High Availability for Database Systems in Geographically Distributed Cloud Computing Environments

by

Huangdong Meng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Abstract

In recent years, cloud storage systems have become very popular due to their good scalability and high availability. However, these storage systems provide limited transactional capabilities, which makes developing applications that use these systems substantially more difficult than developing applications that use a traditional SQL-based relational database management systems (DBMS). There have been solutions that provide transactional SQL-based DBMS services on the cloud, including solutions that use cloud shared storage systems to store the data. However, none of these solutions take advantage of the shared cloud storage architecture to provide DBMS high availability. These solutions typically deal with the failure of a DBMS server by restarting this server and going through crash recovery based on the transaction log, which can lead to long DBMS service downtimes that are not acceptable to users. It is possible to run traditional DBMS high availability solutions in cloud environments. These solutions are typically based on shipping the transaction log from a primary server to a backup server, and replaying the log at the backup server to keep it up to date with the primary. However, these solutions do not work well if the primary and backup are in different, geographically distributed data centers due to the high latency of log shipping. Furthermore, these solutions do not take advantage of the capabilities of the underlying shared storage system.

We present a new transparent high availability system for transactional SQL-based DBMS on a shared storage architecture, which we call CAC-DB (Continuous Access Cloud DataBase). Our system is especially designed for eventually consistent cloud storage systems that run efficiently in multiple geographically distributed data centers. The database and transaction logs are stored in such a storage system, and therefore remain available after a failure up to the failure of an entire data center (e.g., in a natural disaster). CAC-DB takes advantage of this shared storage to ensure that the DBMS service remains available and transactionally consistent in the face of failures up to the loss of one or more data centers. By taking advantage of shared storage, CAC-DB can run in a geographically distributed environment with minimal overhead as compared to traditional log shipping solutions.

In CAC-DB, an active (primary) and a standby (backup) DBMS run on different servers in different data centers. The standby catches up with the active's memory state by replaying the shared log. When the active crashes, the standby can finish the failover process and reach peak througput very quickly. The DBMS service only experiences several seconds of downtime. While the basic idea of replaying the log is simple and not new, the shared storage environment poses many new challenges including the need for synchronization protocols, new buffer pool management mechanisms, approaches for guaranteeing strong consistency without sacrificing performance and new shared storage based failure detection mechanism. This thesis solves these challenges and presents a system that achieves the following goal: if a data center fails, not only does the persistent image of the database on the storage tier survive, but also the DBMS service can resume almost uninterrupted and reach peak throughput in a very short time. At the same time, the throughput of the DBMS service in normal processing is not negatively affected. Our experiments with CAC-DB running on EC2 confirm that it can achieve the above goals.

Acknowledgements

At first, I would like to thank my supervisor Ashraf Aboulnaga for guiding and supporting me through my graduate studies at academic and career levels.

I would also like to thank Professor Kenneth Salem, Li Liu, and Rui Liu, my collaborators on this work, for continuously sharing their insights.

I would also like to thank Professor Kenneth Salem and Professor Lukasz Golab for being my thesis readers.

I would also like to thank my parents for their continuous support and love throughout my entire life.

A special thanks to my girlfriend, Yuke Yang, for her support, patience and great food that helped me through my Master program.

Finally, I would also like to thank the University of Waterloo for providing me a chance to begin my professional life in North America.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today there are numerous distributed storage systems that are designed for a cloud computing environment. These systems are scalable and highly-available, but provide only limited functionality. First, they have a simple data model where the data is stored as (key, value) pairs, and they only support simple operations, e.g., reading and writing values by key. They do not support full SQL, so they they are often called "NoSQL" systems. Second, they have very limited transactional capabilities. For example, Cassandra [22] and Google's Bigtable [15] only provide atomic read and write access to a single row, while atomic multi-row queries and updates are not possible. Google's Megastore [10] does support multi-row atomic access, but multi-row transactions are limited to predefined fine-grained database partitions. In contrast to cloud storage systems, relational database management systems (DBMS) provide a rich query language in the form of SQL, and strong ACID guarantees for transactions. This makes developing application that use relational DBMS much simpler than applications that use cloud storage systems. However, relational database systems are not as scalable and elastic as cloud storage systems.

It is possible to use a cloud storage system as the storage tier for a relational DBMS, which can in turn support the data management needs of hosted applications. We call this approach a *shared-storage DBMS* solution. There have been several shared-storage DBMS solutions, such as running a DBMS on Amazon's S3 storage service [13], ElasTras [19] which uses HDFS [3] as the storage tier, and DAX [23] which uses a Dynamo-like multi-master system as the storage tier.

The shared storage DBMS architecture is a promising architecture for data management on the cloud. This architecture decouples storage scalability and availability (which become the responsibility of the storage system) from query and transaction processing (which become the responsibility of the DBMS). However, all solutions that follow this architecture lack DBMS-level high availability (HA), that is, an always-on hosted DBMS. If a data center fails, the persistent image of the database in the cloud storage tier can survive in another data center through the high availability mechanism of the cloud storage system. However, the cloud storage high availability mechanism does not address DBMS high availability, and a new DBMS server must be launched in another data center and must go through the normal DBMS recovery process, typically based on replaying the transaction log. However, this is a slow process resulting in a significant outage of the DBMS service that may not be acceptable to users. For example, in DAX, the system experiences a downtime of about 270s when a DBMS server fails, and the time to reach peak throughput after a backup server comes online is around 1000s (Figure 10 in [23]). Therefore, it is desirable to provide DBMS-level high availability for shared storage DBMS architectures, and this is the focus of this thesis.

A common approach to providing DBMS high availability is to ship the database transaction log over a network from a primary active DBMS to a backup standby DBMS running on a different server machine. For example, MySQL, which we use in this thesis, has a high availability solution based on *Binlog (binary log) replication* [5]. Binlog replication offers two modes of replication from the active to the standby DBMS: *asynchronous* and *synchronous* log shipping (asynchronous or synchronous to transaction commits). Committed transactions may be lost if asynchronous log shipping is used, while the throughput of the active DBMS will be reduced significantly if synchronous log shipping is used, especially in a multi-data-center deployment. Any log shipping solution would suffer from high message latency if the active and standby servers are in different, geographically distributed data centers. Besides, log shipping solutions do not take advantage of the availability of shared storage between the active and standby in a shared storage DBMS architecture.

In this thesis, we present a new transparent high availability solution for shared storage DBMS architecture, which we call *CAC-DB* (for Continuous Access Cloud DataBase). Our system is designed especially for *eventually consistent* cloud storage systems such as Dynamo [20] or Cassandra [22]. These cloud storage systems run effectively in multiple, geographically distributed data centers and can remain available in the face of failures up to a disaster that crashes a whole data center. Our solution is based on DAX, which provides a multi-tenant database service on top of an eventual consistency cloud storage system. DAX's storage tier provides scalability, elasticity, and availability that is decoupled from the DBMS tenants. DAX works well across data centers and offers strong consistency guarantees to the DBMS tenants without sacrificing performance, which makes it an excellent platform for implementing CAC-DB.

The basic idea of CAC-DB is to provide DBMS high availability by replaying the log

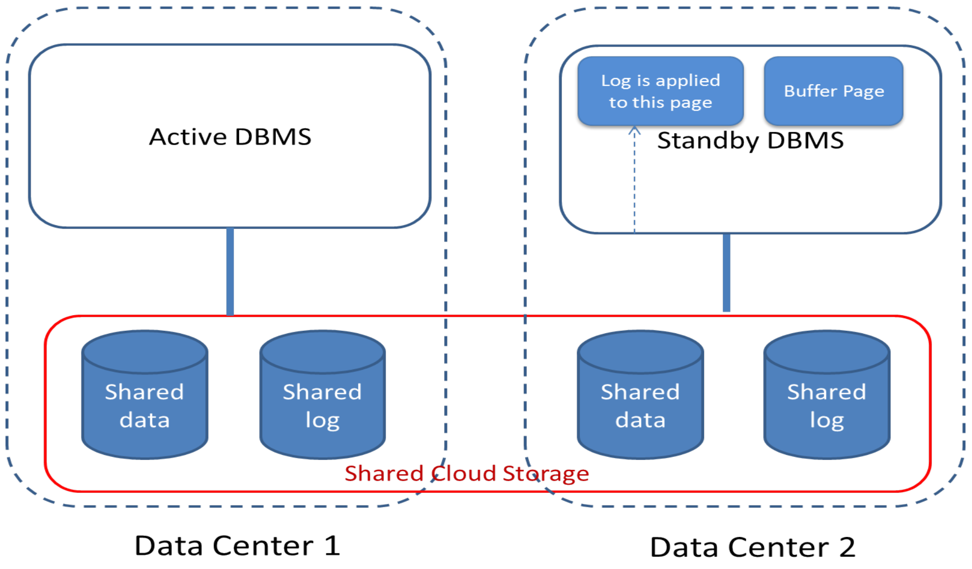

Figure 1.1: CAC-DB System Overview

generated at an active DBMS on a different machine that is hosting a standby DBMS. This idea is not new [5], but the log in our solution is in the shared storage system instead of being shipped from the active to the standby. Figure 1.1 illustrates this architecture.

Having a log and database that are shared between active and standby provides many benefits, but also raises many technical challenges. The main benefit of the shared database and log in DAX is removing the latency of log shipping from the critical path of transaction execution at the active server. The log records still need to be propagated from the active to the standby, but the propagation is handled by DAX with minimal performance overhead for the active. Another benefit is that DAX ensures that the standby can always access a transactionally consistent version of the log that contains all log records of all committed transactions, so no committed transactions are lost in case of a failure. A third benefit is saving storage space, since the standby does not need to have its own copy of the database and log as would be the case in a typical log shipping solution.

The challenges raised by having a shared database and log relate to coordinating access to these shared files between the active and standby. First, in order to ensure that the database and log remain consistent, the shared data can only have one writer at any time. The standby in CAC-DB can read database and log pages, but it cannot write these pages. Therefore, the standby needs novel page eviction mechanisms, since it can no longer flush dirty pages. Second, it is important to guarantee that if the standby is slower than the active and needs to catch up with it, the standby could do so without affecting the throughput of the active. Third, the standby needs a strong consistency guarantee without sacrificing performance during normal operation and during failover. In addition, synchronization protocols between the active and the standby and a lightweight failover detection mechanism based on shared storage are needed.

CAC-DB solves these challenges and achieves the following goals: (1) the overhead during normal operation is low, (2) data remains consistent and accessible after a failure up to the loss of one or more data centers (as in a natural disaster), and (3) the database service experiences minimal outage after a failure and reaches peak throughput fast. We show that CAC-DB has much better performance than a traditional log shipping approach, which is not surprising in a geographically distributed setting.

The difference between CAC-DB and a log shipping solution can be summarized as follows. Compared to asynchronous log shipping, CAC-DB guarantees no committed transactions are lost. Compared to synchronous log shipping, CAC-DB imposes a much lower, almost negligible performance overhead on the active, especially in a multi-data-center deployment. This is because DAX, on which CAC-DB is based, tries to hide the latency of cross-data-center communication and remove it from the critical path of transaction commit. CAC-DB does not require the log to be shipped and does not require an additional local copy of the database at the standby, because the log and database are in shared storage that is accessible to both active and standby. It should be noted that since the persistent log and database are accessible to active and standby, the focus of CAC-DB is on efficiently maintaining the in-memory state of the standby so that it is consistent with the active. That way, if the active fails, the standby can take over managing the shared log and database, go through the log-based recovery process fast since most of the log has already been replayed, and reach peak throughput with minimal service interruption.

The remainder of the thesis is organized as follows. Chapter 2 provides a summary of related work. In Chapter 3, we provide some background about DAX and ARIES recovery, and present the architecture of CAC-DB. Chapter 4 describes CAC-DB during normal operation, discussing buffer pool management, the synchronization protocol between active and standby, and how to ensure that the standby does not regularly fall behind the active and does not slow it down. Chapter 5 discusses failure detection and failover. We present an empirical evaluation of a deployment of our system in Amazon's Elastic Compute Cloud (EC2) in Chapter 6. Chapter 7 concludes the thesis.

Chapter 2

Related Work

There are different types of HA techniques which are used in database systems. Active-standby HA is one of the most important techniques, and it is implemented by many database systems [8, 16, 29, 25]. In active-standby HA based systems, log shipping is often used. Log shipping may be synchronous or asynchronous. With synchronous log shipping, committed transactions are not acknowledged to the client until the active and standby both have made the update durably. Systems based on synchronous log propagation are considered 2-safe systems [21, 28], which means that committed transactions are not lost due to the failure of a single server. However, HA based on synchronous log shipping may impose a large performance penalty on the throughput of the active server, since each transaction commit is delayed until the log records are propagated. In contrast, with asynchronous log shipping, transactions are acknowledged as soon as they are committed at the active. Systems based on asynchronous propagation are 1-safe systems, which incur much lower overhead at the active during normal operation. However, recently committed transactions may be lost if the active fails. We have tested and compared MySQL semi-synchronous and asynchronous Binlog replication, and we discuss their drawbacks in Chapter 6. RemusDB [25] is an active-standby HA system that implements DBMS high availability as a service provided by the virtual machine layer. It uses asynchronous checkpointing to propagate updates at the VM level from active to standby. RemusDB requires the DBMS to run in a virtual machine, and it does not address the issues of geographically distributed deployment; putting the active and standby in different data centers would result in high latency for commits at the primary.

In CAC-DB, which is itself an active-standby system, the active does not need to wait for the standby and therefore there is no additional throughput overhead on the active. In addition, CAC-DB is built on the DAX storage system [23], which guarantees that updates are synchronized to different replicas in the storage tier in a way that is consistent with the semantics of DBMS-level ACID transactions. Therefore, CAC-DB is 2-safe and addresses the drawbacks of synchronous and asynchronous propagation. CAC-DB differs from other database active-standby systems in that it is designed for a shared-storage environment instead of a shared-nothing environment. In addition, a major advantage of CAC-DB is that CAC-DB can provide fast failover across multiple geographically distributed data centers, while other HA solutions require the active and standby to be in the same data center.

There are also several HA approaches for shared access environments, where two or more DBMS instances share a common network attached storage infrastructure, in which the database is stored. The network attached storage stores data redundantly, which makes the data reliable. If one DBMS server fails, other servers which can access to the same database can take over quickly. This technique is implemented by many database systems, such as Oracle RAC [30], and synchronized data nodes accessed through the NDB in MySQL Cluster [8]. CAC-DB differs from these techniques in that it is an active-standby system which is built on top of shared storage. In addition, shared access approaches are designed for single-data-center failover while CAC-DB works in a geographically distributed setting.

In this work, we assume a multi-tenant database service deployed on the DAX shared storage system. Other works have been proposed that deploy a database service on cloud shared storage. ElasTras [19] is an example of a system that is intended to host DBMS on a shared storage tier. However, unlike DAX [23], ElasTras assumes that the storage tier provides read and write operations with strong consistency while DAX assumes a storage tier with eventual consistency. Recent work by Najaran et al. [27] has looked at hosting relational database services on a storage tier based on Sinfonia [9]. However, their work can only provide tuple-level access while DAX provides a block-level interface. In principle, CAC-DB can be adapted to work with any system that uses shared storage, but of course the details of how to do this are highly non-trivial, and are an interesting area of future work for this thesis.

There are also several commercial services which are used as storage tiers for DBMS tenants. A typical example is Amazon EBS, which can provide strongly consistent read and write operations. However, EBS does not work across multiple data centers. EBS is used as the storage tier in our experiments for Binlog replication since the active and standby in this case maintain separate copies of the database in their respective data centers. Brantner et al. [13] consider hosting database systems on Amazon Simple Storage Service (S3). However, S3 only provides weak eventual consistency guarantees, which makes offering transactional guarantees more complex. In contrast, DAX can offer transactional guarantees to the hosted database systems while at the same time providing good performance.

For failure detection, systems typically implement automated or semi-automated mechanisms using some kind of *heartbeat* [6]. Such mechanisms require the active to send regular heartbeat messages to indicate that it is still alive. Failure of the active to send a heartbeat message within a prescribed interval is taken as an indication that it has failed. The active's heartbeat messages may be directed to the standby system or to a third-party system. For example, a Bigtable [15] master system must maintain an active session with the Chubby lock service [14] to ensure that it remains as the master. To keep its session alive, the Bigtable master sends heartbeat messages to Chubby. Another example is the failure detection service in Boxwood [24], which requires that clients send regular heartbeat messages to a set of observers. CAC-DB relies on a heartbeat-based failure detection mechanism that uses DAX storage for reliable, indirect communication of the heartbeats.

Chapter 3

Background and System Overview

In this chapter, we describe the CAC-DB system architecture. Since CAC-DB is built on DAX, we first provide some background about DAX. We also present an overview of the ARIES recovery algorithm, which is used by many DBMS and which CAC-DB assumes will be used.

3.1 Overview of DAX

DAX [23] uses cloud storage to implement a reliable, distributed, shared-block storage service for hosted DBMS tenants. DAX implements a thin software layer called the *DAX client library* that runs on the DBMS tenant machine to intercept the interaction between the DBMS storage manager (InnoDB, in the case of MySQL) and the storage devices to read or write data. The DAX client library converts these interactions to interactions with the underlying cloud storage system (Cassandra, in the case of DAX). To the DBMS tenant, DAX behaves like a disk, but in terms of implementation it is a scalable, reliable, geographically distributed storage service.

The DAX service assumes that all DBMS data access is block-oriented. DAX stores all database blocks in a single Cassandra column family, that is, each row consists of a key and a single data column containing one data block. Each block accessed by a DBMS maps to a single row in this column family, with the block id as the key of the row. The DBMS storage manager remains responsible for data blocks caching, concurrency control, and transactional recovery.

The hosted DBMS in DAX expects that when it reads a data block, it will obtain the most recently written version of that block, but Cassandra only provides eventual consistency for data that is written. That is, the tenant can write one replica of a data block and then read another replica to which the write has not yet propagated, leading to a stale read. To solve this problem, DAX uses a technique called *optimistic I/O* for ensuring consistent reads for hosted database systems, while avoiding most of the performance penalty.

Cassandra provides multiple variants of the read and write operation that enable the application to control how many replicas need to be read or written before the return to the client. This allows the application to control the consistency vs. latency and consistency vs. durability tradeoff. One variant of read is Read(1), which returns the first response received to the client. This operation is cheap but may return stale data if the first replica to respond is stale. Cassandra also provides Read(QUORUM), which waits until a quorum of replicas responds and sends the most recent version from among the data items received from the quorum of replicas to the client. This operation is relatively expensive, especially if the replicas are distributed across geographically different data centers. Similarly, Cassandra provides Write(1) and Write(QUORUM). The client can choose cheap Read(1)/Write(1) operations, which are fast but not necessarily consistent or durable. The client can also choose expensive Read(QUORUM)/Write(QUORUM) operations, which are consistent and durable but may be slow. The goal of DAX is to use cheap reads and writes while getting the consistency and durability of expensive reads and writes. To enable the use of cheap Read(1) calls in DAX, each DBMS tenant maintains an in-memory *version list* that records the most recent version numbers of different blocks. These version numbers are also stored in the blocks in Cassandra. When a DBMS tenant reads a block that is on the version list, the tenant can determine if it received the most recent version of the block by comparing the version number read from storage to the version number on the version list. DAX optimistically uses Read(1) for blocks on the version list, and repeats the read if the returned version is stale. For blocks that are not on the version list, DAX uses Read(QUORUM).

For writes, DAX also uses cheap Write(1) operations. However, this reduces fault tolerance since the loss of the only replica containing an acknowledged write leads to loss of data. To address this problem, DAX implements a technique called *client-controlled synchronization* to complement optimistic I/O. In client controlled synchronization, DAX uses a new type of write operation, called *Write(CSYNC)*. Like Write(1), Write(CSYNC) operations are acknowledged to the client as soon as one replica is updated. In addition, Write(CSYNC) records the key (i.e., the block id) in a per-client *sync-pending list* at the DAX server to which the client is connected. The key stays in the sync_pending list until the remaining replicas have asynchronously acknowledged the update.

DAX provides the client with an API call named *CSync* (similar to fsync) by which the client can tell DAX to ensure that all previously acknowledged writes must be synchronized. This approach is very natural for a DBMS because many DBMS are already designed to use explicit update synchronization points. For example, a DBMS that implements the write-ahead logging protocol will write the log page and then issue a write synchronization operation (e.g., fsync in Linux) to ensure the requirement of durability. When the DBMS needs to ensure that a write is durable (at least k replicas updated), it issues an explicit CSync synchronization request. On a CSync, the DAX server to which the client is connected makes a snapshot of the client's sync_pending list and blocks until at least k replica update acknowledgements have been received for each update on the list. The synchronization request is then acknowledged to the client. A DBMS would typically issue such an explicit synchronization request as part of commit processing. As part of processing such a request, DAX ensures that the k replicas are distributed among various data centers. Therefore, even if an entire data center fails, the database and log are available in other data centers, and reflect the changes made by all committed transactions. A new DBMS instance can be launched in a different data center and can perform log-based database recovery and restore the availability of the database service.

3.2 ARIES Recovery Protocol

ARIES [26] is a recovery algorithm for DBMS. Recovery in ARIES consists of two stages: the redo stage and the undo stage. In the redo stage, the DBMS "repeats history", starting from the latest checkpoint in the log, and restores the database state to what it was at the time of the crash. In the undo stage, the DBMS undoes all the actions of uncommitted transactions, so that only the actions of committed transactions are reflected in the database.

3.2.1 The Transaction Log

The transaction log records the history of actions executed by the DBMS. The log, physically, is a file of log records stored in stable storage. The log file is assumed to survive a crash and can be used to perform recovery. The most recent portion of the log is stored in a *log buffer* in main memory and is periodically forced (i.e., flushed) to stable storage, usually as part of commit processing. Log records are written to stable storage at the granularity of pages, just like data records.

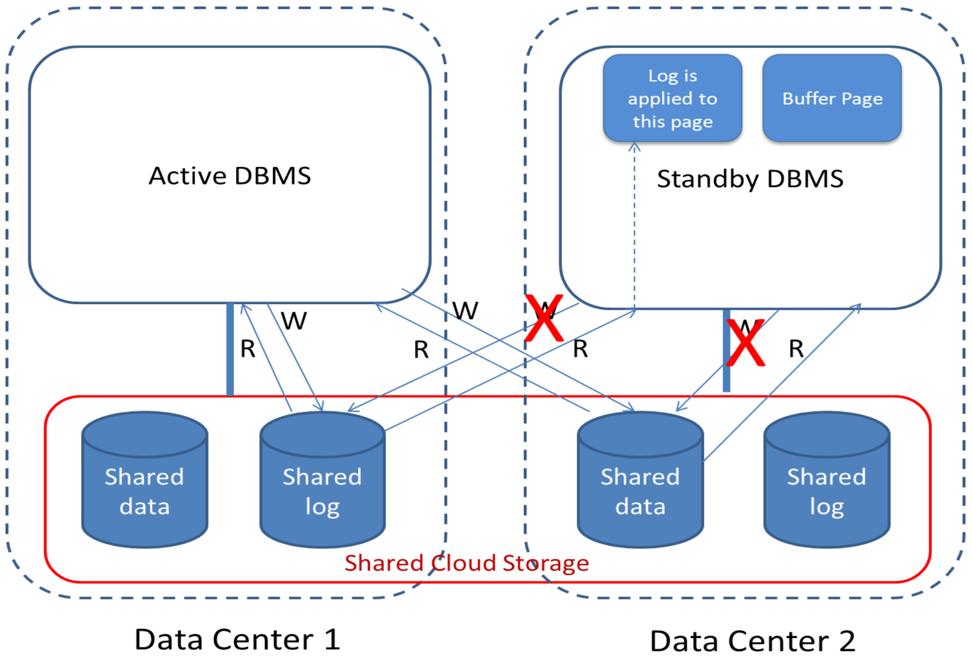

Figure 3.1: CAC-DB Architecture

Every log record has an id called the *log sequence number (LSN)*, which is assigned in monotonically increasing order. In the implementation of InnoDB, LSN mod the size of the log file is the physical offset in the log file. Every page has a *pageLSN*, which is the LSN of the most recent log record that makes a change to this page. In the buffer pool of a DBMS, the pageLSN of a buffer page is the last change to make this page dirty. Based on our system design, the standby has its own interpretation of pageLSN, shown in the definition below:

Definition 1 pageLSN in the standby*: In the buffer pool of the standby, the pageLSN of a buffer page is the last change in the log the standby has applied to this page.*

3.2.2 Checkpointing

When the DBMS takes a checkpoint, it is as if the DBMS takes a snapshot of its state. The DBMS periodically takes checkpoints to reduce recovery time. Conceptually, taking a checkpoint means that all the updates before the latest checkpoint have been flushed to stable storage. In InnoDB, a checkpoint is identified by its *checkpointLSN*. The log file records the checkpointLSN of the latest checkpoint in a known location. All the updates whose LSN is smaller than the latest checkpointLSN are guaranteed to be durable in the storage tier.

The checkpointing technique that InnoDB uses is called *fuzzy checkpointing*. It is inexpensive because it does not require the DBMS to write out pages at the checkpoint. Instead, a background thread is responsible for writing dirty pages to the durable storage periodically. At a checkpoint, InnoDB does not flush any dirty pages and only calculates its new checkpointLSN and records this value in the log. The new checkpointLSN is equal to the smallest oldest-lsn (the first LSN to make a page dirty) of all the dirty pages in the buffer pool.

3.3 Overview of CAC-DB

Figure 3.1 illustrates the architecture of CAC-DB, which is built on DAX. In CAC-DB, every database tenant has a primary server (active) and a backup server (standby) in different data centers. The database and the transaction log are stored in the cloud storage system. The active and standby servers both have (read and write) access to the shared database and log. DAX replicates the database and log in the two data centers with the active and standby, and hides most latencies related to propagating updates from the active data center to the standby data center, so access by the standby typically incurs no cross-data-center latency. It is important to note that even though both active and standby can read and write the database and log, CAC-DB is designed so that *only the active writes (as well as reads) the database and log.* The standby can *read* the database and log, but it never writes to either. The reason for this restriction is that we want every database block to have one writer in order to keep the database consistent without additional concurrency control protocols. This restriction means that the standby cannot flush dirty pages from its buffer pool, which requires changing the buffer eviction mechanism for the standby. We will return to this point in Chapter 4.

The active accepts users requests, executes queries and performs database updates in the shared storage. The standby does not accept user requests. Instead, the standby continuously reads the shared transaction log and replays log records to update its internal memory state (i.e., its buffer pool). This makes the contents of the standby buffer pool closely resemble the most recent state of the active buffer pool. Thus, the standby is "warm" and can take over quickly if the active fails.

If the active fails, the database and log are available to the standby, and their state is consistent with all committed transactions at the active. This is because the storage tier in DAX provides transactionally consistent scalability and HA for the persistent image of the database and log, independent of any DBMS instance. When the standby detects that the active has failed, CAC-DB provides fast failover and atomic handover from active to standby. The standby can then become the new active. The whole operation of a standby server in CAC-DB can be divided into three phases:

1. **Log Replay (phase 1):** In the traditional log-based ARIES recovery algorithm [26], the recovery procedure is based on repeating history, that is, executing exactly the same actions that happened during normal processing before the failure. Repeating history includes redo of log records of both complete and incomplete transactions from the latest checkpoint in the transaction log. In ARIES, recovery, this redo stage is followed by a subsequent undo stage. In CAC-DB, the standby continuously replays the log that is produced by the active. The log replay procedure of CAC-DB is naturally similar to the redo stage of ARIES recovery, so we reuse the code for the redo stage of recovery in our standby log replay procedure. The standby can get the complete transaction log from shared storage. We provide a consistency guarantee for log writes in DAX, similar to optimistic IO, that guarantees (along with the ARIES protocol) that the standby never skips any log records. CAC-DB uses batch based log replay, in which a log file can be divided into many segments. The standby repeatedly reads one segment from storage into memory, replays the log records in this segment, and then reads the following segment.

2. **Failure Detection (phase 2):** For CAC-DB, we use a heartbeat-based failure detection mechanism that uses DAX storage for reliable, indirect communication of the heartbeats. We create a reliable counter in DAX for each active/standby pair to support failover. The standby's failure to read the latest counter value for more than a fixed period is taken as an indication that the active has failed. The failure detection mechanism is presented in Appendix A.

3. **Failover (phase 3):** After detecting the failure of the active, the standby performs ARIES recovery. Since the standby is continuously replaying log records during normal operation, it usually finds that it has already finished the redo stage of ARIES. The standby performs the undo stage of ARIES recovery and becomes the new active. The undo stage of ARIES recovery can proceed while the standby is servicing user requests. That is, the standby actually becomes the active and then performs undo recovery. The new active (the old standby) reaches peak throughput quickly because

(1) it has a warm buffer pool, (2) it typically does not need to perform redo recovery, and (3) it serves user requests while performing undo recovery. The recovery is based on the ARIES algorithm [26] which guarantees atomic handover.

We illustrate the active and standby in the above three phases in a timeline diagram shown in Figure 3.2.

Although the basic idea of replaying the log is not new, the fact that CAC-DB is built on cloud shared storage and tries to take advantage of this storage creates many new challenges. First, during the standby's normal log replay, the standby cannot write to the database, which means that it cannot flush dirty pages, which means that a new mechanism is needed to manage the standby's buffer pool. Second, we have to deal with the case when the standby cannot catch up with the active, without placing any additional overhead on the active. Third, we need to provide a strong consistency guarantee for the standby because the underlying storage tier only supports eventually consistency. Finally, a novel failover detection mechanism based on shared storage is needed. We address these challenges through a series of novel techniques that we present in the next two chapters. Chapter 4 focuses on normal operation and Chapter 5 focuses on failover.

In the implementation of CAC-DB, we choose MySQL as the DBMS, with InnoDB as the storage engine, and DAX as the distributed storage service. For the DBMS tier, CAC-DB can be easily re-implemented in another DBMS, such as PostgreSQL. This would involve changes to the PostgreSQL code, along with the DAX client library for PostgreSQL. For the storage tier, we choose DAX because DAX offers strong consistency guarantees to the DBMS that works across data centers and without sacrificing performance. CAC-DB can use another storage system that provides strong consistency to the DBMS, but the solution would be limited by any limitations of the storage system. For example, if CAC-DB uses HBase as the storage tier, it could work well in a single data center, but not across data centers due to the limitation of HBase. HBase provides strong consistency for each operation in one data center, but this per-operation strong consistency would be expensive in a multi-data-center setting. Thus, HBase would be a good choice for the storage tier when CAC-DB is used to provide DBMS-level HA in one data center. On the other hand, DAX enables CAC-DB is to provide DBMS-level HA which can tolerate the crash of a whole data center (as in the case of a natural disaster).

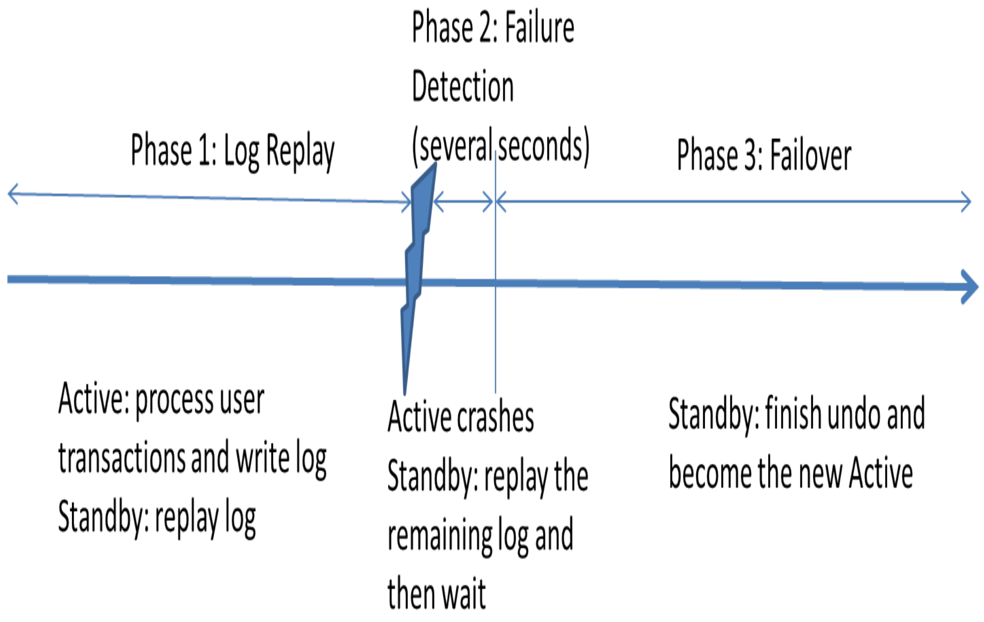
Phase 2: Failure Detection (several seconds)
Phase 1: Log Replay
Phase 3: Failover
Active: process user transactions and write log
Standby: replay log
Active crashes
Standby: replay the remaining log and then wait
Standby: finish undo and become the new Active

Figure 3.2: The Three Phases of Active and Standby in CAC-DB

Chapter 4

CAC-DB Log Replay

This chapter describes the normal operation of CAC-DB. During normal operation (i.e., no failure) the standby replays the log that is written in shared storage by the active. This log replay phase of CAC-DB requires tackling challenges such as synchronizing the active and standby, managing the buffer pool at the standby, and ensuring the consistency of the data read by the standby. We discuss these challenges next.

4.1 Synchronizing Log Replay Between Active and Standby

The basic architecture of CAC-DB is shown in Figure 3.1. The active and standby in CAC-DB access the log and database in a shared storage tier, so there is no need to transfer data between them over the network. The active writes log records to the shared transaction log and the standby reads these log records and replays them to update the in-memory state of its buffer pool.

For this log replay protocol, the standby needs to know when there are new log records to read. The active sends a trigger message to the standby over the network to trigger log replay. This message can be sent periodically (e.g., every second), or whenever the active commits a transaction. In our implementation of CAC-DB, the active sends trigger messages whenever the active commits a transaction, so that the standby can be triggered immediately when new log records come. The active sends trigger messages to the standby over a TCP socket.

The standby also needs to know the *log sequence number (LSN)* of the latest checkpoint. The standby needs this message to (1) manage the buffer pool, and (2) to judge whether the standby has fallen too far behind the active and can therefore see an inconsistent view of the log. Specifically, this LSN is used to judge whether the latest LSN written by the active is going to catch up in the circular log file with latest LSN replayed by the standby. We will return to this point in detail later in this chapter. Therefore, whenever the active completes a checkpoint, it sends the LSN of that checkpoint to the standby over the network.

When the standby receives a trigger message, it reads the most recent log records in the log file and begins replaying them.

4.2 Managing Non-shared Data

The active and standby in CAC-DB reside on a shared storage tier, and the shared storage can have only one writer at any time in order to ensure consistency. During normal operation, this writer is the active. In the next section, we present a buffer pool management technique that enables the standby to replay the log and manage its buffer pool without writing to the database. This approach takes care of database data, but a DBMS needs to write much more than database data. For example, a DBMS needs to write information related to its identity like server name or IP address. A DBMS also needs to write diagnostic and error logs that assist an administrator in tuning and problem solving. As part of query processing, a server may write temporary intermediate data such as hash tables for hash joins. It is not possible for the standby to completely avoid writing this type of server-specific, private data.

Since the data written by the standby is server-specific, we do not need to store it as part of the shared data (shared database and shared log file) on shared storage. Thus, CAC-DB divides all data into two categories: shared data and non-shared data.

- **Shared data**: This data encompasses the database and the transaction log. It is shared between the active and standby and should be stored in the shared storage tier. The active is the only writer of the shared data and the standby can only read this data.
- **Non-shared data**: This is sever-specific data such as the examples given above. This data is specific to the primary or standby, so it is stored on local storage, or in shared storage but separate from the database and log.

We use MySQL as the DBMS in CAC-DB. In MySQL, there are several kinds of non-shared data and we present how CAC-DB manages this data. First, MySQL stores performance data in the *performance_schema* database, and tables in this database are views or temporary tables that use no persistent on-disk storage. MySQL will recreate them when it restarts after a failure. Therefore, this kind of non-shared data does not need to survive a failure. Second, MySQL stores metadata in the *information_schema* database, which is similar to the *performance_schema* database. Third, MySQL creates internal temporary tables while processing queries. MySQL creates a temporary table initially as an in-memory table, then converts it to an on-disk table if it becomes too large. If a failure happens, MySQL will rerun the query and recreate the associated temporary data structures. Therefore, this kind of non-shared data does not need to survive a failure either, and it is stored on local storage. Fourth, MySQL writes error messages to the console or to an error log file. If the error messages are written to a file, and this error log needs to survive a failure, CAC-DB stores this data in shared storage as a separate column family.

The approach adopted for MySQL can be adopted for any other DBMS: ephemeral non-shared data (that does not need to survive a failure) can be stored in local storage, while non-shared data that needs to survive a failure can be stored in shared storage in a server-specific area separate from the shared database and log. In any DBMS that we want to use for CAC-DB, it should be easy to distinguish in the DBMS server code between writes to the database and log on the one hand (shared data), and writes to server-specific data on the other hand (non-shared data).

4.3 Buffer Pool Management in the Standby

A DBMS, whether it is the active or the standby, needs to flush dirty pages periodically when it evicts them from its buffer pool during normal operation. However, the CAC-DB design relies for consistency on the fact that only the active writes the database and log. The standby can read the database and log, but it never writes to shared data to avoid introducing inconsistency. This means that the standby cannot flush dirty pages, which requires changing the buffer eviction mechanism for the standby. The general idea of our solution is that we don't evict pages from the end of buffer pool eviction list (e.g., the LRU list), if these pages may cause inconsistency (we illustrate this with an example below). Instead, we can only evict a buffer page when that page is safe in shared storage. A safe page can be evicted without being flushed, so the standby can avoid all writes to the database.

We present the details of our algorithm in the context of InnoDB, which we use for CAC-DB. Other DBMS will require different implementation details, but the idea of evicting only safe pages will still apply. In InnoDB, there are three lists of pages which are used in buffer pool management. The *flush list* is a list that consists of all the dirty pages in the buffer pool. A background thread flushes the dirty pages from the end of flush list periodically. The pages in the *LRU list* and the pages in the *free list* together contain all the pages in the buffer pool. The buffer pages that have not been used recently are evicted when space is needed to cache newer pages, and this is done using the LRU list. The evicted pages from the end of the LRU list include dirty pages and clean pages. If the evicted pages are dirty, they are flushed to storage. When the DBMS needs a free frame to load a page, it uses one from the free list.

In order to prevent the standby from writing database pages to the shared storage tier, we have to suppress flushing buffer pages from the flush list and LRU list at the standby. The standby has to replay the shared log, which causes it to touch different database pages, so its buffer pool will eventually fill up and thus needs an eviction policy. We cannot simply evict pages from the end of the LRU list or the flush list, since that will cause inconsistency problems. For example, in Figure 4.1, cp_1 represents the latest checkpointLSN received by the standby from the active, and *replay_lsn* represents the point to which the standby has replayed the shared log. At the end of this timeline, we are sure that $P2$ with $update1$ and $P1$ with $update2$ were flushed (by the active) to the storage tier because their updates are before cp_1. However, the active cannot ensure that $P1$ with updates after cp_1 is flushed to the storage tier. Therefore, if the standby evicts $P1$ after applying $LSN = 60$, when the standby applies the log record with $LSN = 70$, it will need to load $P1$ from the shared storage tier. However, because the active cannot ensure that $P1$ with $LSN = 60$ is flushed to the storage tier, the standby may get $P1$ with $LSN = 20$, which misses $update3$, $update4$, and $update5$. This causes inconsistency. On the other hand, $P2$ can be evicted by the standby since we are sure that the latest LSN applicable to this page is reflected in shared storage. That is, $P2$ is *safe* while $P1$ is *unsafe*.

In order to determine which pages can be evicted and which cannot, we first define *safe updates* and *safe pages*.

Definition 2 ***Safe Updates***: *Safe updates are updates that happen before the latest checkpoint at the active. For example,* $update1$ *and* $update2$ *in Figure 4.1 are safe. Because updates before a checkpoint have been flushed to storage, these updates can be recovered by reading the page, so they are safe.*

Definition 3 ***Safe Page***: *A page whose updates are all safe updates is a safe page. In*

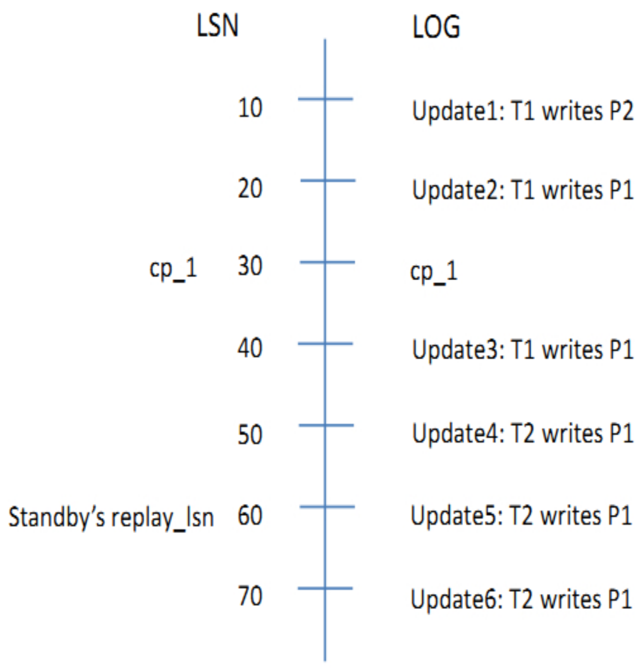

Figure 4.1: Safe Page Eviction

Figure 4.1, page $P2$'s updates are all before cp_1, so $P2$ is a safe page. Since there are updates on $P1$ after cp_1, $P1$ is not a safe page.

In CAC-DB, only the active can take a checkpoint and periodically send the checkpointLSN ($cpLSN$) to the standby. The standby compares the LSN of an update with the latest received $cpLSN$. If an update's LSN is less than or equal to the latest received $cpLSN$, this update is a safe update. If a page's pageLSN ($pLSN$), which records the latest update applied to this page, is smaller than or equal to $cpLSN$, this page is a safe page.

In order to evict only safe pages, we design a new algorithm to manage the LRU list and free list of the standby. Upon receipt of a checkpoint LSN from the active, the standby scans the LRU list and checks the pageLSN of each page on the list to identify safe pages. The standby changes the state of all safe pages to a state that indicates that this page can be replaced, for example, by changing the dirty bit from dirty to clean and removing the page from the flush list. Updated safe pages can now be evicted from the buffer pool.

In our solution, not every receipt of a checkpoint LSN will cause the standby to update the LRU list. The standby updates the buffer pool lists only when there is not enough space in the free list to assign to newly read pages. The standby updates the LRU list and free list and does page evictions only when it needs to.

In our solution, the standby evicts the smallest number of pages possible. For example, if the standby has updated 10 safe pages in the LRU list and the free list needs 2 more pages, the standby will evict exactly 2 pages in order to maintain the utilization of the buffer pool at the standby.

There may be scenarios where the free list is not enough to accommodate new pages and all pages in the standby's buffer pool are unsafe. In this case, the standby can either stop log replay until the next checkpoint arrives so that more pages can be identified as safe, or the standby can ask the active to take a new checkpoint immediately. In our implementation of CAC-DB, the standby asks the active to take a new checkpoint immediately when faced with this situation. The likelihood of this situation arising is related to the buffer pool size of the standby and the size of the log file. The larger the size of the log file, the less frequently checkpoints have to be taken, and the more frequently this situation arises. If the standby has a small buffer pool and the log file is large, this situation will happen relatively frequently. If the buffer pool of the standby is large and the log file is also reasonably small, this situation will not happen very often.

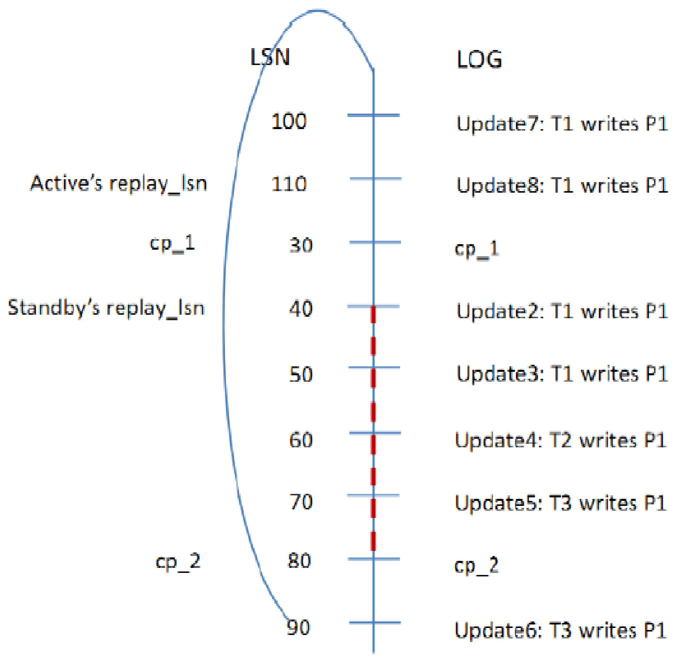

Figure 4.2: Evicting Unsafe Pages at the Standby

4.4 Handling a Slow Standby

The previous section provides a basic HA solution in shared storage environment. However, this basic solution works only under the assumption that the standby can always catch up with the active by replaying all the updates the active has recorded in the shared log. If the standby falls behind the active, in this section, we present techniques for handling this situation.

First, we start with a precise definition of the standby being "caught up" with the active. In modern DBMS, the transaction log is typically implemented as a number of files arranged as a circular file. This is done to keep the amount of space used by the log files bounded. For example, in InnoDB, there are two log files by default in each log group. When InnoDB starts, it writes log records in the first log file. When the first log file is filled, InnoDB begins writing in the second log file. When both log files are filled, InnoDB will return to the start of the first file and overwrite its old log content. InnoDB can overwrite old log content since all the log records before the latest checkpoint LSN are reflected in stable storage and hence will never be used, so they can be overwritten (the log replay procedure in the redo stage of ARIES recovery starts from the latest checkpoint LSN). There is a mechanism to guarantee that new log records will not overwrite log records that have not been checkpointed.

The inconsistencies caused by the standby falling too far behind the active arise if the standby is so slow in applying log records that the active overwrites log content which the standby has not yet replayed. For example, in Figure 4.2, all the log files are filled and the active wraps around and writes log records from the start of the log file. In the figure, the active has written the log record with $LSN = 110$. At that time, the standby has replayed the log only up to the record with $LSN = 40$. If the active continues writing the shared log, the new log records will eventually overwrite log records that the standby has not replayed. If this happens, the standby cannot replay the shared log correctly, causing inconsistency.

It is likely that the standby will not be able to catch up with the active using the default DBMS setting, since the standby issues IO at a lower bandwidth than the active. The active serves multiple concurrent clients that generate several IO streams. In contrast, the standby replays the log sequentially, using only one IO stream. It is likely that the single IO stream of the standby will not be fast enough to keep up with the multiple IO streams of the active.

The specific manifestation of this problem in InnoDB is as follows. InnoDB uses the *asynchronous IO (AIO)* mechanism. In this mechanism, several AIO arrays are used to store AIO requests, and each AIO array is associated with a read/write IO thread. Since we use storage that is accessed over a network instead of local disk, there is a large variance in IO completion time, from single milliseconds to hundreds of milliseconds. The standby performs log replay in batch mode, , where a batch of log records is read and the database pages corresponding to these log records are then read into the buffer pool by IOs that are issued in one batch using the AIO arrays. The standby must finish processing one batch of log records completely before it can load another batch. For every batch, each AIO array has nearly the same number of AIO requests, but the completion time for processing all the AIO requests in different AIO arrays will vary due to variance in IO latency. The faster IO threads have to wait for the slower IO threads, so IO bandwidth is not effectively utilized and the completion time is determined by the slowest thread. The situation does not arise in the active because the different IO threads can work independently without impacting one another. Because of this difference, the IO processing speed in the standby is slower than that the active, causing the standby to fall behind the active.

We present three ways to address the problem in the next sections.

4.4.1 Increasing IO Threads at the Standby

Since the standby falls behind the active due to having less IO intensity, a simple solution is to increase the IO intensity of the standby. In InnoDB, this can be done by increasing the number of IO threads and corresponding AIO arrays. The default number of IO threads and AIO arrays in InnoDB is 4. In CAC-DB, we increase this number at the standby to 10. With this change, we find in our experiments that the standby can easily keep up with the active. Furthermore, the additional IO load generated by the standby does not interfere with the active, so the throughput of the active is the same as it was before. Because the standby does not fall behind the active, no inconsistency arises. In addition, when the standby detects the failure of the active, it always finds in our experiments that it has replayed all of the logs (i.e., the ARIES redo stage is over) and immediately begins ARIES undo almost no system downtime. Furthermore, there is no situation in which the standby must evict extra pages in the buffer pool (as required by the second solution which we describe in the next section), so the buffer pool remains warm and the standby can quickly reach peak throughput.

After failover, the standby decreases the number of read IO threads to the original number (4 by default). Specifically, when the standby finishes the redo stage of ARIES after failover, the AIO arrays are empty. Thus, we destroy these AIO arrays and the associated IO threads, and we re-create the original number of AIO arrays and associated IO threads. The reason we decrease the number of IO threads when the standby takes over as active is to ensure that the IO intensity of the new active is not too high. More IO threads means more IO capacity, and the DBMS is configured with a certain number of IO threads to achieve a certain IO capacity. We increase the number of IO threads at the standby to address the specific problem outlined above, but when the standby takes over as active, it needs to be restored to the configured IO capacity (i.e., the configured number of IO threads).

In our experiments, we have found this solution to be sufficient. We never encounter any case in which the standby is slower than the active. However, there remains a possibility that such a case may occur. If, for example, the standby is slow due to OS problems, hardware problems, or load placed on the machine by other processes. In this case, increasing the number of IO threads at the standby is not enough and we need an explicit synchronization mechanism between the active and standby. We present two possible approaches in the next two sections.

4.4.2 Active Waits for a Slow Standby

The most direct way to prevent the active from overwriting the log records that have not been replayed by the standby yet is to force the active to wait for the standby synchronously. Specifically, every time the standby loads one batch of log records, it will send the start LSN of this batch to the active. When the active wants to write new log records, it checks whether its latest LSN (LSN_a), plus the length of the new log records (l_{newrec}), will pass the start LSN received from the standby ($LSN_{startbatch}$). If Equation 4.1 is satisfied, the active will block until it receives a new start LSN from the standby, at which time it will check the equation again:

$$LSN_a + l_{newrec} \geq LSN_{startbatch} \tag{4.1}$$

Although this solution ensures proper synchronization between the active and standby, it will reduce the throughput of the active if the standby cannot keep up with it. Our experiments show that the throughput of the active with this additional synchronization can be less than 50% of its throughput without the synchronization. One of the primary objectives of this thesis is to avoid solutions that reduce the throughput of the active during normal operation, so we do not use this solution. Instead, we use the solution described next, which does not affect the active's throughput.

4.4.3 Evicting Unsafe Pages at the Standby

Since we do not want the active to wait for the standby, we developed a solution where the synchronization between the active and standby relies on the standby reacting to the actions of the active without stopping them. In particular, instead of forcing the active to wait for the standby, the standby can skip some log records to catch up with the active. Specifically, upon receipt of $cpLSN$ from the active, the standby can compare $cpLSN$ with its latest replayed LSN. There is an existing mechanism to guarantee that the latest LSN at the active will not catch up with $cpLSN$ when the log wraps around the end of the file. Thus, if $cpLSN$ is larger than the standby's latest replayed LSN, the standby will know that it has fallen behind the active, and it can skip all log records up to $cpLSN$. For example, in Figure 4.2, the standby's latest received checkpoint from the active is cp_2. As cp_2 is larger than the standby's *replay_lsn*, the standby will skip log records from the standby's *replay_lsn* to cp_2. However, simply skipping log records without applying the updates they contain would cause the state of the database pages in the buffer pool to be inconsistent. In Figure 4.2, if the standby does not replay the log records between the standby's *replay_lsn* and cp_2 , *update*3, *update*4, and *update*5 to page $P1$ would be missed, whereas *update*6 may be applied. This would leave the copy of $P1$ in the buffer pool inconsistent (recall that the standby only updates pages in the buffer pool not in storage when replaying the log).

The solution to this problem is to simply evict $P1$ from the buffer pool of the standby, relying on the fact that a consistent copy of $P1$ is available in shared storage and can be read by the standby when needed.

Thus, the synchronization mechanism that we propose, and that we use in CAC-DB, is as follows: If the latest checkpoint at the active is later than the latest replayed LSN at the standby, the standby skips the log records until the latest checkpoint. Skipping log records means that some updates will be missed, making some pages *unsafe*. The standby evicts these unsafe pages from its buffer pool.

Definition 4 ***Unsafe Page***: *A page in the standby's buffer pool that misses some updates due to the standby skipping some log records.*

To implement this synchronization approach, at every checkpoint the active sends an array of pairs to the standby. Each pair represents an LSN and the page number to which the LSN applies. The LSNs represented in this array are the LSNs between the previous checkpoint LSN and the current checkpoint LSN, e.g., $\{(P2, 30), (P1, 40), (P1, 50), (P1, 60), (P1, 70)\}$ in Figure 4.2. Note that the array is ordered by LSN. If the standby decides that log records must be skipped, it will do a binary search and find the (page-no, LSN) pairs with LSN larger than standby's *replay_lsn*, e.g., $\{(P1, 50), (P1, 60), (P1, 70)\}$ in Figure 4.2. For each pair with larger LSN, the standby will check whether the page is in its buffer pool. If so, these pages are unsafe pages and the standby evicts them. For example, in Figure 4.2, assuming $P1$ is the in standby's buffer pool, $P1$ will be deemed an unsafe page and will be evicted. The same page $P1$ on the storage tier reflects *update*3, *update*4, and *update*5. When the standby needs $P1$ again later, it can read it from the storage tier with all the necessary updates. After the standby evicts all of the unsafe pages, it will continue to replay the log.

Although this solution avoids inconsistency and has no impact on the throughput of the active, it remains a concern that evicting pages from the buffer pool of the standby will reduce the degree of warmth of the standby's buffer pool. As time passes, if the number of pages evicted is larger than the number of pages loaded, the standby's buffer pool will become colder and colder. In the limit, we may find that the work that CAC-DB does to replay the log and keep the standby buffer pool warm is all lost due to unsafe page evictions, and that failover in CAC-DB is as expensive as starting a new server after failure and performing ARIES recovery. Thus, this synchronization mechanism should be viewed as a way to ensure correctness if the standby *temporarily* falls behind the active. In the steady state, the standby should have sufficient IO capacity so that it rarely falls behind the active, if at all.

To summarize, in CAC-DB the primary mechanism for ensuring that the standby does not fall behind the active is to increase the IO capacity of the standby (Section 4.4.1). CAC-DB also implements the mechanism of skipping log records and evicting unsafe pages at the standby (Section 4.4.3). Forcing the active to wait for the standby when the standby falls behind (Section 4.4.2) is not considered a good solution since it affects the throughput of the active during normal operation.

4.5 Guaranteeing Consistent Reads at the Standby

This section focuses on techniques for guaranteeing strong consistency for the standby when it reads log and data pages, while avoiding some, or most of, the consistency penalty. The combination of optimistic IO and client-controlled synchronization in DAX guarantees that the active always sees consistent versions when it reads log and data pages that it has written (see Sections 3 and 4 in [23]). What we need to guarantee for CAC-DB is that when the standby reads log and data pages written by the active, the standby sees consistent versions.

As explained in Chapter 2, with optimistic IO the active maintains an in-memory version list to record the most recent version numbers of various blocks. The entries in the version list have the latest versions of the associated blocks. The DAX client library assigns the lastest version numbers to blocks when they are written by the active. However, because the standby cannot write to the shared storage, the entries in a version list maintained by the standby would remain unchanged and would not reflect the latest version of the associated blocks (recall that the latest versions are written by the active). Therefore, optimistic IO for the active cannot work for the standby, and a new technique is required for guaranteeing consistent reads at the standby. If the underlying cloud storage system provides strong consistency, this problem would cease to exist. However, one of the goals of DAX and CAC-DB is to avoid strong consistency, since it is too expensive for a system that is deployed across geographically distributed data centers.

In this section, we develop techniques for guaranteeing consistent reads at the standby without requiring the storage system to raise its consistency level all the way to strong

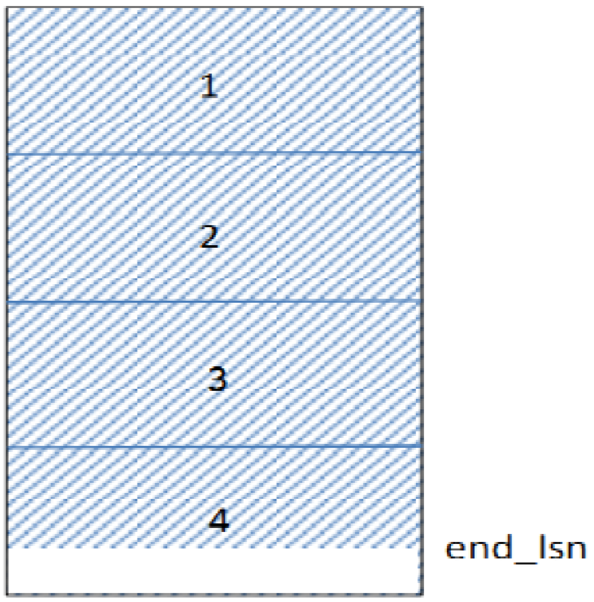

Figure 4.3: Incompletely Written Log Blocks

consistency. We divide data read by the standby into two types: log files and data files. The difference between log and data files is that log files are append-only while data files are not. Because of this difference, we have developed separate techniques to ensure consistent reads of log and data files.

4.5.1 Consistent Reads of Log Files

CAC-DB uses batch-based log replay, in which a log file can be divided into many segments. The standby repeatedly reads one segment from storage into memory, replays the log in this segment, and then reads the following segment.

In the current implementation of InnoDB, if one segment of log files is not fully written (some blocks may be incomplete, e.g., the fourth block in Figure 4.3), the standby will record the end_lsn which represents the standby's replayed position in the log, as shown in Figure 4.3. When the standby reads the next segment, it will read from the last end_lsn, and will replay from that position, not skipping any log records. As log files are append-only, if the standby can record the last replayed position with every read of a log segment, and then replay from that position in the next read of a log segment, the standby will be guaranteed to replay the log in order and not skip any log records. This guarantees the correctness of log reads.

Based on the above, we have developed two solutions to guarantee the correctness of reads from the log file.

SOLUTION 1: The standby can use Read(1), which means that it can potentially get an old version when it reads a new log segment. The standby will replay from the last end_lsn, which records the position to which the standby has replayed the last time, so it will be able to handle stale reads from the log since it would ignore log records that it has previously replayed. After replaying the current log segment, the standby will again record the end_lsn, which is the starting position of log replay for the next log segment read.

Using Read(1) may be sufficient to guarantee correctness of log replay. However, it does not provide strong guarantees that log replay will make forward progress fast enough. If the standby keeps reading stale log segment because of the Read(1) operation, the standby may be much slower than the active and may fall behind. The problems of a slow standby discussed in Section 4.4 would then arise.

SOLUTION 2: A better solution is for the standby to use Read(QUORUM) to read the log file. Read(QUORUM) always obtains the most recent version of the log files, because log writes in DAX are synchronized with at least k replicas as part of commit processing. In this way, the standby can always make forward progress and keep up with the active. In the implementation of CAC-DB, we use this solution, and we have found it to be effective and inexpensive.

4.5.2 Consistent Reads of Data Files

Unlike append-only log files, data files experience multiple updates in different locations. Therefore, the standby cannot use Read(1) and expect consistent reads. To overcome this problem, we have developed a mechanism by which the standby can determine whether a Read(1) has returned most recent version of a block. To achieve this, the standby maintains an in-memory *pageLSN list* in its Dax client library. This is similar to the *version list* used for optimistic IO in DAX [23]. In the pageLSN list, the standby records the latest pageLSN of each page (this pageLSN is not the true largest pageLSN as written by the active, but rather the largest pageLSN seen by the standby). The standby updates the pageLSN of a page in the pageLSN list when it applies a log record to that page during log replay. When a page is read using Read(1), the storage tier returns the first page replica that it is able to obtain, and this returned page contains a pageLSN. The client library at the standby compares the pageLSN returned by the Read(1) with the corresponding pageLSN in the pageLSN list to determine whether the page returned by the Read(1) is stale. The standby does not need the most recent version of a page. If the Read(1) returns a page with pageLSN equal to or larger than the corresponding pageLSN in the pageLSN list, that page is sufficiently up to date for the standby, since the standby will replay the log and update the page. The pageLSN list has a configurable maximum size. If the pageLSN list grows beyond that size, entries are evicted from it using a least recently used (LRU) policy.

The standby will also evict entries from the pageLSN list when unsafe pages are evicted (recall Section 4.4.3). If the buffer pool manager at the standby evicts a safe page, the corresponding entry is not evicted from the pageLSN list. The reason for the difference in treatment between evicted unsafe pages and evicted safe pages is that after an unsafe page evicted, the standby will skip log records to the most recent *cpLSN* to catch up with the active, which means that the corresponding entries in the pageLSN list cannot be used to check for consistency. If the standby does not evict these entries from the pageLSN list and reads a page with the pageLSN larger than the corresponding value in the list but smaller than the most recent *cpLSN*, there is no guarantee that this page is a consistent page. The consistent version of the page in this case is the page with the last update before the latest checkpoint.

For example, in Figure 4.2, the standby skips from its *replay_lsn* to cp_2 to catch up with the active. In Section 4.2, we saw that $P1$ is an unsafe page and is evicted from the buffer pool of the standby. If the standby does not evict the $P1$ entry (with value $LSN = 40$) from the pageLSN list, the following scenario may occur and lead to inconsistency. There are three updates on $P1$ between the standby's *replay_lsn* and cp_2. When the standby replays the log record with $LSN = 90$ and needs $P1$ again, it may read $P1$ with $LSN = 50$ or $LSN = 60$. Because the newly loaded $P1$ is larger than the entry in the pageLSN list, the standby will assume it is a consistent page. However, $P1$ with $LSN = 50$ is not a consistent page. When the standby skips to cp_2, it skips the log records for *update*3, *update*4, and *update*5. If the standby uses the page it reads with $LSN = 50$ or 60, it will miss at least *update*5 whose $LSN = 70$. Since the standby skips log records, it needs to read the most recent version of $P1$, with $LSN = 70$, which is the last update before cp_2. Therefore, the $P1$ entry must be evicted from the pageLSN list.

Since the pageLSN list is maintained in memory, it will be empty when the standby first starts or when it restarts after a failure. The pageLSN list is populated gradually as the standby reads pages. If the standby reads a page for which there is no entry in the pageLSN list, either because the standby (1) has started recently (or restarted after a failure) and is warming up (2) evicted the page as part of unsafe page eviction, or (3) evicted the entry from the pageLSN list because the list got too large, the standby cannot use Read(1), since it will not be able to check the returned version for consistency. To deal with this situation, we propose the concept of a **consistent page**.

Definition 5 ***Consistent Page****: If a page is among the entries of the pageLSN list, then a page in storage with a pageLSN equal to or larger than the corresponding pageLSN in the pageLSN list is a consistent page. Otherwise, a page with the last update before the latest checkpoint is a consistent page.*

When the standby reads a page that is not in its buffer pool, it will always be the case that this is the first time the standby encounters this page after the latest checkpoint. If the standby had previously encountered this page after the last checkpoint, the page would necessarily be in the buffer pool. This is because only safe pages, whose pageLSN is smaller than the checkpoint LSN, can be evicted. Unsafe pages with updates after the latest checkpoint cannot be evicted. If the page being read satisfies the definition of a consistent page, the standby will not skip any updates recorded in the log file for that page.

If a page being read is not in the pageLSN list because its entry was evicted when the list got too large, then the page must have been safely evicted earlier. As a safe page is a page whose last update is before the latest checkpoint, the current read would be the first time the standby encounters this page in the log file after the latest checkpoint. In this case, the standby will also not skip any updates on this page in the log file if the page is consistent according to the definition above.

If a page is not in the pageLSN list or the Read(1) returns a version that is deemed to be stale, we have developed three solutions to guarantee that the standby can obtain a consistent page for the data files.

SOLUTION 1: The standby client library falls back to Read(ALL). It is guaranteed that Read(ALL) will return the latest version of the page if it succeeds, which is definitely a consistent page. However, Read(ALL) is not tolerant to failures of DAX servers. The failure of even a single replica will prevent Read(ALL) operations from completing until that replica is recovered, leading to a loss of availability [23]. This solution is therefore not a good solution.

SOLUTION 2: The standby maintains version list that is identical to that of the active. Every time the active updates the version list, it sends a message to the standby to update its copy of the version list. After an acknowledgement is returned from the standby, the update can be viewed as a successful update. By maintaining the version list, the standby can still use Read(1) and can determine whether a Read(1) has returned the latest version of a page. However, this will negatively affect the performance of the active, especially if the active and the standby are in different data centers. There is a round-trip latency for every update of the version list. Thus, this solution is cheap for the standby, but it is very costly for the active.

SOLUTION 3: Due to the drawbacks described above, we have designed a mechanism that is cheap on both the active and the standby. First, we explain the concept of **durability on shared storage** and **semi-fuzzy checkpoint** in the active.

Definition 6 ***Durability on Shared Storage****: An update is durable on shared storage if update acknowledgements have been received from at least k replicas.*

Definition 7 ***Semi-fuzzy Checkpoint****: The traditional fuzzy checkpoint can guarantee all the updates before the latest checkpoint LSN are durable on disk. In our setting, a semi-fuzzy checkpoint can guarantee all the updates before the latest checkpoint LSN are durable on the shared storage.*

Every time the active takes a semi-fuzzy checkpoint, it ensures that all the updates made before the latest checkpoint are durable. Therefore, after a semi-fuzzy checkpoint, consistent pages are durable on the shared storage with k replicas updated. With this solution, the standby can use Read(QUORUM) to ensure that it obtains consistent pages. In the implementation of a semi-fuzzy checkpoint in CAC-DB, the active directly calls **CSync** every time it needs to take a checkpoint. Note that **CSync** is explained in detail in Chapter 3.

Algorithms 1 and 2 summarizes the final version of the optimistic-IO protocol for data files using SOLUTION 3. This solution is relatively cheap for both the active and the standby, and is the one that we adopt in CAC-DB.

Algorithm 1: Standby Read for Data Files

```
Input: $blockId$
Output: new $pageLSN\_list$, $data$
1  if $pageLSN \in pageLSN\_list$ then
2      $(data, pageLSN) := Read(1)$;
3      while $pageLSN < pageLSN\_list.get(blockId)$ do
4          #retry a stale read;
5          #give up if too many retries and return an error;
6          $(data, pageLSN) := Read(1)$;
7  else
8      #fall back to reading a quorum;
9      $(data, pageLSN) := Read(QUORUM)$;
10 $pageLSN\_list$.put($blockId$, $pageLSN$);
11 return $data$
```

Algorithm 2: Semi-fuzzy Checkpoint by Active

```
1  #blocks until writes on sync_pending list are durable
2  $CSync()$;
3  return $cp\_lsn$
```

Chapter 5

CAC-DB Failover

This chapter describes the active and the standby's operation for failover. Before failover can be triggered, some failure detection approach is needed for the standby to detect whether the active has failed. We use a failure detection approach that was developed independently of this thesis by a co-op student. This failure detection approach is presented in Appendix A. For failover, CAC-DB relies on the ARIES recovery protocol.

5.1 Failover

When the standby detects that the active has failed, the standby will start the failover process. Failover relies on the ARIES recovery algorithm, which consists of redo stage followed by an undo stage. Usually, the standby has already finished the redo stage when failover starts, because the standby is always working to catch up with the active during normal operation. Moreover, there are a few seconds of failure detection in which the active does not generate new log records, and this time is usually sufficient for the standby to finish the redo stage. After redo, the standby must do two things: (1) decrease the number of read IO threads it uses, and (2) update the buffer pool. The need to decrease the number of read IO threads was explained in Section 4.4.1. For updating the buffer pool, the standby scans the LRU list and updates all safe pages to the replaceable state.

After these steps, the standby becomes the new active and can accept new transactions. Many database systems, including InnoDB which we use in this thesis, can perform the undo part of recovery while accepting new transactions at the same time. The DBMS simply keeps locks on the objects accessed by the undo stage.

One issue we have observed is that, the new active needs to clean space in the log for the new log records of the new workload. To clean log space, the new active must flush dirty pages from its buffer pool synchronously or asynchronously, and create a new checkpoint. In InnoDB, the trigger points for synchronous flushing and asynchronous flushing are different threshold ratios for the empty log space in the log file. We have found that the new active typically finds log space almost full at the end of the redo stage, and performs synchronous flushing before accepting new work. In the period of synchronous flushing, the system is not processing any transactions, so this period is part of the system downtime at failover.

To avoid the long system downtime at failover, we provide a solution to prevent the new active from reaching the trigger point for synchronous flushing. We have found the new active performs synchronous flushing because the trigger points for asynchronous flushing and synchronous flushing are too close. Therefore, we lower the threshold for asynchronous flushing so that it is not too close to the threshold for synchronous flushing. Using this approach, the DBMS performs more frequent asynchronous flushes but will almost never need synchronous flushing. At failover, the new active will make asynchronous flushing instead of synchronous flushing, which eliminates downtime for flushing. In our implementation of CAC-DB, we changed the threshold for asynchronous flushing from the default value of $\frac{3}{4}$ to $\frac{1}{2}$.

In addition, CAC-DB can handle the case in which the standby fails, rather than the active. First, our failure detection algorithm can easily be used to detect the failure of the standby. To support failure detection of the standby, we create a reliable counter in DAX for the active/standby pair. This counter is advanced by the standby and read by the active to ensure that the standby is still alive. Second, when the active detects that the standby fails, CAC-DB will immediately deploy a new standby, which begins to replay the log from the latest checkpoint LSN in the shared log file and soon catches up with the active. This reprotection operation is also used after failover: the old standby becomes the new active, and CAC-DB creates a new standby for this active. We describe the reprotection operation next.

5.2 Reprotection

Reprotection is used to launch a new standby for the new active after failover or when the standby fails during normal operation. Reprotection in CAC-DB can tolerate the failure of a data center in a multi-data-center setting, making use of the property that DAX can be deployed across several data centers. The database and the log are replicated by DAX in multiple data centers, so CAC-DB has flexibility in choosing the data center in which it creates the new standby. For example, assume that the active is in data center A and the standby is in data center B. If the entire data center A fails, the standby in data center B can perform the failover and become the new active. Simultaneously, when the old standby detects that the old active has failed, CAC-DB immediately starts a new standby in data center C. This is possible since DAX enables the database and log to be accessible from any data center in which the system is deployed. This time from the instant that CAC-DB starts a new standby to the instant when the new standby begins the log replay is typically very short (around 12s in our experiments). Thus, there is only a short time in which the system is not protected against failures.

Chapter 6

Experimental Evaluation

We conducted an experimental evaluation of CAC-DB with several objectives. First, we want to measure the performance of CAC-DB during failover. We show that CAC-DB provides fast failover and compare its performance to MySQL Binlog replication. Second, we want to test the performance of CAC-DB under different settings of the environment. To do this, we conducted experiments under different buffer pool sizes and database sizes. Third, we illustrate the performance of different design alternatives in our system.

6.1 Experimental Setup

We conducted all of the experiments described in this thesis on Amazon's Elastic Compute Cloud (EC2). Note that EC2 is widely used to host data-intensive services of the kind that we are interested in supporting, so it is a natural setting for our experiments. Moreover, EC2 allows us to run experiments that use multiple data centers. All of our experiments use EC2 large instances (virtual machines), which have 7.5 GB of memory and a single, locally attached 414 GB storage volume holding an ext3 file system. The virtual machines run Ubuntu Linux with a 2.6.38-11-virtual kernel. Our solution is based on the DAX prototype, which is based on Cassandra version 1.0.7, and we use MySQL version 5.5.8 with the InnoDB storage manager as the DBMS tenant.

All of the experiments ran a TPC-C transaction processing workload generated by the Percona TPC-C [7] toolkit for MySQL [8]. We used a 100-warehouse TPC-C database instance with an initial size of 10 GB. The database grows during a benchmark run due to record insertions. For each DBMS, 50 concurrent TPC-C clients were used to generate the

DBMS workload. The performance metric of interest in our experiments is throughput, measured in TPC-C NewOrder transactions per minute (TpmC).

6.2 Behavior of CAC-DB During Failover

In this experiment, we demonstrate the effectiveness of CAC-DB during failover. We consider two distributed CAC-DB configurations. In each configuration, there is one EC2 server running the active MySQL instance, one EC2 server running the standby MySQL instance, and nine EC2 servers running as DAX servers and providing the CAC-DB storage tier. Each database block is replicated six times across the nine DAX servers, for a total data size of approximately 60 GB. The configurations that we test are as follows:

- **One zone:** All eleven servers are located in the same EC2 availability zone (roughly analogous to a data center). For our experiment, we used the US East (N. Virginia) availability zone.
- **Different regions:** The nine DAX servers are distributed evenly over three EC2 availability zones. All three zones are located in geographically-distributed EC2 regions: US East (N. Virginia), US West-1 (N. California), and US West-2 (Oregon). US East houses the active MySQL instance, while US West-2 houses the standby MySQL instance.

In each EC2 configuration, we tested two variants of CAC-DB in addition to DAX without CAC-DB. The different systems tested are as follows:

1. DAX: The original DAX, which supports recovery after failure but no active/standby high availability.
2. CAC-DB without optimization (CAC-DB-WO): CAC-DB without the optimization to reduce synchronous flushing.
3. CAC-DB with optimization (CAC-DB-W): CAC-DB with the optimization to reduce synchronous flushing.

In addition, we perform experiments to compare CAC-DB with MySQL's integrated replication solution, Binlog [5]. For Binlog, we consider three distributed Binlog configurations. In each configuration, there is one EC2 server running the active MySQL instance and one EC2 server running the standby MySQL instance. For the storage tier, we use either the local storage of the machines, where the MySQL server resides, Amazon's Elastic Block Storage service (EBS), or DAX running on other EC2 servers, as the storage tier. In Binlog replication, the active and standby have separate copies of the database, so their storage tiers are independent.

We tested Binlog replication in the one-zone and different-regions settings, as we did with CAC-DB. In each setting, we tested six versions of MySQL Binlog replication. The Binlog versions are as follows:

1. MySQL asynchronous Binlog with local disk (LAS): In this configuration, we use MySQL's asynchronous replication, in which the active writes events to its binary log but does not synchronize with the standby to ensure that it has retrieved and processed them. We use the local storage of the EC2 instance as the storage tier.

2. MySQL semi-synchronous Binlog with local disk (LS): Similar to the configuration above, except that we use MySQL replication's semi-synchronous replication, in which the standby acknowledges receipt of a transaction's events only after the events have been written to its relay log and flushed to disk. The active waits to receive the acknowledgement from the standby.

3. MySQL asynchronous Binlog with Amazon EBS service (EAS): Asynchronous replication, using a single EBS volume for each MySQL instance as the storage tier.

4. MySQL semi-synchronous Binlog with Amazon EBS service (ES): Similar to the configuration above, except that we use MySQL's semi-synchronous replication.

5. MySQL asynchronous Binlog with DAX (DAXAS): In this configuration, we use MySQL's asynchronous replication. We use a separate EC2 instance implementing a DAX storage tier for each of the active and standby MySQL instances.

6. MySQL semi-synchronous Binlog with DAX (DAXS): Similar to the configuration above, except that we use MySQL's semi-synchronous replication.

In the next two sections, we describe experiments for DAX, the two versions of CAC-DB, and the six versions of MySQL Binlog replication solutions. We compare their behavior during failover in the one-zone and different-regions configurations. MySQL is configured with a 5G buffer pool. We run the TPC-C benchmark and plot throughput in transactions per minute (TpmC). The test was run for more than one hour before the failure and more than one hour after the failure. A failure of the active was simulated by stopping the EC2 instance.

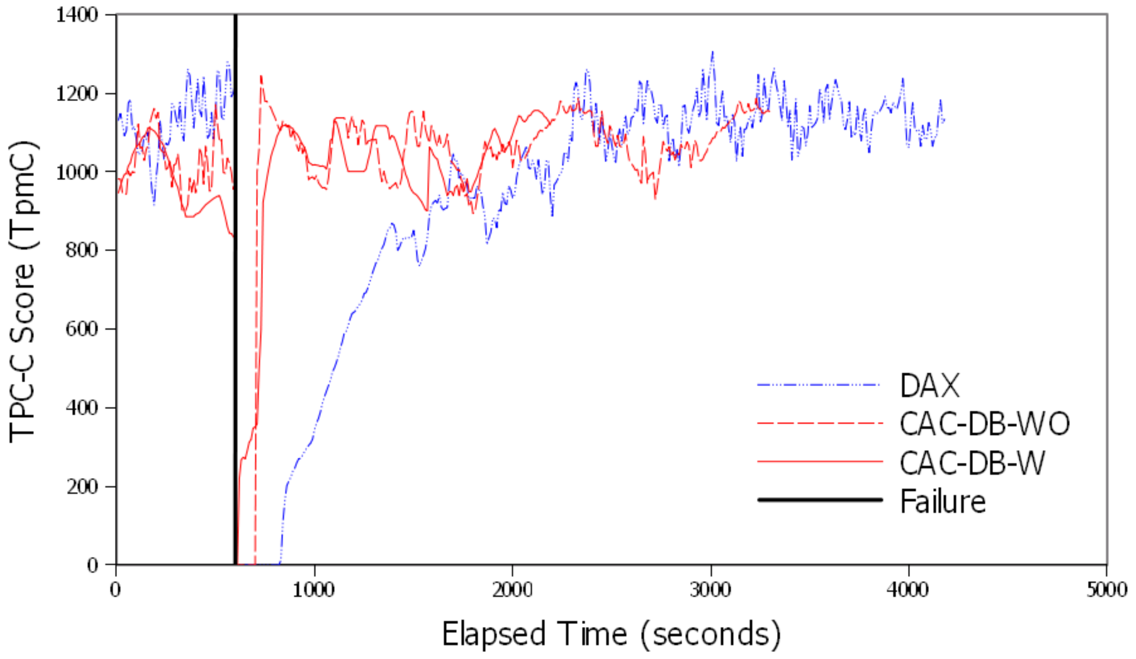

Figure 6.1: Comparison Between DAX and CAC-DB In One Zone

6.2.1 Failover in One Zone

We compare DAX and CAC-DB in the one-zone configuration in Figure 6.1. In the figure, the failure happens at 600s and we show throughput before and after the failure. The throughput shown is the average throughput for a sliding window of 60 seconds. Before the failure, all three methods show similar throughput, ranging from 900 to 1200TpmC. This is expected since the three methods use the same type of DBMS and the same storage tier. After failure, the CAC-DB variants reach their pre-failure performance in a short period. DAX also reaches its pre-peak performance, but it does so after a considerable delay. DAX recovers from a failure by starting a new MySQL instance and performing full ARIES recovery, which takes time and incurs substantial down time.

To observe performance during failover, we zoom in on the period from 500s to 1000s in Figure 6.2. To show the detailed behavior during failover, the throughput in Figure 6.2 is the average throughput for a sliding window of 10 seconds. CAC-DB-WO incurs around 100s of down time, during which it is taking a synchronous checkpoint. After these 100s, it reaches peak throughput almost instantaneously. CAC-DB-W starts accepting new transactions much sooner than CAC-DB-WO, but it reaches its peak throughput more

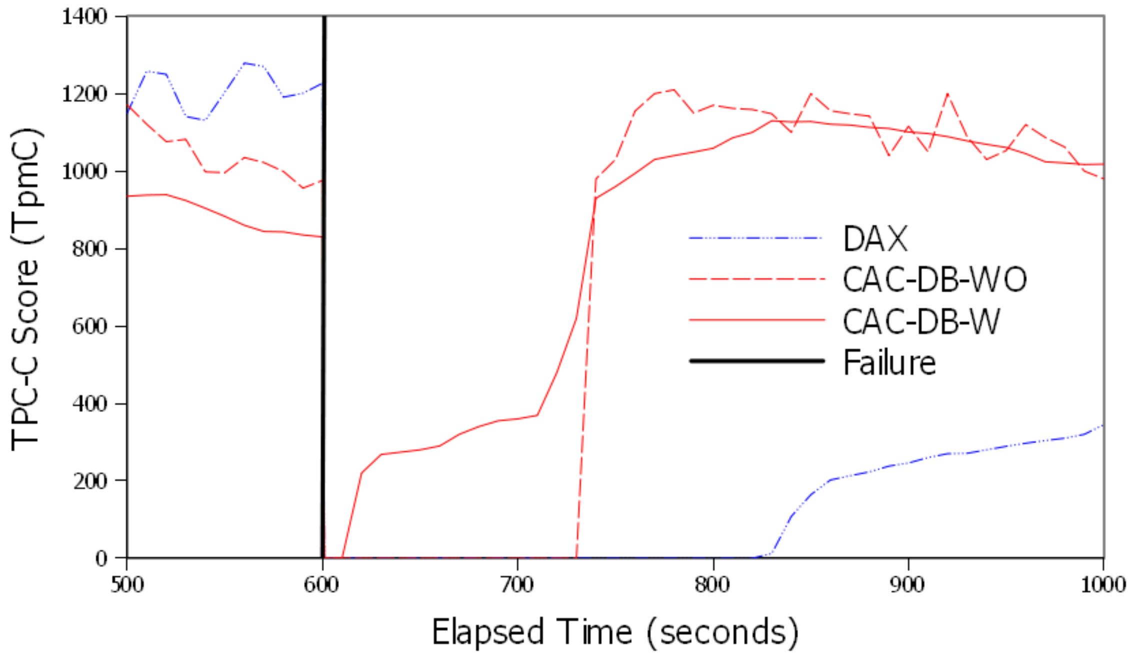

Figure 6.2: DAX and CAC-DB During Failover in One Zone

gradually since it needs several asynchronous checkpoints. We believe that CAC-DB-W is the better alternative in this tradeoff, since it incurs less downtime and is not much slower than CAC-DB-W to recah its peak throughput. DAX is much slower than these two alternatives. By 1000s, it is still slowly warming up its buffer pool.

Figure 6.3 compares results between Binlog replication solutions and CAC-DB-W in the one-zone configuration. First, we compare the throughput before failure. We make two observations about the performance of Binlog replication. First, the throughput of the asynchronous Binlog solutions is higher than that of the semi-synchronous Binlog solutions, because semi-synchronous Binlog solutions put an overhead on the active's throughput. In the semi-synchronous Binlog solution, for every transaction commit, the active cannot continue executing subsequent statements until it receives the acknowledgement from the standby. Second, the throughput of EAS/ES pairing is higher than LAS/LS pairing. We observe that EBS provides better IO performance than local storage, and we study this further in Section 6.4. Third, the throughput of DAXAS/DAXS pairing is higher than LAS/LS pairing because DAX uses disk striping to increase random IO speed. Throughput after failover is higher since the system is not protected so it does not incur the cost of protection: the old standby becomes the active after the failure, and that new active does

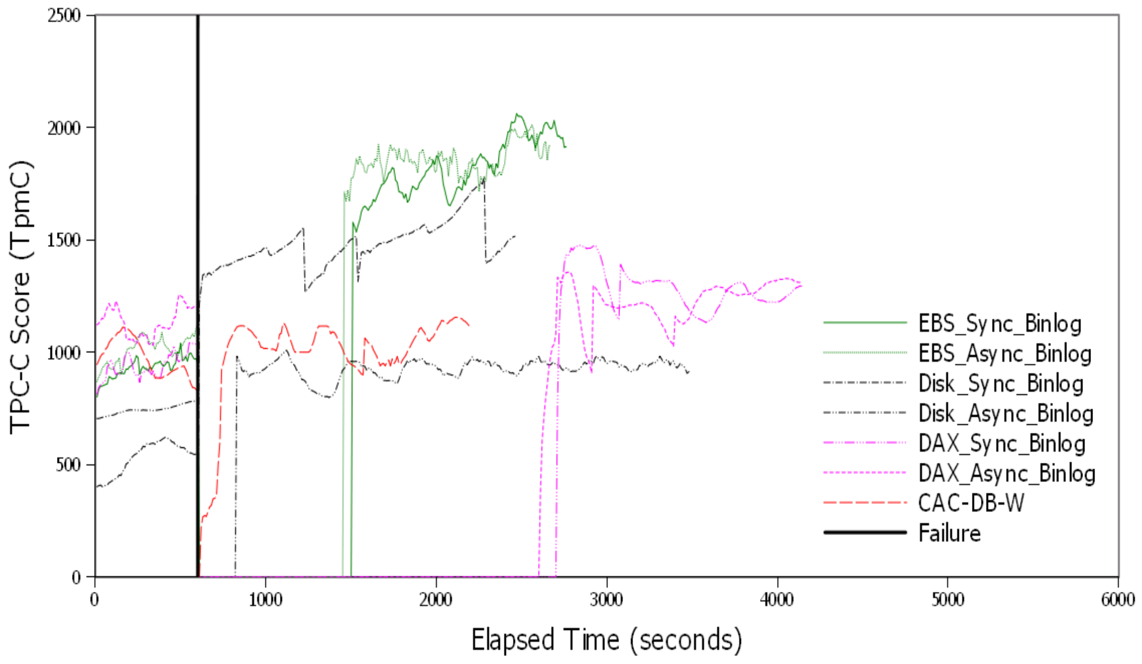

Figure 6.3: Comparison Between MySQL Binlog Replication Solutions and CAC-DB-W in One Zone

not have a standby so does not pay the cost of Binlog replication.

In order to compare the failover performance of different solution, we list several metrics related to failover in Table 6.1. The second row in the table lists downtime in seconds. This is the time from the failure to the time when the new active can accept transaction. The third row in the table shows how far behind the standby is when the active fails. We use the size of the residual relay log that has not been replayed by the standby at the point of failure to measure the standby's lag behind the active. This measure is of relevance only to the Binlog replication solutions, since the other solutions do not use a relay log.

In Table 6.1, we compare DAX with the two versions of CAC-DB. DAX incurs longer downtime and takes longer to reach peak throughput than the CAC-DB methods. This is because DAX needs to launch a new DBMS instance, and the new DBMS instance must go through its recovery procedure, which leads to the longer downtime. DAX takes a longer time to reach peak throughput, because the new DBMS instance needs to warm up the cold buffer pool. CAC-DB takes a relatively short downtime because by the time the standby detects the failure of the active, the standby has usually already finished replaying the transaction log and has thus almost finished the recovery procedure. CAC-DB takes a shorter time to reach peak throughput because the standby becomes the new active with a warmed-up buffer pool. Thus the value of CAC-DB to improve availability is clearly demonstrated.

One-Zone	DAX	CAC-DB-WO	CAC-DB-W	LAS	LS	EAS	ES	DAXAS	DAXS
downtime(s)	220	130	4	220	2	850	900	2000	2100
time to peak throughput (s)	1480	30	180	0	0	0	0	0	0
lag-behind (MB)				18	0.002	79	84	178	179

Table 6.1: Failover Time Comparison for One-Zone

Table 6.1 also compares CAC-DB-WO and CAC-DB-W. As we saw before, CAC-DB-WO has a longer downtime, because CAC-DB-WO needs to flush the dirty pages of the buffer pool synchronously to clear log space for new log entries, during which the new active cannot accept transactions. However, after the synchronous flush, the throughput of CAC-DB-WO can reach peak shortly because of the warmed-up buffer pool. Instead of synchronous flushing, CAC-DB-W uses asynchronous flushing, during which the new active can accept new transactions. Therefore, the new active in CAC-DB-W can accept new transactions as soon as it finishes the recovery (in fact, it is almost always done with recovery by the time the failure is detected). Therefore, the downtime is very close to the failure detection time (approximately 4s). However, repeated asynchronous flushing involves more total work than the synchronous flush, so it takes about 180s for the throughput to reach its peak value. We believe that the shorter downtime of CAC-DB-W makes it the preferred choice. In Section 6.3 and later, we only do experiments with CAC-DB-W.

Table 6.1 also compares the Binlog replication solutions. Observe that LS shows a very short downtime followed by LAS. In contrast, the EAS/ES pairing and DAXAS/DAXS pairing show much longer downtimes. We find that the downtime in the Binlog solutions is proportional to the size of the residual relay log that the standby has not replayed by the time the standby detects the failure. The size of the residual relay log is determined by the speed difference between the active and the standby. The Binlog based solutions on the standby side function as follows: There are two threads on the the standby side. The IO thread receives the binary log from the active and writes the relay log to the standby's storage. Then, the SQL thread reads the relay log from the storage and replays them. The IO workload from the binary log puts an overhead on the standby. In LS, there is a very short downtime, because the low throughput of the active means that it is slow enough for the standby to always keep up with it. In LAS, there is a longer downtime, because the active does not wait for the standby during normal operation, so there is a large residual relay log that the standby has not replayed when the standby detects the failure. In EAS/ES and DAXAS/DAXS, there are much longer downtimes, because the sequential IO operations needed to read and write to the binary log occur over the network, which is slower than the local disk and therefore slows down the standby. When the active fails, the standby is typically far behind the active and a substantial amount of log data has not been replayed.

Table 6.1 quantifies the tradeoff that we expect to see in the Binlog solutions. If we want short downtime, the standby must keep up with the active during normal operation so its throughput would be low (LS). If the active runs faster, the standby has more work to do after failure, so downtime is higher. We have seen that synchronous and asynchronous solutions are comparable in this experiment. The differences between them will be more pronounced in the different-regions experiment in the next section. The main conclusion from the Binlog experiments is that CAC-DB avoids the problems of the different Binlog solutions. CAC-DB has throughput during normal operation that is comparable to the best Binlog solutions. After failure, it recovers faster than these solutions because it relies on shared storage that is updated efficiently. The only Binlog solution that recovers faster than CAC-DB is LS, which has poor throughput during normal operation. Thus, we see that CAC-DB offers an alternative to log-shipping that has better performance. In addition CAC-DB has the following advantages over Binlog replication:

- CAC-DB never loses committed transactions upon a failure, whereas Binlog replication – even semi-synchronous replication – may lose transactions.
- CAC-DB maintains only one copy of the database, whereas Binlog replication maintains two independent copies, one at the active and one at the standby.
- CAC-DB does not require the administrative overhead of setting up log shipping, which is required for Binlog replication.

6.2.2 Failover in Different Regions

In this section, we study performance in the different-region configuration. We compare DAX and the two variants of CAC-DB in Figure 6.4. We compare CAC-DB-W and Binlog

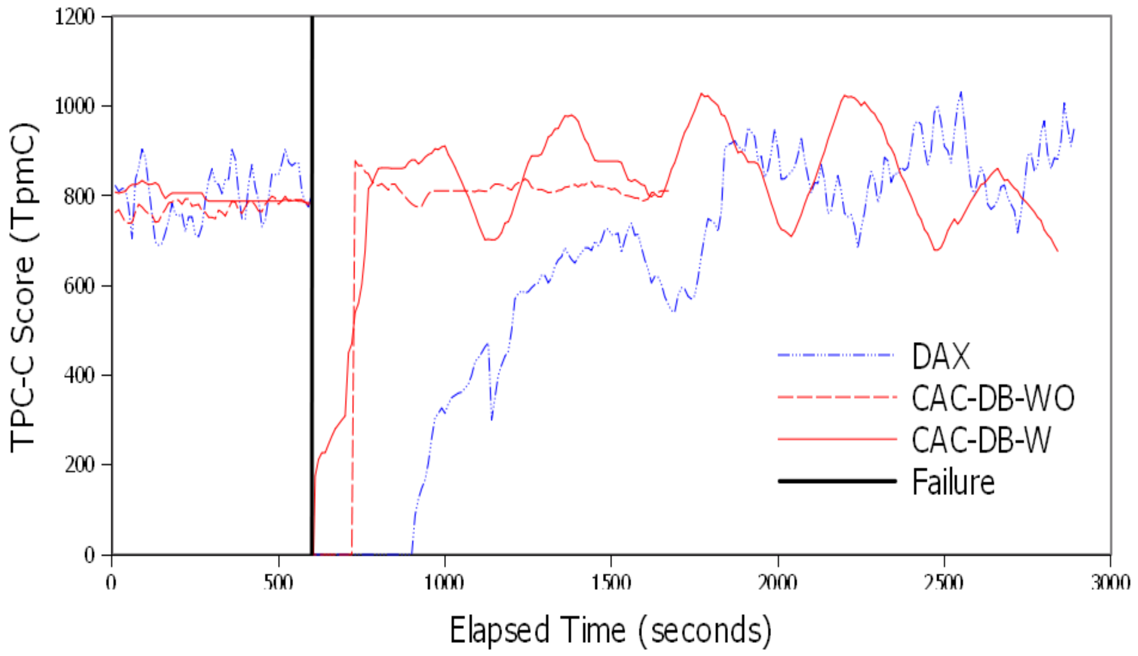

Figure 6.4: Comparison Between DAX and CAC-DB in Different Regions

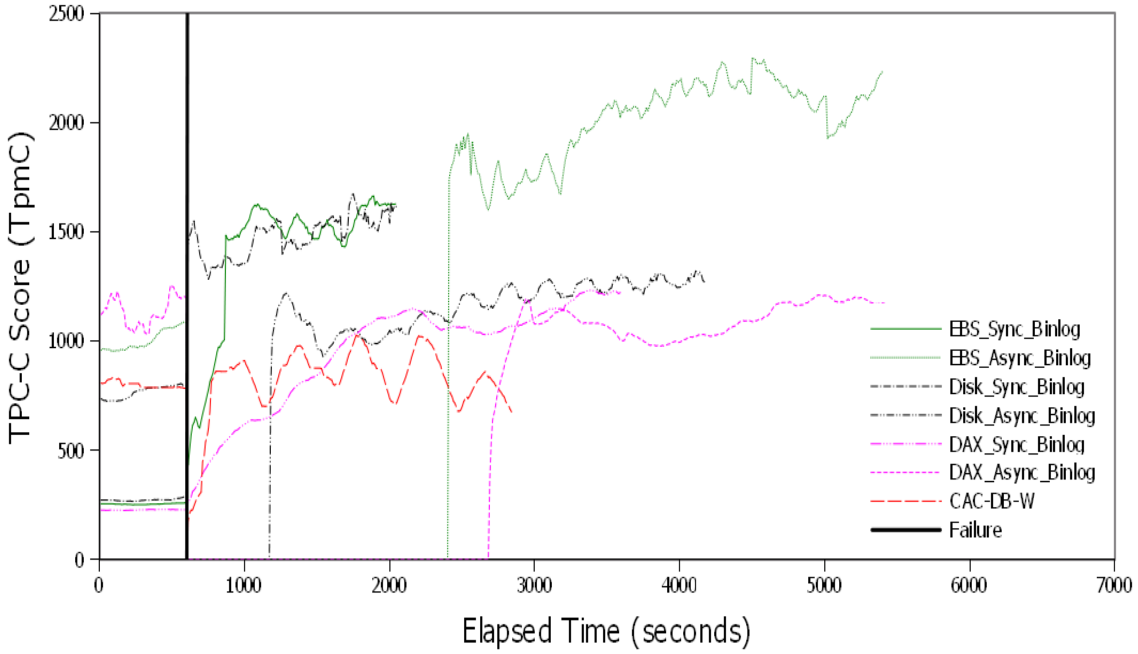

Figure 6.5: Comparison Between MySQL Binlog Replications and CAC-DB-W in Different Regions

replication in Figure 6.5. In these two figures, the active fails at 600s from the start of the measurement period. In this experiment, we put the active in the US East zone and the standby in the US West-2 zone. We list the downtime, the time that the new active takes to reach peak throughput, and how far the standby lags behind the active in Table 6.2. We find that all of the solutions show behavior similar to that shown in the one zone configuration, except that the throughput in the different-region configuration is slightly lower than that in the one zone configuration, due to the inter-region latencies.

Looking at these results in more detail, we see from Figure 6.4 that the two variants of CAC-DB outperform DAX, and we see that CAC-DB-W remains the preferable choice since it does not incur a long downtime. For Binlog replication, we see from Figure 6.5 and Table 6.2 that the semi-synchronous solutions (LS, ES, and DAXS) perform very poorly before the failure, with throughput only one third of that of CAC-DB. In semi-synchronous Binlog solutions, for every transaction commit, the active cannot execute the next statement until it receives an acknowledgement from the standby that the transaction has been propagated. Because the active and the standby are in two different geographic regions, the long inter-region network latencies place a very high overhead on the throughput of the active. The difference between asynchronous Binlog solutions and synchronous Binlog solutions in the different-region configuration is much larger than in the one zone configuration. In addition, LAS, EAS, and DAXAS show similar throughput to the corresponding value in the one zone configuration, because network latency does not greatly affect the throughput of asynchronous Binlog solutions since the active does not wait for the standby to acknowledge.

One-Zone	DAX	CAC-DB-WO	CAC-DB-W	LAS	LS	EAS	ES	DAXAS	DAXS
downtime (s)	300	180	4	570	2	1800	2	2100	2
time to peak throughput (s)	900	30	220	0	0	0	240	0	1200
lag-behind (MB)				31	0.0029	100	0.0019	110	0.0019

Table 6.2: Failover Time Comparison for Different Regions

Turning our attention to behavior after failure, we see that throughput is higher after failover because the system after failover is not protected (has no standby) and so does not incur the cost of protection. The downtime of LS, ES, and DAXS is nearly 0, because the active is slow enough for the standby to always catch up. When the active fails, there is very little residual relay log, so there is nearly no downtime. Second, LAS shows a longer downtime in the different-zone configuration than that in the one-zone configuration, because of the longer network latency between the active and the standby. Both EAS and DAXAS show long downtime in different zones, because both the high network latency and the sequential IO to the storage over the network put a high overhead on the standby, which leads to the standby being slow. When the active fails, there is much log remaining for the standby to replay before completing the failover.

As in the one-zone configuration, CAC-DB represents a good tradeoff that has high throughput during normal operation and low time to recovery after failure. The performance advantage of CAC-DB over Binlog replication is higher in the different-regions configuration than the one-zone configuration because the network overhead is higher. In addition to better performance, CAC-DB has the same advantages over Binlog replication as in the one-zone configuration: never losing transactions, no database replication, and lower database administration overhead.

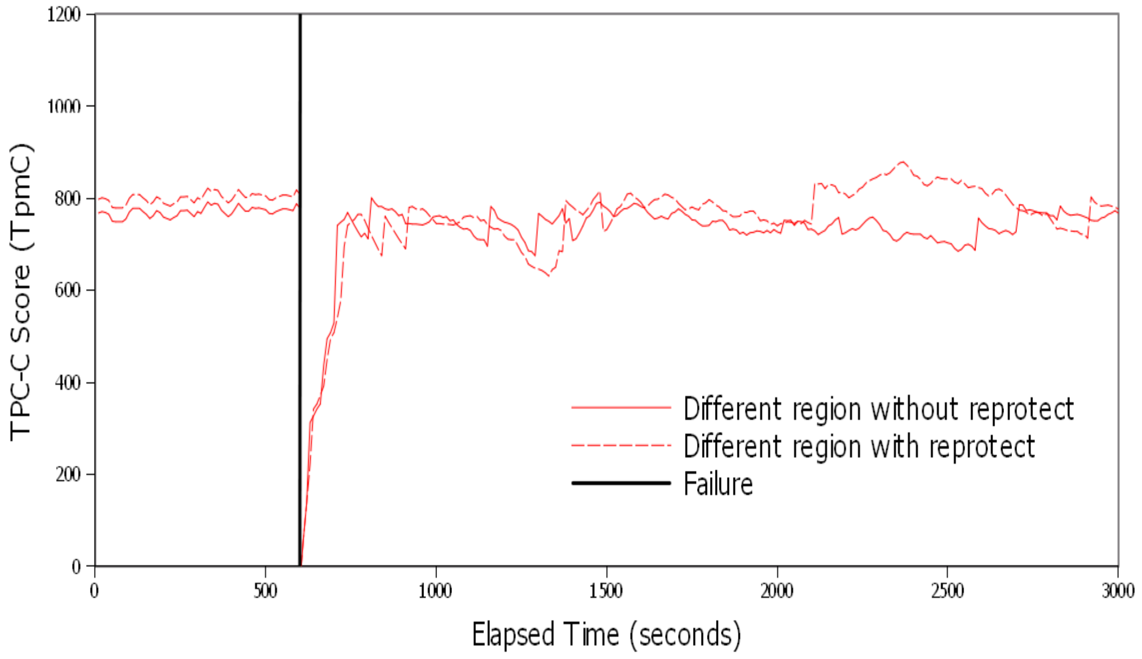

Figure 6.6: Reprotection After a Failure

6.3 Reprotection After a Failure

The reprotection mechanism of CAC-DB is shown in Figure 6.6. Similar to the experiment described in Section 6.2, we run the different-region configuration. We run the test for more than one hour before the failure and more than one hour after the failure. After the failure, the old standby becomes the new active. We start a new standby as soon as failure is detected. The performance of CAC-DB without reprotection is also shown for reference.

In the different region configuration we are able to start the new standby after 4 seconds of the failure. The new standby is up and replaying logs after 12 seconds of being started, meaning that the system is vulnerable to failure for only 4+12 seconds. We can see from Figure 6.6 that reprotection does not significantly impact performance. The only perceptible impact is that reprotection delays the time by which the new active reaches peak throughput by about 30 seconds compared to the case of no reprotection. Thus, we can see that CAC-DB effectively reprotects the DBMS after failover, with little impact on performance.

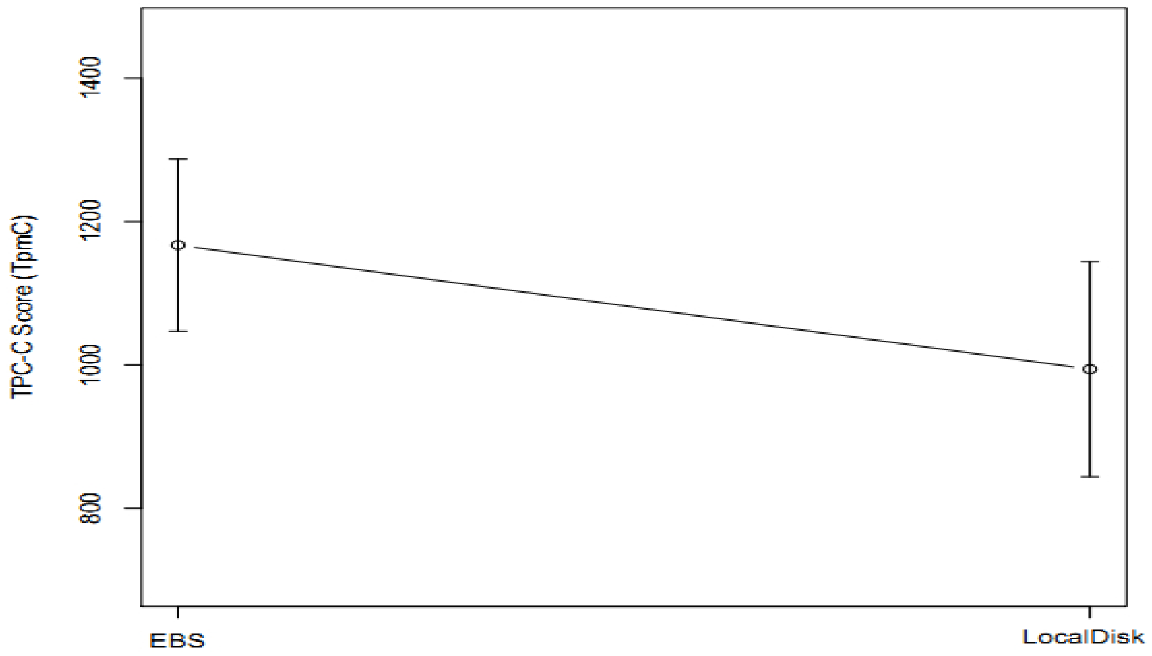

Figure 6.7: Average and 95% Confidence Interval of Throughput on Local Disk and EBS

6.4 Comparison Between Local Disk and EBS

In Figures 6.3 and 6.6, we observe better throughput for MySQL on EBS compared to local disk on the EC2 instance. To analyze the difference between throughput in these two cases, we deploy a single MySQL instance without protection on local disk and EBS and measure throughput. We repeat each experiment ten times and present the 95% confidence interval for the average throughput in Figure 6.7. The figure shows that in our setting, local disk is slower on average than EBS, which can explain the variation in performance observed in our experiments. However, we also see that the variance is high enough that the confidence intervals overlap, so the difference in performance may be due to experimental error.

In addition, we use the “fio” tool to analyze different metrics of IO performance on both local disk and EBS, based on specific IO workloads. The fio tool spawns a number of threads or processes to perform a particular type of IO action, as specified by the user. In our benchmark, we use an m1.large EC2 instance with 1xEphemeral RAID0 local disk and a single EBS volume, and we simulate four kinds of IO workloads: sequential read, sequential write, random read, and random write. We set the parameters in fio to

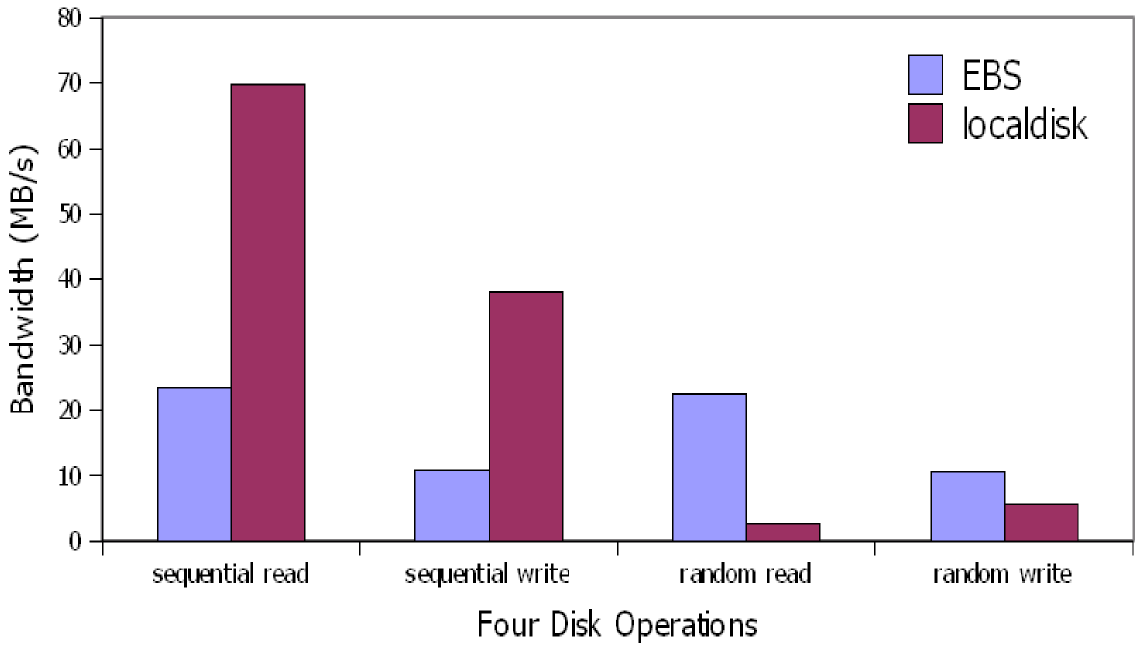

Figure 6.8: Bandwidth Comparison Between Local Disk and EBS

corresponding workload values, the parameter *direct* to “true” to use non-buffered IO, the parameter *ioengine* to “synchronous”, the parameter *filesize* to “2048MB”, and the parameter *blocksize* to “16KB”. We measure the performance of the four kinds of IO workloads separately. Every time we start a new measurement, we clear the buffered content in the memory to ensure a fair comparison. The comparison results are shown in Figure 6.8. The figure shows that LocalDisk outperforms EBS in sequential operations, while EBS performs better at random operations. For both sequential read and sequential write, LocalDisk’s bandwidth is nearly triple that of EBS’s. For random read, EBS’s bandwidth is nearly ten times that of LocalDisk’s. In addition, EBS’s random write’s bandwidth is nearly twice LocalDisk’s. Our benchmark obtains similar results to those shown in [2, 1]. Since transactional database workloads such as TPC-C involve many random reads and writes across the dataset, MySQL replication solutions on EBS show better throughput than those on local disk during normal operations, for both the one-zone and different-regions experiments (Figures 6.3 and 6.5). During recovery, the log is read in a sequential fashion, which is faster on local disk. This contributes to making recovery on local disk faster, although the main reason recovery is faster on local disk is that there is less relay log at the standby after failure.

6.5 Varying Buffer Pool Size

In this section, we study the effects of changing the ratio of database size to buffer pool size on the performance of CAC-DB. We fix the database size to an initial size of 10GB (our default database size) and change the buffer pool size to change the ratio. In the previous sections, we use a 5GB buffer pool. In this section, we conduct experiments comparing DAX and CAC-DB-W with buffer pool sizes of 2GB and 10GB. We use the one-zone and different-regions configurations as before. The results for DAX and CAC-DB-W with 2GB and 10GB buffer pools are shown in Figure 6.9 and Figure 6.10, respectively. Both figures show that different buffer pool sizes do not negatively impact CAC-DB. As expected, a larger buffer pool results in higher throughput during normal operation. After failover, CAC-DB shows very little down time and fast time to peak throughput in both cases.

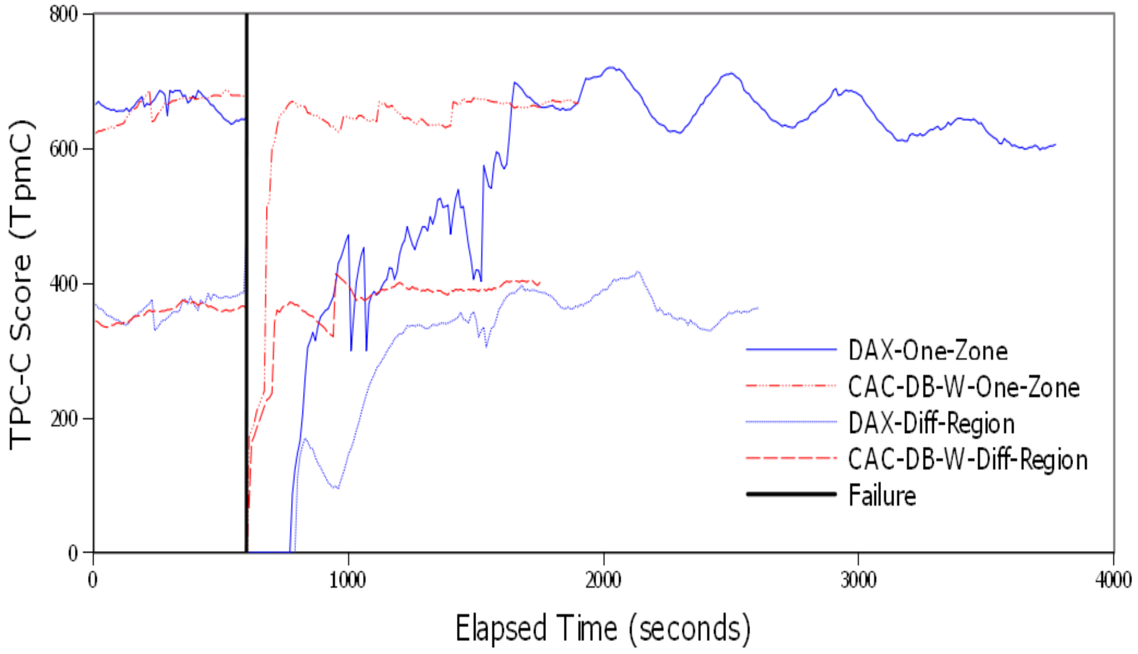

Figure 6.9: Comparison Between DAX and CAC-DB-W With a 2GB Buffer Pool

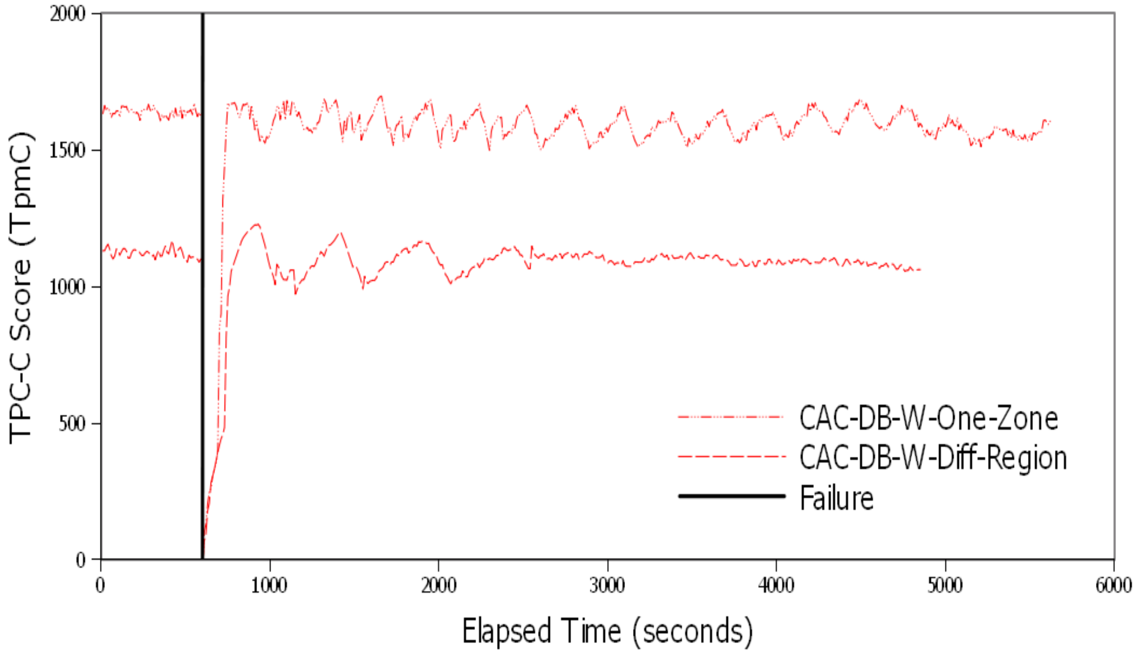

Figure 6.10: Comparison Between DAX and CAC-DB-W With a 10GB Buffer Pool

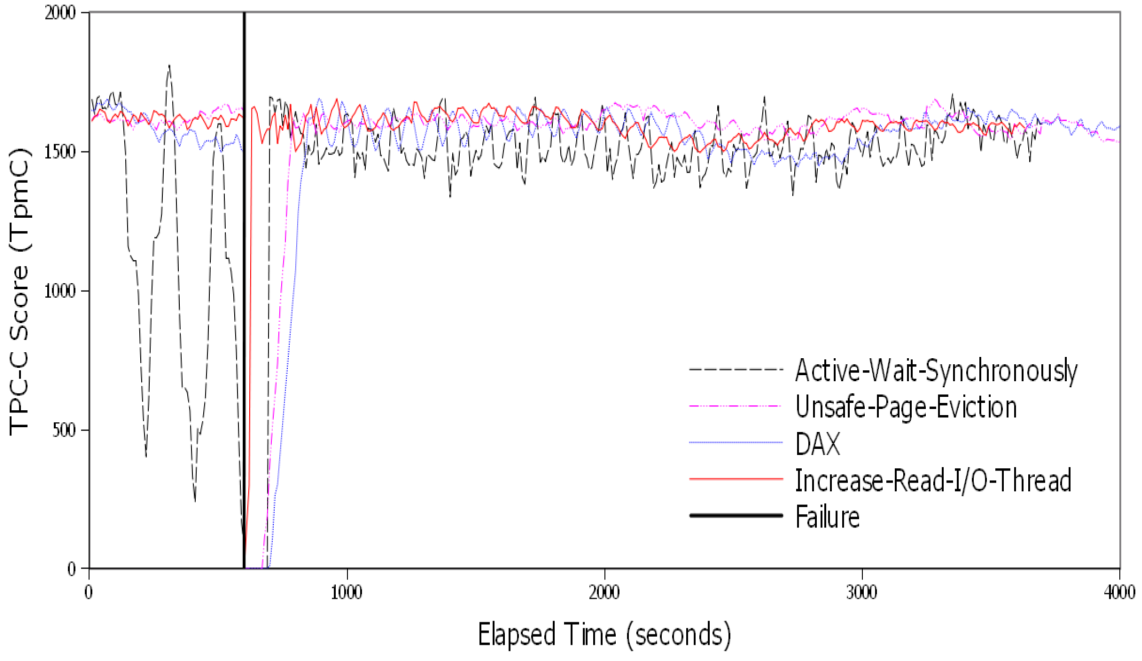

Figure 6.11: Comparison Between Different Methods for Handling a Slow Standby

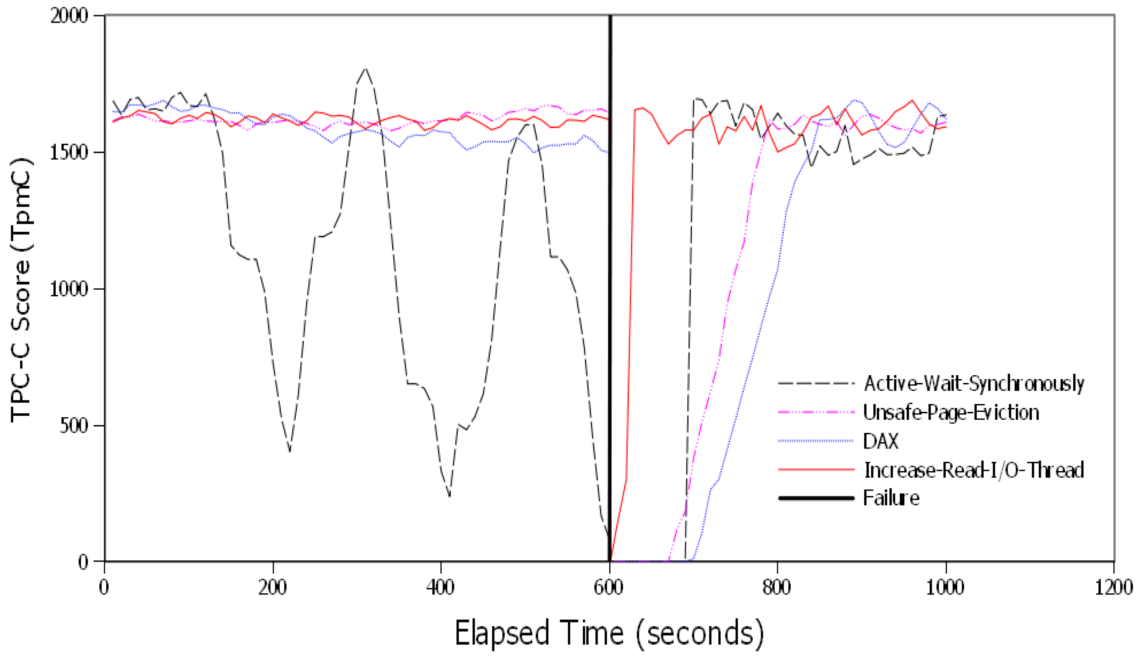

Figure 6.12: Detailed View of the Methods for Handling a Slow Standby

6.6 Different Design Alternatives for Dealing with a Slow Standby

In Chapter 4, three solutions are proposed to deal with a slow standby: (1) the active waits for the standby synchronously, (2) unsafe page eviction, and (3) increasing read IO threads. Our choice for CAC-DB is to increase IO threads. In this experiment, we compare the performance of all three methods to justify our design decision. We use the one-zone setting and the same configuration of EC2 instances as before (one instance for the active, one for the standby, and nine for DAX), but we use a 10-warehouse TPC-C database with an initial size of 1GB. Each database block is replicated six times across the nine DAX servers, for a total data size of approximately 6 GB. The alternative of Active-Wait-Synchronously, Unsafe-Page-Eviction, Increase-Read-I/O-Thread, and DAX are compared in Figure 6.11. We show the throughput of each method for 600s before the failure and 3400s after the failure. Figure 6.12 focuses on the period of 0 to 1000 seconds in Figure 6.11.

Before the failure, Unsafe-Page-Eviction, Increase-Read-I/O-Thread, and DAX show similar throughput. However, the Active-Wait-Synchronously throughput fluctuates significantly from below 500 to above 1500. This is because in this method, the active cannot reclaim log space containing log records that the standby has not yet replayed. When the active fills up the log and needs to reclaim log space, it must wait for the standby (and stop accepting user requests), which causes throughput to drop to below 500. When the standby catches up, the active can reclaim log space and rapidly recover to peak throughput. This pausing and resuming of the active results in the fluctuation observed in the experiment. InnoDB uses circular log files, so the active determines that it needs to wait for the standby when the active wraps around the end of a log file and almost catches up with the standby. We calculated the active's cumulative throughput before failure to be 42.95% lower than the throughput of the other three methods. In addition, we calculated the total synchronous wait time and found that the active is waiting for 45.18% of the total time before the failure. This demonstrates that having the active synchronously wait for the standby is not a viable solution.

Special-case	Active-Wait -Synchronously	Unsafe-Page -Eviction	DAX	Increase-Read -I/O-Thread
downtime (s)	80	80	90	4
time to peak throughput (s)	10	80	150	20

Table 6.3: Failover Time Comparison for Different Methods of Handling a Slow Standby

The disadvantage of Unsafe-Page-Eviction appears in the longer time it needs for recovery. We show the downtime of each method after failure and also the time that the method takes to reach peak throughput in Table 6.3. The table shows that Increase-Read-I/O-Thread outperforms other methods. Increase-Read-I/O-Thread has nearly 0s of downtime and can rapidly reach peak throughput for the reason described in Section 6.2. DAX has longer downtime and longer time to reach peak throughput for the same reason as in Section 6.2. In Active-Wait-Synchronously, the standby has a relatively long downtime, because it needs to go through a similar recovery procedure to that of DAX. When the standby detects the failure, the standby-replayed-lsn is close to the latest checkpoint LSN, because the standby always lags far behind the active. The standby must redo the log from the standby-replayed-lsn, which represents a large number of log records. After redo, the standby can immediately reach peak throughput, because it has a warmed-up buffer pool. For Unsafe-Page-Eviction, the standby also has long downtime, because it

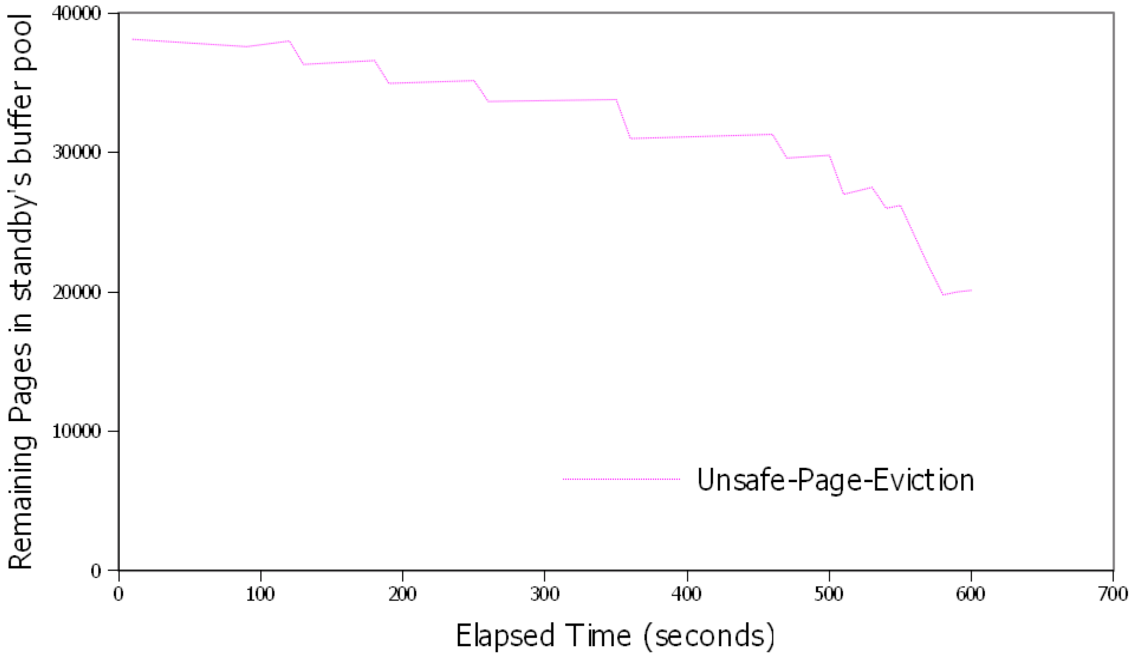

Figure 6.13: Number of Pages in the Standby Buffer Pool

too needs to perform a long recovery procedure (redoes the log from the standby-replayed-lsn). Unsafe-Page-Eviction also requires more time than Active-Wait-Synchronously and Increase-Read-I/O-Thread to reach peak throughput, because the standby begins with a colder buffer pool after the failover. This is because during normal operation before the failure, every time the standby falls behind, it will skip the log to the checkpoint LSN and evict all the unsafe pages from the buffer pool. Though the standby will continue loading new pages from the storage tier, the number of pages loaded is less than the number of pages evicted from the buffer pool. Therefore, the standby's buffer pool will become increasingly cold. To verify this, we record the number of pages in the standby's buffer pool for a 600-second period before the failure in Figure 6.13. The figure shows that although the standby periodically loads new pages, the page eviction rate is greater than the page loading rate. For example, from 90s-120s, the standby loads 404 pages. However, by 120s-130s, the standby evicts 1679 pages. Therefore, the number of pages in the standby's buffer pool decreases nearly 50% in 600s, which explains the longer time required by the standby to reach peak throughput after the failover.

From this experiment, we conclude that we are justified in our design decision of increasing IO threads to deal with a slow standby. This choice has throughput among the best before failure, and has the best behavior after failure.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis proposes CAC-DB, a DBMS-level HA solution that works in a geographically distributed setting and takes advantage of shared storage. CAC-DB relies on a cloud storage system that effectively replicates data across multiple data centers, and in this thesis we use DAX as this storage system. CAC-DB builds on top of DAX, and deploys an active and a standby server in different data centers. The underlying storage system provides shared transactionally consistent access to the database and the log for both the active and the standby. The standby continuously replays log records from the log that it shares with the active, so that the memory state of the standby is always catching up withe the memory state of the active. When the active fails and the standby starts the recovery procedure, the system downtime is typically very short because the memory state is already up to date. This results in a highly available system with very little performance or administration overhead. Our experiments on Amazon EC2 show that the system downtime for CAC-DB is close to zero, even when running in different geographically distributed data centers. Thus, CAC-DB is a novel and effective DBMS high availability solution that leverages cloud shared storage.

7.2 Directions for Future Work

We hope that this thesis will encourage further research into DBMS-level HA solutions that work in a geographically distributed setting and leverage shared storage. We present some specific directions for future work next.

7.2.1 CAC-DB with Other DBMSes and Storage Systems

One interesting direction for future work would be to reimplement CAC-DB with other popular cloud storage systems, such as HBase [4], or another DBMS, such as PostgreSQL. This may expose new challenges and provide opportunities to produce a better solution. Furthermore, if CAC-DB can be deployed successfully with other platforms, it will prove its effectiveness and can have greater impact.

7.2.2 Eliminating the Need for Page Flushing After Failover

Our experiments on Amazon EC2 show that system downtime for CAC-DB is close to zero. However, the time to reach peak throughput may be over 100s, mainly because the standby must flush many pages asynchronously after failover. This is caused by inconsistencies between the *old_lsn* of the buffer pool pages in the active's buffer pool and the standby's buffer pool.

At failover, the standby must flush those pages whose *old_lsn* is smaller than a threshold LSN to clear log space for newly created log records. However, because the standby reads the log from shared storage, it cannot obtain information about the *old_lsn* of the buffer pool pages, and so it cannot update the *old_lsn* of the pages in time. Therefore, there are many pages whose *old_lsn* is small and not updated for a long time, and these pages must be flushed.

One interesting direction for future work is to solve the problem above and make the *old_lsn* for buffer pages in the standby's buffer pool consistent with that in the active's buffer pool. One possible implementation would be that the active writes the *old_lsn* for buffer pages into the log file, so that the standby can obtain this information from the shared storage and update the *old_lsn* for its buffer pages when it replays the log during normal operation. In this way, there will be fewer pages which must be flushed at failover, thereby decreasing the time needed to reach peak throughput.

APPENDICES

Appendix A

Failure Detection in CAC-DB

In this appendix, we present the failure detection approach that the standby in CAC-DB uses to determine that the active has failed. This failure detection algorithm was developed independently of this thesis by a co-op student, and is presented here for completeness.

Before failover can be triggered, some approach is needed for the standby to detect whether the active has failed. This can be accomplished "manually," through human-assisted monitoring of the active. However, automated or semi-automated failure detection is often implemented using some kind of *heartbeat* mechanism. Such mechanisms require that the active sends regular heartbeat messages to indicate that it is still "alive". Failure of the active to send a heartbeat message within a prescribed interval is taken as an indication that the active has failed. The active's heartbeat messages may be directed to the standby system or to a third-party system. For example, a Bigtable [15] master system must maintain an active session with the Chubby lock service [14] to ensure that it remains the master. To keep its session alive, the Bigtable master sends heartbeat messages to Chubby. Another example is the failure detection service in Boxwood [24], which requires that clients send regular heartbeat messages to a set of observers.

CAC-DB uses a heartbeat-based failure detection mechanism that uses DAX storage for reliable, indirect communication of heartbeats. There are several advantages to such a design. First, because the detection mechanism relies only on the storage tier that CAC-DB already uses to store the database and logs, there is no need to deploy and manage any additional service (such as Chubby or Boxwood) for failure detection. In addition, failure detection does not depend on the availability of direct communication between the active and the standby systems. Finally, since there is no separate failure detection service and no direct communication channel between the active and the standby, there are fewer system failure modes to be concerned about. If a storage tier is available, so is the failure detection mechanism.

To support failure detection, we create a reliable counter in DAX for each of the active/standby DBMS pairs for which we wish to support failover. The active system is required to advance the counter regularly as long as it is running. The standby detects failure by reading the counter and determining whether or not it has been advanced. Algorithm 3 shows the algorithm used by the active. Failure of the active to increment the counter at least once every T_{max} time units will be interpreted by the standby as a primary failure. Thus, the active sets an alarm, which is reset each time the counter is successfully incremented. If the alarm times out (e.g., because of loss of communication between the active and the storage tier), then the active must shut itself down. Our implementation uses an operating system timer in the active server to implement the alarm.

Similarly, the standby, shown in Algorithm 4, regularly polls the counter in the storage tier and promotes itself to the active if the counter's value does not advance for more than T_{max} time units. This algorithm does not require clock synchronization between the active and the standby. However, as shown, it does assume that the clocks progress at the same rate on both servers. In practice, to account for clock drift between the active and the standby, the standby may conservatively use a value larger than T_{max} in its test at line 11 in Algorithm 4.

Algorithm 3: Failure Detection - Generating Heartbeats at the Active

```
Input: A shared reliable counter $c$, initialized to 0
Input: Maximum heartbeat period $T_{max}$
Input: Heartbeat period $T$ ($T < T_{max}$)
1  next ← 1;
2  SetAlarm($T_{max}$);
3  try
4      while true do
5          if SetCounter($c$,next)=success then
6              ResetAlarm($T_{max}$);
7              next ← next + 1;
8          Sleep($T$);
9  catch alarm
10     ShutdownActive()
```

Algorithm 4: Failure Detection - Monitoring Heartbeats at the Standby

```
    Input: A shared reliable counter $c$, initialized to 0
    Input: Maximum heartbeat period $T_{max}$
    Input: Polling period $T_{poll}$ ($T_{poll} < T_{max}$)
1   prev ← 0;
2   polls ← 0;
3   repeat
4       Sleep($T_{poll}$);
5       cur ← GetCounter($c$);
6       if cur > prev then
7           prev ← cur;
8           polls ← 0;
9       else
10          polls ← polls + 1;
11  until $T_{max} > T_{poll} *$ polls;
12  PromoteStandby()
```

Our detection mechanism assumes that the shared storage tier implements reliable counters which can be read and updated using the `GetCounter` and `SetCounter` functions. We assume that the storage system provides strong consistency for `GetCounter` operations, i.e., `GetCounter` will return the value at least as recent as that written by the most recent successful `SetCounter` operation. Unsuccessful `SetCounter` operations may or may not be visible: `GetCounter` will see either the result of the latest successful `SetCounter` or that of a more recent by an unsuccessful `GetCounter`.

In our CAC-DB prototype, the storage tier replicates the value of each counter across a configurable number of storage tier servers, just as other data are replicated. `GetCounter` and `SetCounter` are implemented by quorum-based replicated reads and writes. That is, `SetCounter` operations are sent to all counter replicas and succeed only if a majority of replicas acknowledge the operation. `GetCounter` operations are sent to all replicas and return only after a majority of the replicas have responded to the request. The latest (largest) value from this quorum is returned as the value of `GetCounter`. This implementation is similar to Boxwood, which also uses quorums to track the state of the active system.

In our implementation, we set the thresholds for failure detection to the following values: $T = 2s$, $T_{max} = 9s$, and $T_{poll} = 2s$.

References

[1] Amazon EC2 I/O Performance: Local Ephemeral Disks vs. RAID 0 Striped EBS Volumes. [online] http://blog.dt.org/index.php/2010/06/amazon-ec2-io-performance-local-emphemeral-disks-vs-raid0-striped-ebs-volumes/.

[2] EC2 Ephemeral Disks vs. EBS Volumes in RAID. [online] http://victortrac.com/blog/2010/01/02/ec2-ephemeral-disks-vs-ebs-volumes-in-raid/.

[3] Hadoop Distributed File System. [online] http://hadoop.apache.org/.

[4] HBase. [online] http://hbase.apache.org/.

[5] MySQL Replication. [online] http://dev.mysql.com/doc/refman/5.5/en/replication.html.

[6] Linux-HA Project. [online] http://www.linux-ha.org/doc/. 1999.

[7] Percona Tools TPC-C MySQL Benchmark. [online] https://code.launchpad.net/percona-dev/perconatools/tpcc-mysql. 2008.

[8] MySQL Cluster 7.0 and 7.1: Architecture and new features. *A MySQL Technical White Paper by Oracle*, 2010.

[9] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):159–174, 2007.

[10] Jason Baker, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Conference on Innovative Data Systems Research*, pages 223–234, 2011.

[11] Emre Baransel. *Oracle Data Guard 11g R2 Administration Beginner's Guide*. Packt Publishing Ltd, 2013.

[12] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[13] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 251–264, 2008.

[14] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[16] Whei-Jen Chen, Masafumi Otsuki, Paul Descovich, Selvaprabhu Arumuggharaj, Toshihiko Kubo, and Yong J. Bi. High availability and disaster recovery options for DB2 on Linux, Unix, and Windows. Technical Report Redbook SG24-7363-01, IBM, 2009.

[17] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[18] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google globally-distributed database. *Proceedings of OSDI*, page 1, 2012.

[19] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTras: An elastic transactional data store in the cloud. *Proceedings of USENIX Workshop on Hot Topics in Cloud Computing*, 2009.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and

Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[21] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Technology.* Morgan Kaufmann, 1993.

[22] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35, 2010.

[23] Rui Liu, Ashraf Aboulnaga, and Kenneth Salem. DAX: A widely distributed multitenant storage service for DBMS hosting. *Proceedings of the VLDB Endowment*, 6(4), 2013.

[24] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 8–23, 2004.

[25] Umar F. Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent high availability for database systems. *Proceedings of the VLDB Endowment*, 22(1):29–45, 2013.

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

[27] Mahdi T. Najaran, Primal Wijesekera, Andrew Warfield, and Norman C. Hutchinson. Distributed indexing and locking: In search of scalable consistency. *The 5th Workshop on LargeSCale Distributed Systems and Middleware*, 2011.

[28] Christos A. Polyzois and Hector Garcia-Molina. Evaluation of remote backup algorithms for transaction processing systems. *ACM Transactions on Database Systems*, 19(3):423–449, 1994.

[29] Paul S. Randal. Microsoft SQL Server 2008 R2 High Availability Technologies White Paper. [online] http://technet.microsoft.com/en-us/library/ee523927(v=sql.100).aspx. 2010.

[30] Murali Vallath. *Oracle real application clusters.* Digital Press, 2004.