL5.1	1 district	177ms	1.63MB	1.63MB	108ms/MB
PassThru	1 district	404ms	1.83MB	3.46MB	116ms/MB
MySQL5.1	10 districts	2099ms	16.3MB	16.3MB	129ms/MB
PassThru	10 districts	4481ms	18.3MB	34.6MB	130ms/MB

Table 6.1: Performance implications of bandwidth overhead for large results.

By interposing a byte-counting proxy between the client and the database (or PassThrough), we can measure the total bytes sent by the various JDBC protocols. Table 6.1 shows the results of the experiment. As expected, the PassThrough exhibits half the performance of the stand-alone case, as the result must be sent twice: once to the PassThrough and then again to the client. The overall time is governed by total bytes sent (ms per MB of result is comparable). Our result set implementation is slightly less efficient than MySQL for large results (it takes 12% more bytes to send the same result). As is evident from the data from running TPC-C, our JDBC protocol is more efficient for small result sets: when selecting a single row from the customer table, MySQL's result size is 1951 bytes compared with PassThru's 1002 bytes, for a saving of 49%.

One optimization that we could apply, but have not implemented, is pipelined result relay. The JDBC protocol supports incremental fetching of result sets. By requesting the result in chunks and sending each chunk to the client as it arrives, we could hide some of the latency associated with large results. However, not all JDBC drivers support this component of the protocol.

6.2.2 HRDB Overhead

To measure the overhead associated with the HRDB shepherd, we compared running the TPC-C workload against a database directly to running it through the HRDB shepherd. We evaluate the performance of CBS relative to the other potential schemes

the Shepherd implements. The performance of the various schemes depends on the amount of contention in the workload. We produced a number of different contention scenarios by varying the number of TPC-C warehouses. For each scenario, we ran TPC-C directly against the database to determine the zero overhead performance, and also through the PassThrough to determine the maximum middleware performance. Figures 6-4, 6-5, and 6-6 show the results of these tests in comparison to the various concurrency control schemes implemented by the shepherd.

Assuming sufficient CPU and network bandwidth, the major performance difference between *CBS* and PassThrough is that CBS must wait for agreement at commit. The additional wait time adds to transaction duration, which reduces overall performance, especially for high contention workloads. The results for high contention workloads (Figure 6-4: 1 and 3 warehouses) show that the extra latency introduced by CBS at the commit point reduces performance 30-40%. In these cases, the PassThrough also performs badly; CBS is only 11-17% slower than the PassThrough. However, workloads without high contention (Figure 6-5,6-6: 10 and 30 warehouses) show CBS having only 10-15% overall performance loss.

The *Pronto* scheme results in longer wait times at commit than CBS, as it does not pipeline execution on the secondaries with execution on the primary. When Pronto reaches the commit point, it must wait for the all secondaries to execute the entire transaction. Furthermore, the secondaries execute transactions sequentially, ensuring that there is at least a full transaction execution latency between transaction commits on the coordinator. By contrast, CBS can commit batches of transactions at the same time; in some of the tests reaching an average batch size of 7 transactions. With high contention, the system is sensitive to small increases in latency. Our low contention cases occur when the database is I/O limited and transaction execution time becomes significant. Thus, Pronto performs worse than CBS under both high and low contention.

The performance of the *Explicit Partitioning* scheme is entirely dependent on the granularity of the partitioning. Our partitioning scheme uses warehouse as the partitions, acquiring all partition locks upfront in ascending warehouse order, so as to

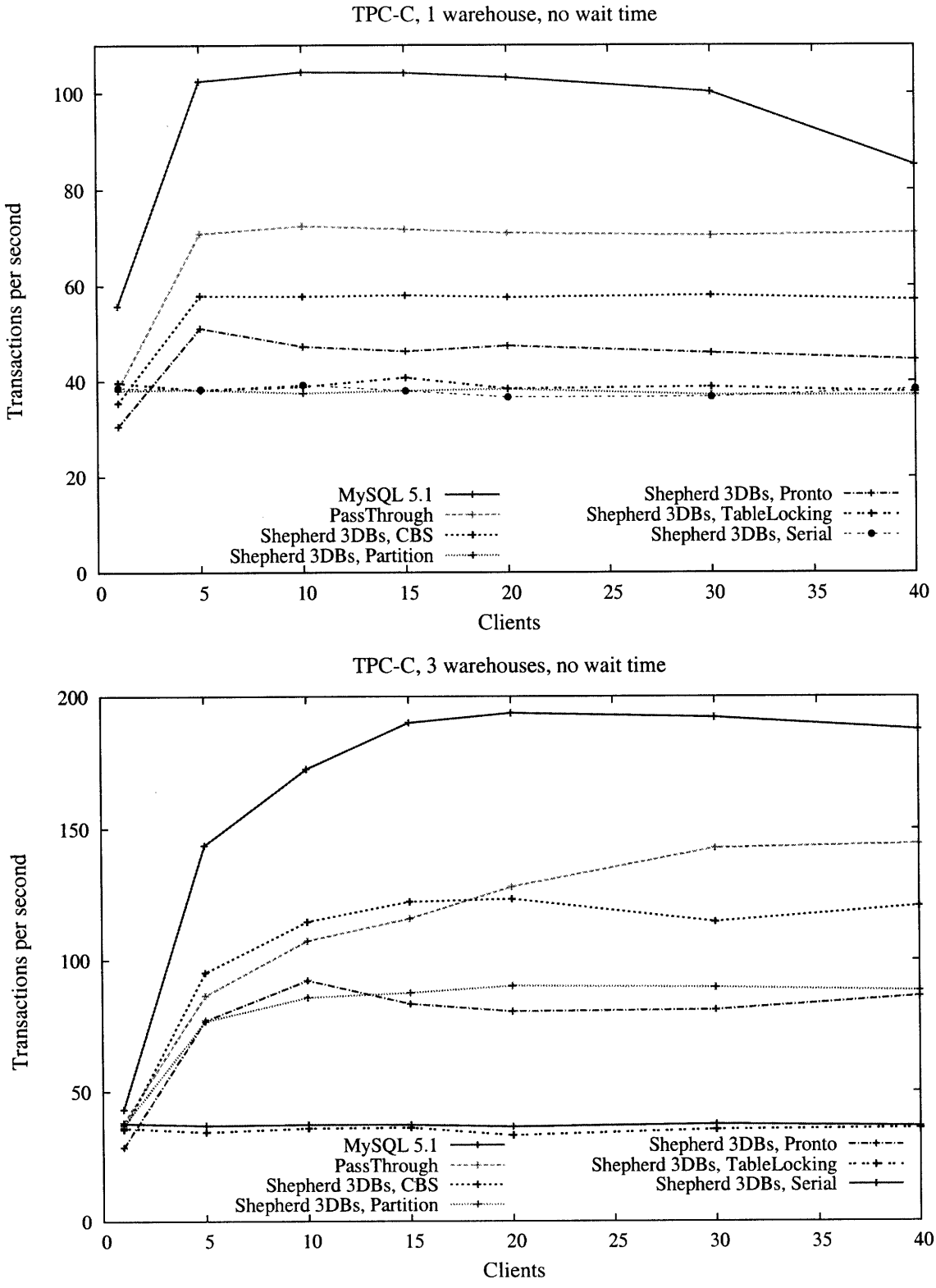


Figure 6-4: TPC-C with 1 and 3 Warehouses, comparing replication methods

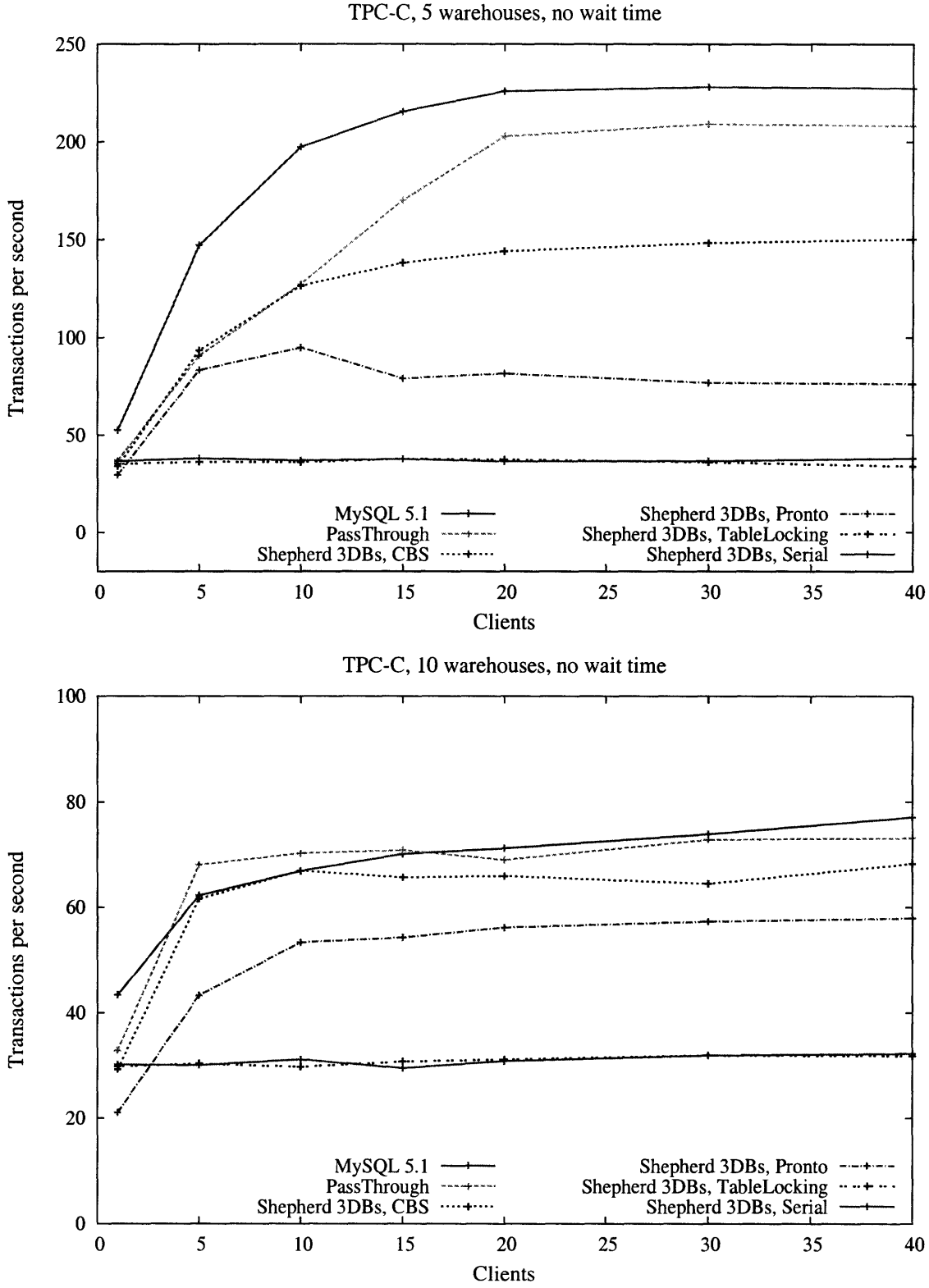


Figure 6-5: TPC-C with 5 and 10 Warehouses, comparing replication methods

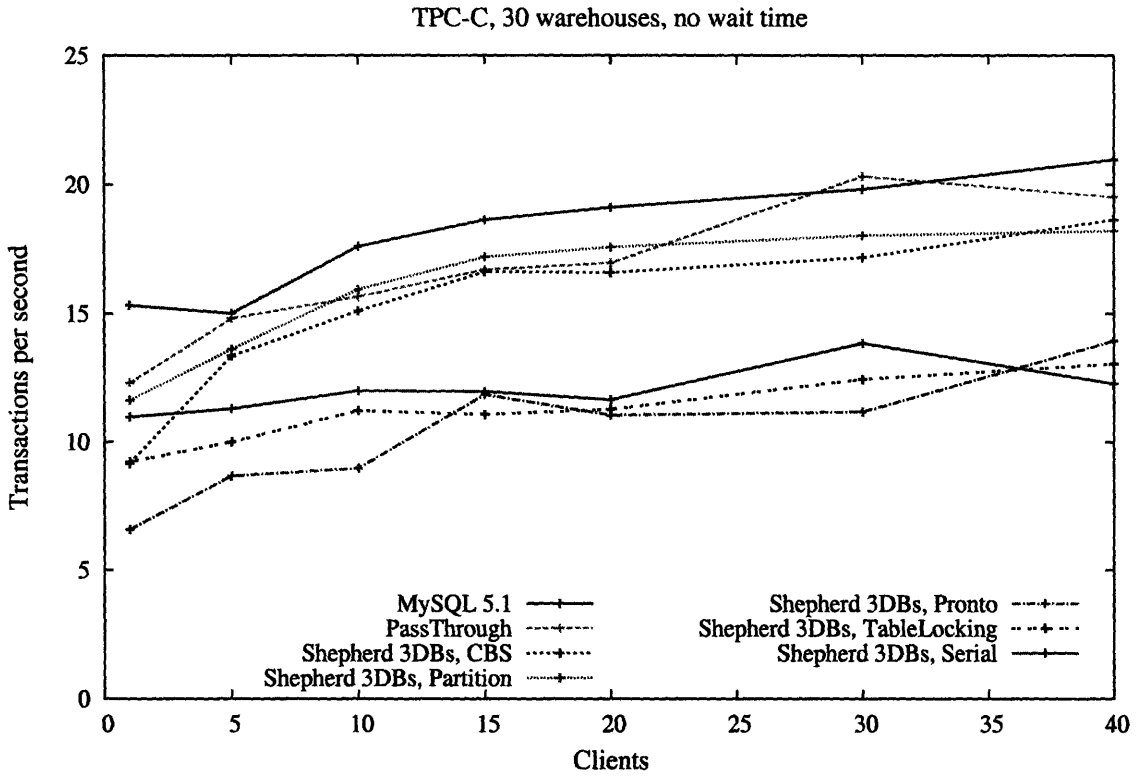


Figure 6-6: TPC-C with 30 warehouses, comparing replication methods

avoid deadlocks. With only 1 warehouse, the partitioning scheme is identical to serial. With 30 warehouses, the number of partitions is sufficient to keep the database busy and partitioning performs very well. Deadlocks between multi-partition transactions can significantly affect performance, but all partitions touched by a transaction are known upfront for the TPC-C workload, thus allowing deadlocks to be avoided. Multi-partition transactions make up about 11% of the workload.

Table Locking performs poorly on TPC-C because most transactions involve writes to the same table. Specifically, the New Order and Payment transactions, which account for 88% of the workload, both write to the `district` table. Thus, Table Locking performs only slightly better than serial execution. Our implementation of TPC-C carefully orders table accesses to minimize deadlocks, but we had to add an explicit table lock acquire to the `delivery` transaction due to an inherent cycle between `new_order` and `delivery` transactions. When the Table Locking scheme experiences deadlocks, its performance drops below that of Serial.

Serial execution restricts performance to that of a single client, regardless of how many clients are present in the test. In our experiments, this results in performance 2 and 5 times worse than a stand-alone database. In all but the highest contention test, CBS beats serial by a factor of 2 to 3.

In summary, Commit Barrier Scheduling performs well in comparison to both the PassThrough and alternative schemes for concurrency control. CBS matches or beats the performance of Explicit Partitioning without any help from the client. Improving the performance of the PassThrough will likely provide significant performance improvements for CBS as well.

6.2.3 Heterogeneous Replication

Now we demonstrate that HRDB is capable of using a heterogeneous replica set, and that a sufficiently blocking primary is a reasonable, but not necessary, assumption. We implemented the TPC-C workload in standard SQL, and thus were able to run it against several different vendor's database implementations. The databases used were MySQL 4.1.22-max and 5.1.16 with InnoDB tables, IBM DB2 V9.1, and Commercial Database X[1]. We also ran experiments with Apache Derby 10.1.3.1 (a Java-based database) but omit performance numbers here as it is much slower than the other systems. The minor SQL translation that HRDB provides makes this possible; however some queries of TPC-C were modified to take into account NULL ordering, timestamp differences, etc.

One issue that crops up is that databases assign different types to the results of computations, making the result sets difficult to compare. Our implementation coerces all database replica results into the types returned by the primary database. While the mechanism works well, a malicious primary could fool the shepherd into believing the replicas are in agreement by selecting types which map many values to few values. For example, returning a single bit results in all replica answers being coerced to a single bit, increasing the likelihood of acquiring $f+1$ matching responses.

We ran the test with 20 clients on a 10 (small) warehouse workload, with the ob-

[1]Database name omitted due to licensing constraints

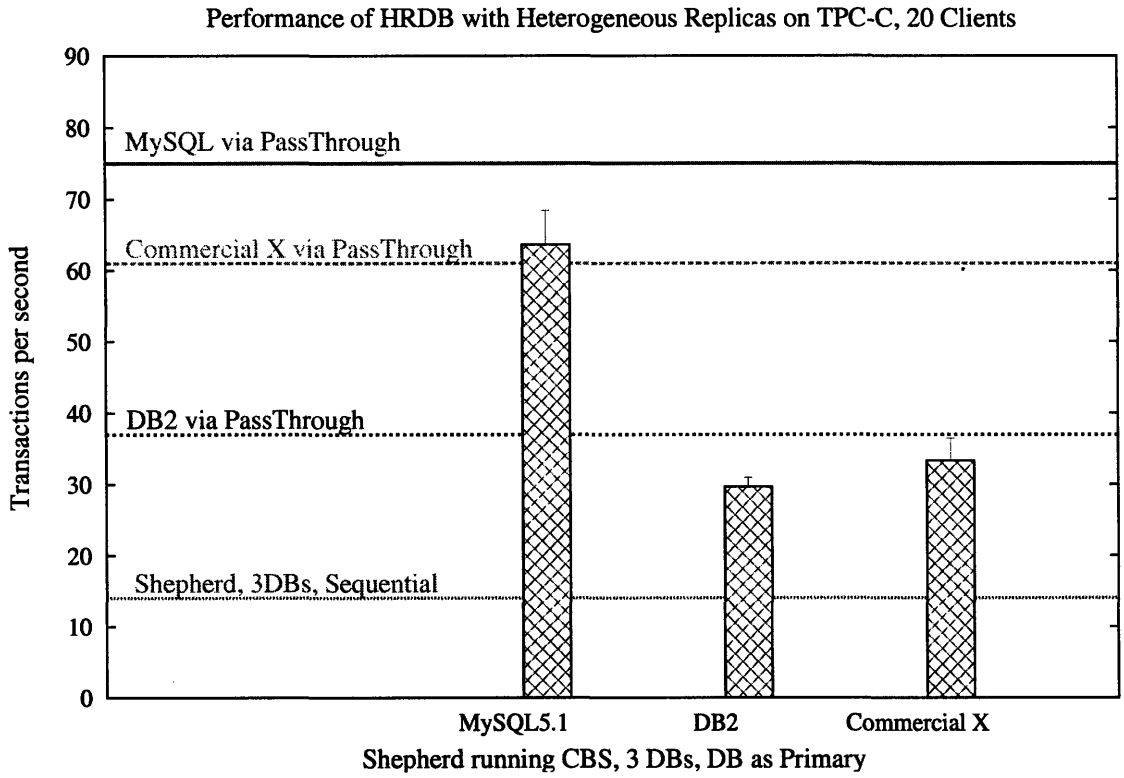


Figure 6-7: Performance of HRDB with a heterogeneous replica set (MySQL, DB2, Commercial Database X).

jective of demonstrating feasibility and relative performance of various configurations (not databases). The implementations of HRDB and TPC-C used in this section are slightly older than the ones used to produce the results in the previous section, so performance numbers do not transfer.

Figure 6-7 shows the throughput results for a number of configurations. The top three horizontal lines show the performance of running TPC-C against a particular database through the PassThrough. They indicate the maximum throughput achievable using that database replica. Each of the three columns shows the performance of running HRDB in a 3 database configuration (MySQL 5.1, DB2, Commercial Database X) with the labeled database as the primary database. The bottom horizontal line shows the performance of running HRDB in a 3 database configuration (MySQL 5.1, DB2, Commercial Database X) where transactions are executed sequentially. No matter which database is selected as the primary, CBS runs twice as fast as sequentially execution on this workload.

CBS runs at the speed of the primary, or the *slowest* of the $f+1$ *fastest* replicas, or whichever is slower. Furthermore, if the primary is not sufficiently blocking for one of the secondaries, that secondary will not be in the $f+1$ fastest replicas most of the time.

- **MySQL as primary** - The primary (MySQL) is faster than the other replicas, so HRDB runs at the speed of the slowest of the $f+1$ fastest replicas, which is Commercial Database X

- **DB2 as primary** - The primary (DB2) is slower than the other replicas, so HRDB runs at the speed of DB2.

- **Commercial Database X as primary** - The primary (Database X) is reasonably fast, but slower than MySQL, so HRDB should run at the speed of Commercial Database X. However, for one query, Commercial Database X does not appear to be sufficiently blocking for MySQL. With MySQL spending time untangling transaction ordering issues, the next fastest replica is DB2, so HRDB actually runs at the speed of DB2.

Sufficiently blocking should be a transitive property. That is, if Commercial Database X is sufficiently blocking for DB2, and DB2 is sufficiently blocking for MySQL, then Commercial Database X should be sufficiently blocking for MySQL. Intuitively, if DB2 allows two transactions to run in parallel with Commercial Database X as the primary, it should also allow those two transactions to run in parallel when it the primary. If MySQL blocks one of the transactions when Commercial Database X is the primary, it should also block one when DB2 is the primary. However, this does not appear to occur during that tests we ran. Our tests do demonstrate that sufficiently blocking holds in most cases and that the system successfully makes progress when it does not.

In conclusion, heterogeneous replication is feasible. Sufficiently blocking is a reasonable assumption, though it does not always hold. Finally, the system runs at the speed of the slowest of the $f+1$ fastest replicas, or the primary, whichever is slower.

6.2.4 Fail-stop faults

To demonstrate that HRDB survives fail-stop faulty replicas, we measured transaction throughput while one replica was crashed and restarted. We ran a 20 client TPC-C workload and recorded the timestamps at which each replica committed each transaction. Figure 6-8 shows a replica being crashed 76 seconds into the test and restarted 40 seconds later. In this time, the rest of the system continues executing because it can tolerate a faulty replica. When the faulty replica restarts, it is 500 transactions behind.

For the overall system, the performance impact of a crashed replica is negligible. When the replica restarts, it is able to catch up with the rest of the replicas because it does not execute any reads. For this workload, the slow replica was able to execute transactions at nearly three times the rate of the rest of the replicas (750 vs. 250 transactions in 20 seconds). This result demonstrates that transiently slow or rebooted replicas can catch up with the other replicas.

6.3 SES Implementation

Our second implementation of the HRDB shepherd uses Snapshot Epoch Scheduling (SES) to provide single-copy snapshot isolation. HRDB running SES uses only replicas that provide snapshot isolation; we show results with PostgreSQL. Our implementation of SES includes both the shepherd and a shim that runs co-resident with the database. Both entities are implemented in Java and together comprise about 7700 lines of code. We first present motivation and implementation details of the shim, then describe interesting implementation details of the shepherd.

6.3.1 Shim Implementation

As mentioned in Section 5.3.1, a malicious network can break network database connections, which the JDBC standard specifies as resulting in the database aborting any currently executing transaction associated with the connection. Once a database has

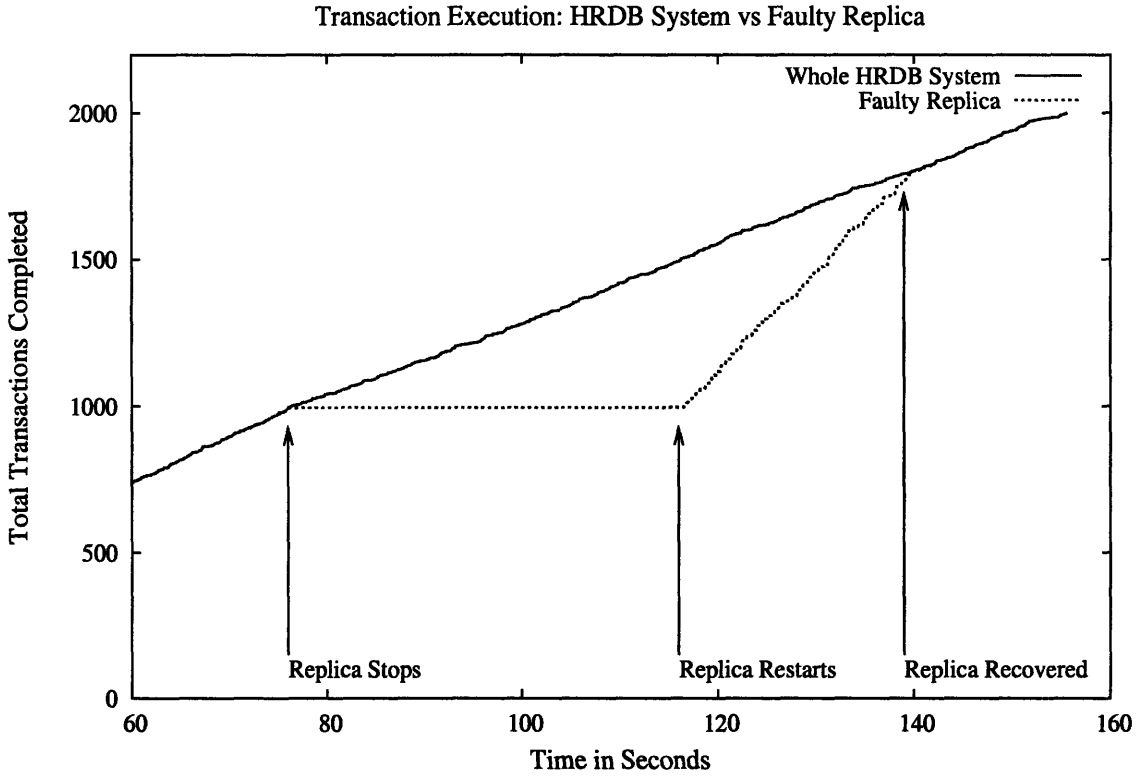


Figure 6-8: Transactions completed over time on TPC-C workload as one replica is crashed and restarted

aborted a transaction, it may be unable to re-execute it, allowing an attacker to prevent a transaction from acquiring the necessary quorum. The primary purpose of the shim is to maintain transactions over connection drops, but once we've placed some code on the database machine, adding some additional functionality is a reasonable proposition.

The shim runs on the same machine as the database and communicates with it via a local, secure, and reliable channel such as an OS socket. The shim, channel, and database all comprise one replica: failure of any of them counts as failure of the whole replica. The shim communicates with the database using the database's standard JDBC driver. In addition to relaying SQL, the shim also can snapshot transactions, extract writesets, and run the epoch scheduling algorithm. The shim snapshots a transaction by executing trivial read: `SELECT 1`. It performs writeset extraction by using an implementation specific mechanism (*e.g.*, hooking into the lower level of the JDBC driver and requesting the writeset). The shim maintains its

own *epoch*, *count*, and *counts[epoch]* values, so it can determine when to advance the epoch. It runs our wire protocol, which supports the operations shown in Table 6.2.

Name	Arguments	Operation
BEGIN	*tid [epoch]*	Begins and snapshots transaction with *tid*, reconnecting to tid if it already existed. If *epoch* is given, blocks snapshot until shim's epoch matches *epoch*.
STMT	*sid SQL*	executes SQL on replica and returns result, caching result. If the same statement id is requested again, returns the cached answer.
WRITESET		Extracts and returns the writeset for the current transaction.
COMMIT	*[epoch]*	Commits the current transaction. If *epoch* is given, blocks commit until shim's epoch matches *epoch*.
ROLLBACK		Rolls back the current transaction.
EPOCH	*epoch count*	Fills in an entry in the shim's *counts[epoch]* array.
ABORT	*tid*	Out of band abort mechanism; aborts transaction *tid* by closing the associated JDBC database connection.
STARTUP	*epoch*	Resets the shim to start in the given *epoch*. Aborts all in-progress transactions.

Table 6.2: Shim Protocol

6.3.2 SES Shepherd Implementation

The HRDB Shepherd running SES interacts with the database shims instead of directly with the databases. Due to the introduction of the shim, we control the interface that the replica manager uses to communicate with the database replica. While JDBC is blocking, our interface need not be, allowing the SES Shepherd to run many fewer threads than the CBS Shepherd. We use a hybrid approach, using a multi-threaded, blocking interface for the primary replica manager, and a non-blocking, event-driven interface for the secondary replica managers.

The replica managers need not enforce epoch scheduling because that functionality is present in the shim: the shim blocks snapshot and commit operations until the appropriate epoch. Each replica manager independently tracks *counts[epoch]*, relaying

new values to the shim as they become available. It does so by maintaining an additional connection to the shim over which is sends control messages like `STARTUP` and `EPOCH`. Once a shim has acknowledged the completion of all events in a particular epoch, the replica manager can discard the *counts*[*epoch*] value for the epoch.

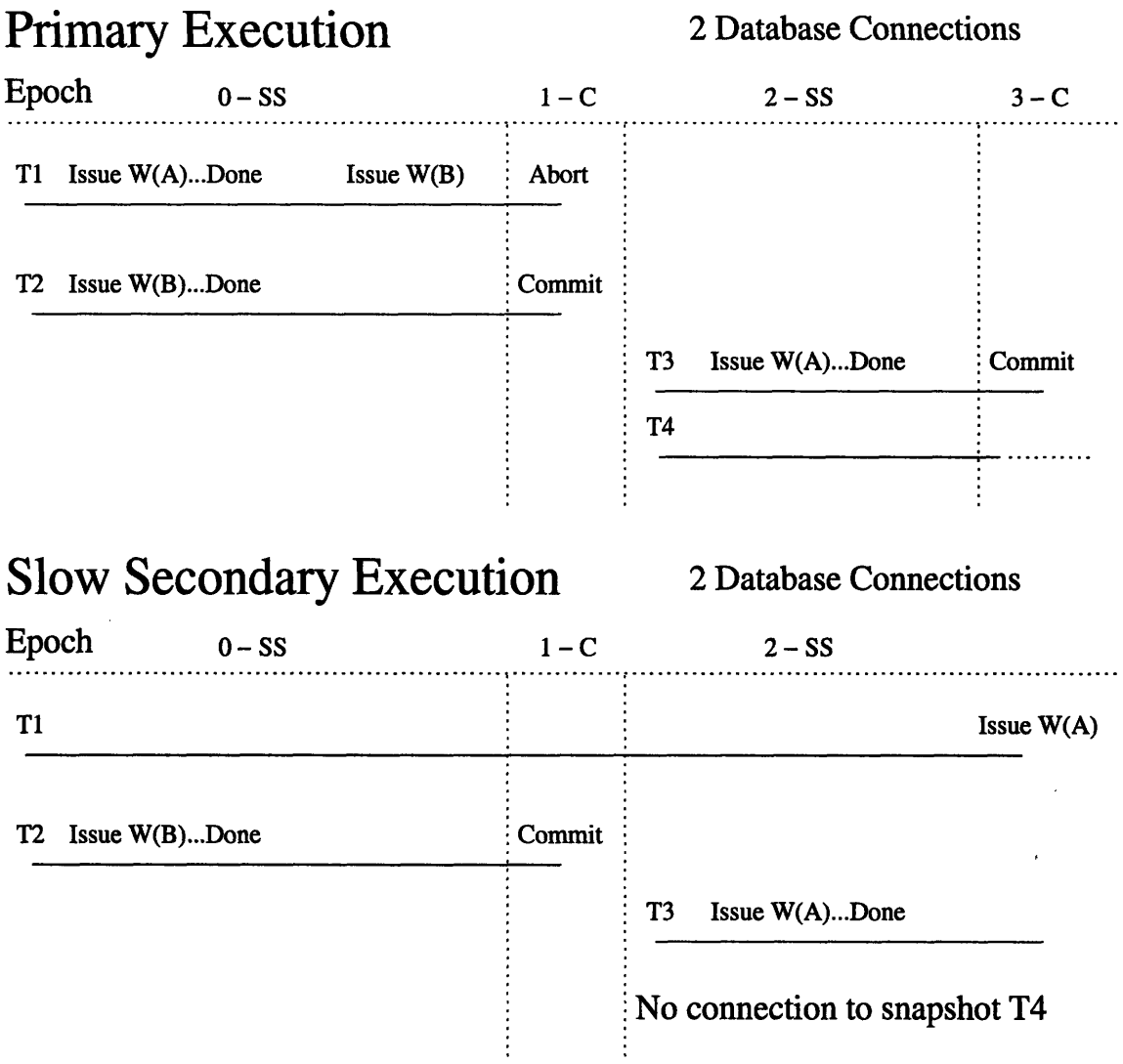


Figure 6-9: Slow secondary gets stuck. An aborted transaction consumes a database connection needed to acquire a snapshot.

The major implementation challenge for the SES shepherd has to do with aborted transactions. Figure 6-9 illustrates the scenario. The top half of the figure shows transaction execution the primary. Transactions T1 and T2 conflict: they both write to B. The primary decides that T2 wins the conflict, committing T2 and aborting T1. Subsequently, the primary starts executing T3 and T4. T3 would conflict with T1

(based on A, not B) if they had executed concurrently. T1 and T2 snapshot in epoch 0. T2 commits in epoch 1. T3 and T4 snapshot in epoch 2. T3 commits in epoch 3. The primary database uses 2 database connections to execute the transactions.

The lower half of Figure 6-9 shows how a slow secondary can get stuck. Two factors cause the problem: SES does not enforce an endpoint on aborted transactions and the shepherd has a limited number of database connections. The secondary is slow and has not realized that T1 has been aborted. Furthermore, it does not issue the W(A) from T1 until after it starts executing T3. At this point, the W(A) from T1 blocks because it conflicts with T3. However, the secondary cannot advance to epoch 3 and commit T3 until after T4 has acquired a snapshot in epoch 2. With both database connections in use, one consumed by T1 and one by T3, none remain available for T4.

There are two straightforward solutions to the problem: more connections and out-of-band abort. Opening another database connection will allow T4 to acquire a snapshot. Once T4 has acquired a snapshot, T3 can commit. When it does so, the database will abort T1 due to conflicting writes, resolving the problem. However, opening a new database connection is an expensive operation. Furthermore, while the situation in the figure only requires 1 additional connection, more complicated scenarios can require many more connections and most databases only support a limited number. The other solution is to force T1 to abort. Sending a rollback message over T1's database connection will not abort the transaction if the database is blocked on a statement and not listening to the connection. While JDBC includes a `cancel` operation for canceling executing SQL statements, drivers are not required to implement it. A solid mechanism for aborting the transaction is to close the associated JDBC connection. Closing and re-opening a JDBC connection is an expensive operation. Since the situation from Figure 6-9 can arise whenever a transaction is aborted, we need a minimally expensive solution.

We chose an escalating approach. Upon transaction abort, we first send an in-band rollback message. If the rollback message is not acknowledged quickly, we switch to an alternate connection. The shim retains several additional connections for this

purpose, making it highly likely that one is available for use. Finally, we start a timer on the shim. If the aborted transaction has not successfully aborted when the timer expires, the shim closes the JDBC connection associated with the aborted transaction. The progressive use of more expensive mechanism ensures that the common case is fast, but still provides eventual progress. The condition is particular to SES because in CBS a transaction that is *ready to commit* never waits for subsequent transactions to perform operations.

6.4 SES Performance Analysis

We present only limited performance results for SES, due to issues with writeset extraction and implementation challenges.

Writeset extraction is not a standard database function and thus requires a database extension. The Postgres-R project developed a modified version of PostgreSQL that supports writeset extraction. While we originally had planned on using Postgres-R, we found that it was too unstable to support our benchmarks. Thus, we augmented the `WRITESET` shim operation to take a size argument: the shim supplies a dummy writeset of the given size. Since writesets are only used during recovery, dummy writesets do not prevent non-faulty performance testing.

We use two workloads to test the shim and the SES shepherd. The first is TPC-C, the industry-standard transaction processing benchmark used to test CBS in the previous section. The other benchmark is one of our own design we call *writes*, where each transaction reads and updates 5 random rows of a 10,000 row table. The benchmark is CPU and write intensive, with low contention and I/O overhead.

While the throughput measurements acquired do not take into account the overhead of actual writeset extraction, they do include exchanges of similarly sized messages. We ran each class of benchmark transaction against Postgres-R, and had it perform writeset extraction. The average writeset sizes are shown in Table 6.3. None of these writesets is sizable compared to the bandwidth available.

Transaction Type		Writeset Size (bytes)
TPC-C	New Order	2704
TPC-C	Payment	788
TPC-C	Order Status	4
TPC-C	Delivery	1327 (per warehouse)
TPC-C	Stock Level	4
Writes		113

Table 6.3: Average writeset size, by transaction type.

6.4.1 Shim Performance

We implemented a JDBC wrapper for the shim's wire protocol so we can run benchmarks directly against the shim. The shim introduces an additional source of latency in transaction execution, albeit one that does not involve an extra network hop. Due to the inefficient implementation of the shim, it consumes significant computation resources, which can result in serious performance loss for computationally bound database workloads. Figure 6-10 shows the results of running the writes and TPC-C workloads against the shim. Since the writes test is CPU-limited, the shim significantly impacts performance. A better shim implementation would probably mitigate much, but not all, of the performance penalty. The shim performs well on TPC-C with 1 warehouse and low client count. Since TPC-C with 1 warehouse is contention-heavy, it is very sensitive to additional latency. Since the shim performs well, it does not introduce much additional latency.

6.4.2 SES Shepherd Performance

Figure 6-11 shows the performance of SES on the writes workload. The *Postgres 8.3 Shim* line shows the performance of running the test against the shim. The *Passthrough* line shows the performance when forwarding middleware is placed between the tester and the shim. The PassThrough does not perform replication, it merely forwards SQL to the shim and results back. By introducing additional operation latency, the PassThrough causes transactions to hold locks longer. The cost of the PassThrough beyond the shim is minimal (3% relative to the shim) due to the lack of contention in the workload.

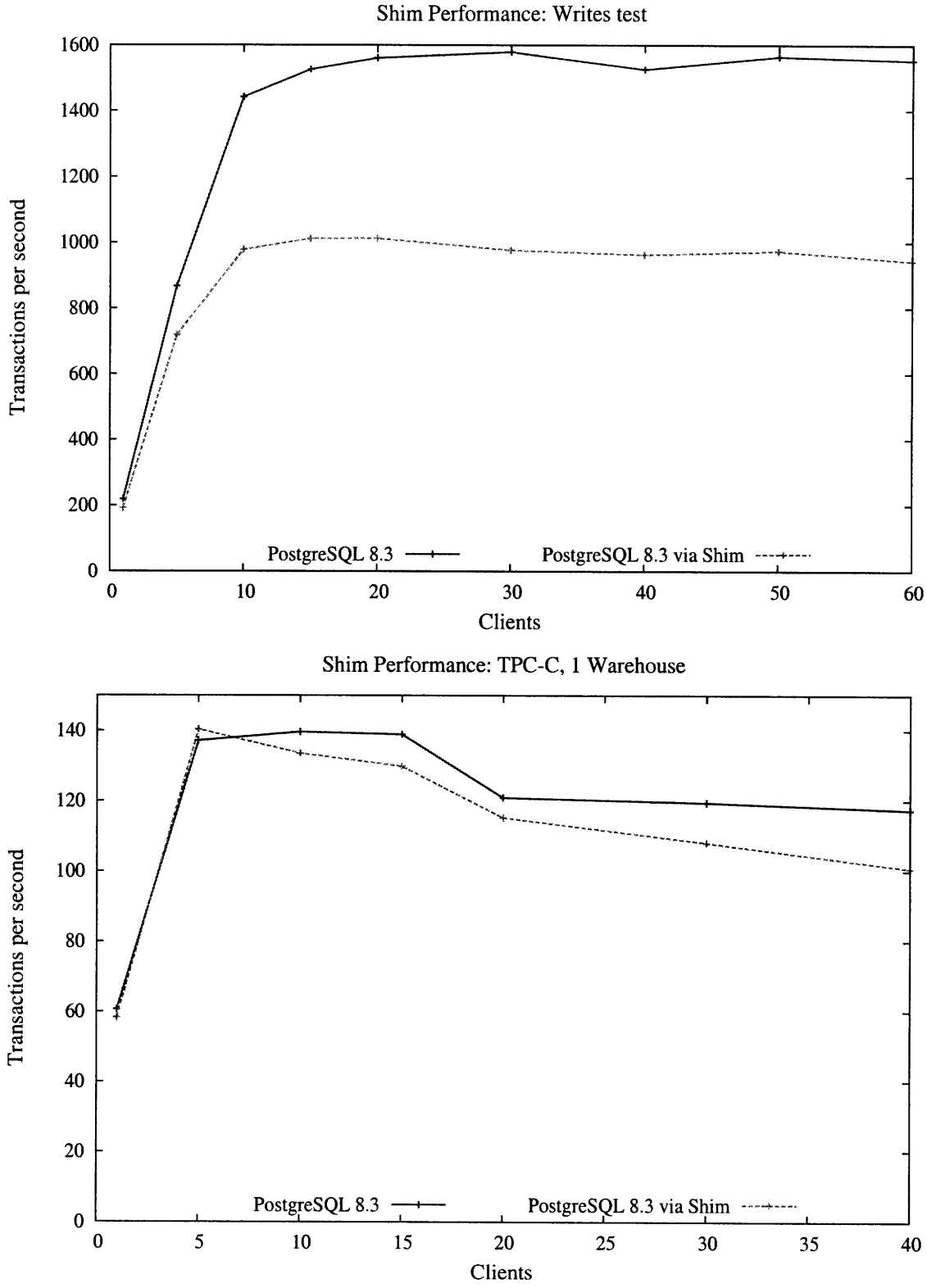


Figure 6-10: Performance of Shim on writes and TPC-C

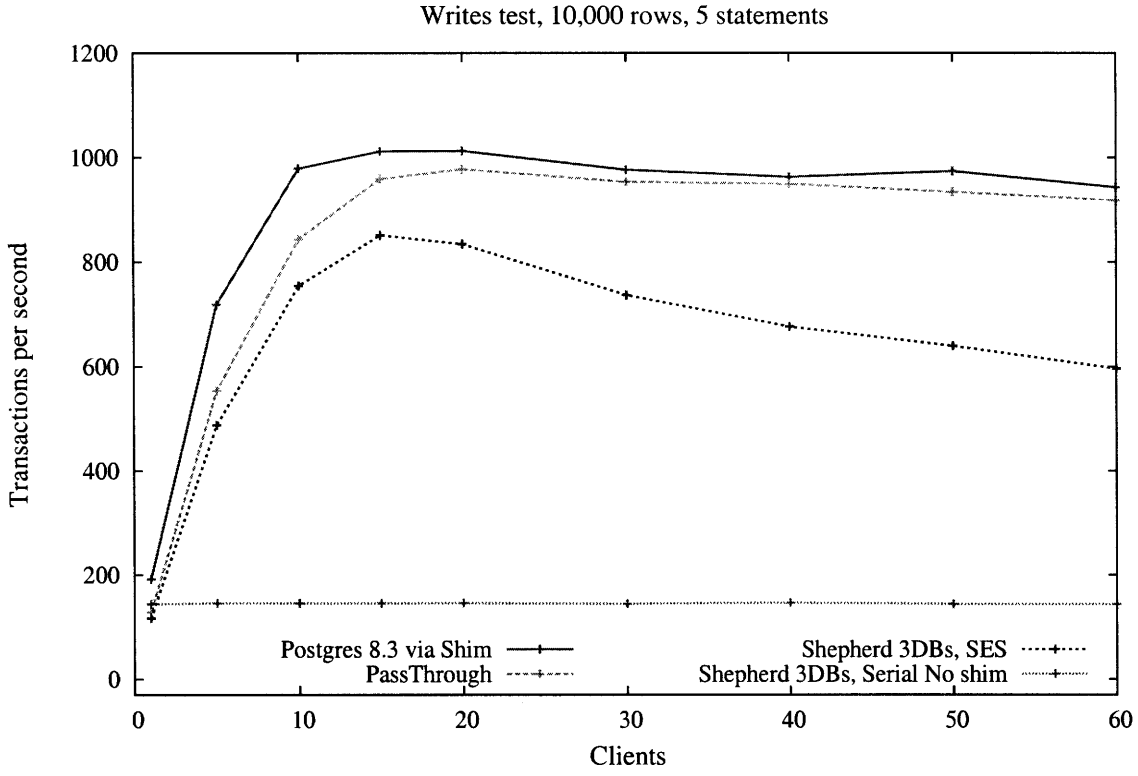


Figure 6-11: Performance of SES Shepherd on `writes` benchmark: each transaction updates 5 rows out of a 10,000 row table.

At 20 clients, *SES* running with 2 PostgreSQL 8.3 database replicas shows good relative performance: 15% performance loss relative to the PassThrough. A performance loss of 15% is similar to CBS running on a low contention workload. However, performance degrades as more clients are added. The performance degradation is due to epoch scheduling: additional clients put more pressure on the shepherd's scheduling algorithm. The scheduling algorithm used for these results is a simple greedy one: if the shepherd is currently in a snapshot epoch and it needs to assign a commit, it ends the snapshot epoch and begins a commit epoch. This approach results in epochs with small sizes (usually 1), which in turn results in less concurrency. A better epoch scheduler would likely resolve the issue.

For comparison, we also show the performance of *serial* execution on the shepherd. The shepherd need not use a shim because a transaction that gets aborted may be safely re-executed if no other transactions are executing on the system. Write-set extraction is similarly unnecessary: crashed replicas can be brought up to date

via transaction replay. Serial execution without a shim runs 85% slower than the PassThrough with a shim. Despite significant overhead, SES performs nearly a factor of 6 better than serial execution on this workload.

We do not show the performance of SES running TPC-C due to implementation issues. TPC-C exhibits the scenario described in Section 6.3.2. Our implementation is not robust enough to perform connection fail-over and thus fails to provide performance numbers for TPC-C with many clients.

6.5 Bug Tolerance and Discovery

To demonstrate HRDB's ability to tolerate non-fail-stop failures, we attempted to reproduce some of the bugs found in our study from Section 3.3.1. We also present details of a new bug in MySQL that was discovered by HRDB during performance testing.

6.5.1 Tolerance of Bugs with HRDB

We focused on bugs that could be triggered by specific SQL statements as they are easier to exhibit, reserving a more general study for future work. We note that we only attempted to reproduce a small number of bugs.

We successfully reproduced seven such bugs in the production versions of database systems that we used (see Figure 6-12). We ran our system with the bugs using the configurations shown in the figure and we were able to successfully mask the failure in each case. Some bugs used SQL that was supported by all vendors; we were able to mask these bugs using databases from different vendors, since none of them appeared across vendors. Other bugs involved vendor-specific syntax and thus were applicable only to certain databases. We were able to mask bugs in MySQL 4.1.16 using MySQL 5.1.11, and to mask bugs in MySQL 5.1.11 using MySQL 4.1.16, demonstrating the utility of our system even within different versions of the same database.

Bug	Description	Faults	No Fault
MySQL #21904	*Parser problem with IN() subqueries*	MySQL 4.1, 5.1	DB2, Derby
DB2 #JR22690	*Query optimizer optimizes away things it shouldn't*	DB2 V9.0	MySQL 4.1
Derby #1574	*UPDATE with COALESCE and subquery*	Derby 10.1.3	MySQL, DB2
MySQL #7320	*Aggregates not usable where they should be*	MySQL 4.1	MySQL 5.1
MySQL #13033	*INNER JOIN and nested RIGHT OUTER JOIN*	MySQL 4.1	MySQL 5.1
MySQL #24342	*MERGE tables use wrong value when checking keys*	MySQL 5.1	MySQL 4.1
MySQL #19667	*GROUP BY containing cast to DECIMAL*	MySQL 5.1	MySQL 4.1

Figure 6-12: Bugs we reproduced and masked with HRDB.

6.5.2 Discovery of Bugs using HRDB

While producing preliminary performance measurements of CBS overhead, we noticed that when running HRDB with 3 identical MySQL replicas, the primary was producing answers that did not match the secondaries' answers. Upon further investigation, we found that MySQL was allowing phantom reads: a query in transaction T1 was seeing inserts from transaction T2 despite T2 being ordered after T1. Isolation level SERIALIZABLE should not permit phantom reads. Since HRDB was running with 3 homogeneous MySQL 5.1.16 replicas, the system suffered correlated failures: both secondary replicas produced the incorrect result and the primary's answer was voted down. Because the primary's answer was deemed wrong, HRDB aborted the transaction, effectively turning a Byzantine fault into a fail-stop fault.

We submitted the details of the case to the MySQL development team, and they quickly verified that it was a bug (MySQL #27197). MySQL failed to acquire the correct locks in `SELECT` queries that use an `ORDER BY ... DESC`. The bug affected every version of MySQL since 3.23; remaining present for 4 years. From looking at the MySQL bug database, it is possible that someone noticed the issues previously but their report was dismissed as unreproducible. It has been patched in MySQL 5

(Version 5.1.20 and onward lack the bug). The results from Section 6.2.2 use MySQL 5.1.23, so as to avoid the bug.

It is surprising that a serious bug in the MySQL locking code remained unnoticed for 4 years, despite being exhibited by a standard benchmark query. The key insight is that the bug is a concurrency bug: it only appears with multiple clients. Most database test suites are single-threaded, as otherwise the test does not know what the correct answers should be. Since HRDB uses voting to determine correct answers and supports concurrent execution, HRDB is admirably suited to finding concurrency bugs in databases.

Chapter 7

Repairing Byzantine Replicas

Our system must repair Byzantine faulty replicas or eventually more than f replicas will be faulty. The two major issues are how and when to repair a replica. The process by which the shepherd repairs faulty replica state must be efficient because databases can be very large. The task is further complicated by the use of heterogeneous replicas, which make solutions involving standard interfaces preferable. Finally, a repair mechanism should also be transactional: it should restore the replica to a state that is identical to an observable state of the overall system.

The shepherd initiates a repair process when it has cause to suspect a replica of being Byzantine-faulty. The shepherd suspects a replica of being faulty when it produces incorrect answers or writesets, regularly misorders transactions, or fails to adhere to the database protocol. However, some faults are silent: the state of the replica gets corrupted, yet does not result in incorrect answers. The shepherd can proactively query replicas to detect silent faults.

We rely on an exterior mechanism to repair faults in the replica software or hardware. Assuming that an external mechanism ensures that the replica is functionally correct, the shepherd must repair the replica's state. The objective of state repair is to make the replica's state consistent with that of the whole replica set.

In this chapter, we begin by presenting a study on efficient "compare and repair" mechanisms. Armed with these tools, we then address how they are applied in the context of HRDB.

7.1 Compare and Repair Mechanisms

A compare and repair mechanism reduces the overhead of correcting a replica's state by only transferring data for the parts of the replica's state that are incorrect. The compare operation expends computation and bandwidth usage to isolate differences between the faulty state and the correct state. Obviously, a compare and repair mechanism provides the most benefit over replacing the entire state of the faulty database when the amount of corruption (difference between the faulty and correct states) is minimal. We believe it is reasonable to assume low corruption, *i.e.,* that faults cause small, localized parts of a database's state to become incorrect. While counterexamples exist, we believe them to be infrequent and thus optimize for the common case. For this chapter, we assume minimal corruption.

Comparing a pair of mostly-identical databases is not an operation exclusive to the shepherd; a number of commercial [33] and open source [8] tools have been created to address the issue. The shepherd does not have any special advantages and has a number of restrictions: it can only use SQL, cannot assume that replicas lay out data similarly, and bugs may cause changes that bypass triggers or other accounting mechanisms on the database. In this section, we describe and evaluate two compare and repair algorithms that adhere to the restrictions imposed by the shepherd.

To simplify the presentation, we consider the problem of attempting to repair a potentially faulty replica's data with a known correct replica's data. No operations other than the repair operation are executing on either replica. The data is made up of *records* from a single table, each of which has a unique *ID* (*e.g.,* primary key). These records are stored on the replica in *pages*, likely sorted by ID. We assume that the number of records, n, is large, that the faulty replica only differs by a minimal amount, and that n is essentially identical for the two replicas.

Our model (shown in Figure 7-1) is one where the faulty replica and the correct replica communicate with a *comparator* over the network. The comparator may ask to be sent records from either replica, or it may ask for aggregate hashes of sets of records. The comparator may issue inserts, updates, and deletes to the faulty replica

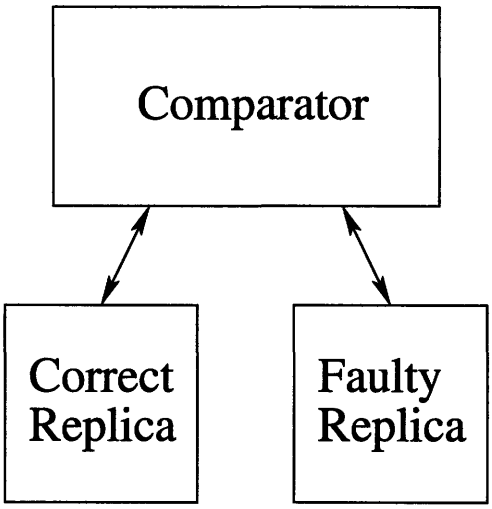


Figure 7-1: Architecture of Compare and Repair mechanism

to make its data match that of the correct replica. Finally, the comparator may use temporary tables on the correct replica to cache information and simplify data extraction. Upon completion, the comparator ensures, with high probability, that the data now stored on the faulty replica matches the data stored on the correct replica. The objective of this work is to develop and evaluate algorithms that are efficient in terms of completion time, computation at the replicas and comparator, and network bandwidth utilized.

The straw-man algorithm, which we will call *table-copy*, is one in which the comparator does a complete data transfer. The comparator instructs the faulty replica to delete its whole data set while requesting that the correct replica send it the whole correct data set. The comparator then inserts the correct data set into the faulty replica. The table-copy algorithm does one pass over the correct data, but sends and inserts the entire data set.

First, we explain how hashing can enable efficient compare and repair mechanisms. Then we discuss two hashing schemes: one from the literature and one we developed. Finally, we show a performance analysis of the strengths of each mechanism.

7.1.1 Computing Summaries: Hashing

Table-copy is very inefficient because it transfers many more records than actually differ on the faulty replica. A correct repair algorithm must transfer *at least* the records that differ, but may transfer more. Efficient repair works by comparing summaries to isolate records that actually differ and transferring only those records. Hashing provides a summary mechanism: if two hashes differ, then, with high probability, the data used to produce them also differs. Conversely, if two hashes match, then, with high probability, the data used to produce them also matches. Hash collisions can result in failure to correctly identify differing records, but the probability of false positives can be made small by choosing a hash with a reasonable number of bits in the output.

As an example, the simplest hashing mechanism is one in which each record is hashed individually. Both replicas hash all records and send the *<ID,hash>* pairs to comparator, which compares them in order of ID. The comparator logs all record IDs whose hashes do not match. The comparator then requests these IDs from the correct replica while deleting records with these IDs from the faulty replica. Finally, the coordinator inserts the records retrieved from the correct replica into the faulty replica.

Compared to table-copy, the simple hashing algorithm does more computation, but consumes less bandwidth. The hashing algorithm does two passes over the data: one to compute the hashes, and a second to extract/insert differing records. The only records that are updated are those that actually differ. For records of reasonable size, the hash is much smaller than the record, which results in transferring less data than table-copy. However, the bandwidth requirement of sending a hash for every record is significant.

7.1.2 Coelho Algorithm

As described in [8] and implemented in PostgreSQL's `pg_comparator`, the Coelho algorithm recursively builds a tree of hashes on each database, then compares the

trees to isolate the records that need to be transferred. The Coelho algorithm in [8] is a not a repair algorithm: it only identifies what rows are different. We extended it to transfer data to make the two tables identical. The extended algorithm has three phases:

- **Build Tree** - The first phase starts by producing a temporary table containing the row's primary key, a hash of said key, and a hash of the row contents (including primary key). Then, it uses the bits of the key hashes to group rows into a tree of aggregate hashes. The algorithm takes a *branching factor* parameter that determines the splay of the tree by controlling how many bits are used at each level of the tree. The height of the tree is determined by measuring the size of the table, and selecting the largest power of the branching factor less than the table size. The Build Tree phase is done on both databases.

- **Compare Trees** - Starting at the root of the tree, the comparator fetches hashes from each database, compares them, and where they differ, the sub-trees are fetched and explored to discover the exact differences. It is possible to arrive at a non-leaf sub-tree that is present on one database but not present on the other. Depending on which database is lacking the sub-tree, either the entire contents of the sub-tree should be transferred, or the entire contents should be deleted.

- **Transfer Rows** - The first step of this phase is to enumerate the ids of rows that must be transferred into a temporary table on the source database. Then, a join of this table to the original table at the correct database produces the contents of all necessary rows, which are then transferred into the faulty database.

Coelho makes two important design decisions: materializing the hash tree, and choosing its primary key to be the hash of the actual table primary key. Materializing the tree requires an amount of storage space linear in the size of the table. The size of a hash relative to the size of a row matters because each leaf of the tree stores a hash of a record, a hash of the record's primary key, and the record's primary

key. For large tables, the temporary tables will not fit in memory and will require disk access. Furthermore, if the number of differences is small, much of the table will never be fetched by the comparator, thus resulting in wasted computation and storage. However, once the tree is built, Coelho can be very efficient (in both time and bandwidth) at isolating differences.

Choosing to hash the primary key provides the benefit of generality at the cost of performance. Coelho can ignore the type of the primary key because hashes can be computed from any data type. It may also safely ignore the distribution of the primary key values because a good hash function will evenly distribute hashed values over the output space. Coelho depends on this even distribution to build a balanced tree. However, grouping records by the hash of the primary key, effectively randomizing record ordering, ensures that Coelho cannot take advantage of the database organization. For example, if a fault corrupts records on a single page, the faulty records are likely to be distributed evenly over the tree, rather than isolated to a particular branch.

7.1.3 N-Round-X-Row-Hash

We developed an algorithm that does not materialize a hash tree, assuming that with a small number of differences, most of the tree would never be used. We begin by producing high level summaries, hashing many records at a time. When the hashes do not match, we produce a new set of more refined summaries from the set of records whose summary hashes did not match. Without a materialized tree, we must have an efficient mechanism for refining a summary. Our algorithm uses the database primary key for efficient re-grouping.

N-Round-X-Row-Hash runs N rounds of computing hash aggregates and then refining where the hashes do not match. Assuming for the moment that the records have numeric IDs that range from 1 to n, on the first round N-Round-X-Row-Hash groups records into buckets based on ID ranges of size X_1 ($1 \ldots X_1$, $X_1 + 1 \ldots 2X_1$, etc). The contents of each bucket is hashed and sent to the comparator. Hashes that do not match imply that at least one record in the bucket differs. In the next

round, N-Round-X-Row-Hash sub-divides buckets whose hashes did not match in the previous round into buckets of size X_2, once again based on contiguous ranges of IDs. The algorithm runs for N rounds, using bucket size X_i on round i. On the last round, N-Round-X-Row-Hash transfers all records in all buckets whose hashes do not match. Thus, the bucket size on round N, X_N, governs the amount of waste: the algorithm will transfer records in chunks of size X_N.

The optimal choice for N and the X_i's is dependent on the number of records, n, the bandwidth cost of a bucket hash compared to the size of a record, H, and the probability that a record differs, P (a measure of corruption). Additional rounds with smaller bucket size trade off additional passes over the records for less bandwidth and waste (unnecessary records transferred). With a well-provisioned network, it may be faster to accept more waste to avoid scanning the records again. For the workloads we have tested, 2 to 3 rounds suffice.

If the bucket hashes are close to the size of the records, the X_i's must be larger to amortize the cost of the hashes. For some tables (*e.g.,* many-to-many mapping tables), the hash may be as large as the record, resulting in no bandwidth savings for single records. However, tables with small record size are not often the biggest tables. In addition to the hash itself, H incorporates the cost of bucket identifiers, both when they accompany the hash on the way to the comparator and when they are sent back to the database to identify records for the subsequent round. Our test workload has big records; an individual hash is 2% the size of the record and the cost per bucket hash is closer to 4% the size of the record.

The probability that a bucket hash does not match depends directly on the probability a record differs. With more differing records, the bucket sizes must be smaller to recover equivalent information about which records differ. For example, if a record differs with probability 1 in a 1000, then buckets of size 100 are likely to match, allowing the comparator to rule out many records. However, if a record differs with probability 1 in 10, then buckets of size 100 are almost guaranteed not to match, preventing the comparator from ruling out any records. In successive rounds, the probability that a record differs has a lower-bound of one over the bucket size in the

previous round ($1/x_{i-1}$). At least 1 record in the previous round's bucket must differ to cause the hash not to match, and thus for the record to be considered in the subsequent round. Since the algorithm becomes less efficient with increasing differences, it gets diminishing returns as the bucket size dwindles in successive rounds.

We developed a model for the approximate bandwidth consumed by N-Round-X-Row-Hash as compared to table-copy, given N, X, H, and P. H is predictable for a given table but P is not: the system must assume a value for the model to work. By comparing N-Round-X-Row-Hash to table-copy, we can factor out the number of records and focus on speedup. The model is recursive because the fraction of buckets whose hashes do not match in round i determines the cost of round $i+1$. The formula is:

$$F(N,X,H,P) = \frac{H}{X_1} + (1-(1-P)^{X_1})G(N,X,H,\max(P,\frac{1}{X_1}),2)$$

$$G(N,X,H,P,i) = \begin{cases} \frac{H}{X_i} + (1-(1-P)^{X_i})G(N,X,H,\max(P,\frac{1}{X_i}),i+1) & \text{if } i \leq N \\ 1 & \text{otherwise} \end{cases}$$

Using the primary key to perform bucketing has some challenges. Since the record ID may be *sparse*, N-Round-X-Row-Hash requests ranges of IDs whose expected number of rows is X_i. For simplicity, N-Round-X-Row-Hash assumes an even distribution and computes the range factor by querying the record count and spread (minimal and maximal record IDs). Multi-column and non-integer primary keys are a subject for future work.

7.1.4 Results

We present details of our implementation and tests, then analyze performance results for Coelho and N-round-X-row-hash separately. Finally, we compare the two algorithms, including testing them in a couple of alternative environments.

Implementation

We implemented the table-copy, N-round-X-row-hash, and Coelho algorithms in Java using JDBC. The two interesting implementation details are how the hashes are computed and how the needed rows are extracted:

- Row hashes are computed by concatenating all the columns together, applying SHA1, and taking the top 64 bits. Aggregate hashes are computed by XOR'ing hashes together.

- When the coordinator has determined which row ranges need to be copied, it creates a temporary table on the correct replica and inserts all the IDs of the rows to be copied. A simple join operation produces all the necessary rows in a single SQL statement.

We measured the time spent and bandwidth consumed by the various repair algorithms. Due to the foibles of Java, the bandwidth consumption was measured by redirecting the JDBC database connections through a byte-counting proxy. While this should degrade the time measurement slightly, it shouldn't affect the bandwidth utilization. The results obtained include all the SQL and JDBC overhead, but do not take into account network level phenomena.

The experiments were run on three machines: two machines running MySQL 5.1 databases, and a comparator machine running the algorithm. The machines are connected by 100MB/s switched Ethernet. The table used was the `customer` table from the TPC-C benchmark, adjusted to use only a single column primary key. We ran the experiments with a number of table sizes, varying from 10,000 rows to 1.3 million rows. A 256K row table is approximately 200M in size, and a 1.3M row table is 1GB in size. The hashes are approximately 2% the size of the row data. For a baseline, doing a TableCopy of the 200M table takes 4.3 minutes and consumes 448 megabytes of bandwidth, while the 1GB table takes 20min to copy and consumes 2.38 gigabytes of bandwidth.

The data corruption tool expresses corruption as the chance that any particular row is incorrect. The tool picks random record IDs until it has the appropriate number

of rows. Then it regenerates the contents of these rows. Thus, the contents are likely to be wildly different from the original row. The amount of difference does not matter to a good hashing function and we aren't considering schemes that attempt to repair partial records.

Analysis of Coelho

Figure 7-2 demonstrates that the Coelho algorithm can be very efficient in bandwidth utilization and does well even when the corruption is high. The performance has a number of interesting features, depending on the branching factor, b, and table size, n. The timing results break down overall execution time into phases. During the *Build Base* phase, the database creates the table of leaves by hashing the contents of each row of the table. The resulting temporary table is 6% the size of the `customer` table. The *Build Base* phase dominates the execution time, implying that the Coelho algorithm is compute and disk I/O bound (it uses 96% of the CPU during this phase). The bandwidth cost of sending the single SQL statement for the phase is negligible.

The database computes and stores the rest of the tree during the *Build Tree* phase. Together, the upper levels of the tree consume less space than the leaf table. Building the tree requires sending a small number of SQL statements; no data is returned. While bandwidth consumed during the *Build Tree* phase is proportional to $\log(n)/b$, the constant factor is small enough to render the bandwidth cost of the phase negligible.

The *Compare Tree* phase is when Coelho compares replica hash trees. The final two phases, *Send IDs* and *Transfer*, are where the algorithm sends the ids of rows to be transferred and performs the transfer. The same rows must be transferred, regardless of how they were discovered. Thus, the variability of the overall bandwidth comes from the *Compare Tree* stage. The results bear this out, as the largest *Compare Tree* time corresponds to the highest bandwidth utilization.

As the branching factor, b, increases, the bandwidth utilization grows linearly, with breakpoints at the square root and cube root of the table size (in rows). Assuming that the number of corrupted rows, C, is small relative to the table size, the

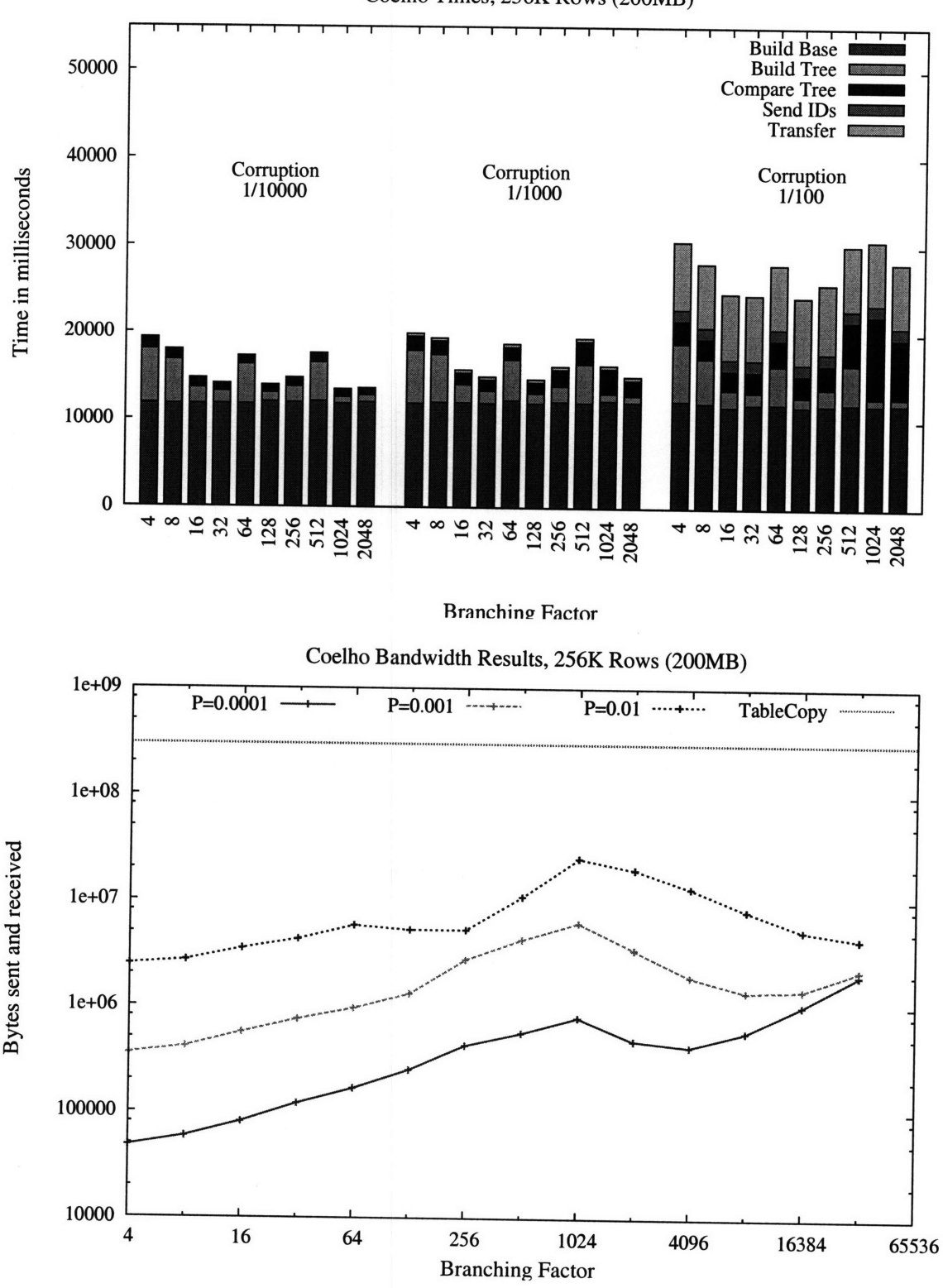


Figure 7-2: Results for recursive hashing Coelho algorithm. The amount of corruption, P, is the probability that any given row is incorrect.

penultimate examination of the hash tree will result in C non-matching hashes. The last examination of the tree will thus result in $C \times b$ leaves fetched, resulting in a linear bandwidth scale up. Over the square root break-point, the leaves of the tree will not be full, resulting in a smaller last fetch. As the branching factor continues to increase, Coelho begins to act more like N-round-X-row-hash.

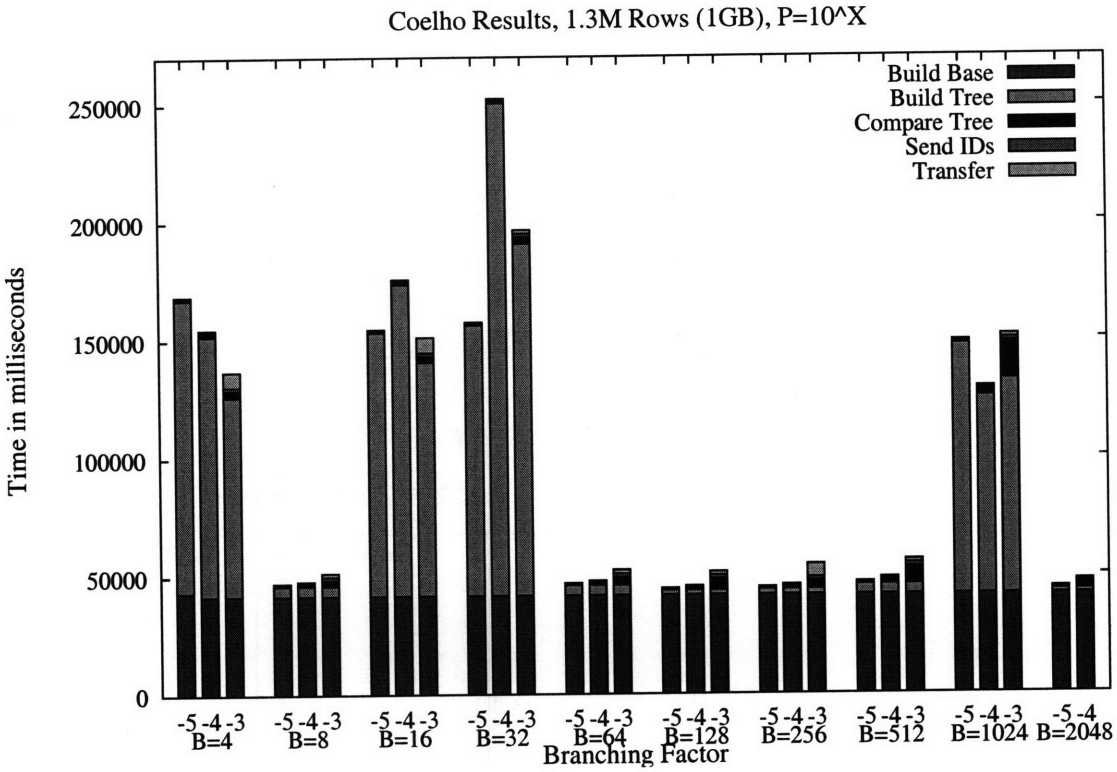


Figure 7-3: Coelho scaled up to run against a 1GB table, varying branching factor (B) and probability a row is incorrect (10^{-5},10^{-4},10^{-3}). The peaks are where the tree leaves do not fit in memory.

Coelho pays a performance cost for materializing the tree. The cost is most noticeable in the case where the tree does not fit in memory, as can occur when the algorithm is scaled up to a 1.3 million row table (shown in Figure 7-3). The high peaks are caused by the *Build Tree* phase and occur at branching factors 2^2, 2^4, 2^5, and 2^{10}. Each of these problematic branching factors turn out to be a factor of 2^{20}, which is slightly less than the actual number of rows in the table. Thus, the *Build Tree* phase will build a tree with a million leaves and it likely spills out of memory while doing so. Since the assignment of rows to leaves is done with an unpredictable `GROUP BY`,

this results in random disk accesses, which account for the abysmal performance. For very large tables, the branching factor must be chosen so that the first level of the tree fits in memory, otherwise catastrophic performance loss results from the spilling of the `GROUP BY` buckets to disk.

The size of the materialized tree affects the performance of Coelho even when the tree fits in memory. The effect is noticeable in Figure 7-2 by looking at the *Build Tree* durations for $B = 4$, $B = 8$, $B = 64$, and $B = 512$. These B values maximize the size of first level of the tree and also correspond to a 25% performance slowdown for situations with low corruption.

In conclusion, the performance of the low-bandwidth Coelho algorithm is limited by the computational and storage overhead of the build tree phase. While the algorithm can be very sensitive to input parameters, it is not sensitive to input conditions. Assuming that input parameters are chosen to avoid problem conditions, the Coelho algorithm performs well.

Analysis of NRoundXRowHash

We ran experiments with varying amounts of corruption and evaluated N-round-X-row-hash's performance in time and bandwidth consumed. As shown in Figure 7-4, the input parameters to NRoundXRowHash play an important role in determining its performance. We used a 2 round hash, except where the initial round bucket size was less than 20. The larger the initial bucket size, the lower the bandwidth of the first round. Since N-round-X-row-hash transfers records in chunks the size of the final round's bucket, large buckets in the final round increase the cost of a final round bucket hash not matching. Thus, N-round-X-row-hash with $N = 1$ has a trade off between the bandwidth cost of sending hashes versus a hash not matching and having to send the whole bucket contents.

The second round mitigates the bandwidth cost of a larger first round size at the expense of the time required to compute the second round. A third round was not worth performing for the database size used in this experiment. For high corruption, a first round bucket size large enough to merit a second round was likely wasted:

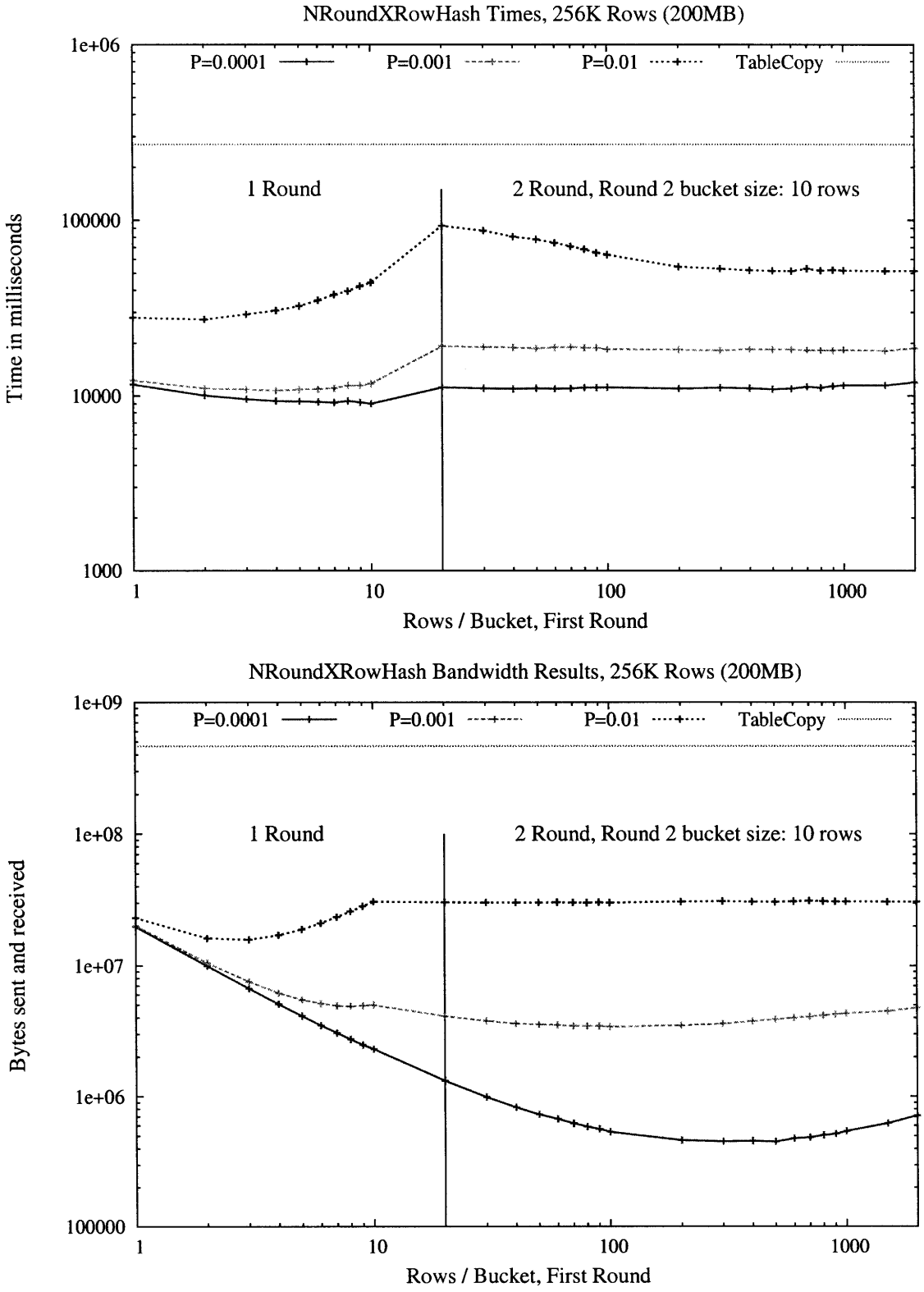


Figure 7-4: Performance of N-round-X-row-hash, P is the probability that any given row is incorrect.

hashes of large buckets are highly likely not to match, preventing the comparator from excluding any rows in the second round. For low corruption, running a second round produced significant reductions in the bandwidth utilized with little additional computational overhead.

The trade off between additional rounds and sending additional rows depends on how well provisioned the replicas are in terms of computational power and bandwidth. During execution, the replica is CPU limited. Our experiments were performed with 100MB/s interconnect, which provides ample bandwidth for less efficient compare mechanisms. Since our target environment is a data center, not a wide-area network, ample bandwidth is a reasonable assumption.

The model provides a good approximation of actual performance, as shown in Figure 7-5. Due to the approximations made in the model, it is most accurate when corruption is low.

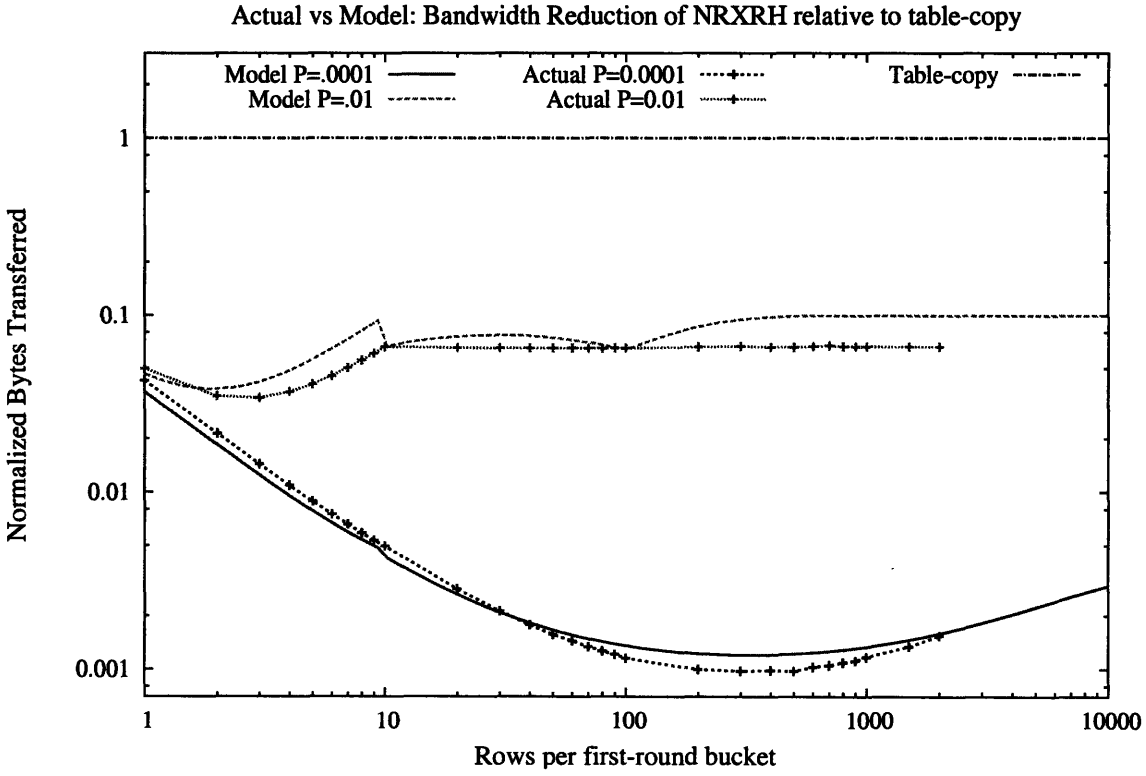


Figure 7-5: Bandwidth model vs actual data for 2 round X row hash with 10 row second round. H (hash overhead compared to row size) is 0.037. P is the probability that any given record is incorrect.

NRoundXRowHash still performs well even when the bucket hashes do not fit in memory. Since the record scan is likely performed in ID order, only one bucket is having rows added to it at a time. Once the database adds a row to bucket $n+1$, it will never add another row to bucket n, allowing bucket n to be written to disk without performance penalty. For very large tables, NRoundXRowHash should avoid unnecessary disk accesses.

NRoundXRowHash vs Coelho

Corruption	Algorithm	Time	Bandwidth
1/100	NRXRH	29s	15MB
1/100	Coelho	24s	8.0MB
1/1000	NRXRH	12s	4.8MB
1/1000	Coelho	15s	1.6MB
1/10000	NRXRH	11s	.44MB
1/10000	Coelho	14s	.27MB

Table 7.1: Time/Bandwidth summary: comparison of NRXRH and Coelho. Parameters used: Coelho used $B = 128$ for each corruption scenario. NRXRH used $X_1 = 2$, $X_1 = 10$, and $X_1 = 200, X_2 = 10$ respectively.

Table 7.1 summarizes data from the Coelho and N-round-X-row-hash results shown earlier, reporting the performance given the best parameters. Coelho uses less bandwidth than NRXRH by a factor of 2 to 3, but with low corruption NRXRH outperforms Coelho in speed by 20%. If the amount of corruption is truly unknown, Coelho may be a better scheme as parameter choice is based primarily on the number of records, whereas NRXRH parameters must also take corruption into account. If corruption is assumed to be low and the network well-provisioned, NRXRH performs well. The performance results summarized in Table 7.1 were selected by taking the results with the fastest time that did not use vast amounts of bandwidth.

Observing that both Coelho and N-round-X-row-hash spend large amounts of computation time hashing the table, we ran some experiments on precomputed hashes. Table 7.2 shows the results of running both algorithms with precomputed hashes. The algorithms perform much faster if they don't have to hash a 200MB table. For low corruption, both algorithms run 2-3 times faster on precomputed hashes. Maintaining

Corruption	Algorithm	Time	Bandwidth
1/100	NRXRH	22s	15MB
1/100	Coelho	15s	7.7MB
1/1000	NRXRH	4.7s	5.7MB
1/1000	Coelho	4.9s	1.5MB
1/10000	NRXRH	3.3s	.43MB
1/10000	Coelho	4.6s	.26MB

Table 7.2: Time/Bandwidth summary: comparison of NRXRH and Coelho with precomputed hashes. Parameters used: Coelho used $B = 128$ for each corruption scenario. NRXRH used $X_1 = 2$, $X_1 = 4$, and $X_1 = 500, X_2 = 10$ respectively.

precomputed hashes would require some sort of trigger in the database that updates the hash whenever the row is updated. These triggers would have a performance impact on system throughput.

For tables that are very large, the story is much the same. For a 1 GB table and a corruption probability of 10^{-5}, the NRXRH model suggests a $N = 2$ round scheme, using $X_1 = 1000$ and $X_2 = 10$. A potential concern is that NRoundXRowHash's second pass would take a long time when the table does not fit in memory. To test this possibility, we ran the test without precomputed hashes, ensuring that the whole gigabyte table must be considered. As usual, $B = 128$ works well for Coelho. The results are as follows:

- **NRoundXRowHash**: 57 seconds, .45 megabytes of bandwidth
- **Coelho**: 73 seconds, .26 megabytes of bandwidth

Even when NRoundXRowHash must run two rounds over a table that does not fit in memory, it still outperforms Coelho in time to completion (by 22%). Since NRoundXRowHash uses the table index on the second round, it only needs to read a small chunk of the table. The second round takes 9 seconds to perform, but by comparison, a single round 1000-row hash takes longer (59 seconds) and uses 50 times the bandwidth (24 megabytes). Precomputed hashes would only improve the performance of NRoundXRowHash: it currently hashes some chunks of the table twice.

The work so far assumed that all rows are equally likely to get corrupted. In practice, corruption is not completely random. Disk corruption or buffer overruns can

Correlation	Corruption	Algorithm	Time	Bandwidth	Params
.5	1/100	Coelho	26s	9.6M	$B = 128$
.9	1/100	Coelho	32s	9.6M	$B = 128$
1	1/100	Coelho	30s	9.6M	$B = 128$
.5	1/100	NRXRH	32s	16M	$X_1 = 4$
.9	1/100	NRXRH	24s	9.7M	$X_1 = 7$
1	1/100	NRXRH	18s	4.7M	$X_1 = 80$ and $X_2 = 10$

Table 7.3: Effect of correlated failures on Coelho and N-round-X-row-hash. Correlation is the probability that the successor row is corrupt given the current row is corrupt. Corruption is the fraction of rows of the table that are corrupt.

corrupt rows in chunks. Table 7.3 shows the results of running the repair algorithms when corruption is likely to corrupt adjacent rows. Rows are ordered by their primary key, assuming that it is a clustered index. With a correlation of .5, rows are likely to get corrupted in pairs. A correlation of .9 means groups of 10 rows and correlation of 1 ensures that all of the corrupted rows are adjacent.

N-round-X-row-hash handles corrupted chunks better than Coelho. Since Coelho hashes the primary key when assigning rows to leaves, it completely randomizes the order of rows. Adjacent corrupted rows are randomly distributed among the leaves, producing no performance benefit. On the other hand, N-round-X-row-hash's bucketing scheme is more likely to catch multiple adjacent corrupted rows in the same bucket. Holding corruption constant, more corrupted rows per bucket means that more bucket hashes match, allowing N-round-X-row-hash to narrow the search more effectively.

7.1.5 Conclusions

Both Coelho and N-round-X-row-hash perform well, reducing execution time by an order of magnitude and bandwidth consumption by several orders of magnitude. Coelho is a general-purpose algorithm that achieves good performance with minimal configuration, but may have trouble with very large tables. N-round-X-row-hash is a special purpose algorithm that is targeted at predictably low or correlated corruption in tables with simple primary key columns. A reasonable solution could involve using

Coelho normally and N-round-X-row-hash when the situation matches its strengths.

7.2 HRDB and Repair

Given the algorithms described in the previous section, HRDB must provide a repair mechanism that guarantees the correctness of the repaired replica. A correct repair mechanism ensures that the logical state of the replica after repair reflects the correct logical state, to the extent possible on the replica. Obviously, if the faulty replica is incapable of having its state correctly updated via SQL (*e.g.,* its disk is full or its software is faulty), then we need an external mechanism to correct the problem. The repair mechanism ensures that the state of the replica after repair reflects the state of the system as was externally visible to the client.

We first present a straightforward repair process that guarantees correctness by quiescing the system. Quiescing the entire system to repair a single replica results in a significant interruption in service. A repair operation may take a long time to complete, while the whole system is unresponsive to clients. A more desirable mechanism would allow client transactions to execute in parallel with repair operations, yet still provide the same correctness guarantees. We present two incremental repair processes, based on whether HRDB has access to writesets or not.

Since most tables in the TPC-C benchmark have multi-column primary keys and our implementations of the compare and repair algorithms do not support them, we haven't produced performance numbers for repair during execution of TPC-C, nor values for the overhead of precomputing hashes while running TPC-C.

7.2.1 Quiescent Repair

The simplest way to ensure that the faulty replica is repaired to a consistent snapshot of the overall system is to quiesce the system while repair is in progress. If the shepherd does not run transactions in parallel with the repair operation then all transactions that committed prior to the repair operation will be reflected in the state of the repaired replica after the repair operation completes. Both Coelho and N-round-X-

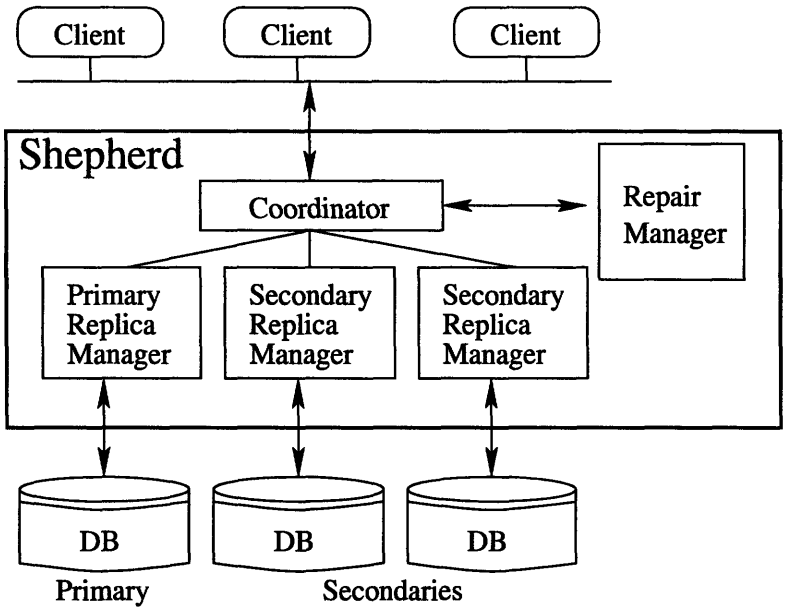


Figure 7-6: The repair manager submits repair transactions through the coordinator like a client, and the coordinator supplies it with the hooks required for successful repair.

row-hash work on a single database table at a time. By running a repair operation for each table in the database, the complete database can be repaired. Once all of the repair operations have completed, the shepherd can recommence executing client transactions.

Repairing a faulty replica requires agreement on what constitutes the correct state. The shepherd already has an agreement mechanism that it uses to verify the correctness of client transactions. The repair system can borrow this mechanism and be guaranteed correctness by submitting transactions as if it were a client: if the repair transaction commits, the values with which it repaired the faulty replica must be correct. HRDB runs a *repair manager* that performs replica repair by communicating with the coordinator if it were a client. The shepherd provides some special hooks for the repair manager to support the operations required for repair. The augmented architecture is shown in Figure 7-6.

Repair transactions have two phases: fault isolation and data transfer. All replicas participate in the fault isolation phase; this part of the transaction is executed just like any client-submitted transaction. Both algorithms above (N-round-X-row-hash

and Coelho) are symmetric during the fault isolation phase: they send identical SQL to both replicas. The shepherd provides a hook whereby the repair manager can examine query results returned by the suspected faulty replica as well as the answer the coordinator sends. Using this hook, the repair manager can compare hashes from the primary with hashes from the faulty replica to determine where the faulty replica's state differs. The result of the fault isolation phase is the set of rows (and their values) that must be updated on the faulty replica.

The data transfer phases consists of `DELETE` and `INSERT` statements to store the correct data and is executed only on the faulty replica. The shepherd provides a second hook that instructs all replicas but the one being repaired to ignore these `DELETE` and `INSERT` statements. The coordinator does not perform agreement on the results of these semi-ignored statements.

When the repair manager attempts to commit the repair transaction, the shepherd will abort the transaction if the primary was faulty and supplied incorrect information. Thus, the shepherd's regular correctness mechanism ensures the correctness of the repair operation. If the primary's results are incorrect, the faulty replica's state will be incorrectly updated, and then all the incorrect updates rolled back. As an optimization, the repair manager may commit the repair transaction at the end of the fault isolation phase to verify correctness, and use one or more subsequent transactions to update the faulty replica. Since many databases have performance issues with transactions that update large numbers of rows, splitting the update operation into several transactions may be more efficient for the database to execute. Finally, once the fault isolation phase is finished and the values of the differing rows acquired, the rest of the replicas are not necessary for the repair operation and could be executing client transactions.

7.2.2 Incremental Repair

With incremental repair, the repair manager repairs small pieces of the database at time, eventually repairing the whole database. Since the repair operation only requires a consistent snapshot of a small part of database, client transactions can successfully

execute on the rest of the database. The granularity of the repair operation determines the amount of interruption and must be balanced against the overhead of running a repair operation. The compare and repair algorithms from Section 7.1 work on whole tables, which for the TPC-C benchmark would be similar to quiescing the system (*i.e.*, similar to Table-level locking performance in Section 6.2.2). Both Coelho and N-round-X-row-hash could be easily modified to work on parts of tables. For simplicity, the rest of this section assumes that the granularity of an incremental repair operation is a whole table.

An incremental repair operation ensures that the state of the repaired region on the faulty replica matches the state of the replica set *at the time the repair operation commits.* The mechanics of incremental repair differ depending on whether HRDB has writesets available or not. In the context of HRDB, writesets are available with SES (snapshot isolation) and not available with CBS (serializable isolation). One subject of future work is using writeset extraction in CBS to ease repair, thus the lack of writesets is not intrinsic to CBS. We start with a description of an incremental repair solution using writesets in the context of snapshot isolation, since it is simpler. Next, we go into a brief digression about how corruption spreads in the database state. The spread of corruption is relevant to the final section where we describe the challenges surrounding incremental repair without writesets in the context of serializable isolation.

Incremental Repair with Writesets

Incremental repair is vastly simplified by the existence of writesets. When the shepherd flags a replica as faulty, it stops executing transactions on the replica. The repair manager repairs each table individually with a separate transaction submitted like any other client transaction, as shown in Figure 7-7. Because the system is not quiesced, client transactions are intermixed with repair transactions. The replica manager of the faulty replica must decide what to do with these intermixed transaction. Since each table is repaired by a different transaction, executing just the repair operations on the replica will leave the tables in an inconsistent state. Consider the example

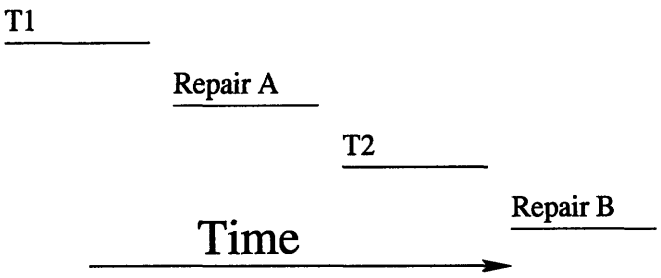


Figure 7-7: Incremental Repair operations repairing first table A then table B.

in Figure 7-7: T2 occurs after the repair of A but before the repair of B. A faulty replica that just executes the repair transactions but not T1 and T2 will end up with the state of table A lacking any updates from transaction T2. However, the state of table B will reflect T2's updates because the repair of table B was subsequent to T2's commit.

The resolution is simple: after all repair operations have committed, replay writesets to the replica just as if it had crashed. Since writeset application is idempotent, applying the writeset for T2 will correctly update table A and leave table B unchanged. Once the faulty replica has applied the writesets of all transactions it did not execute while being repaired, its state is both correct and consistent. In this manner, the repair operation converts a Byzantine-faulty replica into a slow replica.

However, storing all the client transaction writesets for the replica while it performs a potentially lengthy repair operation is costly. Instead, the replica manager can apply writesets as transactions commit. Essentially, the replica manager treats the replica as never having access to the required snapshot for the transaction, forcing it to apply a writeset to commit the transaction. Applying writesets over potentially faulty data is correct because writeset application does not depend on the data in the database. Applying writesets before repair may also result in a performance improvement due to reducing the difference between the correct data and the replica's data; the magnitude of the difference is a key factor in the cost of the repair operation.

Two additional issues arise that are specific to snapshot isolation. The first issue is that repair transactions can get aborted by the database for two reasons: concurrent writes and snapshot expiration. If the repair operation is repairing data that the

client regularly interacts with, then the data is likely to be modified while the repair transaction is running. We can avoid aborts of the repair transactions due to concurrent writes by using a pessimistic locking mechanism for the repair transaction. Most databases support some sort of intentional locking mechanism (typically a `LOCK TABLE` SQL statement) that could be used for this purpose. Snapshot expiration occurs because the database has only a limited amount of storage for snapshots, resulting in long-running transactions having their snapshots overwritten. Repair operations must be quick enough that their snapshots remain intact.

The second issue is that client transactions can wait on repair transactions completing, regardless of whether additional pessimistic locking mechanisms are used. Snapshot isolation uses a pessimistic blocking mechanism when it detects a concurrent write, rather than aborting the conflicting transaction immediately. The longer a single repair transaction runs, the more clients will have attempted to touch the data it is repairing and have blocked. Long running repairs on hot data could quiesce the system unintentionally. Thus, the size of repair operations should be inversely proportional to how often the data is updated.

Spread of Corruption

A fault that corrupts the state of a replica may remain undiscovered by the shepherd for some time. In the interim, corrupted state may propagate from the original fault as transactions are executed. Two factors prevent the spread of corruption: writeset correction and client interaction. If HRDB is using writeset extraction, incorrect rows that would be written by a transaction can be caught prior to commit. Obviously, writeset correction only helps when updates are captured by the writeset extraction mechanism; lower-level faults will continue to elude detection.

The nature of client interaction with the shepherd can also mitigate the likelihood of propagation. Consider the following interaction between the client and the shepherd that inserts a row into a `burnrate` table that reflects the total salary paid out by a company named "Foo Inc.":

```
select sum(salary) from emp where company='Foo Inc.'
```

```
=> 1500000
insert into burnrate (company,rate) values ('Foo Inc.',1500000)
=> 1 row updated
```

Suppose one of the replicas has an incorrect value for the salary of an employee of Foo Inc. If the replica is the primary, it will produce an incorrect answer for the first statement and the transaction will be aborted. If the replica is a secondary, the value it computes for the first statement will be incorrect, but the value inserted in the second statement comes from the (non-faulty) primary, not the faulty secondary. Even with corrupted state involved in the transaction, committing this transaction does not spread corruption on the faulty replica.

However, the following expression of the above process does spread corruption:

```
insert into burnrate select company, sum(salary) as rate
                            from emp where company='Foo Inc.'
=> 1 row updated
```

Instead of pushing the value out through the coordinator's voting process to the client, the result of rate computation is inserted directly into another table. The coordinator does not catch the fault because the return value of "1 row updated" is returned by all replicas, even the one with a bogus value for rate.

In particular, corruption can spread from one table to another if the workload contains the following operations: `INSERT ... SELECT ...`, `UPDATE` with join, `DELETE` with join, or stored procedures. While many workloads contain these elements, some do not (*e.g.,* our implementation of TPC-C).

Incremental Repair without Writesets

The lack of writesets vastly complicates incremental repair. Once again, the replica manager repairs each table individually in a separate transaction as shown in Figure 7-7. From the example in the figure, the replica manager need not execute T1 because both table A and B are repaired after T1 completes, ensuring that any changes made by T1 are reflected in the final state of both tables. T2 presents a problem because

it is ordered between the repair of table A and table B. Transactions like T2 must exist; otherwise the system is effectively quiescent. If the replica manager for the faulty replica never executes T2 then the state of table A on the faulty replica will not match its state on the rest of the replicas. If the replica manager executes T2 after both repairs, then table B will reflect T2 executing twice, which also will not match the rest of the replicas. Clearly, the only time that the faulty replica can execute T2 to ensure consistent repair is exactly when the coordinator ordered T2: between the repair of table A and table B.

However, executing transactions on the faulty replica concurrently with incremental repair operations presents a correctness problem. Once again the problem is with T2: T2 executes after table A has been repaired but before table B has been repaired. If T2 propagates some corruption in the yet-to-be-repaired table B to the already-repaired table A, then after all repairs, some corruption will remain in the database. As described above, there are some workloads that will not propagate faults. Excepting those workloads, there is no general solution to this correctness problem without either using writesets or quiescing the system. From the figure, it is incorrect *not to* run T2, and can be incorrect *to* run T2 if it propagates faults.

We stated earlier that the replica manager for the faulty replica did not need to execute transaction T1 from Figure 7-7 because T1's modifications to the database state would be installed by the repair operations. However, it may be less efficient than actually running T1. The cost of the repair operation is proportional to the amount of data that must be repaired. Not executing T1 before running the repair operations will have one of two effects:

- Executing T1 on the faulty replica causes its state to become further corrupted. The repair operation becomes more costly.

- Executing T1 on the faulty replica correctly updates the state of the replica. This situation will occur when the fault or corrupted state has nothing to do with T1, thus allowing T1 to execute correctly. Any updates correctly applied to the replica state by T1 need not be repaired, effectively reducing the cost of

the repair operation.

Whether to execute T1 depends on the likelihood of further corruption, offset by the relative cost of executing the transaction versus enlarging the repair operation. If the database is large and transactions touch a small number of rows, a fault is unlikely to interfere with many transactions. Transaction processing databases are very efficient at executing transactions and not terribly efficient at computing the large aggregates required by repair. Furthermore, the fault detected by the coordinator may have been transient (*e.g.*, an incorrect answer to a query which did not reflect incorrect replica state). For transient faults, running transactions as usual will result in the repair operation transferring no data.

Chapter 8

Conclusions

This thesis shows, for the first time, a practical way to tolerate Byzantine faults in transaction processing systems while ensuring that transaction statements can execute concurrently on the replicas. We present two new concurrency control schemes that provide correctness and strong consistency while supporting a high concurrency execution. We also describe and evaluate repair mechanisms for correcting a faulty replica's state.

To conclude this thesis, we first present an overview of the contributions in the previous chapters. Next, we discuss directions for future work. Finally, we conclude with a higher level look at where this work fits in.

8.1 Contributions

The first major contribution is commit barrier scheduling, which allows the coordinator in the shepherd to observe transaction execution on the primary and ensure that all non-faulty secondaries execute transactions in an equivalent serial order. CBS ensures that all replicas have the same logical state and that clients get correct answers for all committed transactions. The key insight of CBS is to use one replica as a primary that can efficiently determine the order of transactions.

We have implemented CBS as part of HRDB, which provides a single-copy serializable view of a replicated relational database system using unmodified production

versions of several commercial and open-source databases as the replicas. Our experiments show that HRDB can tolerate one faulty replica with only a modest performance overhead (10-20% for the TPC-C benchmark on a homogeneous replica set). Furthermore, HRDB is 2-3 times faster on TPC-C than a naive implementation that forcibly serializes all transactions. We also showed how HRDB is able to mask faults observed in practice, including one that we uncovered in MySQL.

CBS has a number of attractive properties. First, it requires no modifications to the database systems themselves. Second, it does not require any analysis of SQL queries to detect conflicts. SQL analysis is intractable for complex queries at finer granularities than table-level conflicts and may not work at all for materialized views or stored procedures. Third, CBS is able to preserve substantial amounts of concurrency in highly concurrent database workloads.

The second major contribution is another concurrency control scheme called snapshot epoch scheduling, which is similar to CBS, except it provides single-copy snapshot isolation. SES takes advantage of a recent shift in databases to providing snapshot isolation instead of serializable isolation. Snapshot isolation is a slightly weaker consistency model than serializable, posing its own challenges for implementing a single-copy replication scheme. One result of the weaker model is that we require additional functionality from the database, namely writeset extraction. We implemented SES as another part of HRDB, and our experiments show that it also performs well.

The final major contribution is a repair mechanism that can be used to correct a faulty replica's state. The repair mechanism is essential to a long-lived system, helping to ensure that no more than f replicas are faulty simultaneously. Due to the large size of database state, the algorithms used for repair must be efficient in both time and bandwidth consumed. We analyze two such algorithms, Coelho and N-round-X-row-hash, and determine that both provide order of magnitude performance improvements. In specific situations, N-round-X-row-hash (the algorithm we developed) is 20% faster than Coelho (an algorithm from the literature), though it uses more bandwidth. In a well-provisioned network scenario, this is a minor cost to pay.

8.2 Future Work

There are numerous avenues of future work in the area of Byzantine fault tolerance of databases. We present a limited set here.

The first item for future work is refinement of the SES implementation and acquisition of better performance results. Further work could involve investigation and implementation of the various protocol optimizations presented in Sections 4.6 and 5.7. The optimizations offer performance improvements over the basic protocols, often to deal with particular workloads. Worth special mention is the optimization to SES that allows scale-up: using more replicas to distribute load, potentially providing performance better than a single, non-replicated database.

8.2.1 Repair

As discussed in Chapter 4, the coordinator cannot always know when a replica has suffered a Byzantine failure. Therefore we cannot rely on knowing about the failure in a timely way. For example, some faults are *silent*: the database state is corrupted, but the problem doesn't show up in the responses sent to the coordinator. Silent faults can occur due to hardware problems; they also occur during updates and even during the processing of supposedly read-only queries. Eventually these faults will be detected when the affected tables are read, but this might be so far in the future that it could too late (*i.e.*, by then more than f nodes might be corrupted).

The use of writeset extraction can catch many faults that would otherwise be silent. Specifically, the data updated by write operations is not returned to the client; instead the number of rows updated is returned. Writeset extraction makes the updated values explicit and visible to the coordinator for checking. Writeset extraction has an impact on both performance (overhead of extracting writesets and performing agreement) and deployment (replica databases must be modified to support the operation). While necessary for SES, writeset extraction could be performed in CBS for increased fault tolerance [34].

Additionally, we are exploring a *proactive* approach to detecting silent faults where

the coordinator injects transactions that read aggregates of the database. If a replica's state has been corrupted, it is unlikely to respond correctly to the aggregate operation. Since the expected result of these operations is that the data matches, some variant of N-round-X-row-hash's hash bucketing mechanism would be appropriate. The coordinator can then schedule repairs for databases found to be faulty. The frequency and size of the proactive state checking transactions is dependent on the workload and what guarantees the system must provide about how long corrupt state is allowed to remain undiscovered.

8.2.2 Replication of the Shepherd

The shepherd is single point of failure in an otherwise replicated system. While both CBS and SES support, and can recover from, crash failures of the shepherd, the system is unavailable while the shepherd is down. We can use replication to tolerate crash and even Byzantine failures of the shepherd. Replication of the shepherd requires that the shepherd replicas run an agreement protocol to agree on client operations and their order. Replicating the shepherd for fail-stop fault tolerance requires $2f + 1$ replicas, while replicating for Byzantine faults requires $3f + 1$. Shepherd replicas can be co-resident with databases to reduce the number of hosts required.

In CBS, the only time that the shepherd explicitly assigns ordering to transactions is at the commit point. In SES, there are two ordering events: the snapshot and the commit points. A replication protocol for the shepherd only need run an expensive agreement operation for these points; it need not reach agreement on each individual operation in the transaction. By augmenting the shim that runs adjacent to the database with some additional functionality, the replicated shepherd becomes more a certifier than an intermediary. Removing the shepherd from the statement execution path improves performance. We believe that a fully replicated shepherd can significantly improve the fault tolerance of the system without incurring significant additional performance penalty.

8.2.3 Bug Discovery

We had accidental success at finding new bugs with HRDB. A more determined effort to use HRDB to detect bugs may bear fruit, potentially even in production databases. Due to CBS and SES support for concurrency, HRDB is admirably suited to finding concurrency faults. Concurrency bugs are difficult to find because single-threaded test suites never reveal them and multi-threaded tests are non-deterministic, making it tricky to tell if the result of the test is correct. The shepherd is well-poised to find concurrency faults because it uses voting and guarantees equivalent transaction schedules while not requiring that replicas execute operations in lockstep. The bug we found in MySQL is a good example of such a bug: failure to acquire correct locks resulted in a phantom read in some, but not all, of the replicas.

We have done some preliminary investigation with generating random SQL statements, reminiscent of the RAGS [38] project. In RAGS, they generated random SQL traces and ran them sequentially on a number of databases. By comparing the results, they were able to isolate suspected faulty behavior in a number of commercial databases. However, RAGS did not perform any tests with concurrency due to the non-deterministic nature of database execution. HRDB provides a logical next step.

8.3 Conclusion

In this thesis, we present a practical application of Byzantine fault tolerance to a real-world situation. Databases suffer from Byzantine faults and there is merit (and possibly money) in preventing the effects of these bugs from reaching clients. The two concurrency protocols we developed allow concurrent transaction execution, which is a key component of any viable solution. The repair operation we developed is critical to actually running the system for long periods of time. Our system is useful for tolerating and masking faults in production database systems, in part due to its support for heterogeneity of implementation that is necessary to guarantee failure independence. Lastly, if the cost of the system is too high for production use, it may find a home as novel concurrency testing framework.

Bibliography

[1] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *TODS*, 14(2):264–290, 1989.

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

[3] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *TODS*, 9(4):596–615, 1984.

[4] Anupam Bhide, Ambuj Goyal, Hui-I Hsiao, and Anant Jhingran. An efficient scheme for providing high availability. In *SIGMOD*, pages 236–245, 1992.

[5] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[6] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3), August 2003.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Conference*, 2004.

[8] Fabien Coelho. Remote comparison of database tables. Technical Report A/375/CRI, Centre de Recherche en Informatique, Ecole des mines de Paris, 2006.

[9] Willy Zwaenepoel Emmanuel Cecchet, Julie Marguerite. Raidb: Redundant Array of Inexpensive Databases. Technical Report 4921, INRIA, September 2003.

[10] I. Gashi, P. Popov, and L. Strigini V. Stankovic. On designing dependable services with diverse off-the-shelf SQL servers. *Lecture Notes in Computer Science*, 3069:191–214, 2004.

[11] Goldengate. `http://www.goldengate.com/technology/architecture.html`.

[12] Goldengate veridata. http://www.goldengate.com/technology/veridata.html.

[13] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD*, pages 173–182, 1996.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1992.

[15] IBM. WebSphere Information Integrator SQL Replication. http://www.ibm.com/developerworks/db2/roadmaps/sqlrepl-roadmap-v8.2.html.

[16] ISO/IEC. Sql:1999. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26196.

[17] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 477–484, 2002.

[18] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *TODS*, 25(3):333–379, 2000.

[19] Ramakrishna Kotla and Mike Dahlin. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks*, 2004.

[20] Wilburt Labio and Hector Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 63–74, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[21] John Langford. Multiround rsync, 2001.

[22] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 419–430, New York, NY, USA, 2005. ACM Press.

[23] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. *Distributed Object Management*, pages 79–91, 1993.

[24] Ralph C. Merkle. Protocols for public key cryptosystems. *sp*, 00:122, 1980.

[25] Hector Garcia Molina, Frank Pittelli, and Susan Davidson. Applications of Byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11(1):27–47, 1986.

[26] Oracle. Oracle database heterogeneous connectivity administrator's guide. http://download.oracle.com/docs/cd/B19306_01/server.102/b14232/toc.htm.

[27] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 407–418, New York, NY, USA, 2005. ACM Press.

[28] Hweehwa Pang and Kian-Lee Tan. Verifying completeness of relational query answers from online servers. *ACM Trans. Inf. Syst. Secur.*, 11(2):1–50, 2008.

[29] Fernando Pedone and Svend Frolund. Pronto: A fast failover protocol for off-the-shelf commercial databases. Technical Report HPL-2000-96, HP Laboratories Palo Alto, 2000.

[30] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: replicated database services for world-wide applications. In *SIGOPS European workshop*, pages 275–280, 1996.

[31] Christian Plattner and Gustavo Alonso. Ganymede: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[32] Prabhu Ram and Lyman Do. Extracting delta for incremental data warehouse maintenance. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, page 220, Washington, DC, USA, 2000. IEEE Computer Society.

[33] Red Gate Software. SQL data compare. `http://www.red-gate.com/products/sql_data_compare/index.htm`, May 2008.

[34] J. Salas, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. Lightweight reflection for middleware-based database replication. *srds*, 00:377–390, 2006.

[35] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[36] Oded Shmueli and Alon Itai. Maintenance of views. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 240–255, New York, NY, USA, 1984. ACM Press.

[37] Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker, and Andrew Yu. Data replication in mariposa. In *ICDE*, pages 485–494, 1996.

[38] Donald R. Slutz. Massive stochastic testing of sql. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 618–622, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[39] SQLFairy. `http://sqlfairy.sourceforge.net/`.

[40] Swissql—SQLOne Console 3.0. http://www.swissql.com/products/sql-translator/sql-converter.html.

[41] Sybase replication server. http://www.sybase.com/products/businesscontinuity/replicationserver/.

[42] Transaction Processing Performance Council. Tpc-C. http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[43] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, Australian National Univerisity, 1999.

[44] Matthias Wiesmann, Andre Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. *srds*, 00:206, 2000.

[45] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. *icde*, 0:422–433, 2005.

[46] Min Xie, Haixun Wang, Jian Yin, and Xiaofeng Meng. Integrity auditing of outsourced data. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 782–793. VLDB Endowment, 2007.

[47] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, New York, NY, USA, 2003. ACM Press.