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

Edmond Lau
MIT CSAIL
edmondlau@alum.mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

Any highly available data warehouse will use some form of data replication to tolerate machine failures. In this paper, we demonstrate that we can leverage this data redundancy to build an integrated approach to recovery and high availability. Our approach, called HARBOR, revives a crashed site by querying remote, online sites for missing updates and uses timestamps to determine which tuples need to be copied or updated. HARBOR does not require a stable log, recovers without quiescing the system, allows replicated data to be stored non-identically, and is simpler than a log-based recovery algorithm.

We compare the runtime overhead and recovery performance of HARBOR to those of two-phase commit and ARIES, the gold standard for log-based recovery, on a three-node distributed database system. Our experiments demonstrate that HARBOR suffers lower runtime overhead, has recovery performance comparable to ARIES's, and can tolerate the fault of a worker and efficiently bring it back online.

1. INTRODUCTION

A traditional data warehouse consists of a large distributed database system that processes read-only analytical queries over historical data loaded from an operational database. Such warehouses often do not need traditional database recovery or concurrency control features because they are updated via bulk-load utilities rather than standard SQL INSERT/UPDATE commands. They do, however, require high availability and disaster recovery mechanisms so that they can provide always-on access [3].

Recent years have seen increasing interest in warehouse-like systems that support fine-granularity insertions of new data and even occasional updates of incorrect or missing historical data; these modifications need to be supported concurrently using traditional updates [8]. Such systems are useful for providing flexible load support in traditional warehouse settings, for reducing the delay for real-time data visibility, 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 and to mine relationships.

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 recovery to ensure transactional semantics. One commonly used approach is to implement *snapshot isolation* [4], which allows read-only queries to avoid setting any locks by having them read a historical snapshot of the database starting from some point in recent history.

In such updatable environments, it can also be useful to provide some form of time travel; for example, users may wish to save a version of the database before they insert a number of records so that they can compare the outcome of some report before and after a particular set of changes was made to the database.

In this paper, we look at the performance of a new approach to recoverability, high availability, and disaster recovery in such an updatable warehouse. Our approach, called HARBOR (High Availability and Replication-Based Online Recovery), is loosely inspired by the column-oriented C-Store system [33], which also seeks to provide an updatable, read-mostly store. Our evaluation is conducted, however, on a row-oriented database system to separate the performance differences due to our unusual approach to recovery from those that result from a column-oriented approach.

The gist of our approach is that any highly available data warehouse will use some form of replication to provide availability; we show that it is possible to exploit this redundancy to provide efficient crash recovery without the use of a stable log, forced-writes during commit processing, or a complex recovery protocol like ARIES [23]. We accomplish this 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. As a side effect of this approach to recovery, our system also provides versioning and time travel up to a user-configurable amount of history.

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

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

VLDB '06, September 12-15, 2006, Seoul, Korea.

- To provide high availability, 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.

In this paper, we compare the runtime overhead and recovery performance of HARBOR to those of ARIES and two-phase commit on a three-node distributed database. We show that our system has substantially lower runtime overhead (because we do not require disk writes to be forced at transaction commit) and equivalent recovery performance. Moreover, we show that our system can tolerate site failure and recovery without significantly impacting transaction throughput. Though our evaluation in this paper focuses specifically on distributed databases on a local area network, our approach also works with wide area networks. Finally, our approach is substantially simpler than ARIES.

The remainder of this paper is organized as follows. In Section 2, we lay out our fault tolerance model and provide a high level summary of the techniques and steps used by HARBOR. We describe the mechanics behind query execution and commit processing in our approach in Section 3 and then detail the three phases of our recovery algorithm in Section 4. In Section 5, we show results of evaluating HARBOR against two-phase commit and ARIES. We summarize related work in Section 6 and conclude with our contributions in Section 7.

2. APPROACH OVERVIEW

In this section, we first discuss two features of highly available data warehouses—the requirement for data replication and the concurrency control issues associated with warehouse workloads. We then present a fault tolerance model based on data redundancy and an isolation method based on a technique called *historical queries*; together, these two components form the bedrock of our crash recovery algorithm. We conclude the section by providing a high-level overview of how our recovery algorithm works.

2.1 Data Warehousing Techniques

Data warehouse systems share two important characteristics. First, any highly available database system will use data replication to ensure that data access can continue with little interruption if some machine fails. Each database object, whether it be at the granularity of a tuple, a partition, or a table, will be replicated some number of times and distributed among different sites. Unlike many high availability approaches [25, 27, 22], HARBOR does not require that different sites be physically identical replicas or that redundant copies be stored, partitioned, or distributed in the same way, as long as they logically represent the same data.

This flexibility is key to research efforts that argue for storing data redundantly in different sort orders [33], in different compression formats [33, 1], or in different updatable materialized views [9] so that the query optimizer and executor can efficiently answer a wider variety of queries than they would be able to in situations where the database stores redundant copies using identical representations. C-Store [33], for example, achieves one to two orders of magnitude faster performance over commercial databases on a seven-query benchmark by storing data in multiple sort orders and in different compression formats. 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 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* [4], in which read-only transactions read data without locks, and update transactions modify a copy of the data and either resolve conflicts with an optimistic concurrency control protocol or with locks.

Our approach uses a time travel mechanism similar to snapshot isolation that involves using an explicit versioned and timestamped representation of data to isolate read-only transactions. *Historical queries* as of some past time T read time slices of the data that are guaranteed to remain unaffected by subsequent transactions and hence proceed without acquiring read locks. Update transactions and read-only transactions that wish to read the most up-to-date data can use conventional read and write locks as in strict two-phase locking for isolation purposes. The mechanism was first introduced in C-Store [33]; we explain the details behind such an mechanism in Section 2.3.

2.2 Fault Tolerance Model

A highly available database system will employ some form of replication to tolerate failures. We say that a system provides K-safety if up to K sites can fail, and the system can still continue to service any query. The minimum number of sites required for K-safety is $K + 1$, namely in the case where the $K + 1$ workers store the same replicated data. In our approach, we assume that the database designer has replicated the data and structured the database in such a way as to provide K-safety. Our high availability guarantee is that we can tolerate up to K simultaneous failures and still bring the failed sites online.

If more than K sites fail simultaneously, our approach no longer applies, and the recovery mechanism must rely on other methods, such as restoring from some archival copy or rolling back changes to restore some globally consistent state. However, because our recovery approach can be applied to bring sites online as they fail, the database designer can choose an appropriate value of K to reduce the probability of K simultaneous failures down to some value acceptable for the specific application.

In our fault tolerance model, we assume fail-stop failures and do not deal with Byzantine failures. We do not deal with network partitions, corrupted data, or incompletely written disk pages. We also assume reliable network transfers via a protocol such as TCP.

2.3 Historical Queries

We first describe how historical queries can be supported in a database system before proceeding to provide a high level description of our recovery algorithm. 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. This time travel feature allows clients to inspect past states of the database.

We can support historical queries by using a *versioned* representation of data in which timestamps are associated with each tuple. We internally augment 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, we assign the transaction's commit time to the tuple's insertion timestamp; we set 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, we assign the transaction's commit time to the tuple's deletion timestamp. When a transaction updates a tuple, rather than updating the tuple in place, we represent the update as a deletion of the old tuple and an insertion of the new tuple.

Structuring the database in this manner is reminiscent of version histories [28] 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 whether a particular tuple is visible at time T. A tuple is visible at time T if 1) it was inserted at or before T and if 2) it is either not deleted or deleted after T. This predicate can be straightforwardly pushed down into access methods as a SARGable predicate [29].

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. The amount of history maintained by the system can be user-configurable either by having a background process remove all tuples deleted before a certain point in time or by using a simple "bulk drop" mechanism that we introduce in Section 3.1.

2.4 Recovery Approach

Having established the fault tolerance model and a framework for historical queries, we proceed to describe our recovery algorithm to bring a crashed site online; we defer elaboration of the details to Section 4. The algorithm consists of three phases and uses the timestamps associated with tuples to answer time-based range queries for tuples inserted or deleted during a specified time range.

In the first phase, we use a checkpointing mechanism to determine the most recent time T such that we can guarantee that all updates up to and including time T have been flushed to disk. The crashed site then uses the timestamps available for historical queries to run local update transactions to restore itself to the time of its last checkpoint. In order to record checkpoints, we 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 periodically writes a checkpoint for some past time T by first taking a snapshot of the dirty pages table at time $T + 1$. For each page that is dirty in the snapshot, the system obtains a write latch for the page, flushes the page to disk, and releases the latch. After all dirty pages have been flushed, we record T onto some well-known location on disk, and the checkpoint is complete. In the event that we are recovering a disk from scratch or that the checkpoint has become corrupted, we can recover with a checkpoint time of 0.

In the second phase, the site executes historical queries on other live sites that contain replicated copies of its data in order to catch up with any changes made between the last checkpoint and some time closer to the present. The fact that historical queries can be run without obtaining read locks ensures that the system is not quiesced while large amounts of data are copied over the network; our approach would not be a viable solution if read locks needed to be obtained for the recovery query.

In the third and final phase, the 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. Because the historical queries in the second phase tend to substantially reduce the number of remaining changes, this phase causes relatively little disruption to ongoing transactions, as we show in Section 5.3. The coordinator then forwards any relevant update requests of ongoing transactions to the site to enable the crashed site to join any pending transactions and come online; this step requires that the coordinator maintain a queue of update requests for each ongoing transaction.

Our results show that our recovery approach 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.

3. QUERY EXECUTION

In our 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. Read queries may be distributed 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.

To facilitate recovery, the coordinator maintains an in-memory queue of logical update requests for each transaction it coordinates. A queue for a particular transaction is deleted when the transaction commits or aborts.

In order to support historical queries, the coordinator determines the insertion and deletion timestamps that must be assigned to tuples at commit time. If the database supports more than one coordinator site, we require a time synchronization protocol, such as the one described in [33], for the coordinators to agree on the current time. If there are time lags between sites, the system must ensure that a historical query is run as of a time less than the current time at any worker site handling the query.

For our system, we use timestamps corresponding to coarse granularity epochs that span a few seconds each. We advance epochs synchronously across sites by having a coordinator node periodically send out "advance epoch" announcements to ensure that sites are time synchronized. Finer granularity timestamps would allow for more fine-grained historical queries but would require more synchronization overhead. The database frontend holds responsibility for mapping the times used for historical queries by clients to the internal representation of timestamps in the database.

Because timestamps are assigned at commit time, a worker must keep some in-memory state for each update transaction designating the tuples whose timestamps must be assigned. A worker site can do this by maintaining, for each transaction, an in-memory list of tuple identifiers for tuples inserted by the transaction and another list for those 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. The in-memory lists also contain sufficient information to double as in-memory undo logs for handling aborts.

One caveat is that if the buffer pool supports a STEAL policy [12], uncommitted inserts written to the database should contain a special value in its insertion timestamp field so that the uncommitted data can be identified during recovery and also ignored by queries.

Aside from the versioned representation and the support for historical queries, query execution in HARBOR differs from standard distributed databases in two respects. First, in order to improve performance of recovery queries that must apply range predicates on timestamps, we partition all database objects by insertion time into smaller *segments*. We describe this segment architecture and its implications in Section 3.1. Second, because our recovery approach does not require a stable log, we can reduce commit processing overhead by eliminating workers' forced-writes when using two-phase commit and all forced-writes when using three-phase commit, both of which we describe in Section 3.2.

3.1 Partitioning Objects by Insertion Timestamp into Segments

During normal transaction processing, the insertion and deletion timestamps associated with tuples are used solely to determine a tuple's visibility as of some time T. In data warehouse environments, where historical queries are run relatively close to the present time and where updates to historical data happen infrequently, most of the tuples examined by a query will be visible and relevant. Thus, little overhead will be incurred scanning invisible tuples, and we do not need to build indexing structures on timestamps for this purpose.

As we will see later in the discussion on recovery, however, recovery queries require identifying those tuples modified during specific timestamp ranges and may be potentially inefficient. For recovery, we would like to be able to efficiently execute 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, we 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).

Our 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*. 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 times t and $t + \Delta t$, the next segment containing all tuples inserted between $t+\Delta t$ and $t+2\Delta t$, etc., as illustrated in Figure 1. The size of the time ranges need not be fixed, and we may instead choose to limit the size of each segment by the number of 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. New tuples are added onto the last segment until either the segment reaches some pre-specified size or until we reach the end of the segment's time range, at which point, we create a new segment to insert subsequent tuples.

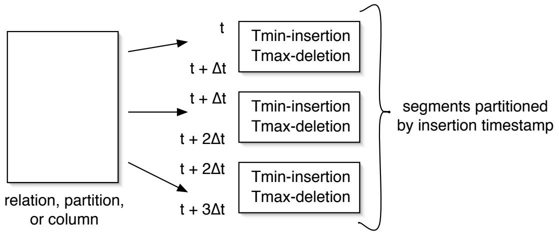

Figure 1: Database object partitioned by insertion timestamp into segments.

To efficiently support the desired range predicates, we 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 timestamps contained within a segment can be deduced from the $T_{min-insertion}$ timestamp of the next segment or is equal to the current time if no next segment exists. We 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, we can efficiently identify those segments with insertions during some time range and also those segments updated since some time T. Note that even without this segment architecture, the recovery approach remains correct but may be inefficient.

These changes to the physical representation require some modification to the query execution engine. Read queries on an object must now be conducted on multiple segments, 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 to find desired tuples.

In return for the segment processing overhead, this solution also provides two concrete benefits to data warehouse environments. Many warehouse systems require regular bulk loading of new data into their databases. Under a segment representation, we can easily accommodate bulk loads of additional data by creating a new segment with the new data 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 upwards of one terabyte of information regarding user clicks on websites. These warehouse systems are only designed to store the most recent N days worth of click-through data. Our time-partitioned segment architecture supports a symmetric "bulk drop" feature whereby we can, for instance, create segments with time ranges of a day and schedule a daily drop of the oldest segment. These bulk load and bulk drop features would require substantially more engineering work under other architectures.

3.2 Commit Processing

One significant benefit that our recovery mechanism confers is the ability to support more efficient commit protocols for update transactions by eliminating forced-writes to a stable log. In this section, we first show how we can tweak the traditional two-phase commit protocol (2PC) [24] to support our timestamped representation of data. We then examine how we can exploit K-safety and our recovery approach to build more efficient variants of 2PC and also of a less widely used, but non-blocking, three-phase commit protocol (3PC) [30]. HARBOR supports both 2PC and 3PC.

3.2.1 Two-Phase Commit Protocol

In the first phase of traditional 2PC, the coordinator sends PREPARE messages to worker sites and requests votes for the transaction. In the second phase, the coordinator force-writes a COMMIT log record and sends COMMIT messages to the workers if it had received YES votes from all the sites; otherwise, it force-writes an ABORT record and sends ABORT messages. During 2PC, worker sites need to force-write PREPARE log records prior to sending out a YES vote, and they need to force-write COMMIT or ABORT records prior to locally committing or aborting a particular transaction.

To support our timestamped representation of data, we augment this framework with two minor changes: 1) COMMIT messages also include a commit time to be used for modified tuples, and 2) the in-memory lists of modified tuple identifiers that a worker maintains for a transaction can be deleted when the transaction commits or aborts.

3.2.2 An Optimized Two-Phase Commit Protocol

Each of the three non-overlapping forced-writes (two by each worker and one by the coordinator) lengthens transaction latency by a few milliseconds. Modern systems use a technique known as group commit [10, 13] to batch together the log records for multiple transactions and to write them to disk using a single disk I/O. Group commit can help increase transaction throughput for non-conflicting concurrent transactions but does not improve latency.

Our 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 lets the recovering site know the transaction's outcome, while the PREPARE record in the absence of any COMMIT or ABORT record lets the recovering site know that it may need to ask another site for the final consensus.

When K-safety exists, however, a crashed worker does not need to determine the final status of transactions. 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 realization that enables worker sites under our approach to eliminate the forced-writes to stable storage and the use of an on-disk log all together.

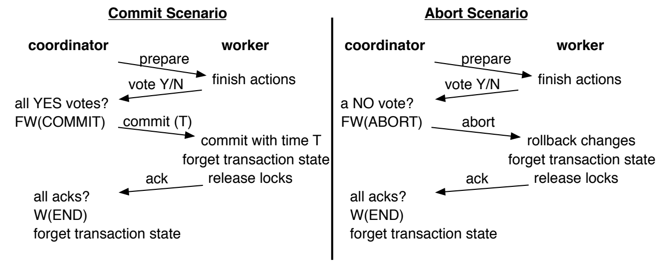

Figure 2: Commit and abort scenarios of our optimized two-phase commit protocol. FW means force-write and W means a normal log write.

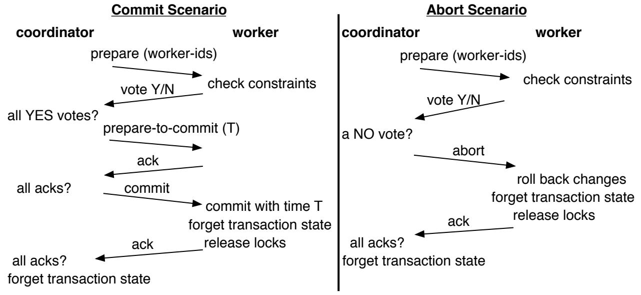

Figure 3: Commit and abort scenarios of our optimized three-phase commit protocol.

Our optimized 2PC therefore looks like the interaction shown in Figure 2. A worker site, upon receiving a PREPARE message, simply finishes its role in the transaction and votes YES or NO. When a worker site receives a COMMIT or ABORT message, it simply assigns the commit time to the modified tuples or rolls back all changes and sends an ACK to the coordinator. No forced-writes or log writes are necessary by the workers because consistency after a failure can be restored by our recovery protocol. We examine the performance implications of eliminating the workers' two forced-writes in Section 5.1.

3.2.3 An Optimized Three-Phase Commit Protocol

Under our modified 2PC, the coordinator is still required to 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 know what to respond to worker inquiries upon rebooting. In this section, we show that by leveraging a less widely used 3PC protocol, we can actually eliminate all forced-writes and all on-disk logs and further improve transaction processing performance.

The key observation behind the canonical 3PC protocol [30] is that by introducing an additional *prepared-to-commit* state between the *prepared* state and the *committed* state, we can enable workers to agree on a consistent outcome for a transaction without depending on the coordinator to ever come back online. This *non-blocking* property of 3PC means that coordinators do not need to maintain stable logs because they do not need to recover the transaction state.

We optimize the canonical 3PC protocol by eliminating all forced-writes and log writes by all participants. Our optimized version of the 3PC protocol is illustrated in Figure 3 and works as follows. In the first phase, the coordinator sends a PREPARE message to all workers with a list of the workers participating in the transaction. Each worker then enters the *prepared* state and responds with a YES or NO vote indicating its ability to commit the transaction.

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 and the interaction proceeds as in 2PC.

Finally in the third phase, after all ACKs have been received, 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.

If a coordinator site fails, we can use a consensus building protocol as described in [30] and whose correctness is proven in [31]. A backup coordinator is chosen by some arbitrarily pre-assigned ranking (or some other voting mechanism) and then decides from its local state how to obtain a consistent transaction outcome, as follows. If the backup had not sent a vote in the first phase, had voted NO, or had aborted, it sends an ABORT message to all workers because no site could have reached the *prepared-to-commit* state. If the backup is in the *prepared* state and had voted YES, it asks all sites to transition to the *prepared* 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, it replays the last two phases of 3PC, reusing the same commit time it received from the old coordinator. If it is in the *committed* state, it sends a COMMIT message to all workers. Workers can safely disregard any duplicate messages they receive.

Using this 3PC in conjunction with our recovery approach, we can support transaction processing and recovery without forced-writes and without maintaining a stable log for any sites. When the network is much faster than the disk, as in true on all modern database clusters, the extra round of messages introduces less overhead than a forced disk write. We evaluate the runtime performance implications of using this 3PC in Section 5.1.

4. RECOVERY

In this section, we first introduce some terminology to be used in our recovery discussion and then present the three phases of the algorithm for bringing a crashed site online.

Let S be the site that failed. We 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, and objects can be recovered in parallel.

Assuming that S crashed while the system still had K-safety for $K \geq 1$, we 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, we can compute a *recovery predicate* such that 1) the sets of tuples obtained by applying the recovery predicate on each recovery object are mutually exclusive, and 2) the union of all such sets collectively cover the object *rec*. We assume that this knowledge is stored in the catalog.

For example, suppose that we have 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*, where the different primary indices are used to more 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 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. Note that this computation is no different than 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, we now discuss in detail the three phases of our recovery approach. We describe virtually all steps of recovery using simple SQL queries. Note that all INSERT, DELETE, and UPDATE statements in the recovery queries refer to their normal SQL semantics rather than the special semantics that we assign to them in our support for historical queries.

4.1 Phase 1: Restore local state to the last checkpoint

Recall from Section 2.4 that a site writes checkpoints by periodically flushing all dirty pages to disk. Call the time of S's most recent checkpoint $T_{checkpoint}$. During recovery, we temporarily disable any new checkpoints from being written. We can guarantee that all updates that committed at or before $T_{checkpoint}$ have been flushed to disk, but we do not know whether changes from subsequent transactions have partially or fully reached disk or if uncommitted changes also remain. In Phase 1, we discard all updates after the checkpoint as well as any uncommitted changes to restore the state of all committed transactions as of $T_{checkpoint}$.

First, we delete 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 will be run on the local site; we will use the keyword `REMOTELY` later on to indicate queries that need to run on remote replicas. The semantics of `SEE DELETED` is that rather than filtering out deleted tuples as a standard query would do, we run the query in a special mode that allows us to turn off the delete filtering and to see both insertion and deletion timestamps. Also, note that by `DELETE`, we mean the standard notion of removing a tuple rather than recording the deletion timestamp. The `uncommitted` value refers to the special value assigned to the insertion timestamp of uncommitted tuples; in practice, the special value can simply be a high value greater than the value of any valid timestamp.

Using the $T_{min-insertion}$ associated with segments, we can efficiently find the tuples specified by the range predicate on insertion time. Assuming that the database is configured to record checkpoints somewhat frequently, executing this query should only involve scanning the last few segments; we evaluate the cost of this scan later in Section 5.

Next, we undelete 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, *i.e.*, an update in place, rather than a delete and an insert. Using the $T_{max-deletion}$ recorded on each segment, we can prune out any segments whose most recent deletion time is less than or equal to the $T_{checkpoint}$. Thus, we pay the price of sequentially scanning a segment to perform the update if and only if a tuple in some segment was updated or deleted after the last checkpoint. In a typical data warehouse workload, we expect updates and deletions to be relatively rare compared to reads and that the updates and deletions would happen primarily in the most recently inserted tuples, *i.e.*, in the most recent segment. Thus, we expect that in the common case, either no segment or only the last segment should need to be scanned.

4.2 Phase 2: Catch up with historical queries

The strategy in the second phase is to leverage historical queries that do not require locks to rebuild the object *rec* up to some time close to the present. 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 HWM are not visible and that all tuples deleted after HWM appear as if they had not been deleted (*i.e.*, their deletion times would appear to be 0).

First, we find all deletions that happened between the $T_{checkpoint}$ and the HWM to the tuples that were inserted at or prior to $T_{checkpoint}$. We require 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. We can then run a set of historical queries as of the time HWM for each recovery object and recovery predicate that we have 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 tuples 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 reduce the number of segments that must be scanned to answer the range predicates on insertion and deletion times; on typical warehouse workloads where historical updates are rare, few segments need to be scanned. For each *(tup_id, del_time)* tuple that S receives, we run the following query to locally update the deletion time of the corresponding tuples 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
```

The predicate on the deletion time ensures that we are updating the most recent tuple in the event that the tuple has been updated and more than one version exists. We assume that we have an index on tuple_id, which usually exists on a primary key, to speed up the query.

Next, we query 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 we insert 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 our versioned representation; it reflects the idea that S is merely copying the requested data into its copy of *rec*. While the second predicate on insertion time is implicit by the definition of a historical query as of the HWM, we make explicit that we 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. At this point, if we have recovered all database objects in parallel up to the HWM, S can optionally record a new checkpoint as of the time HWM to reflect that it is consistent up to the HWM.

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

Phase 3 consists of two parts: running non-historical read queries with locks to catch up to with all committed updates at the current time and joining any ongoing transactions dealing with *rec*. The structure of the read queries highly parallel the structure of those in Phase 2, except that we do not run them in historical mode.

4.3.1 Query for committed data with locks

First, we acquire 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 we assume that the system has some distributed deadlock detection mechanism, by using time-outs 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 are either a) not dealing with *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 that update transactions must update all live copies of the data). 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, S's successful acquisition of read locks implies that any non-pending transactions that updated *rec*'s data must have already completed commit processing and released their locks.

At this point, we are missing, for all tuples inserted before HWM, any deletions that happened to them after the time HWM. To find the missing deletions, we use a similar strategy to the one used at the start of Phase 2. For each recovery object and recovery predicate pair, we execute 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
```

We again rely on the segment architecture to make the two timestamp range predicates efficient to solve. For each *(tup_id, del_time)* tuple in the result, we 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
    WHERE tuple_id = tup_id
      AND deletion_time = 0
```

Finally, to retrieve any new data inserted after the HWM, we run 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, we expect the query to only examine the last segment, but we may need to examine more segments depending on whether segments were created since the beginning of Phase 2. After this query, S has caught up with all committed data as of the current time and is still holding locks. If we have recovered all objects on S in parallel up to the current time, S can then write a checkpoint for one less than the current time.

4.3.2 Join ongoing transactions and come online

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

The protocol for joining all ongoing 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 Section 3, each coordinator maintains a queue of logical update requests for each transaction that it coordinates; we now use this queue to complete recovery.

Let P 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 P. 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. After all transactions in P 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. 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 waiting for locks on the recovery objects can then continue.

4.4 Failures during Recovery

If the recovering site S fails during recovery, it restarts recovery upon rebooting, using a more recent checkpoint if 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, we require 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 our implementation, the detection of an abruptly closed TCP socket connection as a signal for failure. When a recovery buddy detects failure of a recovering node, it overrides the node's ownership of the locks and releases them so that other transactions can progress. The coordinator treats any transactions that S had been joining as having aborted.

If a recovery buddy fails during recovery, the set of recovery objects and recovery predicates computed at the start of recovery are no longer valid; thus, the recovering node S simply releases any locks that it may be holding and restarts recovery with a new set of recovery buddies. The coordinator aborts any transactions S was in the middle of joining.

Coordinator failure during recovery is handled using the consensus building protocol previously described.

5. EVALUATION

In order to evaluate the runtime overhead and recovery performance of our approach, we have implemented a distributed database system, consisting of 11.2 K lines of Java code. The system supports most of the standard database operators and also special versions of the scan, insert, delete, and update operators to support historical queries. We implement distributed transactions and messages using Java's TCP socket library and build special network operators that insert and read tuples to and from socket connections. Both coordinators and workers are multi-threaded and can support concurrent transactions. We have implemented all of the components described in the paper except the consensus building protocol to tolerate coordinator failures and the mechanism to handle failures during recovery. Additional implementation details can be found in [19].

In this section, we compare the runtime overhead and recovery performance of HARBOR to the performance of traditional 2PC and ARIES, both of which we have also implemented, on a three-node distributed database. ARIES [23], the gold standard for log-based database recovery, uses a stable undo/redo log to support atomicity and crash recovery. The buffer pool uses write-ahead logging and forces log records describing modifications to disk before writing a modified page. The ARIES algorithm for crash recovery 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. Unlike HARBOR's recovery algorithm, which relies on a logical interface to the data, ARIES requires intimate interaction with the disk and log file format.

Each node in the system is 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 RAID disk was not used except as a separate disk for the log. The machines are connected together in a local area network with 85 Mb/s bandwidth, as measured experimentally.

We 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.

5.1 Runtime Overhead of Logging

In the first experiment, we compare the runtime overhead of our commit processing against 2PC and 3PC with write-ahead logging using ARIES. Whenever a logging operation is involved, we use group commit [13].

One coordinator node sends update transactions to the other two worker nodes. For simplicity, both workers store the same replicated data in the same format; in general, this need not be the case. We vary the number of concurrent transactions running 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.

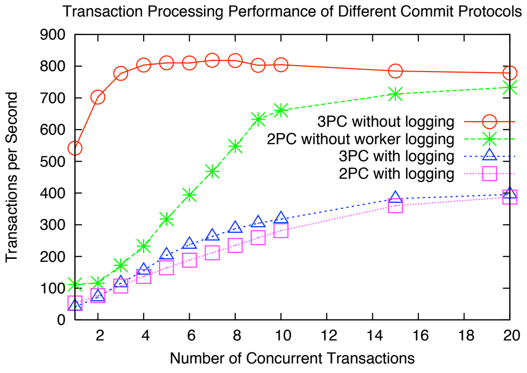

Figure 4: Transaction processing performance of various commit protocols.

Figure 4 shows the transactions per second (tps) in four cases: our optimized 3PC without logging, our optimized 2PC without logging at worker sites, canonical 3PC [30] with logging at workers and no logging at the coordinator, and traditional 2PC with logging. In the case of one transaction running at a time, which illustrates the average latency of a transaction, we observe that our optimized 3PC (latency = 1.8 ms) runs 10.2 times faster than traditional 2PC (18.8 ms). Even in the case of 10–20 concurrent transactions, the throughput of optimized 3PC remains 2–2.9 higher than that of traditional 2PC. Other experiments show that without replication, an ARIES-based system attains only 2-23% higher throughput for various concurrency levels [19]. Our results are encouraging and demonstrate that we can eliminate substantial commit processing overhead using our integrated approach. Of course, with a less update intensive workload, these differences would be less dramatic. Additional experiments on CPU-intensive transactions confirm that optimized 3PC still runs 1.7 faster than traditional 2PC with 4 ms of additional processing time per transaction for 5 concurrent transactions [19].

5.2 Recovery Performance

In the next two experiments, we compare the performance of HARBOR's recovery approach against ARIES and show that our recovery performance is indeed comparable. We believe our benchmark ARIES recovery protocol to be a faithful implementation of ARIES as specified in [23]. Although modern implementations of ARIES and other log-based recovery algorithms include many optimizations that we have omitted, the fact that the performance of our recovery implementation is on par with our ARIES implementation is still highly encouraging.

Both experiments follow a similar setup. The worker sites start by each flushing all dirty pages to disk and writing a checkpoint. Each table on the workers consumes roughly 1 GB, using 101 segments with the last segment half full. The coordinator then runs some number of insert/update transactions with both worker sites; the workers are not allowed to flush dirty pages during these transactions. After the coordinator finishes the transactions, we crash a worker site and measure the time for the crashed worker to recover,

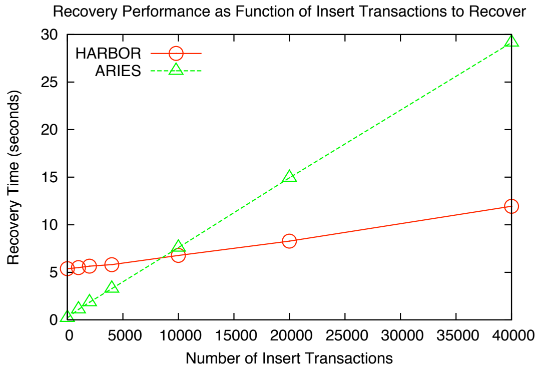

Figure 5: Recovery performance as a function of insert transactions to recover.

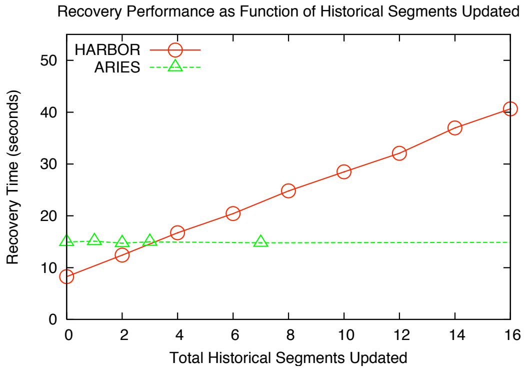

Figure 6: Recovery performance as a function of historical segments updated.

both under HARBOR and under ARIES. Because no transactions occur during recovery, HARBOR does not need to spend time in Phase 3; we show the performance impact of all three phases later in Section 5.3.

Regardless of ARIES's checkpoint frequency, other sites may still receive updates after a site crashes. ARIES's recovery time thus also corresponds to the time ARIES would take to recover after using a conventional recovery mechanism like log shipping [21] to transfer any missing log records.

In the first recovery experiment, the coordinator sends only insert transactions, with one insertion per transaction, and we vary the number of transactions performed. Because newly inserted tuples are appended to the heap file, the new tuples affect only the last segment, though the worker may create a new segment given sufficient insertions. Figure 5 illustrates the results of this experiment.

HARBOR takes 5.3 s to recover 2 insert transactions because the system must scan the most recent segment in Phase 1 for uncommitted data or data committed after the checkpoint. While ARIES performs well on a small number of transactions, ARIES performance degrades over 3 times more rapidly than HARBOR performance, showing that log processing for committed transactions suffers from higher overhead than querying for the committed data from a remote site. In our system, the crossover point where ARIES falls behind HARBOR performance happens at 4581 insert transactions. If our system were to be saturated with insert transactions, this corresponds to 9 s of transactions. Typically, we would expect crashes to last substantially longer.

In the second experiment, we fix the number of transactions at 20 K, but we vary the number of historical segments updated by introducing update transactions that update tuples in older segments of the table. The experiment measures the cost of scanning additional segments for updates that happened to the original 1 GB of data after the time of the last checkpoint, as required by Phase 2. As we can see from the results in Figure 6, HARBOR does well in the cases where few historical segments have been updated and does poorly as the number of segments updated increases. Because ARIES only requires scanning the tail of the log since the last checkpoint rather than scanning the actual data for modifications, its recovery time remains constant regardless of the number of segments updated.

To better understand the performance characteristics of our recovery approach in this second experiment, Figure 7 decomposes our recovery times into their constituent phases for different numbers of updated segments. Phase 1 consumes a constant 3.1 s to scan 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. The time spent by the recovery buddy in Phase 2 to identify updates made to older segments increases roughly linearly with the number of segments updated. A fairly constant 3.7 s is spent in Phase 2 copying over the roughly 20K newly inserted tuples. The cost of Phase 3 is barely visible in this experiment because no updates occurred during recovery.

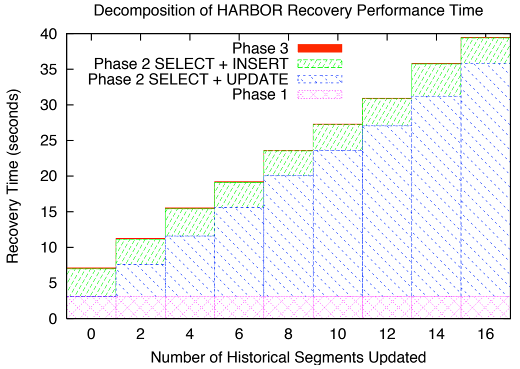

Figure 7: Decomposition of recovery performance by phases, for different numbers of historical segments updated.

Additional results in [19] show that recovering multiple database objects in parallel with HARBOR can reduce recovery time significantly over recovering them sequentially.

Our results indicate that when the update transactions that occurred since the crash consist primarily of insertions and relatively few updates to historical updates, HARBOR actually performs better than ARIES.

5.3 Performance during Failure and Recovery

In our last experiment, we observe the runtime performance of our system in the midst of a site failure and subsequent recovery. The coordinator site continuously inserts tuples into a 1 GB table replicated on two worker sites. Thirty seconds into the experiment, we crash a worker process. Another thirty seconds after the crash, we start the recovery process on the worker.

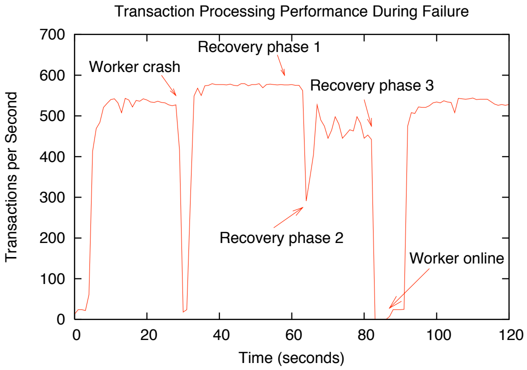

Figure 8: Transaction processing performance during site failure and recovery.

Figure 8 illustrates transaction processing performance as a function of time. The first major dip at time 30 corresponds to the worker crash, the coordinator's detection of the failure, and the abort of any ongoing transaction. After the worker fails, the throughput by the remaining live worker increases by roughly 40 tps because commit processing now only involves one worker participant rather than two. At time 60, the crashed worker begins Phase 1 of recovery, but recovery does not yet affect the overall throughput because the Phase 1 is performed locally. Between time 65 and 81, the system exhibits sporadic but lower average performance as the recovering worker runs historical queries on the live worker to catch up with the HWM in Phase 2. At time 83, an even larger dip in performance appears as Phase 3 begins; the recovering worker queries with a read lock over the data needed by the insertion transactions and obstructs progress for a short time. At time 86, recovery is complete and performance steadies soon 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. Though not shown by this experiment, any read-only transactions or transactions dealing with other tables would not have suffered from this bump nor the performance dip between times 83 and 86.

6. RELATED WORK

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 [7]. This primary copy approach suffers from the setback that the secondary replicas store temporarily inconsistent copies of data and therefore cannot support up-to-date read queries; the replicas play a major role only in the rare 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 [20] as well as in many distributed databases (e.g., Sybase's Replication Server [34], Oracle Database with Data Guard [26], and Microsoft SQL Server [21]).

Another approach to high availability is to maintain multiple identical replicas that are updated synchronously, using a 2PC protocol in conjunction with write-ahead logging to guarantee atomicity. Many places in the literature describe this protocol; see Gray [12] or Bernstein [5] for an overview. This second approach unfortunately suffers in transaction throughput due to the forced-writes of log records required by 2PC; it also requires correctly implementing a complex recovery protocol like ARIES, which is non-trivial.

ClustRa [15] uses a neighbor write-ahead logging [14] technique, in which a node redundantly logs records to main memory both locally and on a remote neighbor, to avoid forced-writes; HARBOR uses redundancy in a similar way but avoids the log all together during crash recovery whereas ClustRa processes a distributed log. Jiménez-Peris *et al.* [16, 18] describe a recovery approach similar to ours in structure, but with many shortcomings. Their scheme requires maintaining an undo/redo log, assumes a middleware communication layer with reliable causal multicast to handle joining pending transactions, and assumes that all sites store data identically; HARBOR imposes none of these assumptions.

In our replication approach, we propagate all writes to $K+1$ sites, which allows us to tolerate failures of up to K sites without sacrificing availability and allows us to process read-only queries at just a single site. Voting-based [11, 17] approaches could be applied to allow us to send updates to fewer machines by reading from several sites and comparing timestamps from those sites. View-change protocols [2] build on these ideas to allow replicated systems to potentially tolerate network partitions and other failures that we do not address here. Allowing the system to proceed without ensuring replicas have all seen the same transactions may make it difficult to achieve one-copy serializability [6].

Most highly available database systems rely on special purpose tools (e.g., the one described in Snoeren *et al.*[32]) to handle failover in the event that a site the client is talking to fails. To enable our system to seamlessly failover and prevent the client from seeing a broken connection, we would need similar tools.

7. CONCLUSIONS

Our core contribution in this paper is recognizing that we can leverage the technique of historical queries and the requirement of data replication in highly available data warehouse environments to build a simple yet efficient crash recovery mechanism. HARBOR, our integrated approach to high availability and recovery, avoids the complexities and costs of an on-disk log. Most of our recovery approach can be described by a few fairly standard SQL queries, and our experiments indicate that our recovery performance is comparable to ARIES and even surpasses ARIES when the workload consists primarily of inserts and updates to recent data, which is the case in data warehouse environments.

By eliminating recovery logs, we also eliminate forced-writes during commit processing, thereby increasing transaction throughput. Our experiments demonstrate that on our distributed database implementation, the optimized three-phase commit protocol that we support can increase transaction throughput by a factor of 2 to 10 over traditional two-phase commit with ARIES on a simple update-intensive workload. The results are encouraging and suggest that our approach to updatable data warehouses is quite tenable.

Acknowledgements We would like to thank David DeWitt, Michael Stonebraker, and Chen Xiao for their helpful feedback and ideas. This material is based upon work supported by the National Science Foundation under Grants 0520032, 0448124, and 0325525.

8. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, June 2006.

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

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

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

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

[6] P. A. Bernstein and N. Goodman. The failure and recovery problem for replicated databases. In *PODC*, pages 114–122. ACM Press, 1983.

[7] A. Bhide, A. Goyal, H.-I. Hsiao, and A. Jhingran. An efficient scheme for providing high availability. In *SIGMOD*, pages 236–245. ACM Press, 1992.

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

[9] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[10] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8. ACM Press, 1984.

[11] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, pages 150–162. ACM Press, Dec 1979.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1992.

[13] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group commit timers and high volume transaction systems. In *HPTS*, pages 301–329, 1989.

[14] S.-O. Hvasshovd. *HypRa/TR - A Tuple Oriented Recovery Method for a Continuously Available Distributed DBMS on a Shared Nothing Multi-Computer*. PhD thesis, Norwegian University of Science and Technology, 1992.

[15] S.-O. Hvasshovd, Ø. Torbjørnsen, S. E. Bratsberg, and P. Holager. The clustra telecom database: High availability, high throughput, and real-time response. In *VLDB*, pages 469–477. ACM Press, 1995.

[16] R. Jiménez-Peris, M. Patino-Martínez, and G. Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In *SRDS*, 2002.

[17] R. Jiménez-Peris, M. Patino-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM TODS*, 28(3):257–294, 2003.

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

[19] E. Lau. HARBOR: An integrated approach to recovery and high availability in an updatable, distributed data warehouse. MEng thesis, MIT, Cambridge, MA, 2006.

[20] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In *SOSP*, pages 226–238. ACM Press, 1991.

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

[22] 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`.

[23] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.

[24] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM TODS*, 11(4):378–396, 1986.

[25] 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`.

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

[27] P. Russom. Strategies and Sybase solutions for database availability, Nov 2001. `http://www.sybase.com/content/1016063/sybase.pdf`.

[28] J. H. Saltzer and M. F. Kaashoek. Topics in the engineering of computer systems. MIT 6.033 class notes, draft release 3.0, Feb. 2006.

[29] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34. ACM Press, 1979.

[30] D. Skeen. Nonblocking commit protocols. In *SIGMOD*, pages 133–142. ACM Press, 1981.

[31] D. Skeen. *Crash recovery in a distributed database system*. PhD thesis, UC Berkeley, May 1982.

[32] A. Snoeren, D. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *USITS*, 2001.

[33] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564. ACM, 2005.

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