Fast Crash Recovery for a Distributed Column-Store Database Management System

Master of Engineering Thesis Proposal

Edmond Lau

May 16, 2005

1 Introduction

A traditional database management system (DMBS) supports both atomic transactions and crash recovery by maintaining an undo/redo log. 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 perform recovery. After a computer failure, a rebooted system can process the log with a crash recovery algorithm, such as ARIES [1], to redo committed transactions and undo uncommitted ones, thereby restoring the system to a consistent state.

This log-based approach to recovery works well in environments that either interact with a small data set on the order of a few gigabytes or that do not demand high recovery performance. In database warehouse applications, however, analysts require highly available and high performance databases in order to execute ad-hoc queries on possibly tens of terabytes of data to extract patterns and to build business intelligence. Standard log-based crash recovery algorithms on these substantial data sets can take days or weeks

Thesis Co-Advisors: Assistant Professor Samuel Madden and Adjunct Professor Michael Stonebraker.
to complete. The use of periodic checkpoints can mitigate the cost but only partially reduces the recovery time and only in the case of non-catastrophic failures where the disk is preserved. Fast recovery of large data sets that can withstand catastrophic failures thus demands a better solution.

For my Master of Engineering thesis, I therefore propose to design, implement, and evaluate fast crash recovery algorithms in the context of C-Store [2], a distributed and column-oriented DBMS geared toward providing high performance and high availability for ad-hoc queries running on terabytes of data. Under a typical C-Store installation consisting of 1000 machines, each with a 100-GB commodity disk with a 200,000 hour disk life, a disk failure will occur approximately every 200 hours; moreover, software problems will further decrease the mean time to failure at a given site. Site failures are therefore frequent phenomena, and efficient recovery procedures that complete in significantly less time than the mean time to failure become crucial for normal functionality.

Unlike traditional DBMSs that employ undo/redo logs for recovery, C-Store maintains an undo log only to support transactional aborts and relies on redundantly stored data at other sites for crash recovery purposes. The crucial insight behind C-Store’s approach to recovery is that for large data sets, crash recovery via copying data over the network, which runs in time proportional to the amount of copied data, should outperform log processing, which not only requires bringing the log up-to-date, but also incurs a time-intensive overhead proportional to the length of the log.

The concept of recovering crashed sites from replicas is not new. To cope with the downtime associated with log processing, commercial data warehouses that require high availability, such as Walmart’s [3] and ones based on Oracle [4], typically run on replicated databases that mirror identical copies of their data at multiple sites. The replication enables the databases to continue servicing query requests despite a site failure. Crash recovery in a replicated database involves state synchronization of the failed site with a live, identical replica in conjunction with standard recovery processing of the log. The redundancy, however, comes at a high cost; the main utility of the redundant copies surfaces only during crash recovery, and expensive log processing is still required to revive a crashed site.

The challenge in C-Store’s recovery problem arises because C-Store stores data redundantly but on non-identical replicas. Instead, C-Store represents
relational tables as sets of columns, called \textit{projections}, and the projections are each stored in different sort orders, horizontally partitioned among multiple distributed sites, and compressed using schemes that depend on the properties of the particular projection. Projections that overlap in some subset of their columns achieve data redundancy, and the query optimizer can select among the different sort orders for the best set of projections to efficiently answer a given query during normal operation. Unlike other DBMSs, C-Store’s redundancy scheme therefore offers both improved reliability and better failure-free performance. Crash recovery, however, becomes more difficult because identical replicas no longer exist for state synchronization during recovery; instead, we must now rebuild projections of crashed sites from redundant data distributed among non-identical replicas in potentially different sort orders and in different compression schemes. Moreover, because the query optimizer depends on the various sort orders to efficiently answer queries, crash recovery of the failed site must be fast.

2 C-Store Architecture

A detailed discussion of the C-Store recovery problem requires some familiarity with C-Store’s overall design goals, high-level system architecture, physical data model, and snapshot isolation mechanism for read-only transactions. In this section, I summarize enough of these ideas to lay the groundwork for a meaningful discussion on recovery.

2.1 Design Goals

In data warehouse applications, transaction systems periodically load new data into historical storage, and analysts execute ad-hoc queries on possibly tens of terabytes of accumulated data in order to gain insight into data patterns. In these environments, DBMSs with row-store architectures that store all fields of a record, or \textit{row}, contiguously on disk have been shown to perform over an order of magnitude more slowly than DBMSs with column-store architectures that store related fields, or \textit{columns}, of different records together. Intuitively, row stores are write-optimized in that they enable the query executor to write all fields of a record to disk with a single disk \textit{I/O}, whereas column-stores are read-optimized in that they enable the query
executor to selectively read only those fields related to a particular query from disk. Existing column-store architectural designs, however, suffer from poor update performance because updates of a record require multiple disk I/O operations to access the non-contiguously stored fields.

C-Store is a distributed column-store DBMS designed to execute ad-hoc queries over an order of magnitude faster than major commercial DBMSs while simultaneously supporting reasonable performance on update queries. To meet this goal, it employs a hybrid column-oriented architecture, illustrated in Figure 1, consisting of a small writable store (WS) component designed to support high performance inserts and updates, a large read-optimized store (RS) designed to support fast read access to large data sets, and a tuple mover that periodically merges batches of tuples from the WS into the RS.

2.2 Physical Data Model

C-Store supports the standard relational logical data model of a database consisting of a collection of tables, and a table consisting of a collection of attributes. Unlike conventional databases, however, C-Store does not physically store the tables directly, but instead stores sets of projections. Each projection is anchored on a specific table and consists of a set of columns from the anchor table along with any number of additional columns from other tables, subject to some constraints. Tuples in a projection are stored

Figure 1: Diagram of C-Store System Architecture.
column-wise, and each column is sorted according to the projection’s sort key, which can be any column or columns from the projection. To reconstruct a table from the various projections anchored on it, C-Store maintains join indices to connect a tuple from one projection to the corresponding tuple in another.

Each projection is further horizontally partitioned into segments according to ranges of sort key values. For each key range, C-Store stores a WS segment and a RS segment containing all tuples whose sort key falls within that range. The WS segment holds recently updated data and is designed to be main-memory resident and quick to update, while the RS segment provides fast read access to the rest of the data. Each segment associates a storage key with every tuple it contains. Storage keys in the RS are numbered sequentially starting from 1 and inferred from a tuple’s storage position in the segment; storage keys in WS are physically stored and are larger than any storage key in the corresponding.

Every projection is therefore represented as multiple pairs of WS and RS segments, one pair for each key range. Corresponding WS and RS segments are collocated, but segments may otherwise be stored at different sites.

### 2.3 Snapshot Isolation

Because the expected workload of C-Store consists of large numbers of ad-hoc queries that read large data sets, a conventional locking mechanism would generate substantial lock contention and lead to poor performance. C-Store therefore supports a form of read-only transactions based on snapshot isolation. Snapshot isolation enables a read-only transaction to time travel and access the database as of some recent time in the past, before which we can guarantee that there are no pending transactions. Because snapshot isolation reads only committed data, no locks are required for serializing the transaction.

The most recent time in the past for which snapshot isolation can run is the high water mark (HWM), and the timestamp authority oversees a mechanism to advance it. Because supporting general time travel would require an expensive time and space budget, the low water mark (LWM) constrains the earliest effective time that a snapshot-isolated query can run.
Answering a snapshot-isolated query requires determining the set of WS and RS records that should be visible at an effective time $T$. A record is visible to the read-only transaction if it was inserted before $T$ and either not deleted or deleted after $T$. Updates are transformed into a delete and an insert in order to fit the semantics of this model. To determine visibility, C-Store introduces coarse granularity timestamps called epochs. Each WS segment has an insertion vector (IV) that records the insertion epoch of each record in the segment, and we ensure that all records in the RS were inserted before the LWM. In addition, each WS and RS segment has an associated deletion record vector (DRV) that records the deletion epoch of each record or a 0 if the record has not been deleted. The IV and DRV are both stored in the in-memory WS to support fast updates.

3 Proposed Work and Current Progress

Having discussed an overview of the C-Store system architecture, I can now elaborate upon my proposed thesis work in more detail. I propose to design fast crash recovery schemes in the context of C-Store, implement those schemes on the existing C-Store research prototype, and evaluate their performance. The algorithms will form a toolkit of recovery protocols that could be applied during different failure scenarios, and they must be capable of recovering, for a failed site, all WS and RS data in the correct sort order, the insertion vector, the deletion record vector, the WS storage keys, and any B-tree indices on these data objects.

Ultimately, I would like to generalize my results beyond C-Store and derive from my work general conclusions regarding crash recovery algorithms, such as an evaluation of the applicability of various recovery approaches, a study of desirable properties of recovery algorithms, or an analysis of when state synchronization from remote replicas outperforms log processing in the context of crash recovery.

In the rest of this section, I discuss some preliminary thoughts on recovery and my current progress. In particular, I specify the C-Store recovery problem (3.1), discuss the applicability of undo logging (3.2), argue that recording WS checkpoints in C-Store is easier than in other DBMSs and can improve recovery performance (3.3), and outline a first stab at a query-based strategy for recovering from catastrophic failure (3.4). Finally, I sketch a projected
3. Research plan for my thesis work (3.5).

3.1 Recovery Problem Specification

Crash recovery of failed sites based on copying data over the network from live remote sites relies on C-Store’s $K$-safety property, which guarantees that there are at least $K$ instances of each piece of data distributed among the sites. Site failures fall into three different categories:

1. **Catastrophic Failure.** Both the WS and the RS may be destroyed by a catastrophic failure, such as a hurricane or a fire, and require complete rebuilding from other sites.

2. **Crash Failure.** The on-disk RS may remain intact but the in-memory WS may become damaged, which can occur if the operating system crashes, the file system crashes, or a power failure occurs. The RS should tend to be safe from damage because it is only written in a constrained fashion by the tuple mover and because the merge-out process writes a new shadow copy of the RS and atomically switches over to the new copy.

3. **Transient Failure.** A site may fail and not suffer any data loss, as in the case of network partition. Under this scenario, the RS and WS of the site are intact but may be missing the most recent updates.

Apart from fast algorithmic runtime, I further specify the following two design goals for crash recovery algorithms:

1. **Minimize network communication.** Recovery time correlates directly with the amount of data transfer required. Moreover, because the other C-Store sites will still be servicing queries, network overload would tend to decrease query performance.

2. **Impose minimal constraints on the rest of the system.** Less constraints on system design translates into additional flexibility for optimizing query performance elsewhere in the system.
3.2 Applicability of Undo Logging

The only logging infrastructure that C-Store needs to support is an undo log used solely for the purpose of aborting any outstanding transactions. Undo records need only be logged for update transactions and not for any read-only transactions.

Two proposals have been developed thus far for handling the log: one based on the view that any recovery log processing should be avoided and the other based on the view that log processing may be an effective method to restore a consistent snapshot from which to begin recovering the WS from remote sites.

In the first proposal, we take the approach of maintaining the undo log only in memory. Intuitively, because the log is not used for crash recovery, it should not need to be written to stable storage. We enforce the write-ahead logging protocol when writing log records to memory (i.e., before updating a page in memory, the appropriate undo log record must be written to the in-memory log) in order to support transactional aborts but allow modified changes to be written to disk at any time.

After a crash that damages the WS, the in-memory log will be unavailable and a partially intact WS on disk may contain data from uncommitted transactions. We therefore scrap the WS and focus on restoring the WS from other remote sites rather than on salvaging any WS remnants that may have been preserved on disk. This proposal offers the benefit that no log records need to be forced to disk, thereby preserving the performance benefits of an in-memory WS. Log records for a transaction can also be rapidly garbage-collected from memory as soon as the transaction commits or aborts. This approach rests on the assumption that log-based recovery of the WS to a consistent state may be slow and introduces additional complexity.

In the second proposal, we adopt a more traditional recovery scheme that requires enforcing the write-ahead logging protocol when writing modified pages to disk. After a crash that damages the WS, uncommitted changes would then be rolled back using the undo log to bring any WS data salvaged from disk to a consistent state, after which, the same state synchronization game could be played. The assumption would be that periodic flushing of log records incurs minimal cost, but that the restoration of a consistent snapshot could substantially improve recovery performance. Morever, because
we expect the WS to be mostly main memory resident, the need to force log records to disk prior to writing modified pages should be relatively low. Further analysis will be required to determine the more effective proposal.

### 3.3 WS Checkpoints to Bootstrap Recovery

Both of the aforementioned proposals may be complemented by periodically recording checkpoints with snapshots of all committed WS data to disk and using the snapshot as a starting point for crash recovery. In traditional databases, checkpoints typically contain uncommitted data as well as committed data unless the system can be quiesced during the checkpointing process. C-Store’s snapshot isolation mechanism, however, enables checkpointing of only the committed data in the WS without quiescing the system. In particular, the timestamp authority advances the HWM to the next epoch only when all transactions in the previous epoch have completed; thus, transactions in C-Store do not cross epoch boundaries. At an epoch \( E \), all transactions that executed prior to \( E \) have been completed, and snapshot isolation can thus be used to obtain a consistent snapshot of the WS as of any time between LWM and HWM. WS recovery using a WS checkpoint as a starting point may substantially reduce recovery time; arguably, the ease of checkpoints downplays the advantages of the second logging proposal.

### 3.4 A General Query-based Recovery Strategy

For my thesis, I plan to develop a collection of strategies to handle the three failure scenarios specified in Section 3.1. In this section, I present a preliminary sketch of a query-based strategy for crash recovery from a catastrophic failure that destroys both the RS and WS. The other two failure scenarios require recovering a subset of the combined RS and WS data, and a strategy for the general case can be refined and adapted to handle the other two.

In a query-based approach, I attempt to think of recovery in terms of snapshot-isolated queries and rely on the query optimizer to decide on an efficient query plan and on the query executor to answer queries correctly. The best that this approach can accomplish is to bring the recovering site up-to-date as of the HWM, which we can view as reducing the problem to
the third failure scenario of missing updates.

To recover a particular RS segment $RS_i$, the recovering site issues a snapshot-isolated query of the following form as of the effective time HWM:

```
SELECT fields-of-RS_i, deletion-epoch
FROM tables-of-RS_i
WHERE join-conditions-of-RS_i AND sort-key in Ki
    AND insertion-epoch < LWM
    AND deletion-epoch = 0
ORDER BY fields-of-Ki
```

To restore the corresponding WS segment $WS_i$, the recovering site issues another snapshot-isolated query as of the effective time HWM:

```
SELECT fields-of-WS_i, storage-key, insertion-epoch, deletion-epoch
FROM tables-of-WS_i
WHERE join-conditions-of-WS_i AND sort-key in Ki
    AND insertion-epoch >= LWM AND insertion-epoch <= HWM
    AND (deletion-epoch = 0 OR deletion-epoch >= LWM)
ORDER BY storage-key
```

The predicates on `insertion-epoch` guarantee that all desired records inserted before the LWM become stored in $RS_i$, and those inserted between LWM and HWM, inclusive, become stored in $WS_i$. The predicates on the `deletion-epoch` guarantee that the recovering site learns about all existing records as well as records deleted at or after the LWM; the deleted ones will be necessary for the site to support snapshot-isolated queries after recovery completes.

The strategy allows the HWM to advance from the time the first query is run to the second query without impacting correctness; the queries will restore $WS_i$ and $RS_i$ to the HWM as of the second query. The LWM is not allowed to advance unless a) the LWM value in the two queries are bound to the same value, b) the tuple mover does not operate on the segments related to the queries after the LWM advances, and c) the query executor allows recovery queries to run with slightly outdated LWM values. The second constraint prevents tuples from appearing in both the $RS_i$ and $WS_i$ segments, or in neither, depending on the order in which the two queries are executed.
The problem remains then to reapply any missing updates and to bring a recovering site $r$ up-to-date. One proposal would be to have some other site $s$ keep a queue of committed updates for $r$ after the system detects that $r$ has failed. When $r$ first comes online, it begins receiving new updates from transaction managers but cannot apply those updates until it has verified that its locally stored segments are up-to-date. The site $r$ therefore buffers the updates as they arrive into its own update queue. Write transactions involving $r$ cannot proceed until $r$ responds; otherwise, update requests may be received and applied out of order at $r$ and violate transaction serialization.

To bring its own locally stored segments up-to-date, $r$ requests from $s$ all of its queued updates and applies them. The site $r$ then starts responding to all the buffered updates in its own update queue until the queue shrinks to zero, at which point it is fully recovered.

More research needs to be done on the restoration of missing updates in order to handle situations where the update queues grow too large or where transactions become blocked for too long of a time.

### 3.5 Research Plan

At this juncture, I have written up a preliminary sketch of possible avenues for crash recovery. This month, I plan to revise this sketch and present the ideas regarding recovery for discussion with other members of the C-Store team. By the end of May, I aim to scope out with Sam and Mike a target set of goals and stretch goals for my thesis work. The goals should be accomplishable within a one-year timeframe and should be flexible enough to take into account the projected ability of the C-Store research prototype to function as a code base for evaluating any recovery schemes that I develop.

Upon returning in early September, I plan to undergo a series of iterations of rigorously reasoning about and designing recovery algorithms, presenting the designs at C-Store team meetings, and incorporating any feedback received. Meanwhile, I also intend to start developing any infrastructure in the C-Store prototype that will prove necessary to eventually evaluate my designs. My goal by the end of the fall is to have developed a concrete vision of what I plan to implement and evaluate in the spring.

Over IAP 2006, I intend to begin implementation of my recovery designs.
In the spring, I plan to evaluate the implementation and write up the results in the form of my Master of Engineering thesis.

4 Related Work

The bulk of the literature on database crash recovery centers around log processing. The gold standard for database crash recovery is the ARIES [1] recovery algorithm, implemented first in varying degrees on IBM's OS/2, DB2, Quicksilver, and a few other systems. Based on write-ahead logging of undo and redo log records, ARIES employs a repeating history paradigm to restore a crashed system to its state directly prior to the crash and then subsequently rolls back all uncommitted transactions.

High performance crash recovery has been explored in a number of database systems. Oracle’s Fast-Start [6] enables a database administrator to impose bounds on recovery time from unplanned power outages by using incremental checkpoints and by adjusting the frequency with which the checkpoint on the log is advanced. Its Data Guard [5] manages standby databases to handle failovers. In the realm of main memory database systems, which may be relevant to C-Store’s in-memory WS, [7] explores the use of an update-frequency partition checkpoint scheme to speed recovery and [8] examines the separation of recovery into a high-speed phase for data immediately required by pending transactions and a slower phase for all remaining data.

More theoretical papers on crash recovery techniques have also been written. Bernstein and Goodman [10, 11] provide an abstract treatment of techniques to serialize operations on replicated databases despite failures and recoveries but assume the existence of recovery procedures to restore a recovered site. Verhofstad [12] provides a general survey of recovery techniques for database systems. Choy et al. [9] survey existing recovery schemes and products and offer a classification of different failures and recovery techniques.

The use of replicas to provide availability and reliability has also been explored outside of the database realm in storage systems and file systems. Petal [13] is a distributed block-level storage system that redundantly stores each data block in a pair of neighboring block servers. The system offers a feature to record a snapshot of the virtual disk that can be subsequently
restored but does not provide a solution to restore the system to the state at the time of a crash. FAB [14] is a distributed disk array that replicates commodity storage bricks and handles brick failure and crash recovery through voting. A state synchronization algorithm dynamically rebalances redundant data to new bricks that are added and to other live bricks when one fails.

5 Conclusion

Log processing techniques for crash recovery on large terabyte data sets take days to complete and do not meet the needs of high performance and high availability data warehouses. These environments require fast crash recovery algorithms that minimize system downtime. I plan to confront this challenge for my Master of Engineering thesis and have proposed to design, implement, and evaluate fast recovery algorithms in the context of a distributed column-store DBMS called C-Store. C-Store’s scheme for redundant data at non-identical replicas in different sort orders, different horizontal partitions, and different compression schemes makes crash recovery a challenging problem.

References


