HARBOR: An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse

by

Edmond Lau

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in
Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author .....
Department of Electrical Engineering and Computer Science
May 5, 2006

Certified by .....
Samuel Madden
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

HARBOR: An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse

by

Edmond Lau

Submitted to the Department of Electrical Engineering and Computer Science
on May 5, 2006,
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Any highly available data warehouse will use some form of data replication to ensure that it can continue to service queries despite machine failures. In this thesis, I demonstrate that it is possible to leverage the data replication available in these environments to build a simple yet efficient crash recovery mechanism that revives a crashed site by querying remote replicas for missing updates. My new integrated approach to recovery and high availability, called HARBOR (High Availability and Replication-Based Online Recovery), targets updatable data warehouses and offers an attractive alternative to the widely used log-based crash recovery algorithms found in existing database systems. Aside from its simplicity over log-based approaches, HARBOR also avoids the runtime overhead of maintaining an on-disk log, accomplishes recovery without quiescing the system, allows replicated data to be stored in non-identical formats, and supports the parallel recovery of multiple sites and database objects.

To evaluate HARBOR's feasibility, I compare HARBOR's runtime overhead and recovery performance with those of two-phase commit and ARIES, the gold standard for log-based recovery, on a four-node distributed database system that I have implemented. My experiments show that HARBOR incurs lower runtime overhead because it does not require log writes to be forced to disk during transaction commit. Furthermore, they indicate that HARBOR's recovery performance is comparable to ARIES's performance on many workloads and even surpasses it on characteristic warehouse workloads with few updates to historical data. The results are highly encouraging and suggest that my integrated approach is quite tenable.

Thesis Supervisor: Samuel Madden
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgements

Many thanks to my advisor, Sam Madden, for his invaluable guidance in weaving the underlying story to my work, for hammering out critical details regarding recovery and commit processing, and for suggesting methods of evaluation. It has been a pleasure working with him. Thank you to Mike Stonebraker for inviting me to join the C-Store team and for initially advising my research with the group. Thanks to David Dewitt for his suggestion to add group commit to my evaluation of commit processing protocols. George Huo helped by providing some code for running distributing queries. I'd also like to recognize Chen Xiao for listening to some of my design issues and helping to bounce some of my ideas a few days before a conference paper deadline. Last, but not least, thanks to Mom and Dad, without whose sacrifices over the years none of this would have been possible.

This research is partially funded by the Deshpande Center and by a Siebel Scholar fellowship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> "Twenty years from now you will be more disappointed by the things that you didn't do than by the ones you did do. So throw off the bowlines. Sail away from the safe harbor. Catch the trade winds in your sails. Explore. Dream. Discover."
>
> — *Mark Twain (1835-1910)*

Every time a sales clerk scans a barcode at the checkout counter of one of Walmart's 5,700 stores [12] worldwide, the sale is saved so that software can transmit the sales data hourly to a growing 583-terabyte data warehouse running on a parallel 1,000-processor system [7]. Every time one of Wells Fargo's 7 million online customers logs onto the bank's online portal, software extracts and builds a personalized spending report from a data warehouse of over 4 terabytes of transaction history [6]. Every time a Best Buy manager reads the morning report summarizing the sales picture of the entire 700-store [14] company, he or she reads the result of a nightly analysis run on the 7 terabytes of data in the company's data warehouse [55].

A traditional *data warehouse* like the ones mentioned consists of a large distributed database system that processes read-only analytical queries over historical data loaded from an operational database. Such data warehouses abound in retail, financial services, communications, insurance, travel, manufacturing, e-commerce, and government sectors, providing valuable sources of information for people to

gather business intelligence, extract sales patterns, or mine data for hidden relationships. They enable, for instance, a Walmart executive to discover for which stores and for which months Colgate toothpaste was in high demand but in short supply, a Wells Fargo customer to view a daily snapshot of his finances, or a Best Buy sales worker to identify unprofitable customers who, for example, file for rebates on items and then return them [17].

These warehouses often do not need traditional database recovery or concurrency control features because they are updated periodically via bulk-load utilities rather than through standard SQL INSERT/UPDATE commands. They do, however, require high availability and disaster recovery mechanisms so that they can provide always-on access [5]. IT system downtime costs American businesses $1 million per hour [33], and much of the value in these systems can be traced back to the business intelligence data captured by data warehouses. A crashed site means fewer queries can be run and fewer analyses can be performed, translating directly into lost revenue dollars. When a site in a data warehouse crashes, queries must remain serviceable by other online sites, and the crashed site must be recovered and brought back online before some other site fails. High availability and efficient crash recovery mechanisms to enable a data warehouse to be up and running as much as 99.999% of the time therefore become critical features for most data warehouses.

1.1 Updatable Data Warehouses

Alongside these traditional data warehouses, recent years have seen increasing interest in building warehouse-like systems that support fine-granularity insertions of new data and even occasional updates of incorrect or missing historical data; these types of modifications need to be supported concurrently with standard SQL updates rather than with bulk-load utilities [13]. Such systems are useful for providing flexible load support in traditional warehouse settings and for supporting other specialized domains such as customer relationship management (CRM) and data mining where there is a large quantity of data that is frequently added to the

database in addition to a substantial number of read-only analytical queries to generate reports. Even users of traditional data warehouses have expressed desires for zero-latency databases that can support near real-time analyses; such functionality would, for instance, empower salespersons to identify customers likely to defect to a competitor or to identify customer problems that have recently been experienced by customers at other store locations [41].

These "updatable warehouses" have the same requirements of high availability and disaster recovery as traditional warehouses but also require some form of concurrency control and recoverability to ensure transactional semantics. Transactions consisting of a sequence of individual actions need to happen *atomically*, and multiple transactions executing concurrently must be *isolated* from one another. In such updatable environments, providing some form of time travel is also useful; for example, users may wish to compare the outcome of some report before and after a particular set of changes has been made to the database.

1.2 Shortcomings with Traditional Recovery Mechanisms

Traditional crash recovery mechanisms for database systems, described in more detail in the next chapter, share three undesirable characteristics:

1. They require an on-disk log to achieve recoverability for transactions. After a crash, the log provides a definitive word as to whether a transaction committed or aborted and contains undo/redo information to roll back or reapply the effects of a transaction if necessary. To eliminate disk seeks between log writes, systems often store the sequentially written log on a separate, dedicated disk; since the log is used only for crash recovery, however, the scheme wastes disk resources during normal processing.

2. They require *forcing*, or synchronously writing, log records to stable storage during strategic points of commit processing to ensure atomicity in a dis-

tributed context. Given today's slow disk times relative to CPU, memory, and network speeds, the forced-writes impose considerable overhead on runtime transaction performance.

3. They require a complex log-based recovery algorithm to restore the database to a consistent state after failure. Log-based recovery is both difficult to reason about and difficult to implement correctly.

These costs may be inevitable in single-site database systems, where the log provides the sole source of redundancy and a mechanism for achieving atomic transactions. No compelling evidence exists, however, to show the necessity of a recovery log in distributed databases where data may already be redundantly stored to provide high availability and where some other mechanism for atomicity may be possible.

In distributed database systems, crash recovery and high availability receive attention as disjoint engineering problems requiring separate solutions. Log-based crash recovery frameworks were originally designed only for single-site databases, but high availability solutions in distributed databases have typically just accepted the log stored on each site as a convenient relic without considering what additional leverage redundantly stored data on remote sites may offer toward crash recovery. A crash recovery scheme for distributed databases designed with high availability in mind may lead to a cleaner and more efficient solution.

1.3 An Integrated Approach to Crash Recovery and High Availability

In this thesis, I design, implement, and evaluate HARBOR (High Availability and Replication-Based Online Recovery)—a new approach to recoverability, high availability, and disaster recovery in an updatable warehouse. HARBOR is loosely inspired by the column-oriented C-Store system [50], which also seeks to provide an updatable, read-mostly store. I conduct my evaluation on a row-oriented database

system, however, to separate the performance differences due to my approach to recovery from those that result from a column-oriented approach.

The core idea behind HARBOR is that any highly available database system will use some form of data replication to provide availability; I demonstrate that it is possible to capitalize on the redundancy to provide simple and efficient crash recovery without the use of an on-disk log, without forced-writes during commit processing, and without a complex recovery protocol like ARIES [37]. Instead, I accomplish crash recovery by periodically ensuring, during runtime, that replicas are consistent up to some point in history via a simple checkpoint operation and then using, during recovery time, standard database queries to copy any changes from that point forward into a replica. HARBOR does not require replicated data to be stored identically, and crash recovery of multiple database objects or multiple sites can proceed in parallel without quiescing the system. As a side effect of this approach to recovery, the system also provides versioning and time travel up to a user-configurable amount of history.

This approach may appear to violate the cardinal rule of recoverability, namely, that transactions need to define some durable atomic action, called the *commit point*, at which they change the database to a new state that reflects the effects of the transaction; the commit point might be the forced-write of a COMMIT log record or the flipping of an on-disk bit to switch shadow pages [42]. In the event of a failure during a transaction, whether stable storage shows that a transaction reached its commit point determines if the database should treat the transaction as having completed successfully. I solve the recoverability problem in HARBOR by recognizing that in highly available systems, the probability that all of the redundant copies of the database fail simultaneously can be made arbitrarily and acceptably small. Therefore, with high probability, some redundant copy will always remain online at any given time and witness the effects of a transaction. Crashed sites can capitalize on the redundancy and query the live sites to obtain any missing information.

Though conceptually simple, there are two major challenges to implementing this approach:

- If done naively, recovery queries can be very slow. HARBOR attempts to ensure that little work needs to be performed during recovery and that little state needs to be copied over the network.

- To provide high availability, the remaining replicas must be able to process updates while recovery is occurring. The details of bringing a recovering node online during active updates while still preserving ACID (atomicity, consistency, isolation, and durability) semantics are quite subtle.

To benchmark my new approach, I compare the runtime overhead and recovery performance of HARBOR with the performance of two-phase commit and the log-based ARIES approach on a four-node distributed database. I show that HARBOR provides comparable recovery performance and that it also has substantially lower runtime overhead because it does not require disk writes to be forced at transaction commit. Moreover, I demonstrate that HARBOR can tolerate site failure and recovery without significantly impacting transaction throughput. Though my evaluation in this thesis focuses specifically on distributed databases on a local area network, my approach also applies to wide area networks. Aside from these strengths, HARBOR is also substantially simpler than ARIES.

1.4 Thesis Roadmap

The remainder of my thesis is organized as follows. In Chapter 2, I summarize the related work that has been completed on crash recovery and high availability. Having painted the context for my work, I then lay out my fault tolerance model and provide an overview of the techniques and steps used by HARBOR in Chapter 3. I describe the mechanics behind query execution and commit processing in HARBOR in Chapter 4 and then detail the three phases of my recovery algorithm in Chapter 5. In Chapter 6, I evaluate my recovery approach against two-phase commit and ARIES on a four-node distributed database implementation and highlight some of the implementation details. Lastly, I conclude with my contributions in Chapter 7.

Chapter 2

Related Work

Much literature has been written to address the separate problems of crash recovery and high availability, but few publications exist on integrated approaches to bring a crashed site online in a highly available system without quiescing the system. In this chapter, I survey the literature on single-site crash recovery, high availability in distributed systems, online recovery of crashed sites, and data replication strategies.

2.1 Single-site Crash Recovery

In the context of single-site database systems, the bulk of the literature on crash recovery revolves around log processing. The gold standard for database crash recovery remains the ARIES [37] recovery algorithm, implemented first in the early 1990's in varying degrees on IBM's OS/2, DB2, Quicksilver, and a few other systems.

Using a log-based approach like ARIES, a database management system (DMBS) supports both atomic transactions and crash recovery by maintaining an on-disk undo/redo log that records all transactions. The standard *write-ahead logging* protocol, which requires a system to force the undo and redo log records describing a modified page to stable storage before writing the modified page to disk, ensures that sufficient information exists to make transactions *recoverable*; a recoverable sequence of actions either completes in its entirety or aborts in such a way that no action within the sequence appears to have occurred. After a computer failure, the

ARIES algorithm then consists of three phases: an *analysis* pass over the log from the last checkpoint onward to determine the status of transactions and to identify dirty pages, a *redo* phase that employs a *repeating history* paradigm to restore a crashed system to its state directly prior to the crash, and an *undo* pass that scans the log in reverse and rolls back all uncommitted changes.

Numerous optimizations have been developed to improve the performance of the original ARIES algorithm. Oracle's Fast-Start [30] employs an incremental checkpointing technique to increase the frequency of checkpoints without introducing much runtime overhead; the optimization also supports data access during the undo phase of recovery. Mohan [36] describes another technique to enable online data access before the redo phase of recovery by defining a set of conditions under which a transaction can safely execute during recovery without bringing the system to an inconsistent state. In the realm of crash recovery for main memory database systems, Hagmann [23] explores the partitioning of data by update frequency to speed recovery, and Lehman and Carey [31] examine the separation of recovery into a high-speed phase for data immediately required by pending transactions and a slower phase for all remaining data.

2.2 High Availability in Distributed Systems

In the distributed context, most of the literature focuses on high availability techniques rather than on crash recovery; most distributed systems presumably still rely on the single-site log-based approach to crash recovery.

A typical high availability framework might involve a primary database server that asynchronously ships update requests or transaction logs to some number of identical secondary replicas, any of which can be promoted to the primary as a failover mechanism [11]. This primary copy approach suffers from the setback that the secondary replicas store temporarily inconsistent copies of data and therefore cannot help load balance up-to-date read queries without sacrificing transactional semantics; the replicas play a major role only in the case of failover, and some other

recovery protocol is necessary to restore a failed site. This approach is widely used in the distributed systems community [32] as well as in many distributed databases (e.g., Sybase's Replication Server [51], Oracle Database with Data Guard [40], Microsoft SQL Server [34], and most other commercial systems).

A second approach to high availability is to maintain multiple identical replicas that are updated synchronously, using a two-phase commit protocol [38] in conjunction with write-ahead logging to guarantee distributed atomicity. Many places in the literature describe this protocol; see Gray [19] or Bernstein [9] for an overview. This second approach unfortunately suffers in transaction throughput because two-phase commit requires the coordinator site to force-write log records during the COMMIT phase and the worker sites to force-write them during both the PREPARE and COMMIT phases. Disk I/O speeds pale in comparison to CPU, memory, and network speeds and become a major bottleneck of the approach. The approach again also requires correctly implementing a complex recovery protocol like ARIES, which is non-trivial.

Most highly available database systems rely on special purpose tools (e.g., an approach like the one described by Snoeren *et al.* [49]) to handle failover in the event that a replica the client is talking to fails. To enable a system using HARBOR to provide a seamless failover and prevent the client from seeing a broken connection, the system would need similar tools.

2.3 Online Recovery of Crashed Sites

The approach most similar to mine is an algorithm for online recovery of replicated data described by Jiménez-Peris *et al.* [27, 29]. Although the approach requires each site to maintain an undo/redo log, the way in which they use the log for distributed recovery follows a similar structure to HARBOR. A recovering site asks an online site to forward it update requests from the online site's log until the recovering site catches up with all past and pending transactions, at which point, the recovering site stops listening to forwarded requests, comes online, and applies

updates from its local queue of update requests.

Their scheme differs from HARBOR, however, in several aspects. First, their scheme requires maintaining a recovery log (two logs in fact) whereas an online site in HARBOR can supply missing updates to a recovering site simply by inspecting its own data. Second, to handle the difficult race conditions surrounding the time at which a recovering site comes online, they assume a middleware communication layer that supports reliable causal multicast and that can deliver messages in the same order to all sites; HARBOR solves this problem using a more conventional locking solution that can be implemented straightforwardly in any database. Third, their scheme assumes that all sites store data identically whereas HARBOR does not. Because HARBOR imposes none of these assumptions, my integrated approach is a more general scheme. To my knowledge, nothing has been written on logless approaches to online crash recovery of databases.

2.4 Data Replication Strategies

Another line of research into high availability focuses on using different data placement strategies for increasing availability. Hsiao and Dewitt [25, 26] propose a chained declustering technique for laying out data to decrease the probability of data loss in multiprocessor database systems. They compare their technique to Tandem's mirrored disk architecture [52] and Teradata's interleaved declustering scheme [53]. These techniques could also be incorporated into a HARBOR-protected system to increase availability.

In my replication approach, a system propagates all writes to $K + 1$ sites and can tolerate failures of up to K sites without sacrificing availability; the approach allows the system to process read-only queries at just a single site. Voting-based approaches [18, 28, 44] could be applied to allow a system to send updates to fewer machines by reading from several sites and comparing timestamps from those sites. View-change protocols [2] build upon these ideas to allow replicated systems to potentially tolerate network partitions and other failures that I do not address.

Allowing the system to proceed without ensuring replicas have all seen the same transactions may make achieving one-copy serializability [10] difficult.

Chapter 3

Approach Overview

In Chapter 2, I painted the backdrop of existing research literature on the crash recovery and high availability approaches of various database systems.

In this chapter, I focus on two specific features of the landscape for highly available data warehouses—the requirement for data replication and the concurrency control issues associated with warehouse workloads. I show that a fault tolerance model built on data replication and a concurrency control technique called historical queries can be together leveraged to build a simple and efficient crash recovery algorithm, and I offer a high-level overview of that algorithm in HARBOR.

3.1 Data Warehousing Techniques

Data warehouse systems share two important characteristics. First, any highly available database system will use some form of replication to ensure that data access can continue with little interruption if some machine fails. Each database object will be replicated some number of times and distributed among different sites. Unlike many commercial high availability approaches [39, 43, 35], HARBOR does not require that the redundant copies be stored in the same way or that different sites be physically identical replicas. Copies of the data can be stored in different sort orders [50], compressed with different coding schemes [50, 1], or included in different materialized views [20, 21, 15] as long as the redundant copies logically

represent the same data. This flexibility in physical data storage enables the query optimizer and executor to more efficiently answer a wider variety of queries than they would be able to in situations where the database stores redundant copies using identical representations. As an illustration of the performance gains available through storing data in multiple sort orders and in different compression formats, C-Store [50] achieves an average of 164 times faster performance over a commercial row-oriented database and 21 times faster performance over a commercial column-oriented database on a seven query TPC-H benchmark [54]. Data redundancy can therefore provide both higher retrieval performance and high availability under HARBOR.

The second characteristic is that data warehouses cater specifically to large ad-hoc query workloads over large read data sets intermingled possibly with a smaller number of OLTP (online transaction processing) transactions. Such OLTP transactions typically touch relatively few records and only affect the most recent data; for example, they may be used to correct errors made during a recent ETL (extract, transform, and load) session. Under such environments, conventional locking techniques can cause substantial lock contention and result in poor query performance. A common solution is to use *snapshot isolation* [8, 56], in which read-only transactions read data without setting locks, and update transactions modify a snapshot of the data and resolve conflicts either with an optimistic concurrency control protocol or a standard locking protocol.

C-Store [50] solves the lock contention problem using a time travel mechanism similar to snapshot isolation. The solution uses an explicit versioned and timestamped representation of data to isolate read-only transactions so that they can access the database as of some time in the recent past, before which the system can guarantee that no uncommitted transactions remain. Such read queries, which I term *historical queries*, can proceed without obtaining read locks because there is no possibility of conflicts. Update transactions and read-only transactions that wish to read the most up-to-date data use conventional read and write locks as in strict two phase locking for isolation purposes.

In HARBOR, I adopt C-Store's model for concurrency control. As a side effect of this model, historical queries also provide a time travel feature that allows clients to inspect the state of the database as of some time in the past; the feature empowers them to, for instance, compare the outcome of a report both before and after a set of changes.

A key contribution in my thesis is recognizing that one can leverage the pre-existing requirement of data replication in highly available data warehouse environments along with the technique of historical queries to build a simple yet efficient crash recovery mechanism. Together, these two elements of data warehouses serve as the foundation of the HARBOR approach to recovery and high availability.

3.2 Fault Tolerance Model

A system provides *K-safety* if each relation in the database is replicated $K+1$ times, and the copies are distributed to sites in such a way that any *K* of those sites can fail without impairing the system's ability to service any query. Copies of a particular relation need not be physically identical as long as they logically represent the same data. A particular copy also does not need to be stored in its entirety at any given site; each copy may be partitioned horizontally or vertically in different ways and the partitions may themselves be distributed to various sites in a way that preserves *K*-safety. A database designer can achieve *K*-safety straightforwardly by distributing each of the $K+1$ copies of a given relation to a different site; however, more complex configurations are also possible.

Note two observations regarding *K*-safety as defined. First, the minimum number of sites required for *K*-safety is $K+1$, namely in the case where the $K+1$ workers all store a copy of the replicated data. Second, a *K*-safe system can tolerate more than *K* site failures as long as no more than *K* of the failures affect each relation. For example, suppose there exists a 1-safe system that stores copies of relation R at sites S_1 and S_2 and copies of relation R' at sites S'_1 and S'_2. Even though the system is only 1-safe, it can tolerate the failures of both S_1 and S'_1 because at most

one failure affected each relation. [1]

In my approach, I abstract away the different possible *K*-safety configurations and assume that the database designer has replicated the data and structured the database in such a manner as to provide *K*-safety. The high availability guarantee of HARBOR is that it can tolerate up to *K* failures of a particular relation and still bring the failed sites online within a reasonable amount of time. As long as each relation remains available at some combination of the online sites, HARBOR can revive all crashed sites.

If more than *K* sites that support a particular relation fail simultaneously, HARBOR no longer applies, and the recovery mechanism must rely on other methods; the other method most likely will involve removing the effects of all updates after some point in time in order to bring all sites to a globally consistent state. However, because HARBOR can be applied to bring sites online as they fail and because the probability of *K* simultaneous failures decreases exponentially with increasing *K*, the database designer can choose an appropriate value of *K* to reduce the probability of *K* simultaneous site failures down to some value acceptable for the specific application.

In my fault tolerance model, I assume fail-stop failures and do not deal with Byzantine failures. Therefore, I do not deal with network partitions or with corrupted data caused by half-written disk pages. I also assume reliable network transfers via a protocol such as TCP.

3.3 Historical Queries

Since HARBOR's recovery approach depends on historical queries, I first describe how they can be supported in a database system before proceeding to paint a high

[1] Technically, a *K*-safe system may be able to tolerate more than *K* failures that affect a given relation. The definition of *K*-safety specifies that a system must tolerate the failure of *any K* sites dealing with the particular relation but leaves open the possibility that the system may actually be able to tolerate failures of certain combinations of more than *K* sites. HARBOR can actually apply as long as some mechanism exists to detect that a relation can still be reconstructed from the remaining live sites, but for simplicity, I do not discuss this case.

insertion_time	deletion_time	employee_id	name	age
1	0	1	Jessica	17
1	3	2	Kenny	51
2	0	3	Suey	48
4	6	4	Elliss	20
6	0	4	Ellis	20

Figure 3-1: Sample sales table with timestamps. Shaded rows indicate that the tuples have been deleted.

level description of the recovery algorithm used in HARBOR. A historical query as of some past time T is a read-only query that returns a result as if the query had been executed on the database at time T; in other words, a historical query at time T sees neither committed updates after time T nor any uncommitted updates.

Historical queries can be supported by using a *versioned* representation of data in which timestamps are associated with each tuple. The system internally augments a tuple $<a_1, a_2, ..., a_N>$ with an *insertion timestamp* field and a *deletion timestamp* field to create a tuple of the form $<$*insertion-time, deletion-time,* $a_1, a_2, ..., a_N>$. Timestamp values are assigned at commit time as part of the commit protocol. When a transaction inserts a tuple, the system assigns the transaction's commit time to the tuple's insertion timestamp; it sets the deletion timestamp to 0 to indicate that the tuple has not been deleted. When a transaction deletes a tuple, rather than removing the tuple, the system assigns the transaction's commit time to the tuple's deletion timestamp. When a transaction updates a tuple, rather than updating the tuple in place, the system represents the update as a deletion of the old tuple and an insertion of the new tuple.

Example: Figure 3-1 illustrates a simple employees table with insertion and deletion timestamps to support historical queries. At time 1, the user inserts the first two tuples into the table. At time 2, the third tuple is inserted. The deletion timestamp of 3 on the second tuple indicates that the second tuple was deleted at time 3. At time 4, the user inserts the fourth tuple. Nothing happens at time 5. At time 6, the user corrects a misspelling in the last employee's name with an update; note that the

update is represented by a deletion of the previous tuple followed by an insertion of the corrected tuple.

Structuring the database in this manner is reminiscent of version histories [45] and preserves the information necessary to answer historical queries. The problem of answering a historical query as of some past time T then reduces to determining the visibility of a particular tuple at time T. A tuple is visible at time T if 1) it was inserted at or before T, and 2) it either has not been deleted or was deleted after T. This predicate can be pushed down straightforwardly into access methods as a SARGable predicate [46]. Because historical queries view an old time slice of the database and all subsequent update transactions use later timestamps, no update transactions will affect the output of a historical query for some past time; for this reason, historical queries do not require locks.

A user can configure the amount of history maintained by the system by running a background process to remove all tuples deleted before a certain point in time or by using a simple "bulk drop" mechanism that I introduce in the next chapter.

3.4 Recovery Algorithm Overview

Having discussed the key elements of data replication and historical queries, I proceed to describe at a high level HARBOR's recovery algorithm to bring a crashed site online. I defer elaboration of the details to Chapter 5. The algorithm consists of three phases and requires executing read queries in a special mode that enables the recovery process to read the insertion and deletion timestamps of both present and deleted tuples; the database can support the special mode simply by treating the insertion and deletion timestamps as regular fields and disabling the machinery for determining tuple visibility.

In the first phase, HARBOR uses a checkpointing mechanism to determine the most recent time T with the property that all updates up to and including time T have been flushed to disk at the recovering site. The recovering site then uses

```
procedure checkpoint():
    let T = current time - 1
    obtain snapshot of dirty pages table
    for each page P in table:
        if P is dirty
            acquire write latch on P
            flush P to disk
            release latch on P
    record T to checkpoint file
```

Figure 3-2: Checkpointing algorithm.

the timestamps available for historical queries to run local update transactions to restore itself back to the time of its last checkpoint.

In order to record periodic checkpoints, I assume that the buffer pool maintains a standard *dirty pages table* with the identity of all in-memory pages and a flag for each page indicating whether it contains any changes not yet flushed to disk. During normal processing, the database writes a checkpoint for some past time T by first taking a snapshot of the dirty pages table at time $T + 1$. For each dirty page in the snapshot, the system acquires a write latch for the page, flushes the page to disk, and releases the latch. After flushing all dirty pages, it records T onto some well-known location on disk, and the checkpoint is complete. Figure 3-2 summarizes these steps. My experimental evaluations on simple insert-intensive workloads suggest that updating this checkpoint once every 1–10 s imposes little runtime overhead; if checkpointing turns out to have an adverse impact on runtime performance, however, database designers can also adopt an incremental checkpointing approach much like the one used by Oracle Fast-Start [30], which has lower runtime overhead.

In the second phase, the recovering site executes historical queries on other live sites that contain replicated copies of its data in order to catch up with any committed changes made between the last checkpoint and some time closer to the present. The ability to run historical queries without obtaining read locks is crucial to ensuring that the system does not need to be quiesced while the recovering site copies potentially large amounts of data over the network. My approach would not be a viable solution if the site needed to acquire read locks for this recovery query.

In the third and final phase, the recovering site executes standard non-historical queries with read locks to catch up with any committed changes between the start of the second phase and the current time. A clever use of read locks ensures that the recovery algorithm and any ongoing transactions preserve ACID semantics. Because the historical queries in the second phase tend to substantially reduce the number of remaining updates, this phase causes relatively little disruption to ongoing transactions, as I show later in my evaluation in § 6.5. The coordinator then forwards any relevant update requests of ongoing transactions to the recovering site to enable it to join any pending transactions and come online; this step requires that the coordinator maintain a queue of update requests for each ongoing transaction.

This overview reveals a critical property of HARBOR's recovery algorithm. Unlike other recovery approaches, it is not log-based; rather, it leverages the redundantly stored data at other sites to perform recovery. Hence, as I show in the next chapter, a HARBOR-protected system can eliminate much of the runtime performance overhead associated with maintaining on-disk logs, undo/redo log records, and forced-writes.

My results in Chapter 6 demonstrate that HARBOR's recovery algorithm works well in data warehouse-like environments where the update workloads consist primarily of insertions with relatively few updates to historical data. Its performance surpasses the recovery performance of a log-based processing protocol like ARIES under these conditions.

Chapter 4

Query Execution

In the previous chapter, I introduced the notions of *K*-safety and historical queries and offered a high level overview of HARBOR's recovery approach. In this chapter, I drill down into the necessary design modifications to a standard database's query execution engine in order to support HARBOR's data replication, concurrency control, and crash recovery goals.

Query execution in HARBOR differs in three respects from standard distributed databases. First, the versioned representation of data and the support for historical queries bring about some changes to the distributed transaction model, which I outline in § 4.1. Second, in order to improve performance of HARBOR's recovery queries that must apply range predicates on timestamps, I partition all database objects by insertion time into smaller chunks called *segments*. I describe this segment architecture, its features, and its implications in § 4.2. Third, because HARBOR does not require worker sites to log updates to disk, I can reduce commit processing overhead by eliminating workers' forced-writes when using two-phase commit and all forced-writes when using three-phase commit. I describe these optimized variants of both commit protocols in § 4.3.

4.1 Distributed Transaction Model

In my framework, each transaction originates from a *coordinator* site responsible for distributing work corresponding to the transaction to one or more *worker* sites and for determining the ultimate state of the transaction; the coordinator site may also be a worker site for the same transaction. The coordinator may distribute read queries to any sites with the relevant data that the query optimizer deems most efficient. Update queries, however, must be distributed to all live sites that contain a copy of the relevant data in order to keep all sites consistent; crashed sites can be ignored by update queries because the sites will recover any missing updates through the recovery algorithm described in Chapter 5.

To facilitate recovery, the coordinator maintains an in-memory queue of logical update requests for each transaction it coordinates. Each update request can be represented simply by the update's SQL statement or a parsed version of that statement. The coordinator can safely delete this queue for a particular transaction when the transaction commits or aborts.

In order to support historical queries, the coordinator determines, at transaction commit time, the insertion and deletion timestamps that must be assigned to any tuples inserted, deleted, or updated at the worker sites. If the database supports more than one coordinator site, HARBOR requires some consensus protocol for the coordinators to agree on the current time; [50] describes one such protocol in which a designated timestamp authority decides with the coordinators to advance the system time. The timestamps can be arbitrarily granular and need not correspond to real time. For instance, one might choose to use coarse granularity epochs that span multiple seconds or timestamps with second granularity. Finer granularity timestamps allow for more fine-grained historical queries but require more synchronization overhead if there are multiple coordinator sites. The database frontend holds responsibility for mapping the times used by clients for historical queries to the internal representation of timestamps in the database.

As previously mentioned in the discussion on historical queries, a worker assigns

timestamps not as it executes update requests but rather at transaction commit time; therefore, a worker must keep some in-memory state for each update transaction designating the tuples whose timestamps must later be assigned. Furthermore, in order to handle aborts, a worker must be capable of identifying and rolling back any changes associated with a particular transaction. A traditional database would handle aborts using the same on-disk undo/redo log used for crash recovery. With HARBOR, however, the log is not used for crash recovery, and so the log can be stored in memory without using conventional protocols like forced-writes or write-ahead logging. Moreover, because updates and deletes do not overwrite or clear previously written data, the "log" does not even need to store undo/redo information; it merely needs to identify modified tuples.

A worker site can support both normal commit processing and transaction aborts straightforwardly by maintaining, for each transaction, an in-memory *insertion list* of tuple identifiers for tuples inserted by the transaction and another in-memory *deletion list* for tuples deleted by the transaction; note that an updated tuple would go in both lists. To commit the transaction, the worker assigns the commit time to the insertion time or deletion time of the tuples in the insertion list or deletion list, respectively. To roll back a transaction, the worker simply removes any newly inserted tuples identified by the insertion list; no tuples need to be undeleted because deletion timestamps are assigned at transaction commit.

One caveat is that if the buffer pool supports a STEAL policy [19] and allows uncommitted data to be flushed to disk, uncommitted inserts written to the database should contain a special value in its insertion timestamp field so that the uncommitted data can be ignored by queries and identified in the event of crash recovery. Because deletes only assign deletion timestamps and because a site cannot determine the deletion timestamps until commit time, there is no reason for the database to write uncommitted deletions to the buffer pool.

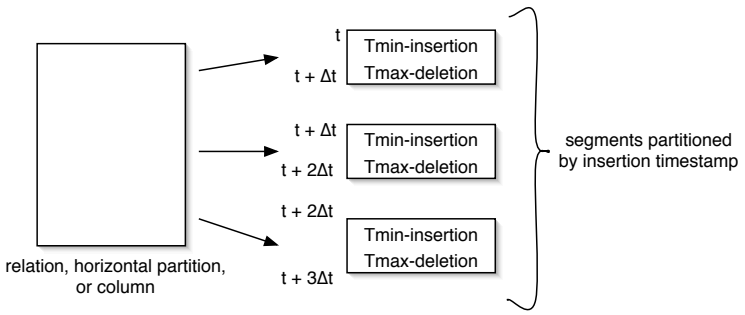

Figure 4-1: Database object partitioned by insertion timestamp into segments.

4.2 Segment Architecture

During normal transaction processing, the query executor uses the insertion and deletion timestamps associated with tuples solely to determine a tuple's visibility as of some time T. In data warehouse environments where users typically run historical queries as of some time relatively close to the present and where updates to historical data happen infrequently, most of the tuples examined by a query are visible and relevant. Thus, the executor incurs little overhead scanning invisible tuples, and one does not need to build indices on timestamps for this purpose.

As I explain later in the discussion on recovery, however, recovery queries require identifying those tuples modified during specific timestamp ranges and may be potentially inefficient. Specifically, recovery requires the efficient execution of the following three types of range predicates on timestamps: *insertion-time* $\leq T$, *insertion-time* $> T$, and *deletion-time* $> T$. Moreover, given that recovery is a relatively infrequent operation, one would like to be able support these range predicates without requiring a primary index on timestamps (a secondary index would not be useful in this case).

My architectural solution to this problem is to partition any large relations, vertical partitions, or horizontal partitions stored on a site by insertion timestamp into smaller chunks called *segments*, as illustrated in Figure 4-1. For example, a horizontal partition for the relation *products* sorted by the field *name* that is stored on one site might be further partitioned into one segment containing all tuples inserted

between time t and $t + \Delta t$, the next segment containing all tuples inserted between $t + \Delta t$ and $t + 2\Delta t$, etc. The size of the time ranges need not be fixed, and one may instead choose to limit the size of each segment by the number of constituent pages. Each segment in this example would then individually be sorted according to *name*. If the original *products* table required an index on some other field, say *price*, each segment would individually maintain an index on that field. The query executor adds new tuples to the last segment until either the segment reaches some pre-specified size or until the current time passes the end of the segment's time range; in either case, when a segment becomes full, the executor creates a new segment to insert subsequent tuples.

To efficiently support the desired range predicates, I annotate each segment with its minimum insertion time $T_{min-insertion}$, which is determined when the first tuple is added to a newly created segment. An upper bound on the insertion time of a segment can be deduced from the minimum insertion time of the next segment or is equal to the current time if no next segment exists. I also annotate each segment with the most recent time $T_{max-deletion}$ that a tuple has been deleted or updated from that segment; recall that an update is represented as a deletion from the tuple's old segment and an insertion into the most recent segment.

With this structure, the database can greatly reduce the amount of data that must be scanned to answer a range query on timestamps during recovery. To find tuples satisfying the predicate *insertion-time* $\leq T$, the database ignores all segments with $T_{min-insertion} > T$; similarly, to find tuples satisfying the predicate *insertion-time* $> T$, it ignores all segments coming before a segment with $T_{min-insertion} \leq T$. To find tuples with *deletion-time* $> T$, it prunes all segments with $T_{max-deletion} \leq T$. The segment timestamps thus enable recovery queries to vastly reduce their search space for valid tuples.

These changes to the physical representation require some modifications to the standard query execution engine. Read queries on a segmented object must now be conducted on multiple segments and may require an additional seek or index lookup per segment, and the results from each segment must be merged together;

however, this merge operation is no different from the merge operation that must occur on any distributed database to combine results from different nodes, except that it is performed locally on a single node with results from different segments. Update queries may similarly need to examine multiple segments or traverse multiple indices, starting from the most recent segment, to find desired tuples. The distributed database implementation that I evaluate in Chapter 6 shows some of the overhead for sequential scan queries, indexed update queries, and inserts on a segmented architecture.

In return for the segment processing overhead, this solution also provides two concrete benefits to data warehouse environments. Many warehouse systems require daily or hourly bulk loading of new data into their databases. Using a segment representation, a database system can easily accommodate bulk loads of additional data by creating a new segment and transparently adding it to the database as the last segment with an atomic operation.

Recent years have also seen the rise of massive clickthrough warehouses, such as Priceline, Yahoo, and Google, that must store over a terabyte of information regarding user clicks on websites. These warehouse systems are only designed to store the most recent N days worth of clickthrough data. My time-partitioned segment architecture supports a symmetric "bulk drop" feature whereby one can, for instance, create segments with time ranges of a day and schedule a daily drop of the oldest segment to make room for fresh data. These bulk load and bulk drop features would require substantially more engineering work under other architectures.

4.3 Commit Processing

In distributed database systems, a distributed commit protocol enables multiple coordinator and worker sites to atomically agree to commit or abort a particular transaction; reasons for which workers may abort a transaction include deadlocks and consistency constraint violations. The multi-phased commit protocols apply only to update transactions; for read transactions, the coordinator merely needs to notify

the workers to release any system resources and locks that the transaction may be using. One significant benefit that HARBOR confers is the ability to support more efficient commit protocols for update transactions by eliminating forced-writes to an on-disk log.

In this section, I start with the traditional two-phase commit protocol (2PC) [38] used in conventional distributed databases and explain how my versioned representation of data can be supported with a few minor adjustments. I then dissect certain parts of 2PC and show that many of the expensive log-related steps of 2PC can be eliminated to create an optimized variant of 2PC by leveraging the guarantees of *K*-safety and HARBOR. Finally, I present an observation that enables a HARBOR-protected database system to employ a logless variant of a less widely used, but non-blocking, three-phase commit protocol (3PC) [47] that achieves even higher runtime performance.

4.3.1 The Traditional Two-Phase Commit Protocol

The traditional mechanism for supporting atomicity in distributed transactions is the two-phase commit (2PC) protocol [38], illustrated in Figure 4-2. Variants of 2PC including Presumed Abort and Presumed Commit [38] also exist. A coordinator initiates 2PC after the client commits the transaction and after the last worker has acknowledged that it has completed the last update request for that transaction.

In the first phase of traditional 2PC, the coordinator moves a transaction from the *pending* state to the *in-doubt* state and sends a PREPARE message to each worker. Each worker willing to commit the transaction then force-writes a PREPARE log record, moves the local state of the transaction from the *pending* state to the *prepared* state, and sends a YES vote to the coordinator; each worker wishing to abort the transaction force-writes an ABORT log record, moves the transaction to the *aborted* state, rolls back any changes, releases the locks for the transaction, and sends a NO vote to the coordinator.

In the second phase, the coordinator force-writes a COMMIT log record if it

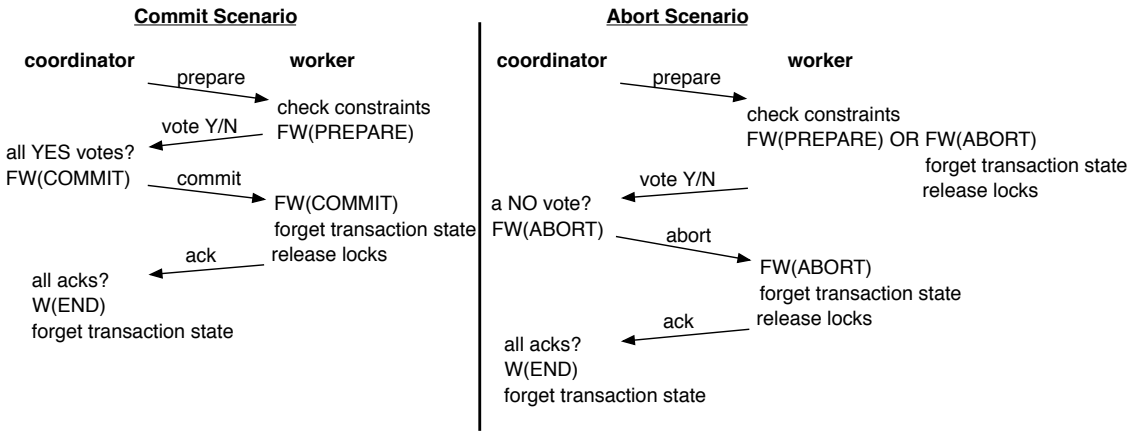

Figure 4-2: Commit and abort scenarios of the traditional two-phase commit protocol. FW signifies force-write, and W signifies a normal log write. In the abort scenario, only workers that did not abort in the first phase continue onto the second phase.

received YES votes from all workers and then sends COMMIT messages to all workers; otherwise, if it received a NO vote or if a worker site crashed, the coordinator force-writes an ABORT log record and sends an ABORT message to all workers in the *prepared* state. In either case, the logging of the COMMIT or ABORT record is known as the *commit point* of the transaction; once the log record reaches stable storage, the transaction is considered to be *committed* or *aborted*, and the coordinator can notify the client of the outcome.

If a worker receives a COMMIT message in the second phase, the worker moves the transaction to the *committed* state, releases the locks for the transaction, force-writes a COMMIT record, and sends an ACK to the coordinator. If a worker receives an ABORT message, the worker moves the transaction to the *aborted* state, rolls back any changes, releases the locks for the transaction, force-writes an ABORT record, and sends an ACK to the coordinator. When the coordinator receives ACKs from all workers that it sent a message to in the second phase, it logs an END record and forgets the transaction state.

To support my timestamped representation of data, I augment 2PC with two minor changes: 1) COMMIT messages also include a commit time to be used by worker sites for all tuples modified by a particular transaction, and 2) the in-memory inser-

tion and deletion lists of that a worker maintains for an ongoing transaction can be deleted when the transaction commits or aborts.

4.3.2 An Optimized Two-Phase Commit Protocol

The three non-overlapping forced-writes of log records (two by each worker and one by the coordinator) increase transaction latency by several milliseconds each and can easily become the performance bottleneck of update-intensive workloads. The evaluations in § 6.3 indicate that the forced-writes can increase transaction latency by over an order of magnitude on a simple insert-intensive workload on a four-node distributed database system. Modern systems use a technique known as group commit [16, 24] to batch together the log records for multiple transactions and to write the records to disk using a single disk I/O. In the distributed context, sites must still ensure that forced-writes reach stable storage before sending out messages, but group commit can help increase transaction throughput for non-conflicting concurrent transactions. Latency, however, is not improved.

One key observation regarding 2PC is that the forced-writes by the worker sites are necessary only because log-based recovery requires examination of the local log to determine the status of old transactions. After a worker fails and restarts, the forced COMMIT or ABORT record for a transaction informs the recovering site about the outcome of the transaction, while the PREPARE record in the absence of any COMMIT or ABORT record informs the recovering site that it may need to ask another site for the final consensus.

If a worker can recover without using a log, then the forced-writes and, in fact, any log writes become unnecessary; this condition exists within HARBOR's framework. When *K*-safety exists, a crashed worker does not need to determine the final status of individual transactions for recovery purposes. As long as all uncommitted changes on disk can be identified from the special insertion timestamp value associated with uncommitted tuples, worker sites can roll back uncommitted changes and query remote replicas for all committed updates at recovery time. It is this re-

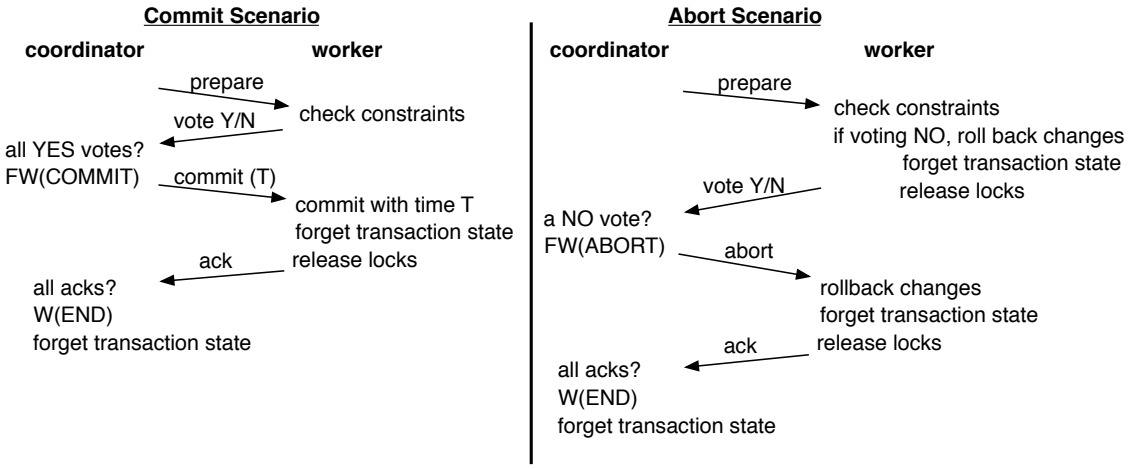

Figure 4-3: Commit and abort scenarios of the optimized two-phase commit protocol. FW signifies force-write, and W signifies a normal log write.

alization that enables worker sites under the HARBOR approach to eliminate both the forced-writes to stable storage and the use of an on-disk log all together.

My optimized 2PC therefore looks like the interaction shown in Figure 4-3. A worker site, upon receiving a PREPARE message, simply checks any consistency constraints and votes YES or NO. When a worker site receives a COMMIT message, it simply assigns the commit time to the modified tuples; when a worker site receives an ABORT message, it rolls back all changes. In either case, the worker then sends an ACK to the coordinator. I examine the performance gains of eliminating the workers' logs and their two forced-writes in § 6.3.

Note that despite the absence of a log write in the PREPARE phase, the protocol still requires workers to be *prepared* because the database schema may have integrity constraints that can only be verified at the end of a transaction; in other words, a particular worker site may still need the option of aborting the transaction after the transaction's operations have completed. In special frameworks where the database schemas have the property that workers can verify integrity constraints after each update operation, the PREPARE phase can be removed from this protocol to produce a one-phase commit protocol [4, 3] without logging; in general, however, these assumptions do not hold, and I do not further consider this specific case in my thesis.

Handling Failures during Optimized 2PC

If a coordinator receives no response from a worker during the optimized 2PC protocol, either because a faulty network dropped the worker's response or because the worker crashed, the coordinator assumes the worker aborted the transaction and voted NO. In accordance with this logic, if a worker crashes, recovers, and subsequently receives a vote request for an unknown transaction, the worker responds with a NO vote.

If a coordinator fails, a worker site can safely abort the transaction if either a) the worker has not received a PREPARE message for the transaction, *i.e.*, the transaction is still *pending*, or if b) the worker has received a PREPARE message but has voted NO. Otherwise, the worker needs to wait for the coordinator to recover before completing the transaction. That the worker needs to wait for the coordinator to recover stems from the *blocking* nature of 2PC; I describe an optimized 3PC protocol that avoids this problem in the next section.

4.3.3 An Optimized Three-Phase Commit Protocol

Under my optimized 2PC, the coordinator still must force-write a COMMIT or ABORT log record prior to sending out COMMIT or ABORT messages to workers. The reason is that if a coordinator fails after informing the client of the transaction's outcome but before receiving ACKs from all of the workers, it needs to examine the log to reconstruct the transaction's status prior to the crash so that it can respond to worker inquiries regarding the transaction's final outcome upon rebooting.

The key insight in this section is that if workers can eliminate their dependency on a crashed coordinator and resolve a transaction by themselves, then the coordinator's log becomes unnecessary because a recovering coordinator can be absolved of responsibility for transactions that it coordinated prior to the crash. Unfortunately, 2PC does not provide workers with sufficient information to recover a transaction after a coordinator crash. Suppose that a coordinator fails after sending a COMMIT message to one worker, and the worker also fails; then the remaining

workers cannot proceed because they do not have sufficient information among themselves to determine whether the transaction has committed, has aborted (possibly due to a timeout from a slow network), or is still pending.

The canonical 3PC protocol solves this *blocking* problem by introducing an additional *prepared-to-commit* state between the *prepared* state and the *committed* state. The additional state enables the workers to use a consensus building protocol, described in the next section, to agree on a consistent outcome for a transaction without depending on a crashed coordinator or other worker site to ever come back online. Under this framework, workers execute the consensus building protocol if they detect a coordinator crash. The coordinator, upon restart, responds to worker inquiries by asking the workers to execute the consensus building protocol rather than following the conventional "if no information, then abort" rule of 2PC [38]. The *non-blocking* property of 3PC means that coordinators in 3PC do not need to force-write log records or maintain an on-disk log. [1]

No part of this discussion affected the ability of workers to recover without a log. With *K*-safety, the system can tolerate *K* failures to a database object and still recover any worker data through the recovery algorithm to restore committed data and the consensus building protocol to resolve any ongoing transactions. Combining the canonical 3PC protocol, which does not require coordinator log writes, with my 2PC optimizations, which eliminate worker log writes, therefore leads to an optimized 3PC protocol that eliminates all forced-writes and log writes by all participants. My optimized 3PC protocol is illustrated in Figure 4-4 and works as follows.

In the first phase, the coordinator sends a PREPARE message to all workers with a list of site identifiers for the workers participating in the transaction. A worker willing to commit the transaction after checking any consistency constraints enters the *prepared* state and responds with a YES vote; a worker unable to commit enters the *aborted* state and responds with a NO vote.

[1] Note that the original proposal for 3PC [47] does not explain log writes for the protocol and that my interpretation of 3PC differs from the one presented by Gupta *et al.* [22]; their 3PC still requires coordinators to maintain a log, presumably because they do not abandon the "if no information, then abort" rule.

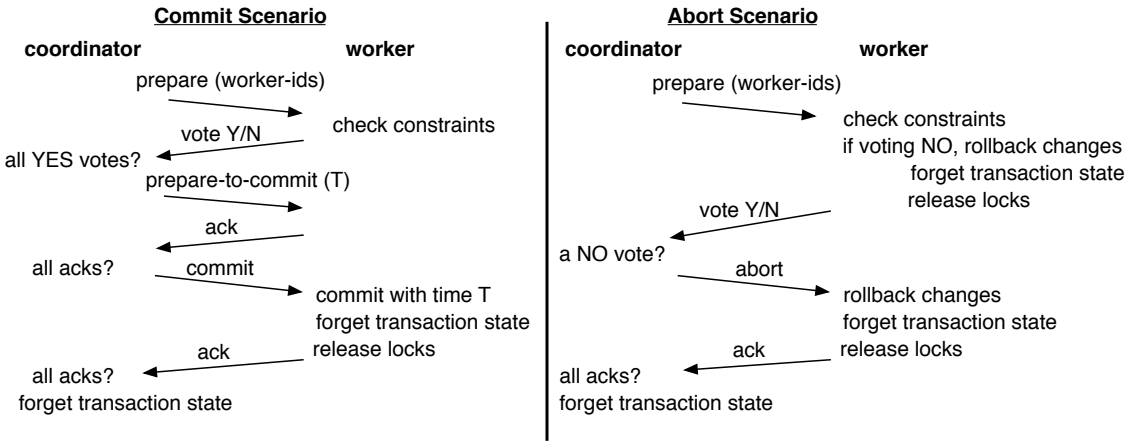

Figure 4-4: Commit and abort scenarios of the optimized three-phase commit protocol.

In the second phase, if the coordinator receives all YES votes in the first phase, it sends a PREPARE-TO-COMMIT message with the commit time to all the workers. A worker enters the *prepared-to-commit* state after receiving the message and replies with an ACK. When all ACKs have been received, the coordinator has reached the commit point in the transaction. If the coordinator had instead received a NO vote, it sends an ABORT message to the workers still in the *prepared* state, and the protocol terminates after the coordinator receives all ACKs.

Finally, in the third phase for transactions to be committed, the coordinator sends the final COMMIT message. Upon receiving the COMMIT, workers enter the *committed* state and can assign the commit time to modified tuples, forget any state for the transaction, and release its locks.

Correctness of my logless 3PC protocol can be seen assuming a correct consensus building protocol, which I describe in the next section, and a correct logless recovery algorithm, which I describe in the next chapter. If a worker fails during a transaction, the coordinator behaves as with 2PC: it commits with the remaining workers if it has already received a YES vote from the failed worker and receives YES votes from all other workers; otherwise, it aborts. In either case, the failed worker, upon executing the recovery algorithm, undoes any uncommitted changes from that transaction and then copies state from other replicas to become consistent

with respect to all transactions. If a coordinator fails, the workers use the consensus building protocol to obtain a consistent transaction outcome, and the coordinator, upon recovery, is no longer responsible for the transaction.

Using this 3PC protocol in conjunction with HARBOR, one can support transaction processing and recovery without forced-writes and without maintaining an on-disk log structure for any sites. When the network is substantially faster than the disk, the extra round of messages introduces less overhead than a forced disk write. I evaluate the runtime performance implications of using this 3PC in § 6.3.

Consensus Building Protocol to Handle Coordinator Failures

If a worker detects a coordinator failure before a transaction's commit processing stage or if a recovered coordinator indicates no knowledge for a particular transaction, the worker can safely abort the transaction as with 2PC. If the coordinator site fails during commit processing, however, a worker that detects the crash can use a consensus building protocol like the one described in [47] and whose correctness is proven in [48]. In the protocol, a backup coordinator is chosen by some arbitrarily pre-assigned ranking or some other voting mechanism from the transaction's worker set and then decides from its local state how to obtain a globally consistent transaction outcome, as follows.

The backup can either be in the *pending* (a.k.a. *unprepared* state), the *prepared* state, the *prepared-to-commit* state, the *aborted* state, or the *committed* state. To aid in the exposition, Figure 4-5 shows the 3PC state transition diagram for a worker site under normal transaction processing. Note that because the state transitions proceed in lock-step, no worker site can be more than one state away from the backup coordinator.

There are four cases:

- If the backup either 1) had not voted in the first phase, 2) is prepared and had voted NO, or 3) had aborted, it sends an ABORT message to all workers because no site could have reached the *prepared-to-commit* state.

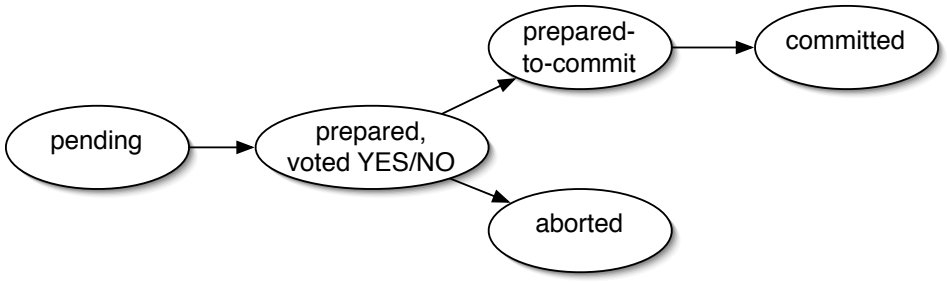

Figure 4-5: State transition diagram for a worker site using three-phase commit.

- If the backup is in the *prepared* state and had voted YES, then no sites could have yet reached the *committed* state (they could have at most reached the *prepared-to-commit* state). The backup therefore asks all sites to transition to the *prepared* state if it is at some other state and waits for an ACK from each worker; it then sends an ABORT message to each worker.

- If it is in the *prepared-to-commit* state, then no site could have aborted. Thus, it replays the last two phases of 3PC, reusing the same commit time it received from the old coordinator, and commits the transaction.

- If it is in the *committed* state, then clearly no site aborted, and so it sends a COMMIT message to all workers. Workers can safely disregard any duplicate messages they receive.

Table 4.1 summarizes the backup's actions for the various states. In order for the backup coordinator to seamlessly handle the failed connection with the client, one can use a special purpose tool like the one described in [49].

Backup Coordinator State	Action(s)
pending	abort
prepared, voted NO	abort
prepared, voted YES	prepare, then abort
aborted	abort
prepared-to-commit	prepare-to-commit, then commit
committed	commit

Table 4.1: Action table for backup coordinator.

Protocol	Messages per Worker	Forced-Writes by Coordinator	Forced-Writes per Worker
2PC	4	1	2
optimized 2PC	4	1	0
3PC	6	0	3
optimized 3PC	6	0	0

Table 4.2: Overhead of commit protocols

4.3.4 Cost Summary of Commit Protocols

The standard strategy for profiling commit protocols, as done in [22], tabulates the number of messages and the number of forced-writes required for a protocol and measures the overhead via a simulation or an online system. Table 4.2 shows the messages and forced-writes required for the four commit protocols discussed thus far, in terms of the number of messages sent by a coordinator to each worker, the number of forced-writes by a coordinator, and the number of forced-writes by a worker. [2]

I evaluate the commit processing overhead of these four protocols on my four-node distributed database implementation in § 6.3.

4.3.5 Additional Advantages of HARBOR

Aside from eliminating the need for forced-writes and an on-disk log, HARBOR also confers two additional advantages to transaction processing not available to conventional database systems. In a conventional system, a coordinator must abort a transaction if a worker crashes in the midst of a transaction. In a HARBOR-protected system, however, if a worker crashes before commit processing for a transaction has begun, the coordinator can opt instead to commit the transaction with *K-1*-safety rather than aborting it. The crashed worker will simply recover the committed data when it runs the recovery algorithm to come back online.

As a corollary to this first advantage, a coordinator can also "crash" a worker

[2] Note that the 3PC analysis in [22] assumes that the coordinator also logs, which is not strictly necessary.

site that is bottlenecking a particular *pending* transaction due to network lag, deadlock, or some other reason and proceed to commit the transaction with *K-1*-safety; the "crashed" worker site would then be forced to undergo recovery to recover the committed changes. Obviously, the coordinator must take measures to ensure that *K*-safety is not lost if it takes this action (or if multiple coordinators decide to execute this technique), but HARBOR at least allows the flexibility for such a technique to exist.

Chapter 5

Recovery

Having discussed the design modifications to query execution required to support the HARBOR framework, I now proceed to discuss the central recovery algorithm of this thesis. In this chapter, I first introduce some terminology to clarify my recovery discussion. I then present the three phases of HARBOR's recovery algorithm to bring a crashed site online and conclude with how HARBOR deals with site failures during recovery.

5.1 Terminology

Let *S* be the failed site. I describe recovery in terms of bringing a particular database object *rec* on *S* online, where *rec* may be a table, a horizontal or vertical partition, or any other queryable representation of data. Indices on an object can be recovered as a side effect of adding or deleting tuples from the object during recovery. Site recovery then reduces to bringing all such database objects on *S* online. HARBOR permits multiple *rec* objects and even multiple sites to be recovered in parallel; each object proceeds through the three phases at its own pace.

Assuming that *S* crashed within the specifications of the fault tolerance model (*i.e.*, while the system still maintained *K*-safety for $K \geq 1$), the system can still continue to answer any query. Therefore, there must exist at least one collection of objects *C* distributed on other sites that together cover the data of *rec*. Call each

of these objects in *C* a *recovery object*, and call a site containing a recovery object a *recovery buddy*. For each recovery object, one can compute a *recovery predicate* such that a) the sets of tuples obtained by applying the recovery predicates on their corresponding recovery objects are mutually exclusive, and b) the union of all such sets collectively cover the object *rec*.

> *Example:* Suppose that a database stores two replicated tables EMP1 and EMP2 representing the logical table *employees*. EMP1 has a primary index on *salary*, and EMP2 has a primary index on *employee_id*; the database designer has designated different primary indices on the two copies to efficiently answer a wider variety of queries. Suppose EMP2A and EMP2B are the two and only horizontal partitions of EMP2; EMP2A is on site *S1* and contains all employees with *employee_id* < 1000, and EMP2B is on site *S2* and contains all employees with *employee_id* ≥ 1000. Finally, let *rec* on the recovering site *S* be the horizontal partition of EMP1 with *salary* < 5000. In this example, *S1* would be a recovery buddy with the recovery object EMP2A and the recovery predicate *salary* < 5000; similarly, *S2* would be another recovery buddy with the recovery object EMP2B and the same recovery predicate.

I assume that the catalog stores the information illustrated in the example, which is not unreasonable because the computation of recovery objects and predicates is the same as the computation that the query optimizer would perform in determining which sites to use to answer a distributed query for all employees with *salary* < 5000 when *S* is down.

Having established this terminology, I now discuss in detail the three phases of HARBOR's recovery algorithm. For simplicity, I describe virtually all steps of recovery declaratively using SELECT, INSERT, DELETE, and UPDATE SQL queries and define the semantics of special keywords as they arise. All INSERT, DELETE, and UPDATE statements in the recovery queries refer to their normal SQL semantics in standard databases rather than to the special semantics used for historical queries.

5.2 Phase 1: Restore local state to the last checkpoint

Recall from § 3.4 that a site writes checkpoints during runtime by periodically flushing all dirty pages to disk. Call the time of *S*'s most recent checkpoint $T_{checkpoint}$. During recovery, HARBOR temporarily disables periodically scheduled checkpoints. The last checkpoint guarantees that all insertions and deletions of transactions that committed at or before $T_{checkpoint}$ have been flushed to disk. The disk may also contain some uncommitted data as well as some, but probably not all, data from transactions that committed after the checkpoint. In Phase 1, HARBOR executes two local queries to discard any changes after $T_{checkpoint}$ and any uncommitted changes in order to restore the state of all committed transactions up to $T_{checkpoint}$.

First, HARBOR deletes all tuples inserted after the checkpoint and all uncommitted tuples by running the following query:

```
DELETE LOCALLY FROM rec
SEE DELETED
WHERE insertion_time > T_checkpoint
  OR insertion_time = uncommitted
```

The `LOCALLY` keyword indicates that this query runs on the local site; I will use the keyword `REMOTELY` later to indicate queries that need to run on remote replicas. The semantics of `SEE DELETED` are that rather than filtering out deleted tuples as a standard query would do, HARBOR executes the query in a special mode with delete filtering off so that both insertion and deletion timestamps become visible as normal fields. Also, note that the `DELETE` in the query refers to the standard notion of removing a tuple rather than recording the deletion timestamp. The special `uncommitted` value refers to the special value assigned to the insertion timestamp of tuples not yet committed; in practice, the special value can simply be a value greater than the value of any valid timestamp, which has the benefit that uncommitted tuples are added to the the last segment.

The segment architecture discussion of § 4.2 describes the usage of $T_{min-insertion}$ and $T_{max-deletion}$ annotations on segments to improve the performance of recovery queries. Using the minimum insertion timestamp $T_{min-insertion}$ associated with each

segment, HARBOR can efficiently find the tuples specified by the range predicate on `insertion_time` for this query. Assuming that the database is configured to record checkpoints somewhat frequently, executing this query should only involve scanning the last few segments; I evaluate the cost of this scan later in Chapter 6.

Next, HARBOR undeletes all tuples deleted after the checkpoint using the following query:

```
UPDATE LOCALLY rec SET deletion_time = 0
SEE DELETED
WHERE deletion_time > T_checkpoint
```

Like the `DELETE` in the previous query, the `UPDATE` in this query corresponds to the standard update operation (an update in place), rather than a delete and an insert. Using the maximum deletion timestamp $T_{max-deletion}$ recorded on each segment, HARBOR can prune out any segments whose most recent deletion time is less than or equal to $T_{checkpoint}$. Thus, recovery performance pays the price of sequentially scanning a segment to perform an update if and only if a tuple in that segment was updated or deleted after the last checkpoint. In a typical data warehouse workload, one expects that updates and deletions will be relatively rare compared to reads and that any updates and deletions will happen primarily to the most recently inserted tuples, *i.e.*, in the most recent segment. Thus, in the common case, either no segment or only the last segment should need to be scanned. [1]

After the two queries in the first phase complete, the state of *rec* reflects only those transactions that committed at or before $T_{checkpoint}$.

Example: Figure 5-1 illustrates a sample run of recovery Phase 1 on a simplified version of a sales table. A real warehouse table would

[1] An alternative data representation that would greatly reduce the cost of this recovery query and subsequent recovery queries with range predicates on the `deletion_time` would be to store a separate deletion vector with the deletion times for all tuples in that segment. Recovery could then simply scan the deletion vector to find the desired deleted tuples rather than scan the entire segment. If the deletion vector replaced the `deletion_time` column, however, standard read queries would incur the additional cost of performing a join with the deletion vector to determine tuple visibility. If it were stored redundantly in addition to the `deletion_time` column, updates would need to write the deletion timestamp in two locations. Either recovery optimization may be a potentially worthwhile tradeoff. Ultimately, the deletion vector representation is better suited for column-oriented databases where each column is already stored separately, and range predicates on columns are already the norm.

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	6	Dell Monitor
5	0	Crest
uncommitted	0	Playstation

(a) Table on the crashed site with checkpoint at time 4. Shaded rows indicate rows inserted after the checkpoint and will be deleted by the `DELETE` query.

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	6	Dell Monitor

(b) Table after the `DELETE` query. Shaded cells indicate deletions after the checkpoint that must be undone by the subsequent `UPDATE` query.

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	0	Dell Monitor

(c) Table after recovery Phase 1.

Figure 5-1: Sample run of recovery Phase 1 for a checkpoint recorded at time 4.

contain millions of rows and 20–40 columns. Figure 5-1(a) shows the state of the table on the crashed site. Assuming the site recorded its most recent checkpoint at time 4, the `DELETE` query removes all the shaded tuples inserted either after the checkpoint or by uncommitted transactions to obtain the table shown in Figure 5-1(b). The shaded cell shows a deletion that happened after the checkpoint; the `UPDATE` query undoes this deletion, resulting in the table of Figure 5-1(c).

5.3 Phase 2: Execute historical queries to catch up

The strategy in the second phase is to capitalize on the data redundancy available at remote sites and to leverage historical queries to rebuild *rec* up to some time close to the present. Because historical queries do not require locks, this phase of recovery does not quiesce the system. Let the *high water mark* (HWM) be the time right before *S* begins Phase 2; thus, if *S* begins Phase 2 at time T, let the HWM be $T - 1$. Recall that a historical query as of the time HWM means that all tuples inserted after the HWM are not visible and that all tuples deleted after the HWM appear as if they had not been deleted (*i.e.*, their `deletion_time`s would appear to be 0).

First, HARBOR finds all deletions that happened between $T_{checkpoint}$ (exclusive) and the HWM (inclusive) to the data that was inserted at or prior to $T_{checkpoint}$. HARBOR's recovery algorithm requires that each tuple have a unique tuple identifier (such as a primary key) to associate a particular tuple on one site with the same replicated tuple on another. HARBOR can then accomplish this task by running a set of historical queries as of the time HWM for each recovery object and recovery predicate computed for *rec*:

```
{(tup_id, del_time)} =
  SELECT REMOTELY tuple_id, deletion_time FROM recovery_object
  SEE DELETED HISTORICAL WITH TIME HWM
  WHERE recovery_predicate AND insertion_time <= T_checkpoint
    AND deletion_time > T_checkpoint
```

The `HISTORICAL WITH TIME HWM` syntax indicates that the query is a historical query as of the time HWM. The query outputs a set of tuples of the type <*tuple_id, deletion_time*>, where all the *tuple_id*s refer to tuples that were inserted at or before $T_{checkpoint}$ and deleted after $T_{checkpoint}$. Both of the $T_{min-insertion}$ and $T_{max-deletion}$ timestamps on segments help to reduce the number of segments that must be scanned to answer the `insertion_time` and `deletion_time` range predicates; on typical warehouse workloads where historical updates are rare, few segments should need to be scanned. For each *(tup_id, del_time)* tuple that *S* re-

ceives, HARBOR runs the following query to locally update the deletion time of the corresponding tuple in *rec*:

```
for each (tup_id, del_time) in result:
  UPDATE LOCALLY rec SET deletion_time = del_time
  SEE DELETED
  WHERE tuple_id = tup_id AND deletion_time = 0
```

The `UPDATE` again happens in place, as opposed to being a separate insert and delete, to reflect that *S* is simply copying any new deletion times from its recovery buddies. The predicate on the `deletion_time` ensures that HARBOR updates the most recent tuple in the event that the tuple has been updated and more than one version exists. I assume that an index exists on `tuple_id`, which is usually the primary key, to speed up the query.

Next, HARBOR queries for the rest of the data inserted between the checkpoint and the HWM using a series of historical queries as of the HWM on each recovery object and associated recovery predicate, and it inserts that data into the local copy of *rec* on *S*:

```
INSERT LOCALLY INTO rec
  (SELECT REMOTELY * FROM recovery_object
   SEE DELETED HISTORICAL WITH TIME HWM
   WHERE recovery_predicate AND insertion_time > T_checkpoint
     AND insertion_time <= hwm)
```

The semantics of the `INSERT LOCALLY` statement are in the sense of a traditional SQL insert, without the reassignment of insertion times as would occur under my versioned representation; it reflects the idea that *S* is merely copying the requested data into its local copy of *rec*. While the predicate `insertion_time <= hwm` is implicit by the definition of a historical query as of the HWM, it is shown here explicitly to illustrate that HARBOR can again use the $T_{min-insertion}$ timestamps on segments to prune the search space.

After running this query, the recovery object *rec* is up-to-date as of the HWM. Because objects will be recovered at different rates, HARBOR cannot write a checkpoint stating that stable storage reflects all committed changes up to time T for

some $T > T_{checkpoint}$ until recovery of all objects completes. If the recovering site or a recovering buddy crashes and recovery needs to be restarted, however, the recovering site would benefit from the knowledge that some database objects may already be more up-to-date than $T_{checkpoint}$. To accommodate this observation, *S* adopts a finer-granularity approach to checkpointing during recovery and maintains a separate checkpoint per object. Using the object-specific checkpoints, *S* records a new checkpoint as of the time HWM for *rec* to reflect that *rec* is consistent up to the HWM. The site resumes using the single, global checkpoint once recovery for all objects completes.

Note that if *S* has crashed for a long time before starting recovery, or if *S*'s disk has failed and must be recovered from a blank slate, Phase 2 may require a substantial amount of time to copy the missing data. After Phase 2, the HWM may lag the current time by a sizeable amount, and many additional transactions may have committed since Phase 2 started. If the HWM differs from the current time by more than some system-configurable threshold, Phase 2 can be repeated additional times before proceeding to Phase 3. The benefit of re-running Phase 2 is that the remaining queries in Phase 3 proceed with transactional read locks and block ongoing update transactions, whereas the historical queries of Phase 2 do not.

> *Example:* Figure 5-2 continues the recovery story from the previous section. Figure 5-2(a) shows the table on the recovering site after Phase 1. Phase 2 uses a HWM of 10, and Figure 5-2(b) depicts the recovery buddy's table with the missing updates to be copied during Phase 2 shaded. Note that the recovery buddy's table uses a different sort order but logically represents the same data. Because the HWM is 10, the tuple with an insertion time of 11 will not be copied, and the tuple deleted at time 11 appears undeleted to historical queries as of time 10. After executing the `SELECT` and `UPDATE` queries to copy over the deletion timestamps for tuples deleted between the checkpoint and the HWM, the recovering site holds the table shown in Figure 5-2(c); the copied deletion time is shaded. Figure 5-2(d) reflects the table at the end of

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	0	Dell Monitor

(a) Table on the recovering site after recovery Phase 1 with checkpoint at time 4.

insertion_time	deletion_time	product_name
11	0	Bose Speaker
6	9	Chapstick
1	0	Colgate
4	8	Dell Monitor
10	11	iPod
2	3	Poland Spring

(b) Table on the recovery buddy, sorted by `product_name`. Shaded cells indicate insertions, deletions, and updates that the recovering site must copy during Phase 2 of recovery. Note that the insertion and the deletion at time 11 are not visible to a historical query running as of time 10.

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	8	Dell Monitor

(c) Table on the recovering site after the `SELECT` and `UPDATE` queries of Phase 2. The shaded cell indicates the deletion time copied from the recovery buddy.

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	8	Dell Monitor
6	9	Chapstick
10	0	iPod

(d) Table on the recovering site after the nested `SELECT` and `INSERT` query of Phase 2. Shaded tuples indicate tuples copied from the recovery buddy.

Figure 5-2: Sample run of recovery Phase 2 for a checkpoint recorded at time 4 and a HWM of 10.

Phase 2, after running the nested `SELECT` and `INSERT` query to copy the shaded tuples with insertion timestamps between the checkpoint and the HWM.

5.4 Phase 3: Catch up to the current time with locks

Phase 3 consists of two parts: 1) executing non-historical read queries with locks to catch up to with all committed updates up to the current time and 2) joining any pending transactions dealing with *rec*. The structure of the read queries highly parallel the structure of those in Phase 2, except that they are not run in historical mode.

5.4.1 Query for committed data with locks

First, HARBOR acquires a transactional read lock on every recovery object at once to ensure consistency:

```
for all recovery_objects:
  ACQUIRE REMOTELY READ LOCK ON recovery_object
  ON SITE recovery_buddy
```

The lock acquisition can deadlock, and I assume that the system has some distributed deadlock detection mechanism, by using timeouts for instance, to resolve any deadlocks. Site *S* retries until it succeeds in acquiring all of the locks.

When the read locks for all recovery objects have been granted, all running transactions on the system are either a) not touching *rec*'s data, b) running read queries on copies of *rec*'s data, or c) waiting for an exclusive lock on a recovery buddy's copy of *rec*'s data (recall from § 4.1 that update transactions must update all live copies of the data, including the one on the recovery buddy). In other words, after site *S* acquires read locks on copies of data that together cover *rec*, no *pending* update transactions that affect *rec*'s data can commit until *S* releases its locks. Moreover, no *in-doubt* transactions that updated *rec*'s data can remain in

the database because *S*'s successful acquisition of read locks implies that any such transactions must have already completed commit processing and released their locks.

After Phase 2, the recovering site *S* is missing, for all tuples inserted before the HWM, any deletions that happened to them after the time HWM. To find the missing deletions, I use a strategy similar to the one used at the start of Phase 2. For each recovery object and recovery predicate pair, HARBOR executes the following query:

```
{(tup_id, del_time)} =
  SELECT REMOTELY tuple_id, deletion_time FROM recovery_object
  SEE DELETED
  WHERE recovery_predicate AND insertion_time <= hwm
    AND deletion_time > hwm
```

HARBOR again relies on the segment architecture to make the two timestamp range predicates efficient to solve. For each *(tup_id, del_time)* tuple in the result, HARBOR locally updates the deletion time of the corresponding tuple in *rec*:

```
for each (tup_id, del_time) in result:
  UPDATE LOCALLY rec
  SET deletion_time = del_time
  WHERE tuple_id = tup_id AND deletion_time = 0
```

Finally, to retrieve any new data committed inserted after the HWM, HARBOR runs the following insertion query that uses a non-historical read subquery:

```
INSERT LOCALLY INTO rec
  (SELECT REMOTELY * FROM recovery_object
   SEE DELETED
   WHERE recovery_predicate AND insertion_time > hwm
     AND insertion_time != uncommitted)
```

The check `insertion_time != uncommitted` is needed assuming that the special `uncommitted` insertion timestamp is represented by some value larger than any valid timestamp and would otherwise satisfy the `insertion_time > hwm` predicate. In the common case, the query only examines the last segment, but

HARBOR may need to examine more segments depending on whether new segments were created since the time HWM at the beginning of Phase 2. After this query, *S* has caught up with all committed data for *rec* as of the current time and is still holding locks on its recovery buddies' recovery objects. *S* can then write a checkpoint for *rec* timestamped at the current time minus one (the current time has not expired, and additional transactions may commit with the current time).

> *Example:* Figure 5-3 illustrates the remaining queries for missing updates. Figure 5-3(a) shows the recovering table after Phase 2, and Figure 5-3(b) shows the table on the recovery buddy; shaded cells in the recovery buddy's table indicate data to be copied. The recovering site obtains a read lock on the recovery buddy's table at the start of Phase 3. After the recovering site queries for the deletions after the HWM, the recovering table looks as shown in Figure 5-3(c) with the copied deletions shaded. Figure 5-3(d) shows the recovering table after it has copied all insertions up to the current time.

5.4.2 Join pending transactions and come online

At this point, there may still be some ongoing update transactions that the recovering site *S* needs to join. The simple approach would be to abort all pending non-historical transactions at coordinator sites and restart them with the knowledge that *rec* on *S* is online; however, one would like to save the work that has already been completed at other worker sites.

The protocol for joining all pending update transactions begins with site *S* sending a message *M* to each coordinator saying "*rec* on *S* is coming online." Any subsequent transactions that originate from the coordinator and that involve updating *rec*'s data must also include *S* as a worker; read-only transactions can optionally use *rec* on *S* because it already contains all committed data up to this point. As previously mentioned in § 4.1, each coordinator maintains a queue of update requests for each of its transactions; the system now uses this queue to complete recovery.

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	8	Dell Monitor
6	9	Chapstick
10	0	iPod

(a) Table on the recovering site after recovery Phase 2.

insertion_time	deletion_time	product_name
11	0	Bose Speaker
6	9	Chapstick
1	0	Colgate
4	8	Dell Monitor
10	11	iPod
2	3	Poland Spring

(b) Table on the recovery buddy, sorted by `product_name`. Shaded cells indicate insertions, deletions, and updates that the recovering site must copy during Phase 3 of recovery.

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	8	Dell Monitor
6	9	Chapstick
10	11	iPod

(c) Table on the recovering site after the `SELECT` and `UPDATE` queries of Phase 3. The shaded cell indicates the deletion time copied from the recovery buddy.

insertion_time	deletion_time	product_name
1	0	Colgate
2	3	Poland Spring
4	8	Dell Monitor
6	9	Chapstick
10	11	iPod
11	0	Bose Speaker

(d) Table on the recovering site after the nested `SELECT` and `INSERT` query of Phase 3. Shaded tuples indicate tuples copied from the recovery buddy.

Figure 5-3: Sample run of the standard lock-based queries to catch up to the current time during recovery Phase 3 with a HWM of 10.

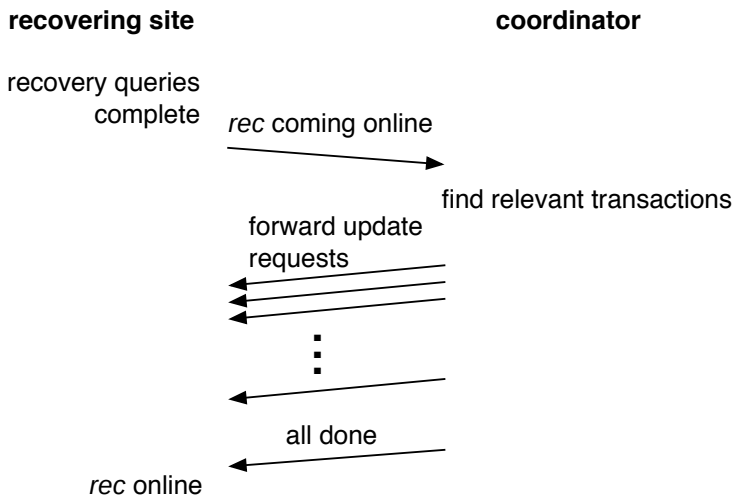

Figure 5-4: Protocol to join pending transactions and come online.

Let PENDING be the set of all *pending* update transactions at a coordinator when message M is received. To complete recovery, S needs to join all *relevant* transactions in PENDING. A transaction T is relevant if at some point during the transaction, it deletes a tuple from, inserts a tuple to, or updates a tuple in the recovering object *rec*. The coordinator determines if a transaction T is relevant by checking if any updates in the transaction's update queue modify some data covered by *rec*. To have S join a relevant transaction, the coordinator forwards all queued update requests that affect *rec* to S. When the transaction is ready to commit, the coordinator includes S during the commit protocol by sending it a PREPARE message and waiting for its vote, as if *rec* on S had been online since the beginning of the transaction.

The coordinator cannot conclude that S will not join a particular transaction T until either the client commits the transaction or the transaction aborts because otherwise the transaction may still update *rec*'s data. After all transactions in PENDING have been deemed either relevant or irrelevant to *rec*, and after S has begun joining all relevant transactions, the coordinator sends an "all done" message to S. Figure 5-4 summarizes this protocol for the recovering site to join *pending* transactions and come online. When S receives all such messages from all coordinators, it releases its locks on the recovery objects for *rec* on remote sites:

```
for all recovery_objects:
  RELEASE REMOTELY LOCK ON recovery_object
  ON SITE recovery_buddy
```

Object *rec* on *S* is then fully online. Transactions that were waiting for locks on the recovery objects can then continue to make progress.

5.5 Failures during Recovery

HARBOR's recovery algorithm tolerates a variety of site failures during recovery, including failure of the recovery site, failures of recovery buddies, and failures of coordinators.

5.5.1 Failure of Recovering Site

If the recovering site *S* fails during Phase 1 or Phase 2 of recovery, it restarts recovery upon rebooting, using a more recent object-specific checkpoint if one is available. If *S* fails during Phase 3, however, it may still be holding onto remote locks on its recovery objects, and the coordinator may have been attempting to let *S* join ongoing transactions. To handle this situation, HARBOR requires some failure detection mechanism between nodes, which already exists in most distributed database systems. The mechanism may be some type of heartbeat protocol in which nodes periodically send each other "I'm alive" messages; or, it may be, as in my implementation, the detection of an abruptly closed TCP socket connection as a signal for failure. Regardless of the mechanism, when a recovery buddy detects that a recovering node has failed, it overrides the node's ownership of the locks and releases them so that other transactions can progress. For any pending transactions that *S* had been joining, the coordinator treats the situation as it normally treats worker failures during transactions. *S* then restarts recovery.

5.5.2 Recovery Buddy Failure

If a recovery buddy fails during recovery, the remaining read locks on the remaining recovery buddies no longer cover a full copy of *rec*. Because the distributed transaction model only requires that updates be sent to all live worker sites with the relevant tuples, transactions that update the part of *rec*'s data stored on the crashed recovery buddy may be able to slip past *S*'s locks. Thus, the recovering node *S* must release any locks on the remaining replicas that it may be holding, abort any transactions that it may be joining, and restart recovery with a new set of recovery objects computed from the remaining online sites. The coordinator and the other recovery buddies behave as if *rec* on *S* had actually failed.

5.5.3 Coordinator Failure

If a coordinator fails during recovery, and the system is configured to use optimized 2PC, recovery may block in Phase 3 as *S* waits for any read locks held by workers participating in the coordinator's update transactions. *S* can proceed only after those transactions complete and release their locks, which may require waiting for the coordinator to recover.

The non-blocking, optimized 3PC protocol supports recovery without waiting for a failed coordinator to recover. In the rest of this section, I show that the consensus building protocol used to handle coordinator failure during normal processing continues to preserve ACID semantics.

Suppose that a coordinator fails during Phase 1 or Phase 2 of recovery. All of the coordinator's transactions will either commit or abort based on the rules of optimized 3PC and the consensus building protocol. The aborted transactions do not affect the results of any recovery queries because the queries only copy committed data. All other committed transactions that affect *rec* fall into two categories:

1. The transaction commits at or before the HWM. In this case, *S* begins Phase 2 after the transaction committed and will copy the transaction's updates with its historical queries to catch up to the HWM.

2. The transaction commits after the HWM. In this case, *S* could not have obtained the read locks to its recovery objects until the transaction completed. Therefore, by the time that *S* acquires its remote read locks, it will be able to copy the updates through its non-historical queries with locks to catch up to the current time.

Suppose that a coordinator instead fails during Phase 3, after *S* has obtained read locks on its recovery objects. In this case, any of the coordinator's ongoing transactions that affect *rec* must still be in the *pending* state and waiting for a lock held by *S*; otherwise, *S* would have been unable to acquire its locks. Therefore, any transactions affecting *rec* will abort and not affect *S*'s recovery.

Chapter 6

Evaluation

In order to evaluate the runtime overhead and recovery performance of HARBOR, I have implemented a distributed database system, consisting of 11.2 K lines of Java code. In this chapter, I first highlight some implementation details not covered previously; I then compare the runtime overhead and recovery performance of HARBOR to the performance of traditional two-phase commit and ARIES on a four-node distributed database.

6.1 Implementation

Figure 6-1 shows an architecture diagram of the query execution, query distribution, and transaction management modules in my implementation. I have implemented all of the components described thus far except the consensus building protocol to tolerate coordinator failures and the functionality to handle failures during recovery.

The implementation supports two independent recovery mechanisms—HARBOR and the traditional log-based ARIES approach. Furthermore, I have implemented the canonical and optimized variants of both the 2PC and 3PC protocols, for a total of four different protocols. For ease of experimental evaluation, a few command-line parameters enable the system to easily switch between the two recovery frameworks and the various commit protocols.

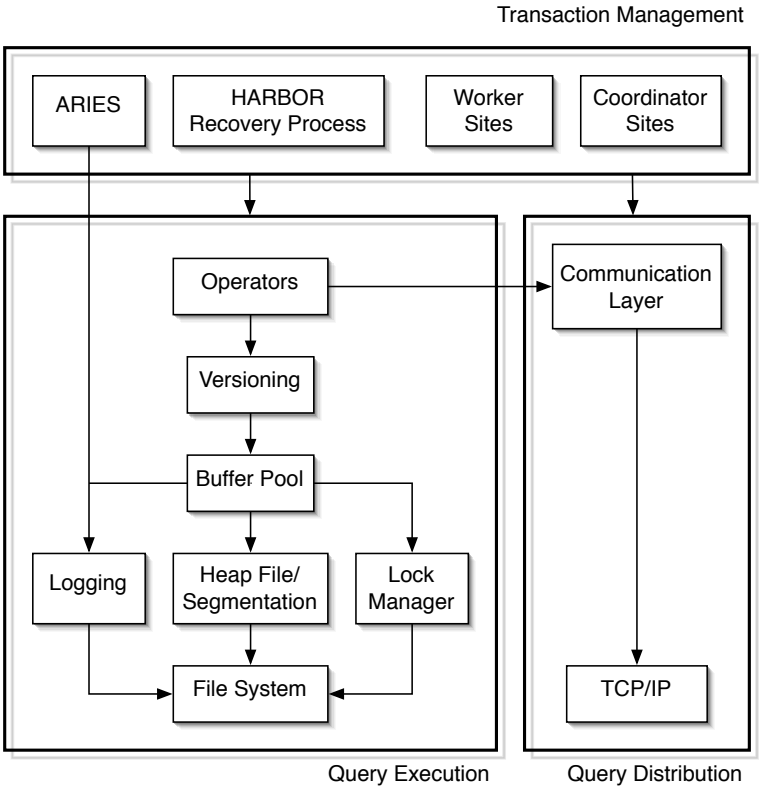

Figure 6-1: Architecture diagram of query execution, query distribution, and transaction management modules in the database implementation. Boxes indicate key components, and arrows indicate dependencies.

6.1.1 Physical Data Storage

The database system stores relations in segmented heap files with 4 KB pages using standard files in the underlying file system. Each segmented heap file stores in a header page a series of $<T_{min-insertion}, T_{max-deletion},$ *start-page*$>$ triplets that identify all the segment timestamps and boundaries within that file, in accordance with the segment architecture specifications of § 4.2. The system assigns the *start-page* for a segment at creation time, and updates to a particular segment may modify the segment's timetamps as appropriate during runtime. The physical data model reserves two columns in each row of a relation for the insertion and deletion timestamps.

To reduce the cost of sequential scans, the system densely packs heap files by inserting tuples into any available slots before appending additional heap pages to the file; to reduce the cost of insertions, heap files and heap pages maintain pointers to the first empty slot once the buffer pool loads them into memory.

6.1.2 The Lock Manager

A local lock manager on each site handles the concurrency control logic for the tables on that site. During normal transaction processing, the lock manager supports locking at page granularity in order to offer more concurrency than table-level locking while avoiding the excessive bookkeeping that would be required by tuple-level locking. The lock manager exposes the following API to the buffer pool to regulate shared and exclusive access to pages:

```
// lock manager API
void acquireLock(TransactionId tid, PageId pid, Permissions perm)
boolean hasAccess(TransactionId tid, PageId pid, Permissions perm)
void releaseLocks(TransactionId tid)
```

The API call `acquireLock` blocks if a lock on the page with the specified read or write permissions cannot currently be granted to the transaction. To detect deadlocks, the call employs a simple timeout mechanism and throws an exception if a

timeout occurs. The call `hasAccess` checks if a transaction has appropriate privileges to a page, and the call `releaseLocks` releases all of a transaction's locks at the end of the transaction. For recovery purposes, the lock manager also exposes a table-granularity version of the locking API to enable a recovering site to obtain read locks on entire recovery objects of recovery buddies.

To synchronize concurrent access to the lock manager, I declare the API methods for acquiring and releasing locks with Java's `synchronized` attribute so that at any given time, only one thread can read or write the critical data structures within the lock manager.

6.1.3 The Buffer Pool

The buffer pool enforces a STEAL/NO-FORCE paging policy [19] (though other paging policies have also been implemented) to manage the reading and writing of pages from and to disk and uses a random page eviction policy to deal with buffer pool saturation. The buffer pool presents the following API to the database operators:

```
// data access and modification API
// reads the page, either from memory or from disk,
// acquiring any necessary locks
Page getPage(TransactionId tid, PageId pid, Permissions perm)
// inserts/deletes/updates a specified tuple
RecordId insertTuple(TransactionId tid, int tableId, Tuple t)
void deleteTuple(TransactionId tid, Tuple t)
void updateTuple(TransactionId tid, RecordId rid, UpdateFunction f)
```

In the data access and modification API implementation, the buffer pool uses the lock manager API to regulate operator access to data across concurrent transactions. Prior to returning a page in `getPage` to an operator, the buffer pool calls `hasAccess` to determine whether the transaction already holds a lock with the necessary read or write permissions, and if not, acquires one with `acquireLock`. Update operators then use `insertTuple`, `deleteTuple`, or `updateTuple` to modify a page in the buffer pool.

One point worth noting is that when inserting new tuples into a table, a transaction obtains a shared lock over a heap page before scanning it for empty slots and upgrades to an exclusive lock on a page when an empty slot is found. The shared lock prevents the race condition of another concurrent transaction filling the last empty slot on a page just as the current transaction is intending to do so.

The buffer pool exposes a separate API to coordinators and workers to handle transaction commit and abort:

```
// commit processing API
void prepareTransaction(TransactionId tid)
void prepareToCommitTransaction(TransactionId tid)
void commitTransaction(TransactionId tid)
void abortTransaction(TransactionId tid)
```

Each call updates the local state of a transaction. If the particular commit processing protocol in effect requires logging, each call also appends an appropriate log record, forcing the log to disk if necessary. If a transaction commits, the buffer pool releases all locks using the lock manager's `releaseLocks` call; if a transaction aborts, the buffer pool rolls back any logged changes and subsequently releases the locks.

6.1.4 Versioning and Timestamp Management

A versioning and timestamp management wrapper around the buffer pool abstraction re-exposes the buffer pool's data access and modification API and its commit processing API but decorates the calls with additional timestamp-related functionality. The call `insertTuple` acquires a lock on the tuple's page, adds the tuple's identifier to the in-memory insertion list, assigns 0 to the tuple's deletion timestamp, writes the special `uncommitted` value to the tuple's insertion timestamp to indicate that it has not yet been committed, and finally delegates the remaining work to the buffer pool's `insertTuple` call. The `deleteTuple` call simply acquires an exclusive lock on the affected page and adds the tuple identifer to the in-memory deletion list without yet engendering any actual page modifications; the

lock ensures that the page with the tuple can be subsequently modified at transaction commit time. The `updateTuple` call deletes the old tuple and inserts the new tuple via the wrapper.

The versioning layer's `commitTransaction` assigns the commit time to the insertion and deletion timestamps of the tuples in the insertion and deletion lists, respectively, prior to deleting the transaction's in-memory state and delegating to the buffer pool's `commitTransaction` implementation. If logging is disabled, `abortTransaction` uses the insertion list to roll back updates at transaction abort and then deletes the transaction's in-memory state prior to delegating lock management to the buffer pool's `abortTransaction` implementation; otherwise, it simply deletes the in-memory state and delegates the remaining work to the buffer pool.

6.1.5 Database Operators

The database implementation supports most of the standard database operators, including sequential scans, primary indices based on tuple identifiers, predicate filters, aggregations with in-memory hash-based grouping, nested loops joins, projections, updates, deletes, and inserts. The implementation also supports special versions of the scan, insert, delete, and update operators that are aware of timestamps and historical queries. All operators export the standard iterator interface found in row-oriented database systems:

```
// standard iterator interface
void open()
Tuple getNext()
void rewind()
void close()
// returns the relational schema for the operator's output tuples
TupleDesc getTupleDesc()
```

The database implementation does not yet have a SQL parser frontend; query plans must be manually constructed before they can be executed.

6.1.6 Distributed Transactions

Coordinators and workers interact through a basic client-server model and communicate via TCP socket connections. Each worker runs a multi-threaded server that listens for incoming transaction requests, and multi-threaded coordinator sites send requests to the workers. Each TCP socket connection manages a single transaction at any given time, but connections can be recycled for subsequent transactions; coordinators execute concurrent transactions by establishing multiple socket connections simultaneously.

I implement distributed transactions by building a communication abstraction layer over Java's TCP socket library and by creating special network operators that writes tuples to and read tuples from socket connections. The communication layer serializes messages to and deserializes messages from a socket connection.

6.1.7 Recovery

As previously mentioned, I instrumented my database implementation with two recovery mechanisms, ARIES and HARBOR. I believe my benchmark ARIES implementation to be a faithful implementation of the ARIES recovery protocol as specified in [37]. Modern implementations of ARIES and other log-based recovery approaches undoubtedly include many optimizations [30, 36] that I have omitted, but my implementation suffices as a benchmark to validate HARBOR's recovery approach. One small implementation detail to note is that because workers assign timestamps to tuples at commit time, ARIES requires writing additional log records for the timestamp updates after the PREPARE phase, on top of any log records that may have been written for the tuples prior to the PREPARE phase.

The HARBOR recovery implementation follows directly from the recovery discussion of Chapter 5. A recovering worker leverages the machinery described in this implementation discussion to execute and distribute the recovery SQL queries. A coordinator site runs a recovery server on a well-known port to listen for recovery messages and to enable recovering workers to join ongoing transactions.

6.2 Evaluation Framework and Objectives

Each node in the four-node distributed database system runs on a 3 GHz Pentium IV machine with 2 GB of memory running RedHat Linux. In addition to a 200 GB disk with 60 MB/s bandwidth, each machine is also equipped with a three-disk 750 GB RAID with 180 MB/s bandwidth. The database did not use the RAID disks except as a separate disk for the log when a log is required. Unless otherwise noted, whenever a logging operation is involved, the database uses group commit without a group delay timer [24]; test experiments showed that various group delay timer values ranging from 1–5 ms only decreased group commit performance. The machines are connected together in a local area network with 85 Mb/s bandwidth.

Table sizes vary in the different experiments, but I implement each table as a heap file with a segment size of 10 MB. The tuples in each table contain 16 4-byte integer fields, including the insertion and deletion timestamp fields used for historical queries.

The objective of the evaluation is to profile and better understand the performance characteristics of optimized 3PC and HARBOR relative to traditional 2PC and ARIES. Because disk speeds today fall roughly six orders of magnitude behind main memory access speeds, I would expect optimized 3PC to win with higher transaction throughput and lower transaction latency by trading away its forced-writes for an additional round of messages on a fast local area network. In fact, the experiments on the runtime overhead of logging and commit messages in § 6.3 confirm this hypothesis. Given sufficient network bandwidth, I would also hypothesize that copying data over the network for recovery would be cheaper than processing an on-disk recovery log, but that HARBOR's recovery performance would decrease if many historical segments needed to be scanned for recovery; the recovery experiments presented in § 6.4 validate my suspicions. To validate that HARBOR embodies an end-to-end recovery mechanism, I show in § 6.5 that my system can tolerate a site failure and bring the site back online while still processing transactions.

6.3 Runtime Overhead of Logging and Commit Messages

In the first set of experiments, I compare the runtime overhead of my optimized commit processing protocols against two-phase and three-phase commit with write-ahead logging using ARIES. Only three of the four nodes participate in the experiments to measure runtime performance. One coordinator node sends update transactions to the two other worker nodes. For simplicity, both workers store the same replicated data in the same format; in general, HARBOR does not constrain replicated data to use the same storage format at different nodes.

For these experiments, I configure the system to flush all dirty pages and record a checkpoint once per second. A few separate experiments that I ran showed that setting the checkpoint frequency between 1–10 s affected transaction throughput by no more than 9.5%.

6.3.1 Transaction Processing Performance of Different Commit Protocols

In the first experiment, each transaction simply inserts a single 64-byte tuple into a target table on the worker sites. I vary the number of concurrent transactions to observe the effect of group commit in alleviating the overhead of forced-writes. To eliminate the effect of deadlocks on the results, concurrent transactions insert tuples into different tables so that conflicts do not arise; for example, an experimental run with 10 concurrent transactions uses 10 different tables.

Figure 6-2 reports the throughput in transactions per second (tps) of four commit processing protocols: optimized 3PC without logging, optimized 2PC without logging at worker sites, canonical 3PC with logging at workers and no logging at the coordinator, and traditional 2PC with logging. To better illustrate the effects of group commit and the costs of replication, I also include for comparison the throughput of 2PC with no group commit and the throughput of 2PC without repli-

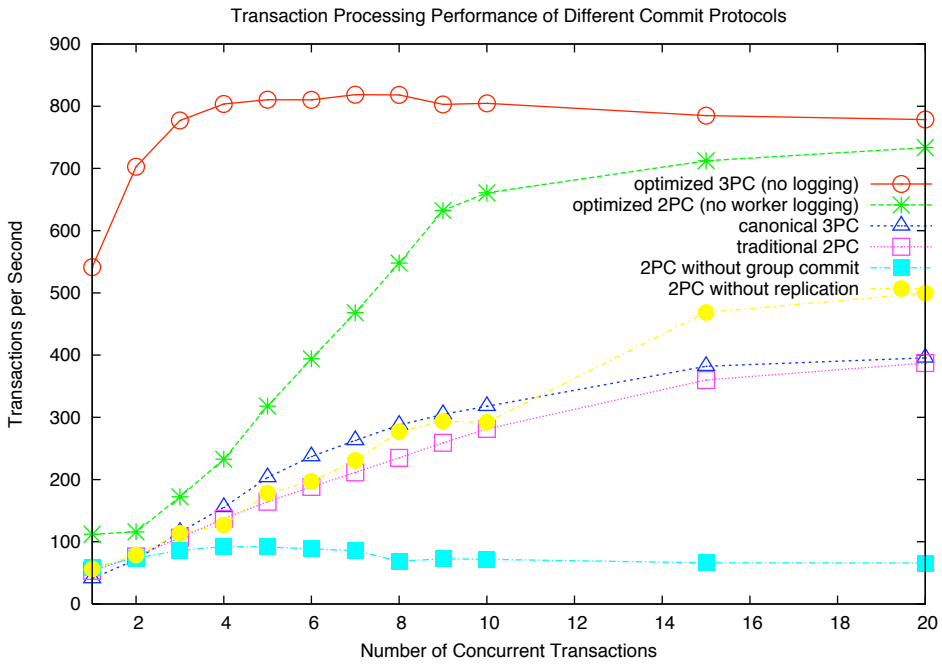

Figure 6-2: Transaction processing performance of different commit protocols.

cation, *i.e.*, with only one worker site. I obtained the measurements in the figure by recording the average steady-state throughput on a workload with N transactions, where N equals 10000 times the number of concurrent transactions for the particular run.

The case of no concurrency, *i.e.*, a single running transaction, illustrates the average latency of a transaction and shows that optimized 3PC (latency = 1.8 ms) runs 10.2 times faster than canonical 2PC (18.8 ms), 13.2 times faster than canonical 3PC (23.4 ms), and 4.8 times faster than optimized 2PC (8.9 ms). The overhead of the 3 forced-writes in traditional 2PC and canonical 3PC far eclipse the overhead of adding an additional round of messages to 2PC to obtain optimized 3PC.

Additional degrees of concurrency enable the overlap of CPU, network, and disk utilization to increase transaction throughput; furthermore, group commit reduces the cost of forced-writes significantly, as shown by the upward trends of the logging protocols with group commit in contrast to the stable line for 2PC without group commit. Without group commit, concurrency does not increase throughput beyond

58–93 tps because the synchronous log I/Os of different transactions cannot be overlapped. Nevertheless, even with 10–20 concurrent transactions, the throughput of optimized 3PC still remains a factor of 2–2.9 higher than the throughput of traditional 2PC with group commit. Surprisingly, optimized 2PC performs nearly as well as optimized 3PC on highly concurrent workloads despite requiring a forced-write by the coordinator; this result suggests that group commit can effectively batch the single log write of optimized 2PC across transactions.

Optimized 3PC and optimized 2PC taper off with throughput at 778 tps and 733 tps, respectively, because they become CPU-bound. Production database systems achieve higher transaction throughput by increasing the number of processors per machine, the number of coordinator sites accepting transactions, and the number of worker sites performing work on behalf of different transactions. Such additions would also increase the overall transaction throughput of my implementation.

Canonical 3PC and traditional 2PC level off with throughput at 396 tps and 387 tps, respectively, because they become disk-bound. Group commit cannot perfectly overlap the I/O across concurrent transactions; some of the 3 forced-writes of some transactions will inevitably incur overhead waiting for the disk to become available.

Despite requiring the same number of forced-writes and an additional round of messages compared to traditional 2PC, canonical 3PC still manages to achieve roughly 14% higher throughput. This apparent oddity can be explained by noting that each of canonical 3PC's forced-writes for a given transaction can be batched via group commit with any of the 3 forced-writes for any other transaction running on the worker; in contrast, a coordinator's forced-writes in 2PC can only be batched with other forced-writes at the coordinator, and a worker's forced-writes can only be batched with other forced-writes at the worker. Thus, forced-writes in canonical 3PC have more opportunities to be batched by group commit than those in traditional 2PC.

The results for 2PC without replication suggest that replication on two sites costs between 2 to 23% of transaction throughput. Optimized 3PC without replication has been omitted from the experiments and is not too meaningful because

the data redundancy is the primary reason why HARBOR's recovery guarantees are possible without logging. The tradeoff for the runtime performance of replication is increased availability during failures; however, the substantial gains in runtime performance observed when comparing optimized 3PC with traditional 2PC more than justify the replication costs.

The results are encouraging and demonstrate that one can eliminate substantial commit processing overhead using HARBOR. Of course, with a less update-intensive workload, these differences may be less dramatic. Moreover, update transactions typically may require some computational overhead, and the computation would tend to dampen the relative advantage of a faster commit protocol. To observe the effects of CPU overhead and also to gain a better grasp of the performance characteristics of commit processing on a less update-intensive workload, I rerun a similar experiment with more CPU-intensive transactions.

6.3.2 Transaction Processing Performance with CPU-Intensive Workload

In this second experiment, each transaction still inserts a single tuple, but worker sites must perform some duration of work for a transaction prior to committing the transaction. I simulate CPU work required for a transaction by invoking a loop for some number of cycles at the worker sites. The simulated work could represent the ETL (extract, transform, and load) processing of new tuples, the compression of new data prior to storage, the generation of derived fields from the original fields of an operational database, the update of multiple materialized views, or any other CPU-intensive actions. Figure 6-3 shows the tps throughput of different commit protocols while varying the amount of simulated CPU work (measured in number of cycles) at the worker sites per transaction. I repeat the exercise for 1, 5, and 10 concurrent transactions.

Aside from the obvious trend that increasing the amount of work per transaction decreases transaction throughput and increases transaction latency, the experiment

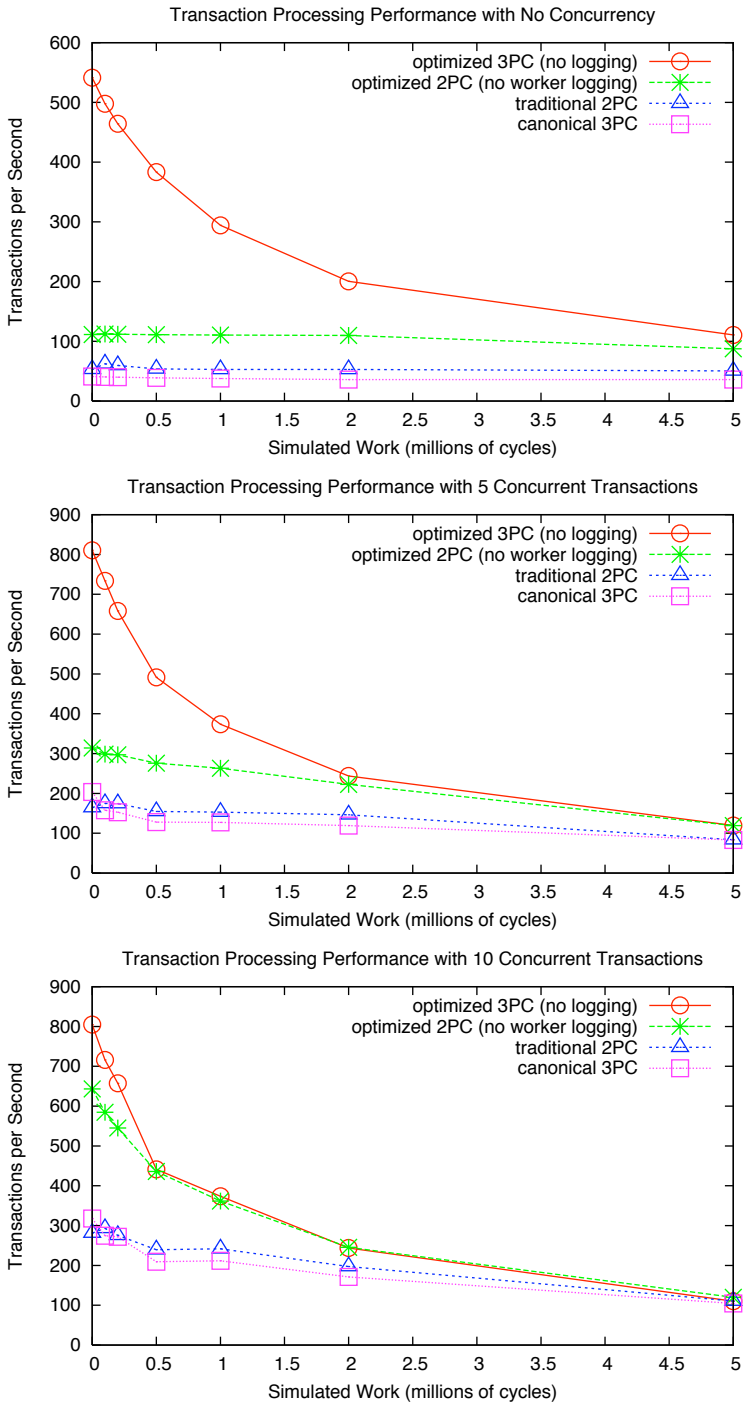

Figure 6-3: Transaction processing performance of different commit protocols, with simulated work at the worker sites and various degrees of concurrency.

illustrates two additional trends:

1. *The relative performance differences among protocols decrease with increasing CPU work.* As the duration of simulated CPU work increases, the absolute amount of work required for a transaction begins to overshadow any relative commit costs associated with the log-based commit protocols. For example, in the case of 5 concurrent transactions, optimized 3PC has 4.9 times higher throughput than traditional 2PC when transactions perform 0 cycles of simulated CPU work, but only 1.4 times higher throughput when transactions perform 5 million cycles of simulated CPU work. Similarly, optimized 3PC throughput drops from being 4.0 times higher than canonical 3PC to 1.4 times and from being 2.6 times higher than optimized 2PC to roughly equal performance. The graphs for no concurrency and for 10 concurrent transactions exhibit a similar trend.

2. *The relative performance differences among protocols decrease with increasing concurrency.* The previous section's experiment already alluded to this trend, but this experiment confirms the trend for CPU-intensive workloads as well. At 1 million cycles of CPU work per transaction and no concurrency, optimized 3PC provides 5.6 times higher throughput than traditional 2PC, 7.8 times higher throughput than canonical 3PC, and 2.7 times higher throughput than optimized 2PC. Holding the CPU work constant and increasing concurrency to 5 transactions cause those performance ratios to drop to 2.4, 2.9, and 1.4 respectively. At 10 concurrent transactions, those ratios become 1.5, 1.8, and 1.0.

   Two explanations account for these observations. First, as previously noted, group commit increases the throughput of the logging protcols by batching log writes; thus, higher degrees of concurrency benefit the logging protocols. Second, a worker site cannot overlap the CPU work of concurrent transactions because the processor can only dedicate itself to one transaction at a time; thus, increasing the number of transactions, each with some CPU cost,

translates to increasing the absolute amount of total work and therefore overshadows any commit costs even further. In this respect, CPU resources differ from disk and network resources, which can be shared across concurrent transactions via batching.

6.4 Recovery Performance

In the next two experiments, I compare the performance of HARBOR's recovery approach against ARIES and demonstrate that my recovery performance is indeed comparable. Though modern implementations of ARIES and other log-based recovery algorithms include optimizations that I have omitted from my benchmark ARIES implementation, the following experiments still shed light on the recovery performance. The fact that the performance of my recovery implementation is on par with my ARIES implementation is highly encouraging.

Both experiments use all four nodes (one coordinator and three workers) and follow a similar setup to measure recovery performance as a function of the number of transactions to recover. Each table used is replicated identically on each worker site, and each worker site starts with a clean buffer pool and a fresh checkpoint. The coordinator then executes some number of insert/update transactions with the three worker sites; the workers do not flush any dirty pages during these transactions, but they periodically flush the log if ARIES is being evaluated. After the coordinator finishes the transactions and after any and all log writes have reached disk, I crash a worker site and measure the time for the crashed worker to recover under four scenarios:

1. The inserts/updates all target one 1 GB table, and the crashed worker uses ARIES to recover from the log.

2. The inserts/updates all target one 1 GB table, and the crashed worker uses HARBOR to recover from another worker site.

3. The inserts/updates are equally distributed among two 1 GB tables, and the crashed worker uses HARBOR to recover each table, in parallel, one from each of the remaining worker sites.

4. The inserts/updates are equally distributed among two 1 GB tables, and the crashed worker uses HARBOR to serially recover each table from other worker sites.

The purpose of the four scenarios is 1) to profile and understand the performance characteristics of ARIES and HARBOR on my distributed database implementation and 2) to better understand if and how parallel recovery of multiple database objects may reduce overall recovery time. Though a system would never actually have a reason to use serial recovery over parallel recovery (assuming that parallel recovery finishes faster than serial recovery, which it does), I include serial recovery to help shed light on the performance gains provided by parallelism. Profiling recovery performance as a function of transactions to recover can also be interpreted as measuring the potential reductions in recovery time from increasing checkpoint frequency.

Each 1 GB table in the scenarios uses a segment size of 10 MB and consumes 101 segments, with the last and most recent segment half full. Because no transactions occur during recovery, HARBOR spends minimal time in Phase 3; I show the performance impact of all three phases on runtime performance later in § 6.5.

6.4.1 Recovery Performance on Insert Workloads

In the first recovery experiment, the coordinator sends only insert transactions, with one insertion per transaction, and I vary the number of transactions performed. Because newly inserted tuples are appended to the end of a 1 GB heap file, the newly inserted tuples affect only the last segment, though the worker may create a new segment given sufficient insertions. Figure 6-4 illustrates the recovery performance of all four scenarios as functions of the number of insert transactions to recover.

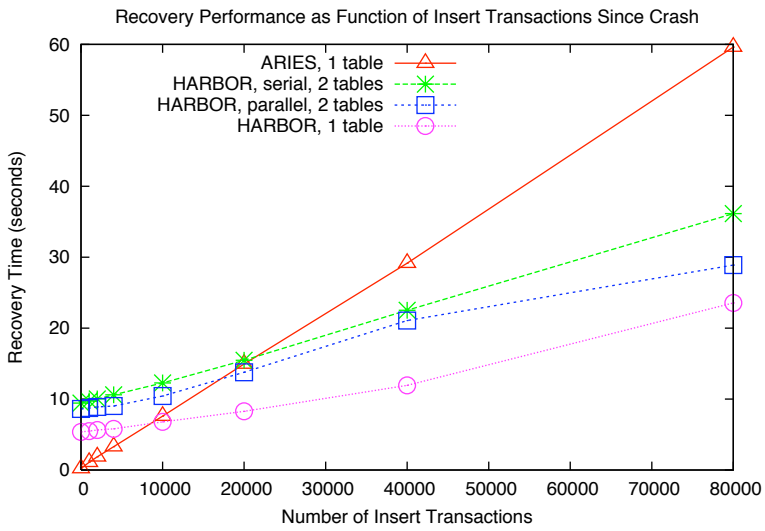

Figure 6-4: Recovery performance as a function of insert transactions since crash.

While ARIES performs well on a small number of insert transactions, ARIES performance degrades over 3 times more rapidly than HARBOR performance in the same single table scenario; HARBOR requires an additional second of recovery time for every additional 4400 insertions, but ARIES needs an additional second after only 1347 tuples. In my implementation, the crossover point where ARIES performance falls behind HARBOR performance in the same single table scenario occurs at 4581 insert transactions. These observations show that, at least for my implementation, log processing for committed transactions incurs higher overhead than querying for the committed data from a remote replica does.

The 5.3 s that HARBOR requires to recover 2 insert transactions indicate the fixed cost that HARBOR must invest 1) to send the recovery SQL requests and 2) to scan the most recent segment in Phase 1 to search for uncommitted data or data committed after the checkpoint. Aside from the fixed cost, recovery time for HARBOR grows roughly linearly as a function of the number of insert transactions. Serial recovery of two 1 GB tables demands more time than recovery of a single 1 GB table even when both scenarios involve an equivalent total number of insert transactions because the fixed cost must be expended once per table.

The results show that parallel recovery proceeds on average only 1.4 s faster

than serial recovery up to a certain point (40000 transactions) but that the performance gap widens substantially to 7.3 s at 80000 transactions. The observation that parallel recovery of two 1 GB tables is only slightly faster than serial recovery on small numbers of transactions demonstrates that the recovering site wins only a small performance gain by attempting to parallelize the work to scan the two tables' last segments. The worker's disk head can only scan one segment at a time in Phase 1, and the fixed costs dominate most of the recovery time. The larger performance difference at higher numbers of transactions suggests that parallel recovery scales well with the number of insert transactions to recover and that Phase 2, unlike Phase 1, can benefit from the parallelism. In the parallel recovery scenario, Phase 2's recovery queries copy over inserted tuples simultaneously from both recovery buddies; thus, the recovery buddies can overlap the network costs of sending tuples, and the recovering site essentially receives two tuples in the time to send one.

One point worth mentioning but not illustrated by the results is that because HARBOR only queries for committed data, aborted transactions do not impact HARBOR's recovery performance. On the other hand, because ARIES must redo all transactions and undo aborted ones, an aborted transaction after a checkpoint actually costs ARIES twice as much work as a committed transaction.

6.4.2 Recovery Performance on Update Workloads

In the second recovery experiment, I fix the number of transactions at 20 K and vary the number of historical segments updated by introducing update transactions that update tuples in older segments of the table. The purpose of this experiment is to measure the cost of scanning additional segments for updates that happened after the checkpoint to the original 1 GB of data, as required by Phase 2. Figure 6-5 reports the recovery time of the four scenarios as functions of the number of historical segments updated. Note that the number of historical segments does not include the most recent segment in a table, which must already be scanned

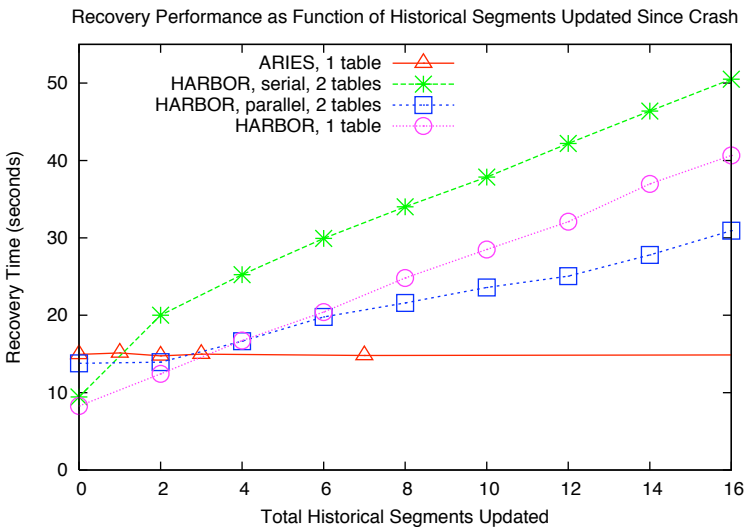

Figure 6-5: Recovery performance as a function of segments updated since crash.

by Phase 1 of recovery; furthermore, in the scenarios involving both tables, the segment count reflects the total number of historical segments updated in both tables.

In the single table scenarios, the graph illustrates that HARBOR performs extremely well in the situations where few historical segments have been updated since the last checkpoint; in fact, HARBOR's recovery performance exceeds ARIES performance in the case of 3 or fewer segments. As the number of historical segments increases, HARBOR's recovery performance degrades linearly because HARBOR needs to scan the segments whose $T_{max-deletion}$ timestamps occur after the checkpoint in order to find the particular tuples updated; on the other hand, because ARIES only scans the tail of the log since the last checkpoint rather than the actual data to find updates, its recovery time remains constant regardless of the number of segments updated. In database systems where checkpoints occur frequently and where the update workload consists mostly of inserts and relatively few OLTP updates to historical data, one expects that the system would tend to exhibit recovery performance in the region of the graph with few historical segments; these conditions hold true in data warehouse environments.

The scenarios with two tables shed additional light on HARBOR's parallel re-

covery performance. While parallel recovery of two tables performed worse than recovery of a single table in the previous experiment, this experiment establishes an opposite trend; as the total number of historical segments increases, parallel recovery actually exhibits better performance than serial recovery. Even though the work in Phase 1 cannot be overlapped, the work in Phase 2 can be parallelized because each live worker site can share in the responsibility of scanning segments from different tables during recovery. The net effect of the concurrency is a substantial reduction of recovery time over the serial case.

The results indicate that when the transactions to recover consist primarily of insertions and of updates concentrated on a few historical segments, HARBOR actually performs better than ARIES.

6.4.3 Analysis of Recovery Performance

To better understand the performance characteristics of the HARBOR recovery approach, I decompose HARBOR's recovery time from the previous experiment's single table scenario into four constituent parts: Phase 1, Phase 2's remote `SELECT` and local `UPDATE` of historical tuples deleted between the checkpoint and the HWM, Phase 2's remote `SELECT` and local `INSERT` of tuples inserted between the checkpoint and the HWM, and Phase 3. Figure 6-6 shows the decomposition of the recovery times for different numbers of updated segments.

The time required for Phase 1 remains constant at 3.1 s regardless of the number of historical segments updated and reflects the time the system spends in Phase 1's `DELETE` query, scanning the last segment for tuples inserted after the checkpoint. Because no updates reached disk after the checkpoint, the worker uses the $T_{max-deletion}$ timestamp associated with segments to avoid scanning segments for deletions after the checkpoint as part of Phase 1's `UPDATE` query; hence, the cost of Phase 1's `UPDATE` query is negligible. Had the worker crash happened after updates to historical segments reached disk but before a new checkpoint could be written, the worker would need to invest recovery time scanning those segments;

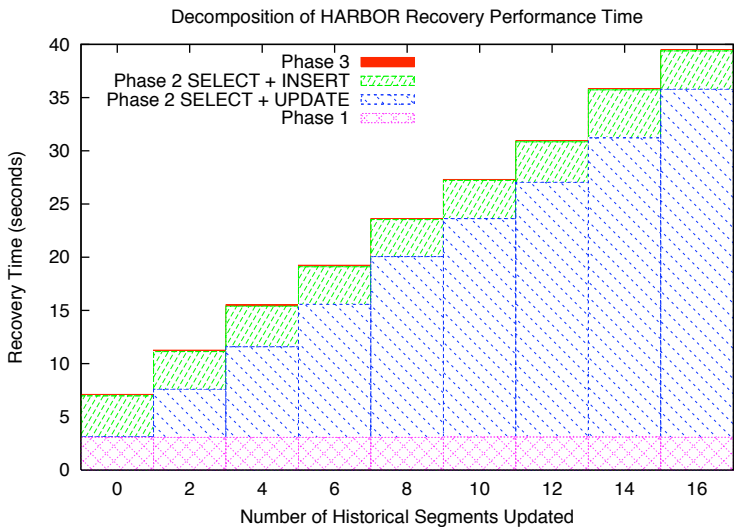

Figure 6-6: Decomposition of recovery performance by phases, with different number of segments.

given a checkpointing frequency of a few seconds and a warehouse workload with few historical updates, however, this situation is rare.

The amount of time spent by the recovery buddy in Phase 2's `SELECT` and `UPDATE` queries to identify updates made to older segments increases linearly with the number of segments updated, at a rate of 2.0 additional seconds of recovery time for each additional historical segment updated. The site spends the bulk of this time scanning segments with eligible $T_{max-deletion}$ timestamps for updated tuples. Phase 2's `SELECT` and `INSERT` queries consume a fairly constant cost of 3.7 s to have the recovery buddy send over the roughly 20 K newly inserted tuples to the worker site.

Phase 3 consumes an insignificant fraction of recovery time and is barely visible in the figure because the coordinator executes no inserts or updates during recovery; therefore, the crashed worker copies all the missing tuples during Phase 2's recovery queries. The next and last experiment shows how transactions during recovery impact recovery performance.

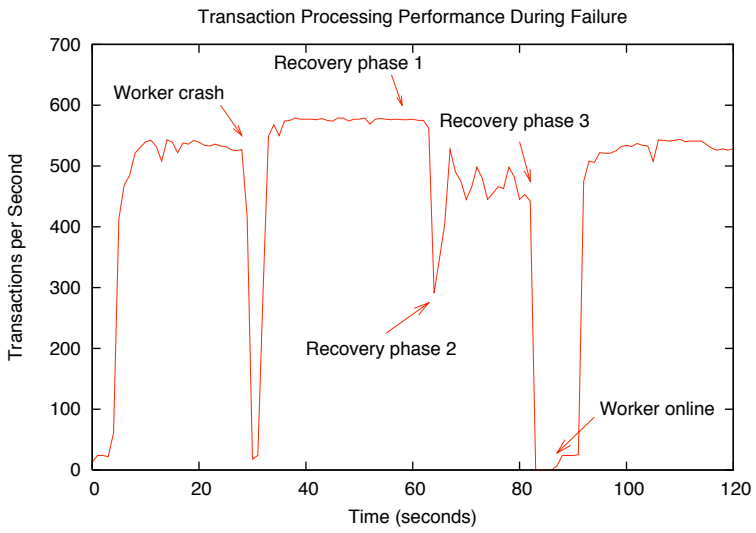

Figure 6-7: Transaction processing performance during site failure and recovery.

6.5 Transaction Processing Performance during Site Failure and Recovery

In the last experiment, I capture the runtime performance of my database system in the midst of a site failure and subsequent recovery. The coordinator site, with no concurrency, continuously inserts tuples into a 1 GB table replicated on two worker sites. Thirty seconds into the experiment, I crash a worker process. Another thirty seconds after the crash, I start the recovery process on the worker.

Figure 6-7 illustrates transaction processing performance as a function of time. Transaction throughput steadies at 530 tps prior to the worker crash. The first major dip at time 30 corresponds to the worker crash, the detection of the failure by the coordinator, and the abort of any ongoing transaction. After the worker fails, the throughput by the remaining live worker increases by roughly 40 tps to 570 tps because commit processing now only involves one worker participant rather than two.

At time 60, the crashed worker initiates Phase 1 of recovery, but recovery does not yet affect the overall throughput because the first phase proceeds locally. The small dip at time 64 indicates the start of Phase 2. Between times 65 and 81, the sys-

tem exhibits sporadic but lower average performance as the recovering worker runs historical queries on the live worker to catch up with the HWM, thereby draining some of the live worker's CPU and disk resources away from transaction processing. At time 83, an even larger dip in performance appears as Phase 3 begins; the recovering worker obtains a read lock over the data needed by the insertion transactions, thereby obstructing progress for a short time. By time 86, however, recovery is complete and performance soon steadies back to its initial point.

The small bump between times 86 and 91, where the system averages roughly 25 tps, results due to TCP slow-start and Java's overhead for opening new socket connections. One can observe a similar bump at the start of the graph from time 0 to time 4. Though not shown by this experiment, any read-only transactions or transactions dealing with other tables would neither have suffered from this bump nor the final performance dip between times 83 and 86. The read locks to ensure transactional consistency only affect update transactions, and the re-establishment of TCP socket connections only affects the recovery objects.

The experiment demonstrates that HARBOR can tolerate the fault of a worker site and efficiently bring it back online without significantly impacting transaction throughput. Online recovery of crashed sites from remote replicas is indeed viable and effective in updatable data warehouses.

Chapter 7

Contributions

Walmart executives leverage their 583-terabyte data warehouse to extract consumer sales patterns; Best Buy managers read daily reports summarizing business intelligence gathered from 7 terabytes of data; Wells Fargo customers and bankers use a 4-terabyte warehouse of transaction history to serve up a 360° view of personal spending reports; Google, Yahoo, Priceline, and other internet companies all strive to personalize the user experience and do so by storing terabytes of user click-through data in updatable data warehouses—data warehouses that not only support data mining but also service operational inserts and updates. The thread of data warehouses runs through retail, financial services, e-commerce, communications, travel, insurance, and government sectors.

The strong dependency on data warehouses introduce requirements for five or six 9's of availability (being up and running 99.999% or 99.9999% of the time). When downtime can cost businesses upwards of $1 million an hour [33], fast recovery becomes critical. This thesis represents my attempt at a viable solution.

My main contributions in this thesis include the following:

1. Designing a simple yet efficient crash recovery algorithm for an updatable, distributed data warehouse. Unlike traditional log-based approaches to recovery, which require intimate interaction with disk and file layout, most of HARBOR's recovery algorithm can be built over the standard SQL interface

that relational databases already present. Through a segmenting scheme that partitions database objects by insertion time, HARBOR attains recovery performance comparable to ARIES performance and even exceeds it on warehouse workloads consisting of mostly inserts or updates to recent data; the segment architecture also significantly reduces the engineering work required to build oft-demanded bulk load and bulk drop features.

2. Integrating the recovery algorithm with a high availability framework that supports non-identical replicas and defining a fault tolerance model based on *K*-safety that clearly delineates HARBOR's recovery guarantees. Though the performance benefits of storing data redundantly in non-identical formats are not examined in this thesis, recent research has shown that storing multiple orderings of data can achieve upwards of one to two orders magnitude better query performance [50]. Many log-based high availability approaches [51, 40, 34] require identical replicas and lose this class of advantages.

3. Developing an optimized three-phase commit protocol that eliminates the overhead of synchronous forced-writes and the maintenance of an on-disk log, while preserving transactional semantics. The protocol leverages the logless nature of the recovery algorithm and the guarantees of *K*-safety to achieve correctness. On a simple insert-intensive workload, the optimizations result in 10 times less latency and 2-10 times higher throughput than the traditional two-phase commit protocol with ARIES.

4. Implementing in Java a four-node distributed database system, with support for multiple commit protocols, the ARIES recovery algorithm, and the HARBOR recovery algorithm, and capturing the design details required to engineer a fully functional distributed database system.

5. Evaluating the runtime overhead of logging and of the commit protocols and verifying empirically the efficiency of HARBOR's parallel recovery approach on the four-node database. Furthermore, I have analyzed in detail the experi-

mental results and decomposed the cost of HARBOR's recovery approach into its constituent phases to shed additional light on recovery performance.

The results of my thesis work are highly encouraging and suggest that updatable data warehouses and my integrated approach to solving their recovery and high availability problems are both quite tenable.

Bibliography

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, June 2006.

[2] Amr El Abbadi and Sam Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, 1989.

[3] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. One-phase commit: does it make sense? In *Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, pages 182–192. IEEE Computer Society, 1998.

[4] Yousef J. Al-Houmaily and Panos K. Chrysanthis. 1-2PC: the one-two phase atomic commit protocol. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 684–691, New York, NY, USA, 2004. ACM Press.

[5] Hisham Alam. High-availability data warehouse design. *DM Direct Newsletter*, December 2001. `http://www.dmreview.com/article_sub.cfm?articleId=4374`.

[6] Deena M. Amato-McCoy. One-stop banking. *FinanceTech*, November 2005. `http://www.financetech.com/showArticle.jhtml?articleID=173402232`.

[7] Charles Babcock. Data, data, everywhere. *InformationWeek*, January 2006. http://www.informationweek.com/story/showArticle.jhtml?articleID=175801775.

[8] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, New York, NY, USA, 1995. ACM Press.

[9] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[10] Philip A. Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *PODC '83: Proceedings of the Second Annual ACM symposium on Principles of Distributed Computing*, pages 114–122, New York, NY, USA, 1983. ACM Press.

[11] Anupam Bhide, Ambuj Goyal, Hui-I Hsiao, and Anant Jhingran. An efficient scheme for providing high availability. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 236–245, New York, NY, USA, 1992. ACM Press.

[12] Alex Biesada. Wal-Mart Stores, Inc. Hoovers. http://www.hoovers.com/free/co/factsheet.xhtml?ID=11600.

[13] Michele Bokun and Carmen Taglienti. Incremental data warehouse updates. *DM Direct Newsletter*, May 1998. http://www.dmreview.com/article_sub.cfm?articleId=609.

[14] Joe Bramhall. Best Buy Co., Inc. Hoovers. http://www.hoovers.com/free/co/factsheet.xhtml?ID=10209.

[15] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *ICDE '95: Proceed-*

ings of the 11th International Conference on Data Engineering, pages 190–200, Washington, DC, USA, 1995. IEEE Computer Society.

[16] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, New York, NY, USA, 1984. ACM Press.

[17] Joshua Freed. The customer is always right? Not anymore. July 2004. `http://www.sfgate.com/cgi-bin/article.cgi?f=/news/archive/2004/07/05/national1332EDT0564.DTL`.

[18] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, New York, NY, USA, Dec 1979. ACM Press.

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

[20] Ashish Gupta and José A. Blakeley. Using partial information to update materialized views. *Information Systems*, 20(9):641–662, 1995.

[21] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: problems, techniques, and applications. pages 145–157, 1999.

[22] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. Revisiting commit processing in distributed database systems. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 486–497, New York, NY, USA, 1997. ACM Press.

[23] Robert B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, 35(9):839–843, 1986.

[24] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group commit timers and high volume transaction systems. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 301–329, London, UK, 1989. Springer-Verlag.

[25] Hui-I Hsiao and David J. Dewitt. Chained declustering: a new availability strategy formultiprocessor database machines. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 456–465, February 1990.

[26] Hui-I Hsiao and David J. Dewitt. A performance study of three high availability data replication strategies. *Distributed and Parallel Databases*, 1:53 – 79, January 1993.

[27] Ricardo Jiménez-Peris, M. Patino-Martínez, and Gustavo Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In *SRDS '02: Proceedings of the IEEE International Symposium on Reliable Distributed Systems*, 2002.

[28] Ricardo Jiménez-Peris, M. Patino-Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, 2003.

[29] Bettina Kemme. *Database Replication for Clusters of Workstations*. PhD dissertation, Swiss Federal Institute of Technology, Zurich, Germany, 2000.

[30] Tirthankar Lahiri, Amit Ganesh, Ron Weiss, and Ashok Joshi. Fast-Start: Quick fault recovery in Oracle. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 593–598, New York, NY, USA, 2001. ACM Press.

[31] Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 104–117, New York, NY, USA, 1987. ACM Press.

[32] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the harp file system. In *SOSP '91: Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 226–238, New York, NY, USA, 1991. ACM Press.

[33] Robert Manning. Managing the costs of system downtime. *CIO Update*, September 2004. `http://www.cioupdate.com/budgets/article.php/3404651`.

[34] Microsoft Corp. Log shipping. `http://www.microsoft.com/technet/prodtechnol/sql/2000/reskit/part4/c1361.mspx`.

[35] Microsoft Corp. SQL server 2000 high availability series: Minimizing downtime with redundant servers, November 2002. `http://www.microsoft.com/technet/prodtechnol/sql/2000/deploy/harag05.mspx`.

[36] C. Mohan. A cost-effective method for providing improved data availability during DBMS restart recovery after a failure. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 368–379, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[37] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[38] C. Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.

[39] Oracle Corp. Oracle database 10g release 2 high availability, May 2005. `http://www.oracle.com/technology/deploy/availability/pdf/TWP_HA_10gR2_HA_Overview.pdf`.

[40] Oracle Inc. Oracle database 10g Oracle Data Guard. http://www.oracle.com/technology/deploy/availability/htdocs/DataGuardOverview.html.

[41] Mark Rittman. Implementing real-time data warehousing using oracle 10g. *DBAzine.com*, February 2006. http://www.dbazine.com/datawarehouse/dw-articles/rittman5.

[42] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[43] Philip Russom. Strategies and Sybase solutions for database availability. Technical report, Nov 2001. http://www.sybase.com/content/1016063/sybase.pdf.

[44] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58, New York, NY, USA, 2004. ACM Press.

[45] Jerome H. Saltzer and M. Frans Kaashoek. Topics in the engineering of computer systems. MIT 6.033 class notes.

[46] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, New York, NY, USA, 1979. ACM Press.

[47] Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142, New York, NY, USA, 1981. ACM Press.

[48] Dale Skeen. *Crash recovery in a distributed database system*. PhD thesis, University of California, Berkeley, May 1982.

[49] Alex Snoeren, David Andersen, and Hari Balakrishnan. Fine-grained failover using connection migration. In *USITS '01: Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2001.

[50] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A column-oriented DBMS. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564. ACM, 2005.

[51] Sybase, Inc. Replicating data into Sybase IQ with replication server. `http://www.sybase.com/detail?id=1038854`.

[52] Tandem Database Group. Nonstop SQL. a distributed, high-performance, high-reliability implementation of SQL. In *High Performance Transaction Systems Workshop*, September 1987.

[53] Teradata. DBC/1012 database computer system manual release 2.0, November 1985. Doc. C10-0001-02.

[54] Transaction Processing Performance Council. TPC benchmark H (decision support): Standard specification revision 2.3.0, August 2005. `http://www.tpc.org/tpch/spec/tpch2.3.0.pdf`.

[55] Kelli Wiseth. Find meaning. Oracle, 2001. `http://www.oracle.com/oramag/oracle/01-sep/o51cov.html`.

[56] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 422–433, Washington, DC, USA, 2005. IEEE Computer Society.