A Snapshot Utility for a Distributed Object-Oriented Database System

Chuang-Hue Moh
MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139
chmoh@lcs.mit.edu
August 14, 2002

Abstract

Thor is an object-oriented database management system that is designed to be scalable in a distributed environment. Thor allows objects to be stored persistently and accessed efficiently while providing ACID transaction semantics. In this paper, we propose to extend Thor with a snapshot utility. Snapshots are useful for a wide range of applications, including data analysis, data mining and database recovery. In order to provide high performance in Thor, it is essential that the snapshot utility be efficient in both the archiving and the retrieval of snapshot so that interference with concurrent transactions in the system is minimized. We propose a lazy gossip-based approach to taking snapshots efficiently. In this approach snapshots are propagated throughout the system via a lazy pairwise exchange of gossip messages between servers. The propagation of messages is either done in the background or piggybacked on other communication messages so that they are non-intrusive to concurrent transactions. In addition, we propose to create snapshots in an incremental and lazy manner to further improve the efficiency and to minimize the space requirements of storing the snapshots. Our approach is novel in that it makes use of the server’s write buffer to allow the lazy propagation and creation of snapshots.

In our system, the data pages of a snapshot are automatically archived after they have been created. These pages are later retrieved on-demand when the user runs a transaction on a snapshot of the database. The archival and retrieval of snapshots is completely transparent to the user. The retrieved data pages of snapshots are installed into the client’s persistent cache. We extend Thor’s cache management mechanism to support the caching of multiple versions of the database at the same time, and also to allow compaction of the database across these versions without cache contamination.

Finally, we propose a peer-to-peer infrastructure for the underlying storage facility required for archiving the snapshots. Peer-to-peer systems have the desirable qualities of high availability, fault-tolerance, automatic load-balancing and self-configuration. More importantly, the cooperative storage paradigm in peer-to-peer systems allows us to utilize the otherwise unused storage in the participating storage servers. We extend the DHash layer in the Cooperative File System for archiving data pages of snapshots by providing persistent storage and adding a self-repair mechanism to improve data survivability.

1 Introduction

Thor [21, 20] is a scalable, distributed object-oriented database management system (OODBMS) that incorporates many novel high-performance approaches to cache management [6, 13], concurrency control [2] and garbage collection [24]. We extend Thor to provide a snapshot utility in a distributed database environment.

Snapshots are transaction-consistent copies of the database [1]. They are essential for many database applications, such as data analysis, data mining and database recovery. Our recent work on software upgrades in a OODBMS [23] could also make use of snapshots to perform the upgrades. In this paper, we propose to extend Thor with a snapshot utility for creating and retrieving snapshots. Snapshots are automatically archived after they are taken, and retrieved from the archival store as
and when they are needed. The archival and retrieval of snapshots are completely transparent to the users.

The efficiency of the snapshot utility affects the performance of the entire database system. As performance is a key requirement for our system, the main challenge of providing this snapshot utility is how to efficiently archive and retrieve snapshots with minimum interference with concurrent transactions in the system. We propose to use a lazy, gossip-based approach [12, 22] to propagate snapshots throughout the system. In addition, snapshots are created lazily using a copy-on-write scheme i.e., the data pages of a snapshot are created just before they are modified. Snapshots are archived automatically after they have been created and retrieved on-demand when the user runs a transaction on a snapshot of the database. The archival and retrieval process is completely transparent to the user.

As the client in Thor also caches a copy of the data that it uses, we propose to extend this caching scheme to accommodate multiple versions and to allow compaction of data pages while avoid cache contamination. Supporting multiple versions and compaction of data pages across these versions is essential to prevent the performance of the system from degrading due to thrashing and inefficient cache management.

Snapshots are long-lived data, so even though they are created incrementally and lazily, the archival store for these snapshots requires a potentially large amount of storage. We propose to use a peer-to-peer (P2P) storage system for the archival store. As demonstrated in recent research [35, 10, 33, 32, 18, 5], P2P systems exhibit the qualities of high availability, fault-tolerance, automatic load-balancing and self-configuration. More importantly, the cooperative storage paradigm in P2P systems allows us to utilize the otherwise unused storage in the participating storage servers. We implement our archival store for the snapshot pages by extending the DHash layer of the Cooperative File System (CFS) [10].

The rest of the paper is organized as follows. We provide some essential background details on Thor and CFS in Section 2. Section 3 discusses our design for the snapshot utility and the archival store. Section 4 presents some related work in the fields of distributed and cooperative storage systems, database archival and network file storage solutions. Finally, we conclude our proposal in Section 5 and discuss some of the future work and the current status of this work.

2 Background

2.1 Thor Distributed Object-Oriented Database System

Thor is implemented on a client-server architecture and applications interact with Thor by calling methods on objects; these calls occur within atomic transactions. The client part of Thor runs the Front End (FE) while the server part runs the Object Repository (OR). A two-phase commit protocol (2PC) is used when a transaction spans more than one server. A language-independent type system is also provided within Thor to support heterogeneity of programming languages.

2.1.1 Front End (FE)

The disk storage on the ORs is organized in fixed-size pages. FEs fetch data from the ORs in pages, and an in-memory page cache is maintained in every OR to speed up client fetch requests. As an optimization, the FEs caches copies of persistent objects and communicate with the ORs to fetch pages and commit modifications to objects. The FE implements the the Hybrid Adaptive Caching (HAC) [6] scheme for managing its cache; the essence of the HAC scheme is that it allows FEs to flexibly switch between a object caching scheme and a page caching scheme. This flexibility allows the HAC scheme to combine the advantages of both these schemes while avoiding their drawbacks.

In the HAC scheme, the data is fetched from the OR in pages but cache replacement is done at the granularity of objects. When an update request at the FE commits, only the modified objects are sent to the OR. The FE’s cache is partitioned into page-sized frames. Frames can be either intact or compacted. Intact frames holds pages fetched from the OR while compacted pages hold objects that were retained when their containing pages are compacted. When a disk page is fetched and some of the objects that it contains is already cached in a page frame, the copy of the object in the page frame will be used. The objects are installed into the page only when the frame is freed.
The FE performs *pointer swizzling* i.e., replaces the object references by virtual memory pointers to speed up pointer traversals. HAC uses *indirect pointer swizzling* where the object reference is translated to a pointer into an *indirection table* entry. The entry of this table in turn, points to the target object. The indirection table is called the *Resident Object Table (ROT)* and this indirection reduces overhead when evicting objects from the cache. All references in an object is swizzled at the same time and if the object references another object that is not in the client’s persistent cache, a *surrogate* is created in its place. The pointer is then swizzled to point to the surrogate. A surrogate is a small object containing the location-dependent reference to the missing object.

When a persistent object is modified, it needs to be sent to the OR during the transaction commit phase; the modified object needs to be unswizzled before they are sent. In addition, during unswizzling process, volatile objects that are being made persistent by the transactions may be discovered. These objects are sent over to the OR as well. When the transaction commits at the OR, it sends *invalidation messages* to other FEs in the system that have cached copies of the objects modified in the transaction. These FEs will then discard the invalidated objects by turning them into surrogates. If a current transaction at the FE is using a modified object when it receives an invalidation message for the same object, the transaction is aborted.

2.1.2 Object Repository (OR)

To reduce the write performance degradation due to object shipping, the OR implements a *Modified Object Buffer (MOB)* [13] in memory. Objects that are shipped back to the OR are not written back to its disk immediately but instead resides in the MOB until the background *flusher* thread flushes them to disk. When a page to be modified is not present in the OR’s memory, an *installation read* is performed to bring the page into memory before the OR can modify the page and write it back to disk.

The MOB improves write performance by allowing the OR to delay *installation reads* until the contents of the MOB have to be flushed to disk. Delaying installation reads allow for efficient disk scheduling and improves system performance [28]. In addition, the MOB makes it possible to batch modifications to a single page from different transactions into a single modification using the MOB. The MOB further improves performance by allowing *write absorptions* i.e., i.e., if an object is modified multiple times in the MOB, only the latest modification needs to be written out to disk.

Using the MOB does not compromise the correctness of the system since modifications are inserted into the MOB as soon as the commit has been recorded into the persistent transaction log in the OR. Only the transaction log is required for correct operation - the MOB can be reconstructed from the transaction log if the OR crashes. As objects are removed from the MOB and flushed to disk, the corresponding records are also removed from the log.

The batching of multiple updates to a single page into a single update may potentially create holes in the log representation of the MOB. Consequently, the MOB uses a *scan list* data structure to find the set of pages that needs to be modified so that the MOB contents can be discarded in log order. The log is scanned in log order when updating the scan list.

The MOB also contains an *object table*, which is a hash table that enables the OR to find objects in the MOB given an object identifier or a page identifier. The purpose of the object table is to satisfy client fetch requests for newly modified objects that have not been written out to disk yet. It is also used by the flusher thread to identify the set of objects belonging to a particular page that is to be written to disk.

Disk writes are performed in segments, which are larger than the size of pages. The current implementation has a page size of 8 Kbytes and a segment size of 4 pages, which corresponds to 32 Kbytes of contiguous locations on the disk.

2.2 Chord Peer-to-Peer Routing Protocol & Cooperative File System

Chord is a P2P routing and location protocol provides one service to the user: given a key, it returns the node that is responsible for the key. The Chord protocol also specifies
how new nodes join the system and how to handle the failure of existing nodes.

Central to Chord is the use of a consistent hashing mapping of keys to nodes. Consistent hashing balances load and minimizes key movements when nodes join or leave the system. Each node in Chord is assigned an identifier using the SHA-1 secure hash function of its IP address. The nodes are arranged into a logical ring topology ordered by this identifier. Keys in Chord are also assigned identifiers in the same manner i.e., the SHA-1 of the key itself.

Chord uses two data structures for key location, namely the **successor list** and the **finger table**. The successor list of a node contains the identifiers of its next immediate successors in the Chord ring. This suffices for correct operation of the protocol but it is inefficient. The finger table is used to optimize this protocol. Assuming that there are \( m \) bits in the identifier, the finger table will contain \( m \) entries. The \( i^{th} \) entry of the finger table will contain the identity of the first node that succeeds the current node by at least \( 2^{i-1} \) on the identifier ring. In addition to the successor list and the finger table, each node in Chord also maintains a **predecessor** variable to simplify the joining and leaving/failure of nodes.

**CFS** is a P2P read-only file system that is built on top of the Chord protocol. Only the publisher is allowed to update the file system stored in CFS. In addition, CFS stores data for an agree-upon finite time interval; upon expiration of this time interval, CFS no longer guarantees persistent storage for the file. Expired files stored by a node can be deleted when its storage space runs low.

CFS segments files into fixed-sized blocks for storage. Storing blocks instead of whole files has the advantage of better utilization of storage space. Each block is assigned an identifier, which is the hash of the block’s content. The node whose identifier immediately succeeds the block’s identifier will be responsible for storing the block.

DHash replicates each block on the \( k \) CFS servers immediately after the block’s successor on the Chord ring. Replicas can be easily identified by the server’s successor list, where \( r \geq k \). Since nodes that are close in the identifier space are not necessarily close physically, servers managing the replicas can be assumed to have independent failures.

To avoid overloading servers that hold popular data, DHash also caches data blocks along the lookup path. The effectiveness of DHash’s caching mechanism is based on the fact that a Chord lookup takes progressively shorter hops in the identifier space as it gets closer to the target. Cache blocks are stored in a different storage area than regular file blocks and are managed in a LRU fashion.

### 3 System Design

The main design goal of our system is to provide an efficient snapshot utility in Thor. The snapshot execution should be efficient and impose minimal interference to concurrent transactions in the system. In addition, the underlying archival store should guarantee survivability and high availability of the snapshots. Our design provides only read-only snapshots - supporting updatable snapshots is an area of future work.

When a snapshot is taken by the user at the FE, the state of the entire database is captured at the instance of the snapshot. Ideally, all the ORs in the system will learn of the snapshot at the time the snapshot was taken. To achieve this, system needs to run a transaction involving every OR in the system. Such a transaction can be very expensive and does not scale well with the number of ORs. Furthermore, a transaction involving every OR would mean that the entire system needs to wait until the snapshot completes before it can proceed with other transactions. This *stop-the-world* approach would adversely affect the performance of our system.

Instead of using a transactional approach to create snapshots, snapshots are considered to be taken once it is registered at a centralized snapshot coordinator. The snapshot is then propagated lazily throughout the entire system. The implication of this is that every OR will need to maintain an undo log for the recent modifications so that modifications to the data pages on its disk can be undone to produce pages required for the snapshot when the snapshot message eventually arrives at the OR.

Snapshots are also created lazily i.e., the data pages for the snapshots are created just before they are modified.
by the first update transaction on the page that occurred after the snapshot. The fundamental idea is that the original page from the disk is saved as the data page for the snapshot before applying the modifications. Creating snapshots lazily minimizes the space requirements for archiving pages of the snapshot since we do not need to create a copy of the entire database on every snapshot - only pages that were modified since the last snapshot needs to be copied. This approach is similar to incremental dumping mechanisms in databases and copy-on-write snapshot schemes in file systems [15, 16, 30].

As update transactions in Thor are buffered in the in-memory MOB and written out to disk by a background flusher thread, the pages of a snapshot are created and archived as part of the background process. This also helps to reduce interference on concurrent transactions in the system.

Pages of a snapshot are archived in a P2P archival store for long-term storage. Archiving pages into the P2P archival store involves sending the pages and ensuring that these pages are archived without any errors. Such a process can take a relatively long time and hence should be done asynchronously in the background. Consequently, pages to be archived are stored into an on-disk buffer before the are archived into the P2P archival store.

3.1 Correctness Criteria

In a distributed system like Thor, there is no notion of a global time, therefore it is impossible to have perfect time synchronization among the nodes in the system. Current time synchronization technologies like the Network Time Protocol (NTP) [25, 26] allows computers on a network to be synchronized within 1 to 50 milliseconds of each other. Such a loosely synchronized clock is also assumed in Thor. Without loss of generality, we assume that the clock skew between any two ORs is \( \pm \epsilon \) (milliseconds).

We assume the following correctness criteria for taking a snapshot in Thor:

- A snapshot taken a time \( t \) reflects all the transactions that ran no later than \( \epsilon \) milliseconds before the snapshot was taken.
- The snapshot may also reflect transaction that ran as late as \( \epsilon \) milliseconds after the snapshot was taken.
- The snapshot however, does not reflect transactions that occurred \( \epsilon \) seconds after the snapshot was taken.
- A snapshot must respect the atomicity of transactions i.e., it either reflects the effects if a transaction or it does not reflect the transaction at all.

The last criterion, in particular, refers to transactions that involves multiple ORs. The snapshot should either reflect the transaction in all the ORs or none of the ORs. It cannot reflect the transaction in only some of the ORs involved.

3.2 Snapshot Naming

The name of a snapshot uniquely identifies it within the archival system. The naming convention for snapshots should be such that the data pages of snapshots can be stored and retrieved efficiently and correctly. One intuitive way of naming snapshots is to use system-generated version numbers such as in Lotus Notes. These numbers are monotonically increasing and consecutive snapshots will be assigned consecutive version numbers. Version numbers allows the system to detect inconsistencies in the snapshots. For example, if an OR receives snapshot messages for versions 5 and 7, it knows that snapshot version 6 is missing. In this way, external consistency of the snapshots is maintained.

The main drawback of using system generated version numbers is that it is not intuitive for the users when retrieving snapshots. For example, if a user wants to retrieve a snapshot on 4th July, 2003 at 12 pm, he or she must remember that the corresponding snapshot version number for the snapshot taken at that time. An alternative to using version numbers is to use timestamps as snapshot names [34]. In this case, the user only needs to provide the time of the desired snapshot and the system will return the snapshot that is closest to the desired time. Using timestamps however, does not provide an OR with the ability to maintain external consistency. Suppose that an OR receives two snapshots with timestamps 104 and 106 respectively, it cannot conclude whether a snapshot was taken at time 105.

Although Thor assumes a loosely synchronized clock, clock skews on the order of tens of milliseconds can
still occur between computers in the system. This gives rise to another problem when using timestamps to name snapshots. For example, at time \( t \), a snapshot is taken and at time \( t + 1 \), ORs \( \text{OR}_x \) and \( \text{OR}_y \) execute an update transaction. Due to the clock skew of the system clock in \( \text{OR}_x \), its update transaction had a timestamp of \( t - 1 \). When the snapshot arrives at both \( \text{OR}_x \), it would appear that the snapshot was taken after the update transaction, whereas the snapshot is deemed to have happened after the update transaction in \( \text{OR}_y \). The problem becomes more tricky when the snapshot and update transaction are assigned the same timestamp. Our correctness criteria breaks down under these circumstances.

We avoid the first problem by using a single timestamp per transaction - in the case where the update transaction involves only a single OR, we make use of the timestamp of the OR; if the transaction involves more than one OR, the 2PC protocol coordinator will produce the timestamp for the transaction. In this way, we ensure that the snapshot is serialized with respect to the concurrent transactions in the system i.e., the snapshot can occur either before or after a transaction but never in the middle of a transaction. To address the problem of snapshots and transactions having the same timestamps, we define snapshots to have higher priority over transactions in that a snapshot always reflects a transaction that occurred at the same time.

We choose the rich semantics of using timestamps, as opposed to using system generated version numbers, to name snapshots. Our lazy snapshot propagation scheme includes a mechanism for efficiently maintaining external consistency in the system without using system generated version numbers. The basic idea is for the OR to keep track of the time which it possess complete information about snapshots and communicated the snapshot coordinator or its peers to update this information. We leave the discussion of the details of this mechanism to the next section.

### 3.3 Snapshot Propagation

A snapshot is a special system-wide transaction that requires the participation of all the ORs in the system. Executing such a transaction can take a long time and may block other concurrent transactions until it is completed. Thor uses a 2PC protocol for transactions involving more than one OR. Obviously, a 2PC transaction-based approach to taking snapshots is expensive and does not scale with the number of ORs in the system.

We propose a lazy snapshot propagation scheme to allow snapshots to execute efficiently. In this scheme, snapshots are administered by a centralized coordinator OR called the snapshot coordinator or \( \text{OR}_{ss} \). An FE requesting to take a snapshot of the database will send its request to the \( \text{OR}_{ss} \); and the \( \text{OR}_{ss} \) will determine the name of the snapshot using the time when the snapshot request was received. When the FE receives a reply from the \( \text{OR}_{ss} \), the snapshot transaction is considered to be complete. The snapshot is then propagated lazily to the entire system.

#### 3.3.1 Basic Centralized Snapshot Propagation

Our basic mechanism requires that each OR communicates with the \( \text{OR}_{ss} \) every \( k \) minutes to obtain updated snapshot information. Every OR in the system maintains a snapshot list, which is a list of all the timestamps of snapshots that the OR has heard of. In addition, each OR also maintains a timestamp, \( \text{TS}_{\text{curr}} \) that contains the most recent timestamp that the OR has heard of from the \( \text{OR}_{ss} \). The \( \text{TS}_{\text{curr}} \) determines how up-to-date is an OR’s snapshot list.

To obtain updated snapshot information from \( \text{OR}_{ss} \), an OR will simply send the value in \( \text{TS}_{\text{curr}} \) to the \( \text{OR}_{ss} \). In response, the \( \text{OR}_{ss} \) will reply with its current timestamp and a list of snapshots that were taken since \( \text{TS}_{\text{curr}} \). This information will allow the OR to update its snapshot list and \( \text{TS}_{\text{curr}} \) variable.

The snapshot list can be truncated when the information is not longer needed. An entry \( t_1 \) can be discarded from this list when there are not outstanding modifications in the OR’s write buffer with timestamps older than \( t_1 \). The main advantage of this basic centralized scheme is its simplicity and the low communication overhead involved.

#### 3.3.2 Gossip-based Snapshot Message Propagation

In the basic centralized scheme, every OR in the system will need to communicate periodically with the \( \text{OR}_{ss} \) to
obtain updated snapshot information. This can pose a potential bottleneck in the system. Furthermore, in the wide-area, communicating with the ORs frequently can incur expensive communication cost if an OR is located far away from the ORs.

To eliminate this bottleneck, we extend our basic centralized approach to snapshot propagation with a gossip-based message exchange propagation scheme. Gossip-based mechanisms, also refereed to as epidemic mechanisms, have been employed in a wide range of applications such as replicated database management [11, 3], garbage collection [19], concurrency control systems [37], network multicast [4, 36] and membership management [14].

Apart from the snapshot list and TS_curr, every OR also maintains a TS_discard variable, which contains the timestamp of the latest snapshot that it discarded from the snapshot list when the list was truncated. A snapshot is propagated via the pair-wise message exchange between two ORs, OR_send (sender) and OR_recv (receiver). The message propagation almost always propagates complete information about snapshots. Whenever OR_send communicates with OR_recv, it sends the message \(< OR_{send}.TS_{curr}, OR_{send}.TS_{discard}, OR_{send}.List_{ss} >\), where List_{ss} is the snapshot list.

OR_recv processes this message as follows:

- If \( OR_{send}.TS_{discard} > OR_{recv}.TS_{latest} \) (where \( TS_{latest} \) is the most recent snapshot in the snapshot list), then OR_send has truncated its snapshot list beyond the most recent snapshot that OR_recv knows. In this case, OR_recv will need to communicate with the ORs to obtain the latest update information as in the basic centralized scheme.

- Else if \( OR_{send}.TS_{curr} > OR_{recv}.TS_{curr} \) i.e., the latest snapshot that OR_send knows is more recent than that of OR_recv, then OR_recv will update \( OR_{recv}.TS_{curr} := OR_{send}.TS_{curr} \) and \( OR_{recv}.List_{ss} := OR_{recv}.List_{ss} \cup OR_{send}.List_{ss} \).

- Else if \( OR_{send}.TS_{curr} < OR_{recv}.TS_{curr} \) i.e., the latest snapshot that OR_recv knows is more recent than that of OR_send, then OR_recv will respond with the message \(< OR_{recv}.TS_{curr}, OR_{recv}.TS_{discard}, OR_{recv}.List_{ss} >\) to allow OR_send to update its snapshot information.

Finally, if \( OR_{send}.TS_{curr} = OR_{recv}.TS_{curr} \), then both the OR_recv and OR_send have the same snapshot information and no updates are required.

In this gossip-based scheme, there is a high probability that the message exchange will result in either one of the two ORs being updated since snapshots are propagated slowly. Furthermore, as snapshots are expected to happen relatively infrequently, the messages that need to be exchanged between the two ORs will be small.

### 3.4 Server Buffer Management

In our system, snapshots are propagated lazily throughout the system, and their data pages are created incrementally and lazily to optimize performance. An update transaction can therefore race ahead of snapshot that was taken before the update transaction. Consequently, the ORs may not have complete information about snapshots in the system when they commit update transactions to disk. As a result, the states of the data objects before update transactions must be saved until the OR can ascertain whether the data objects need to be included as part of a snapshot that occurred before the update transaction.

#### 3.4.1 Anti-MOB Undo Log

We make use of the Anti-MOB for the purpose of saving the pre-modification states of data objects when they are modified. The Anti-MOB is a log-based buffer that is similar to the MOB. Unlike the MOB however, this is an undo log rather than a redo log, hence the name Anti-MOB. The Anti-MOB saves the state of data objects before they are modified so that the OR can later undo the modifications from a page on the disk to restore it to the state just before the snapshot.

The MOB improves performance of the system by allowing installation read to be performed in a background flusher thread. In addition, modifications to the same pages by different transactions are batched into a single modification and written out to disk in one shot. Modifications to the same data object is also absorbed into a single update that reflects only the latest modification. This process is called write absorption.

Two details associated with the working of the MOB needs to be addressed when implementing the snapshot
utility in the ORs. Firstly, modifications are not written out to disk in log order due to the batching of updates to different objects in the same page into a single update. Secondly, not all modifications are reflected in the OR due to write absorption. Intermediate states between two modifications to the same data object may not be observed by the OR. We need to cater for situations when a snapshot is taken between two modifications to the same page that are batched together, and between two modifications to the same data object that caused write absorption of the older modification.

If two modifications to a single page $P$ are batched into a single disk write, the later modification will be removed from the MOB once the disk write is completed. Consequently, when a snapshot occurs between these two modifications, it will miss creating a snapshot page for $P$ because the later modification that would have otherwise triggered the creation of a page for the snapshot had already been installed earlier and removed from the MOB. One way of overcoming this problem is to create the snapshot pages when the MOB is scanned, since the scanning occurs in log order. However, this approach will force the OR to perform a disk read to bring the pre-modified page into memory for every page that it scans. As pointed out in [28], delaying installation reads to a convenient time improves the performance of the system. If we force the OR to perform installation reads during the scanning process, we may neutralize the performance gain of using the MOB. Furthermore, since we do not know when the snapshots messages are to be delivered, saving entire pages can quickly fill up the buffer space allocated in the OR for storing snapshots.

To prevent performance degradation, we make use of the Anti-MOB to save the original state of data objects before they are modified when the MOB is flushed. That is, during the installation read, the data objects to be modified are first written to the Anti-MOB before we install the modifications and write the page back to disk. The pre-modified states of these objects are inserted into the Anti-MOB in timestamp order of the transactions that caused the modifications. By using the Anti-MOB to log the pre-modified state of the objects, we avoid having to perform disk reads during the scanning process. The relationship between the MOB and the Anti-MOB is depicted in Figure 1.

![Diagram of MOB and Anti-MOB](image)

Figure 1: The Relationship Between the MOB and the Anti-MOB

Write absorption in the MOB can also eliminate some intermediate states of objects required by a snapshot, therefore these intermediate states needs to be captured as well. Unlike the batching of modifications of objects in a single page, write absorptions occur when a more recent modification that supersedes an older modification gets inserted into the MOB. We say that the later modification displaces the older one from the MOB. Subsequently, during the installation process, only the latest modification is installed.

In order to capture the intermediate states of a data objects, we need to intervene when an older modification is about to be displaced by a more recent one. This is done by recording down the value of the object due to the older modification as the pre-modification state of the more recent modification that displaced it from the MOB. Furthermore, we add a *write absorbed* field to every MOB entry. The use of this write absorbed field is as follows:

- A NULL value in the write absorbed field indicates the corresponding modification did not result in any write absorption.
- If a modification $M$ results in a write absorption and the displaced entry $E$ in the MOB has a NULL write absorbed field, then the write absorbed field of the MOB entry corresponding to $M$ will contain $E$.
- If a modification $M$ results in a write absorption and the displaced entry $E$ in the MOB contains another entry $F$ in its write absorbed field, then the write ab-
sorbed field of the entry corresponding to \( M \) will contain \( F \) i.e., it inherits the field’s value from \( E \).

Finally, every entry in the Anti-MOB is appended with an applied field to indicate whether the corresponding modification has actually been written out to disk. Since the Anti-MOB records the pre-modification state of an object and the OR undo modifications by overwriting the later state of the object in the page with an earlier state in the Anti-MOB entry, all entries in the Anti-MOB with valid pre-modification states are considered to be have been applied. In other words, an entry will have a FALSE applied value if and only if the corresponding pre-modification state is not available. In our system, only the very first modification of a series of write-absorbed modification will have applied its applied value set to FALSE since the pre-modification state is still on the OR’s disk. When the last modification of this series is written out to disk, the an installation read will be performed to bring the page from the disk, causing he pre-modification state of the first entry of the series of write-absorptions to be available. We update this first entry with the corresponding pre-modification state of the object from the disk and set its applied value to TRUE.

We illustrate our scheme with an example as shown in Figure 2. Suppose at time \( t \), the OR encounters a modification data object \( X \), which has an initial value of 0. The modification changes \( X \) to 1. This is inserted in entry \( n \) of the MOB as shown in the figure. At time \( t+1 \). The write absorbed (WA) field of the corresponding MOB entry (\( n \)) will be empty. At time \( t+1 \), the OR encounters another modification to object \( X \) that changes its value to 2. This is inserted into the MOB as entry \( n+1 \) and displaces entry \( n \). Its WA field will then contain \( n \) i.e., the entry that it displaced from the MOB. Two entries are also created in the Anti-MOB. The first entry (entry \( n \)) will be an empty entry with applied=FALSE. The second entry (entry \( n+1 \)) contains the pre-modification state of entry \( n+1 \) of the MOB, which is the value of the first modification being displaced from the MOB.

Subsequently, this entry is displaced by another more recent modification to \( X \). We can see that in each case when an entry of an older modification is displaced by that of a more recent modification, the resultant value of the older modification is recorded as the pre-modification state of the more recent modification in the Anti-MOB. Furthermore, the value of WA is inherited from one modification to the next modification to the same data object. Finally, when the modification is written out to disk, the WA entry will enable the value of \( X \) on the data page to be update into entry \( n \) in the Anti-MOB. The OR then sets the applied field of entry \( n \) to TRUE.

### 3.4.2 Creating Snapshot Pages from the Anti-MOB

A snapshot can only be processed (i.e., data pages are created for the snapshot) when all the modifications that occurred before the snapshot have been written out to disk. Otherwise, the OR will be unable to compute the correct state of a data page for the snapshot since the modifications that need to be reflected in the data page may still be sitting in the MOB. Furthermore, a snapshot will only be processed after the OR obtains a complete history of preceding snapshots that occurred before the snapshot i.e., the snapshot occurred earlier than \( T_{SS} \). Without knowing all the preceding snapshots, the OR will be unable discard entries in the Anti-MOB after the corresponding page for the snapshot has been created. In summary, a snapshot with timestamp \( T_{SS} \) can only be processed if:

- \( T_{SS} \leq T_{TSS} \), where \( T_{TSS} \) is the timestamp of
the latest transaction that has been flushed from the MOB to the OR’s physical disk.

- \( T_{ss} \leq T_{curr} \), i.e., the OR has complete history of all preceding snapshots.

The creation of snapshot pages for a snapshot is done by a background flusher thread lazily. This flusher thread will scan the Anti-MOB and snapshot list to create the data pages for snapshots. To compute a page for a snapshot, the flusher thread needs to first perform an installation read to bring the disk page into memory. Next, it needs to rollback all the modifications to objects on the page that occurred after the snapshot. Finally, it also needs to install write-absorbed modifications to objects that occurred before the snapshot.

To undo the modifications to objects in the page after the snapshot is a straightforward task of scanning the Anti-MOB from the time of the snapshot to the end of the Anti-MOB. Since every entry in the Anti-MOB contains the pre-modification state of the object modified, the first entry that is encountered during the scanning process that modifies a particular object will contain the state of the object just before the modification is written out to disk. This is the state of the object just before the snapshot since the OR is scanning the Anti-MOB for only entries that occurred after the snapshot. Consequently, if the object is modified more than once, subsequently entries that the OR encounters in the Anti-MOB during the scan is ignored.

To install absorbed modifications that occurred before the snapshot is equivalent to installing the pre-modification state of the first entry in the Anti-MOB that is later than the snapshot and that modifies the same object. In effect, the scanning process for installing absorbed modification is the same as the scanning process for uninstalling modifications that occurred after the snapshot.

In addition, a scanned entry with a FALSE applied value will not be physically installed into the page. However, these entries still need to be included into the scan will prevent later entries that modify the same object from being installed into the page. This special case only arises when an entry with a FALSE value is the first modification to the object after the snapshot. This implies that the page on the disk is the correct version because our scheme dictates that all later modifications to the object would not have been physically installed into the page; otherwise, the first entry will have its applied field set to TRUE. Consequently, the page on the disk will be the page required for the snapshot.

Entries in the Anti-MOB older than the next snapshot in the snapshot list can be deleted once they have been used in creating a page for the snapshot. If the entry is later than subsequent snapshots in the snapshot list, it is still required to undo the modifications for the later snapshots that precedes it. Consequently, only entries representing modifications to a snapshot page that occurred between the current snapshot that the flusher thread is processing (i.e., the oldest snapshot) and the next snapshot can be removed after the snapshot page has been created.

The processing of a snapshot completes when all entries this snapshot and the next snapshot in the Anti-MOB has been removed i.e., all pages that were modified between this snapshot and the next snapshot have been created and archived.

To prevent a page from being archived twice for a snapshot, the OR maintains a page bitmap (i.e., one bit per page) for every snapshot that it knows. This bitmap will determine if a page had already been created for the snapshot.

### 3.4.3 Optimizing the Creation of Snapshot Pages

When modifications in the MOB are flushed, installation reads are performed to bring the affected pages into memory before the modification can be applied to the page and written out to disk. As an optimization, we can create snapshot pages for the affected pages during the installation read process when the MOB gets flushed, provided that the transactions causing the modifications are older than \( T_{curr} \). Creating snapshot pages during MOB installation reads can avoid one extra disk read per page, which would have been otherwise incurred if we were to create these snapshot pages later. Furthermore, due to performance reasons, the OR has a tendency to keep modifications in the MOB for a relatively long period of time before writing them out to disk. Therefore, we believe that substantial performance improvements can be achieved in this way.
The process of creating the snapshot page during the MOB flushing process is similar to that described in Section 3.4.2 i.e., the Anti-MOB still needs to be scanned to undo modifications after the snapshot and apply modifications that have been absorbed.

The complexity associated with this approach is that the flushing of modifications to disk may not necessarily occur in log order. Modifications to the same page are flushed to the disk at one shot. In fact, modifications that are written out in the same batch are ordered to ensure maximum disk write efficiency. One consequence of flushing modifications to disk in batches is that not all modifications can trigger the creation of snapshot pages. In Figure 3 for example, the OR cannot create the corresponding snapshot page when the modifications that occurred after Snapshot 2 are written out to disk because the OR cannot be certain that all modifications preceding Snapshot 2 (in timestamp order) has already been flushed to disk. This prevents the OR from correctly determine the state of the data page just before Snapshot 2.

Our snapshot creation scheme dictates that snapshots page can only be created for snapshots that are older than the oldest entry in the MOB. In effect, during the installation read process, only pages belonging to snapshots that is older than all the modifications being flushed can be created e.g., only pages for Snapshot 1 can be created in our example in Figure 3. Modifications that occurred after Snapshot 2 are logged into the Anti-MOB.

### 3.5 Archive Log

As committing pages to the P2P archival system requires sending the page over the wire to some remote node, it can take a relatively long time. The archive log is used as a buffer between the process of computing pages for archival and archiving the pages themselves. This allows the computation of snapshot pages to be done asynchronously with the archival process. The archive log is an on-disk physical log in the OR that saves data pages for snapshots. Every page stored in the archive log is tagged with the timestamp or name of the snapshot that it belongs to.

There are two thresholds the govern the working of the archive log, namely the *high watermark* and the *low watermark*. The high water mark determines when the OR should start to send pages to the P2P archive store and the low water mark determines when the OR should stop sending these pages. In other words, if the percentage of the archive log used is greater than the high watermark, the OR will start the archiving process. The archiving stops when the percentage of the archival log used is less than the low watermark. Pages are sent to the archival store in timestamp order.

### 3.6 Retrieving Snapshots

A FE requests a snapshot page from the OR by sending a page fetch request with a non-NULL timestamp field. When the OR receives this request, it must determine if the page had already been archived. To quickly ascertain the availability of the pages, the OR maintains two variables, $TS_{alog, start}$ and $TS_{alog, end}$. These two variables records the timestamps of the oldest and the most recent page present in the archive log.

The OR compares the timestamp on the page fetch request:

- If the requested page’s timestamp is larger than $TS_{alog, start}$, the snapshot page has not been created yet. In this case, the FE replies with a *snapshot not ready* message to the FE.

- If the requested page’s timestamp is between $TS_{alog, start}$ and $TS_{alog, end}$, the required page may be in the archive log. We do not know for sure because pages may not have been inserted into the archive log e.g., due to write absorptions which may cause the page to be create at a later time. The OR searches its archive log and return the page if it is found. Otherwise, it replies with a snapshot not ready message.
If the requested page’s timestamp is smaller than $T_{alog\text{-}end}$, the page had already been sent to the archival store. In this case, the OR makes the request to the archival store on behalf of the FE. When the archival store replies with the page, the OR forward the page to the FE.

If the archival store returns with a page not found message, the OR returns the page on its own disk storage because the page had not been modified since the requested snapshot. Since the OR does not know the snapshot that the page belongs to, it will have to contact the OR with the required snapshot’s timestamp to determine the timestamp of the most recent snapshot that is older than the required timestamp. This will be tagged to page as its snapshot timestamp.

### 3.7 FE Cache Management

The FE maintains a $T_{DB}$ to indicate the timestamp of the snapshot that it is currently running at. A NULL value indicates that the FE is running on the current database. The cache management is unchanged if the current transaction and previous transaction are running on the same snapshot version. When the current transaction changes from one snapshot to another, the FE’s ROT and cache needs to be cleared so that the FE can correctly run the transaction on the correct version of the database. Clearing the entire cache may degrade the performance of the database because the FE will need to refill its cache and hot object belonging to the previous version of the database will be discarded. We propose a scheme to allow the FE to maintain multiple version of the database in its cache at the same time. We further propose a compaction scheme that allows the FE to perform cache management across these different versions.

#### 3.7.1 Page Mapping

The FE uses the ROT to perform pointer swizzling, causing the references to objects to point to a cached copy of the object on the FE’s cache. When the FE runs a transaction on a snapshot of the database, the ROT needs to be cleared. When a object is not the ROT and the FE accesses the object, it consults either the page map ($pmap$), which maps $< OR\#, PageID >$ to the index of the page in the FE’s persistent cache. We extend this mapping to map $< OR\#, PageID, T_{DB} >$ to the index of the page in the persistent cache. When an object that is accessed is not in the ROT, the FE will first search its persistent cache to find the containing page and it uses the $pmap$ for this purpose. We include the timestamp of the snapshot that the current transaction is running on in the mapping and returns the page if and only if the page with the same timestamp is present in its persistent cache. Consequently, every page in the FE is augmented with a timestamp field to indicate which version of the snapshot it belongs to. When the $pmap$ misses, the FE will request the OR for the corresponding page with the page identifier, as well as the timestamp. This will allow the OR to retrieve the correct version of the page for the request.

In the HAC scheme, an object that is contained within a page that was fetched could already be contained in some other page frame due to compaction. Since the copy of the object in the page frame is already installed into the ROT, this copy will be used instead of the copy in the page. The object is copied from the page frame into the page only when the page frame is freed. In our scheme, the ROT is cleared whenever we switch from one snapshot version to another. This will cause the objects in the page that is recently fetched to be used instead of the copy in the page frame. This does not affect the correctness of the system because both copies of the object are consistent when the transaction starts. Furthermore, this does not affect the object usage computation for cache replacement because HAC uses a scheme that values recency more than frequency.

#### 3.7.2 Page Compaction

Page compaction can cause an object to move from one page frame to another. Since the FE keeps pages from different snapshots in its persistent cache at the same time, compacting different pages from different snapshots can cause objects from one snapshot version to reside in a page frame in another snapshot version. This may cause objects of the wrong version to be used in a transaction. One method of preventing cache contamination is to prevent page compaction. This is however not desirable because it may cause thrashing if the hot objects in the cache are from many different pages.
To prevent cache contamination while allowing compaction, we modify the HAC’s compaction mechanism to compact only pages with the same timestamp. The selection of victim frames for compaction does not consider the timestamp; the timestamp is used only to ensure that the target frame and victim frame are of the same snapshot version.

3.8 Peer-to-peer Archival Store

The archival store for storing snapshot data pages is built on the DHash layer of CFS, which is in turn, layered on top of the Chord P2P routing and location infrastructure. CFS is a read-only file system and does not provide an explicit delete operation. Consequently, it uses an expiration mechanism to discard outdated data. Archived data, on the other hand, requires permanent, durable storage.

We assume that in our system, storage space is abundant. Given the rapidly decreasing cost of disk storage space and the ability of P2P systems to amalgamate and utilize unused storage in the participating computers, we believe that this assumption is a valid one. Consequently, pages in the archival store are archived permanently and must not be deleted. We therefore remove the expiration mechanism in CFS to provide permanent storage in the archive store. DHash also provides replication of the data stored for fault tolerance, which improves the data survivability of the archive store. In addition, we further extend the archival store with a self-repair mechanism to improve data durability.

CFS is designed to tolerate the constant joining, leaving and failure of nodes in the Chord ring. In our system however, storing of snapshot data pages in random nodes on an untrusted network affects both the data survivability and performance of our archival store. Since these are two key metrics of our system, the more rational approach will be to store these data page in a network of trusted database servers or dedicated storage servers. Moreover, the amount of data stored in our archival store is expected to be large, hence the constant movement of nodes will result in a large amount of data transfer in the network. This will inevitably stress the system and its underlying network, resulting in a performance degradation. Therefore, we expect nodes to join and leave rarely. Of course, even reliable storage servers can still fail. In fact, old storage servers may be shut down due to upgrading and new ones may be plugged into the network to increase its capacity. The self-configuration feature of the P2P architecture, which copes with nodes leaving and joining, allows these events to take place with little or no human intervention.

3.9 Data Page Identifier

In DHash, the key of the disk block is determined by the secure hash (SHA-1) of its contents. This method of determining the key of a disk block preserves the integrity of the data block since a change in the data of the block will cause the key of the block to be different from its original key. Such a mechanism allows the the successor nodes of the data blocks to detect corrupted data blocks and replace them with clean data blocks in the replicas. This enables the system to maintain the integrity of the data that it stores.

Although it is attractive to make use of a data block’s key (i.e., its content hash) to maintain data integrity, such a scheme will cause different snapshot versions of a data page to be mapped to different successor nodes. During the snapshot retrieval process, the user provides the system with the timestamp of the desired database snapshot. As we have seen in Section 3.6, the correct page to return for a particular timestamp is the page belonging to the youngest snapshot that occurred just before the timestamp provided.

We see that retrieving the correct snapshot version of a page involves searching through the different versions and examining the timestamps in every version of the archived page. Dispersing the different versions of a data page will hence incur expensive communication costs during the retrieval process, as the system will need to make a request to a remote successor node for every version of the page it wishes to examine.

We instead propose to determine the key of a data block by its page number and OR number of the containing OR. The combination of these two numbers uniquely identifies a data page within the entire database system.

Key(Page) = SHA-1(concatenate(OR#, Page#))

In such a scheme, different snapshot versions of a data page will have the same key and hence mapped to the same node. The timestamp or name of the snapshot
tagged to the page will allow the successor node to uniquely identify each version of the same page. Placing all the versions of a page on the same successor node allows us to efficiently search through the different snapshot versions of a data page. Traditional B-tree or B+-tree algorithms can also be used to speed up the search process.

In addition, it may be possible to perform compaction when all the different versions of a page are stored in a single by storing only the difference between the pages rather than an entire page for each version. However, this will incur additional cost when retrieving the snapshot pages as additional computation is required to produce the page. Since we are assuming that there is abundant storage, we choose to store entire pages to improve the performance of the retrieval process.

Note that request for a snapshot version of a page will first go through the containing OR, which will in turn, search its own archive log before propagating the request to the archival store since the snapshot page required by the user may still be residing in the OR’s archive log.

3.10 Self-Repair

A self-repair mechanism involves a successor node detecting if a data block is corrupted and making the necessarily replacement to this block if it is indeed corrupted. Since we do not derive the key of a data block from its content hash, we cannot rely on the data block’s key to maintain data integrity. As a result, we need an alternative scheme to allow the successor nodes to check that all the blocks that it stores are not corrupted. We propose to use a CRC checksum scheme to achieve this goal. The CRC checksum of for each block is calculated and append it to the block during the storage process. CRC checksum algorithms have the advantages of having small overhead, good error detection and ease of implementation.

The CRC checksum stored with each data page in the archival system will allow the successor to check if the data page is corrupted. The integrity checking of the successor node is done as a background process so the impact on the performance of archival system is minimal. When a data page is found to be corrupted, the successor node simply makes a request for the page from one of its replicas and replaces the corrupted page.

3.11 The Archival Store Interface

The P2P archival store provides two services to the ORs, namely:

- put(Page, Key, Timestamp, CRC Checksum): Uses the Key to determine the successor of the Page. This, together with the Timestamp and the CRC Checksum, is sent to the successor node for storage. The node calculates the CRC checksum based on the contents of the page and returns a success if it matches with the CRC checksum sent by the callee. Otherwise, it returns an error message.

- get(Key, Timestamp): Fetches and returns the Page associated with the Key and the Timestamp i.e., the version of the Page keyed by Key and having the largest timestamp that is smaller or equal to the indicated Timestamp.

To store a page into the archival store, the OR first calculates the key of the page from its page number and OR number. Next it calculates a CRC checksum on the page and appends it to the page before sending it to the archival store using the put function. The archival store will ensure that a correct copy of the page is received and written to its disk by using its CRC checksum. To do this, the successor node in the archival store will first write the page to its disk. It then reads the page back to compute its CRC checksum, which is compared to the value of the checksum appended to the end of the page. If the page is stored correctly, the archival store will return a success message and the archival is completed. Otherwise, an error message will be returned and the OR will try to archive the page again.

To retrieve a page from the archival store, the OR simply calculates the key of the required page by using its page number and OR number. The client will have to specify a timestamp of the required snapshot. This timestamp, together with the derived key, will be used to retrieve the archived page using the get function. Upon receiving the request, the successor node of the page in the archival store will perform an integrity check of the page before sending it back to the OR. If the page from its disk is corrupted, it will forward the request to a replica. When the replica returns with the page, the successor node
will overwrite its own copy of the page with the copy it received from the replica. At the same time, it will send the received copy back to the OR. The OR also uses the CRC checksum on the page to determine its integrity. If the page is corrupted during the transmission, the OR discards the corrupted page and resends the request to the archival store.

4 Related Work

4.1 Database Snapshots & Archive Copies

Archival utility (AC) is a commonplace among commercial database management systems [9, 17]. AC is mainly used as a method for database backup and recovery. A full AC (FAC) archives all the pages of the database while an incremental AC (IAC) archives only pages that are updated since the last AC. In addition, a fuzzy AC is an AC that allows concurrent updates during the AC execution.

The AC utility in DB2 [9] supports both FAC and IAC. The data structure used for its AC utility includes a table of archive copy bit (ACB), one for each data page in the system, and a flag at each data page. The ACB is set when the page is first modified by an update transaction. In addition, the flag of the page is also set when the page is first modified. When a data page in the system is to be updated and the flag is not set, the database will check to ACB map to determine which pages needs to be copied to the archive. If the flag is already set, then the data page to be updated has already been modified since the last AC and would have already been archived, hence no special actions are required. In order to serialize page updates by a transaction with the maintenance of the flag by the AC utility, a lock is required on every page as it is accessed. Mohan and Narang [27] further improved this scheme by avoiding locks and eliminating the need for the flag in every data page.

On the other hand, Informix provides fuzzy IAC and FAC utilities using timestamps and shadowing [17]. A logical timestamp is stored into a data page’s header to detect which pages have changed since the last AC. An AC_Begin checkpoint is taken at the beginning of an AC. Since Informix does shadowing using a physical log, which stores the pre-modification state of a page every time it is first modified after a checkpoint, pages that have been modified since the beginning of the AC are checked for their eligibility of inclusion in the IAC by retrieving their AC_Begin checkpoint’s state.

The methods employed by DB2, Informix and POSTGRES are based on a centralized database system. As a result, their requirements and considerations are significantly different from ours.

4.2 Archival Storage Systems

In the Myriad system [7], the authors present an alternative to mirroring in achieving disaster tolerance in a multiple site geoplex. The main benefit of Myriad is the reduction in the storage space required as compared to a mirroring solution. The system makes use of cross-site checksums i.e., checksum blocks are computed from a group of blocks, one from each site, using the Reed-Solomon erasure code. The checksum blocks are used to recover data for a site should the local redundancy fail to recover the data. The checksum blocks are updated incrementally. The main assumptions of this work are that data should be dispersed over multiple sites and the applications on each site should perform only local computation (otherwise, the cost of the WAN bandwidth will exceed the benefit of Myriad, which is a result of reduced physical storage).

Venti [31] is network storage system designed for archiving data. Venti uses a similar approach to CFS in that it uses the SHA-1 of a block’s content as its identifier. This way of identifying blocks allows the data integrity of the blocks to be maintained. It also allows duplicate blocks to be coalesced to reduce storage consumption. However, in a highly survivable system, coalescing of blocks reduces the amount of redundancy and hence reduces its ability to achieve fault tolerance. The authors also pointed out that archived data should be immutable to prevent accidental deleting or modification of data. This reduces the probability of data block corruption.

Snapshot [29] is an asynchronous mirroring utility that makes use of incremental file system snapshots and a copy-on-write scheme to identify blocks for transfer. The system maintains an active map for every snapshot in the system. The active map has one bit for every block
in the volume. By comparing active maps between two snapshots, the system can determine which blocks should be involved in an update transfer between the mirrors. The log-structured operation of the copy-on-write scheme also allows these updates to be batched, thereby reducing the bandwidth and write latency of maintaining a mirror compared to a synchronous solution. The system uses an eager update scheme where the source periodically sends a snapshot of the data to its mirrors. Since all the data for a snapshot is present in a single location, the creation of snapshots at the source volume and its application at the destination volume are considered to be atomic operations. Furthermore, updates to the file system is always done at the source volume and hence the system does not deal with the problem of concurrent transactions in the system.

The Elephant file system [34] offers protection from accidental deletion of files by the user by retaining important versions of the file. The file system provides the user with two distinct services, namely the ability to undo changes within a short time frame, and also keeps long-term history of important versions. Versions of a file are defined by the full file name (with path name) as well as the date and time of creation of the file i.e., timestamp of file creation. The versions are also indexed by this timestamp. The naming scheme for file versions in the Elephant file system is similar to our snapshot naming scheme. However, the retrieval of files in their system is based on a specific time indicated by the user rather than the latest copy before the snapshot.

Apart from Myriad, Venti, SnapMirror and the Elephant file system, earlier file systems such as WAFL [15], Plan 9 [30] and AFS [16] also provide backup services using a snapshot feature. A snapshot, as defined in the context of a file system, is a consistent read-only view of the file system at a particular point of time. These snapshots are implemented using the copy-on-write or lazy mechanism. A new copy of a block is created when the block is modified and the original block is left unchanged. Unmodified blocks are shared between the active file system and its snapshots.

4.3 Distributed Storage Systems

The Archival Intermemory [8] is a subscriber-based cooperative distributed block-level storage system that provides highly survivable and available data in the wide-area. Data durability is provided using erasure resilient codes to encode the blocks before distributing them into the system. The addressing of blocks to the host that stores the blocks is done offline. The repair mechanism in the system replaces dead storage servers with new Intermemory daemons.

Another wide-area persistent P2P storage system that uses erasure resilient codes to provide data durability is the OceanStore [18] project from University of California, Berkeley. OceanStore is built on top of the Tapestry [38] routing infrastructure. In contrast to the Archival Intermemory, OceanStore stores whole files rather than blocks. These files are uniquely identified by a globally unique identifier (GUID), which is the collision-resistant cryptographic hash of the file’s contents for immutable objects. OceanStore also supports updating of the files that it stores; in this case, the GUID is the hash of the public key and the username of the owner rather than the hash of the file’s content. To support updates in an untrusted infrastructure, OceanStore employs a primary tier of master replicas that runs a Byzantine agreement protocol to choose the final commit order for updates.

A similar system to OceanStore is the PAST [33] system from Microsoft Research. Like OceanStore, PAST aims to provide persistent storage in a wide-area P2P network. Unlike OceanStore however, PAST achieves data availability and durability using replication of the files stored in the system. In addition, files stored in PAST are immutable. PAST is layered on top of the Pastry [32] routing infrastructure, which is very similar to Tapestry used in OceanStore in that both systems use some form of prefix/suffix routing to forward a message one digit closer to the destination in the logical namespace at each hop.

While PAST stores whole files in the system, CFS [10] segments each file into equally sized blocks before storing them. Like PAST, replication is the main engine to achieve high availability and fault-tolerance. In addition, both system make use of collision-resistant content hash to calculate the key of the file/block, hence providing data integrity to the system. The block-level storage in CFS allows better utilization of free space and more efficient retrieval of files e.g., a file’s block can be retrieved in parallel from the different servers that store the blocks. The
routing layer used by CFS is Chord [35], which is an overlay network that organizes nodes in a ring topology. Each node in Chord is uniquely identified an identifier, which is the secure hash of the nodes IP address and a virtual node index.

5 Conclusion

In this paper, we proposed to extend Thor with a snapshot utility. This utility will enable the user to preserve the state of the database at a particular point of time. Read-only transactions can then be performed on snapshots of the database. We presented a lazy, gossip-based approach to propagating snapshots throughout the system to minimize interference to concurrent transactions. We also proposed to create data pages for snapshots incrementally and lazily to improve performance and storage requirements.

Pages of snapshots are stored in an archival store, which is implemented on the DHash layer of CFS, a distributed file system that is in turn, implemented on top of the Chord P2P routing and location infrastructure. P2P systems provide us with the desirable qualities of automatic load-balancing, self-configuration and fault-tolerance. More importantly, they provide us with the ability to harness unused storage space to provide a potentially huge storage arena to cater to the the potentially large amount of storage that our archive store may require.

An on-demand scheme is used when running transactions on snapshots of the database to reduce the amount of data movement during the transaction. Snapshot pages are installed into the persistent cache of the FE. We extend the FE’s caching scheme to support multiple database versions and present a scheme for allowing page frame compaction without contaminating the cache.

References


