View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Waterloo's Institutional Repository

Database High Availability using SHADOW Systems

by

Xin Pan

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

Waterloo, Ontario, Canada, 2014

© Xin Pan 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Various High Availability DataBase systems (HADB) are used to provide high availability. Pairing an active database system with a standby system is one commonly used HADB techniques. The active system serves read/write workloads. One or more standby systems replicate the active and serve read-only workloads. Though widely used, this technique has some significant drawbacks: The active system becomes the bottleneck under heavy write workloads. Replicating changes synchronously from the active to the standbys further reduces the performance of the active system. Asynchronous replication, however, risk the loss of updates during failover. The shared-nothing architecture of active-standby systems is unnecessarily complex and cost inefficient.

In this thesis we present SHADOW systems, a new technique for database high availability. In a SHADOW system, the responsibility for database replication is pushed from the database systems into a shared, reliable, storage system. The active and standby systems share access to a single logical copy of the database, which resides in shared storage. SHADOW introduces write offloading, which frees the active system from the need to update the persistent database, placing that responsibility on the underutilized standby system instead. By exploiting shared storage, SHADOW systems avoid the overhead of database-managed synchronized replication, while ensuring that no updates will be lost during a failover. We have implemented a SHADOW system using PostgreSQL, and we present the results of a performance evaluation that shows that the SHADOW system can outperform both traditional synchronous replication and standalone PostgreSQL systems.

Acknowledgements

I would like to thank Professor Kenneth Salem for being my supervisor. I admire his patience, that made the completion of this thesis possible, and I am grateful for his thorough critical comments, that helped me to make it more consistent.

I am grateful to University of Waterloo for providing funding for my studies and being an exciting place.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Figures **ix**

1 Introduction **1**

1.1 Background 1

1.2 SHADOW systems 2

1.3 Thesis Organization 3

2 SHADOW Overview **5**

3 SHADOW Operations **8**

3.1 Assumptions 9

3.2 Stand Alone State 10

3.3 Protect 10

3.4 Active + Standby State 12

3.5 Failover 13

3.6 Re-protect 13

3.7 DBMS Recovery 14

3.8 Correctness of SHADOW 15

4 Large Databases **17**

4.1 Stale Read Detection 17

4.2 Correcting Stale Reads 18
4.3 Discussion 20
4.3.1 Alternative Stale Read Correcting Methods 20

5 Prototype Implementation **21**
5.1 DBMS for SHADOW-NFS 22
5.2 Shared Storage for SHADOW-NFS 25
5.3 Alternative Shared Reliable Storage 25
5.3.1 Dynamo 25
5.3.2 DAX 26

6 Evaluation **27**
6.1 Experiment Methodology 27
6.2 Large Memory Case 29
6.3 Small Memory Case 30
6.4 Large Dataset Case 32
6.5 Protection and Failure Handling 34
6.5.1 Protection 34
6.5.2 Failover 36
6.5.3 Re-protect 37

7 Related Works **39**
7.1 Overview 39
7.2 MySQL 41
7.3 PostgreSQL 42
7.4 Oracle 43
7.5 RemusDB 44
7.6 Spanner 44

8 Conclusion **46**

References **47**

List of Figures

2.1 SHADOW System Architecture 6

3.1 Operational States of a SHADOW System 9
3.2 Protection Operation Timeline 11
3.3 DBMS Persistent State After Linked Backup 12

4.1 Reading a Page at the Active DBMS 19

5.1 The SHADOW-NFS Prototype 22

6.1 Database Writes (KB/s) between Active and NFS server 29
6.2 50th and 99th Percentile NewOrder Transaction Latency (ms) 29
6.3 TPC-C Throughput, Large Memory 30
6.4 TPC-C Throughput, Small Memory 31
6.5 TPC-C Throughput, Large Dataset 33
6.6 TPC-C Throughput, Large Dataset 35
6.7 Write I/O between the Active and the NFS, Large Dataset 35
6.8 Protection Time in SHADOW-NFS and SR 36
6.9 Failover Time in SHADOW-NFS and SR 37
6.10 TPC-C Throughput During a Standby Failure 38

7.1 Replication design dimensions 40

Chapter 1

Introduction

1.1 Background

High Availability refers to the ability of a system to remain accessible to users in the presence of either software or hardware failures. Highly Available DataBase systems (HADB) are widely used nowadays. Various techniques [2, 4, 19, 1, 16, 6] are used to achieve database system high availability.

In this thesis, we focus on the active-standby configurations with one active system and a standby system. In a typical one active and one standby configuration, the active and standby systems each maintain a private copy of database in their private persistent storage. The active accepts read/write workloads from users and updates its own copy of database in persistent storage. It also ships logs to the standby so that the standby can replay the operations from the active and update its own copy of database. The whole system is able to remain available during a single-site failure.

Although active-standby systems have been widely used, they have some significant drawbacks:

complexity: Two logically distinct copies of the database are required, one managed by the active system and one managed by the standby. Synchronization of these two databases, which is normally done by log shipping, must be managed by the database systems, and configured and controlled by database administrators (DBAs). This introduces extra complexity to the already very complex DBMS.

performance: To minimize the risk that updates will be lost as a result of a failure of

the active system, active-standby systems can use *synchronous* log shipping. This means that updates performed by a transaction at the active must be shipped to the standby system, and acknowledged by the standby, before the active transaction commits. Such synchronization can add considerable latency to transaction commit processing at the active system, hurting its performance.

lost updates: To avoid the performance penalties associated with synchronous log shipping, the active system can ship logs *asynchronously*, after the logged transaction has been committed at the active system and acknowledged to client applications. Although this hides the latency of log shipping, updates performed by recently committed transactions may be lost when the active system fails, since those updates may not have been shipped to the standby system prior to the failure. Many asynchronous replication mechanisms implemented by DBMS cannot guarantee the ACID [1] properties during failover. Log shipping and transaction commit cannot be done in an atomic manner easily.

cost and cost efficiency: An active-standby system is essentially twice as expensive to deploy and operate as an unprotected active system alone, since there are two database systems and two databases to store and update. Despite this extra cost, an active-standby system sometimes performs worse than an unprotected active system, because of the overhead of log shipping and the latencies associated with update synchronization. Thus, the cost per transaction may be more than twice as high in an active-standby system as in a standalone active. This is a steep price to pay for high availability.

1.2 SHADOW systems

In this thesis we propose SHADOW (SHAred Database with Offloaded Writes) systems, an alternative active-standby high availability architecture for DBMS that addresses these weaknesses. SHADOW systems incorporate two key ideas. First, in a SHADOW system, the active and standby database systems share access to a *single logical copy of the database and a single copy of the log.* The shared database and log reside in a reliable and highly available storage tier which is accessible to both the active system and the standby system. The storage tier replicates and distributes data to achieve fault tolerance and high availability. By pushing replication out of database systems and into the storage tier, complexity in the database system tier is reduced. Durably committing a transaction does not require coordination between two database systems. There is no risk that the

[1] Atomicity, Consistency, Isolation and Durability

active and standby databases will diverge, since there is only one (logical) database in a SHADOW system.

The second idea incorporated by SHADOW systems is called *write offloading*. In a SHADOW system, the active DBMS retains responsibility for processing client requests. However, the active DBMS never updates the persistent database in the underlying storage tier. It writes only log records to the persistent storage system. The task of updating the persistent copy of the database falls instead to the standby system. In some scenarios, such as when the active and standby database systems have large amounts of memory, write offloading allows a SHADOW system to achieve higher transaction throughput than a standalone, active DBMS, while ensuring high availability. A SHADOW system is still more expensive than a standalone DBMS, but this cost buys *both* high availability and improved performance.

This thesis makes the following research contributions:

- We propose the SHADOW system architecture, and present algorithms for initiating a SHADOW standby system and for reading from the persistent database at the active DBMS.
- We present a prototype implementation of the SHADOW architecture. The prototype uses a network file system to provide shared persistent storage for the database and the log.
- We present an evaluation of the prototype, and a comparison of its performance against several baselines. Our evaluation shows that the SHADOW prototype provides better performance that existing hot standby approaches, while ensuring that no transactions are lost as a result of failures. It also demonstrates that, because of write-offloading, a SHADOW system can outperform a standalone DBMS, although the former provides high availability while the latter does not.

1.3 Thesis Organization

The remainder of the thesis is structured as follows. Section 2 provides a general overview of the SHADOW architecture. Section 3 describes the operation of a SHADOW system under the assumption that there is sufficient local memory at the active and standby database servers to hold a complete copy of the database. This is the target environment for the SHADOW design. In Section 4, we describe how this assumption can be relaxed.

Section 5 describes our SHADOW prototype implementation, called SHADOW-NFS, and Section 6 presents the results of a performance evaluation that compares the prototypes to several different baselines. Section 7 summarizes related work, and Section 8 concludes.

Chapter 2

SHADOW Overview

Figure 2.1 illustrates the architecture of a SHADOW system. Clients connect to the active DBMS, which processes all client transactions. The clients can also send read-only queries to the hot standby. Both the active and the standby have access to a shared, reliable persistent storage system, which holds a single (logical) copy of the database and the transaction log.

We assume that the shared persistent storage system is itself highly available. There are numerous widely used techniques for implementing highly available shared storage. For example, Amazon Web Services (AWS) provides the Elastic Block Store (EBS), which replicates data across multiple servers. EBS provides network-accessible storage volumes which can be attached to servers running in the AWS cloud. Alternatively, highly available shared storage could be provided by a cluster file system, such as VMFS [26], with the file system data mirrored or striped across multiple storage devices using RAID [21] techniques. Network file systems, such as NFS, can also be made highly available through the use of server pairs. For the purposes of this section, we are not concerned with the specific implementation of the shared storage system. In Section 5 we describe the details of an implementation of the SHADOW architecture, named SHADOW-NFS. It is based on the PostgreSQL DBMS and an NFS shared file system.

The SHADOW architecture targets transaction processing (OLTP) workloads, for which fast efficient transaction processing is critical. In particular, the SHADOW architecture targets applications in which most or all of the database can fit in memory at the active and standby servers. This is a common configuration for OLTP applications. Thus, Figure 2.1 shows local database caches at both the active and standby systems. In a SHADOW system, all storage local to the active and standby systems is treated as *ephemeral*, and

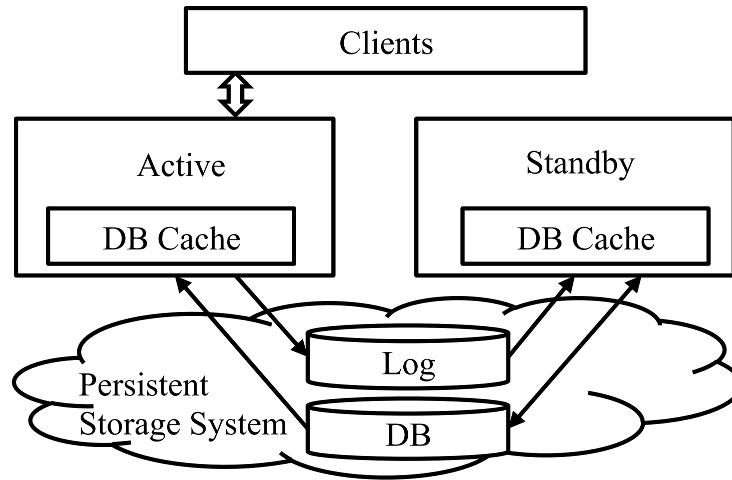

Figure 2.1: SHADOW System Architecture

data stored locally (including the contents of the local database cache) are not expected to persist across a failure of the local system. Thus, for a transaction to be considered durably committed, its updates and commit record must be present in the persistent storage system.

In a SHADOW system, the active DBMS performs updates in its local, cached copy of the database. The active DBMS also performs write-ahead logging to push records of database updates and transaction commits to the persistent log. These log writes are the *only* writes the active DBMS performs on the persistent storage system. It never updates the persistent copy of the database.

The standby reads log records generated by the active and replays those logged updates against its local cached copy of the database. In addition, the standby gradually propagates database changes from its local database copy to the persistent copy. Thus, only the standby system updates the persistent copy of the database. This offloading of database updates from the active DBMS to the standby distinguishes SHADOW systems from other active-standby techniques.

In a SHADOW system, a transaction is committed once its commit record is present in the persistent log. There is no need for coordination between the active and standby systems during transaction commit. SHADOW is designed to handle varioius failure senarios. In case of a failure of the active DBMS, the standby finishes replaying any remaining log records, and then takes over as the new active system. This *failover* procedure is very

similar to failover procedures in other hot standby database systems. The standby DBMS is able to survive repeated failures, as long as the shared persistent storage is always available. When both the active and the standby systems are down, the standby is recovered first. The standby first replays all the logs left over by the active. Then a failover operation is performed to promote the standby to the new active. More details are described in Section 3.

Chapter 3

SHADOW Operations

Figure 3.1 shows the possible operational states of a SHADOW system, and the transitions among those states. Transitions shown in dashed lines are failure transitions, which occur when either the active or the standby fails. Transitions shown in solid lines are SHADOW system operations, which move the system from one operational state to another. Other active-standby database systems would have a state diagram similar to the one shown in Figure 3.1. However, because a SHADOW system divides responsibility for managing a single persistent database between the active and standby systems, the behavior of a SHADOW system in the various states may differ from that of other systems. Similarly, SHADOW system operations, such as PROTECT, will differ from those in other active-standby systems.

As described in Section 5, we have implemented SHADOW prototypes using the PostgreSQL DBMS. However, the SHADOW architecture should also be implementable using other database systems. Thus, in this section we describe the SHADOW operations in terms of a more generic DBMS, and we defer discussion of PostgreSQL specifics until Section 5.

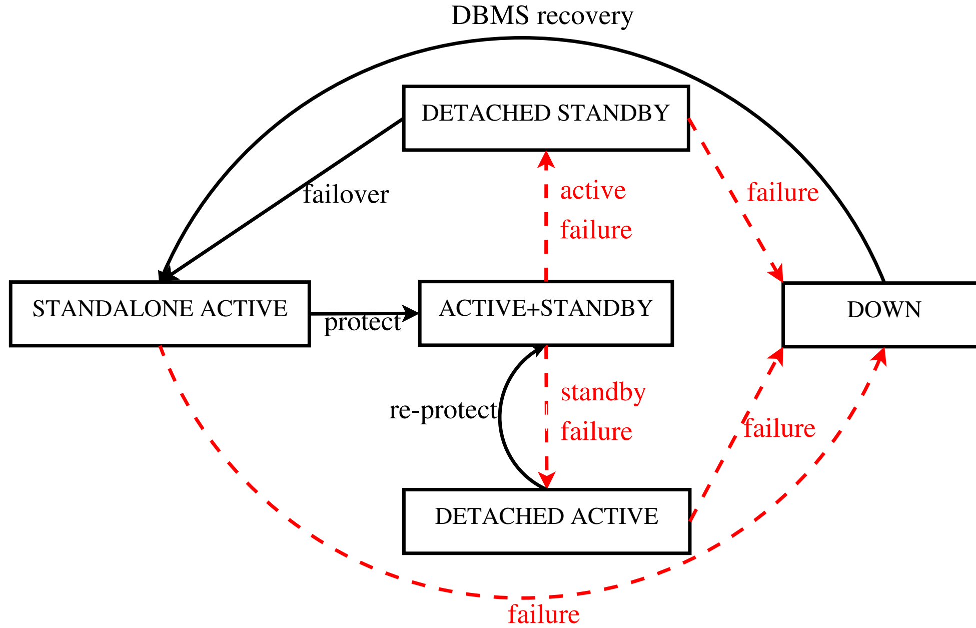

Figure 3.1: Operational States of a SHADOW System

3.1 Assumptions

We designed and implemented SHADOW based on some techniques, such as ARIES [20], that are widely used in modern relational database systems. We assume that these techniques are already implemented. In this section we describe these assumptions.

We assume the database systems use physiological [12] or physical write-ahead logging (WAL). Each physiological log and physical log record describes the changes to a single page. Replaying a log record will, as a result, changes only the single page that is described by the log record. WAL ensures that whenever a change in database page happens, the log record is written to the persistent storage first.

We assume that replaying a log record will result in changes to a database page that are *equivalent* to the changes that resulted from the original update. In the case of physiological logging, the physical page state resulting from log replay may differ from the physical page state resulting from the original update. However, the two states are logically equivalent from the perspective of the DBMS. Hence, the database pages written by the standby can

be recognized by the active. Oracle 12c [1] takes advantage of this property to fix block corruption at the active by using a block from standby, and vice versa.

We assume that when an update is applied on a page, a Log Sequence Number (LSN) is generated. The LSN is stored in both the updated page and the log record that describes the update. LSNs are mononitically increasing. Therefore, LSNs can be used to determine the freshness of a page.

Finally, we assume that the database systems can perform periodic checkpoint operations, and that as part of a checkpoint operation, the DBMS will identify a new *log recovery point*, which is the LSN from which log replay should start in the event of a DBMS failure after the checkpoint. As part of its checkpoint operation, the DBMS is assumed to record the new log recovery point in a *recovery file* in persistent storage. We assume that the DBMS is free to discard, at any time, log records with LSNs earlier than the current log recovery point, since such log records would not be needed to recover the database in the event of a failure. We refer to this process as *log truncation*.

In the following, we describe the states and operations shown in Figure 3.1 in more detail. For the purposes of the discussion in this section, we will assume that both the active and standby systems can hold a complete copy of the database in their local cache. We describe how to relax this assumption in Section 4.

3.2 Stand Alone State

At the beginning, only one active DBMS is running. We call it the STANDALONE ACTIVE state. This is not a high availability state. The system loses availability when the active is down or recovering. The recovery time can be reduced by checkpointing more frequently. However, checkpointing has a significant overhead for the database system. The PROTECT operation, which is discussed in next section, can bring the standalone system into a high availability state.

3.3 Protect

The PROTECT operation is used to create a standby DBMS, moving the system into the ACTIVE+STANDBY state. The ACTIVE+STANDBY state is the normal high availability operational state of the system.

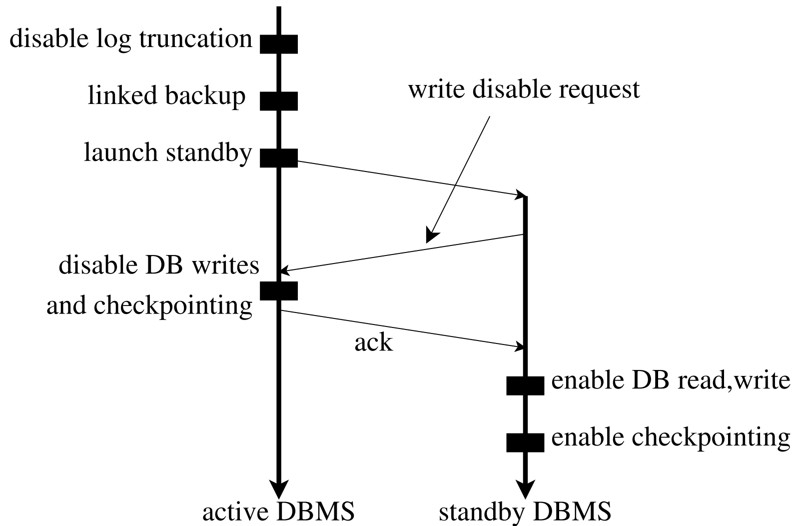

Figure 3.2: Protection Operation Timeline

A typical approach to creating a standby DBMS is to create a backup copy of the active database and then deploy the standby. The backup database copy serves as the initial state of the database managed by the standby. As part of the process of creating the backup, the active DBMS identifies a *replay start LSN*, from which the standby will begin replaying the log. The replay start LSN is chosen to ensure that any updates that may not be present in the backup will be replayable from the log. The active also needs to ensure that enough logs are reserved so that all log records with LSN larger than the replay start LSN are available when the standby starts up.

As an optimization, the active will perform a checkpoint operation before creating the backup copy. In this way, the backup copy represents a more recent state and the standby can catch up with the active system much faster. In addition, a checkpoint allows the active DBMS to truncate the logs.

Figure 3.2 illustrates how the PROTECT operation is performed in a SHADOW system. It differs in several ways from this typical procedure. First, because of the shared storage assumption, there is no need to create a new copy of the database for use by the standby. We assume that the persistent state of each DBMS includes the log, the database, and additional instance-specific metadata, such as configuration files and the recovery file, as shown in Figure 3.3. To deploy the standby system, the active DBMS copies only the data shown in the dashed line in Figure 3.3, not the database and the log. We refer to this

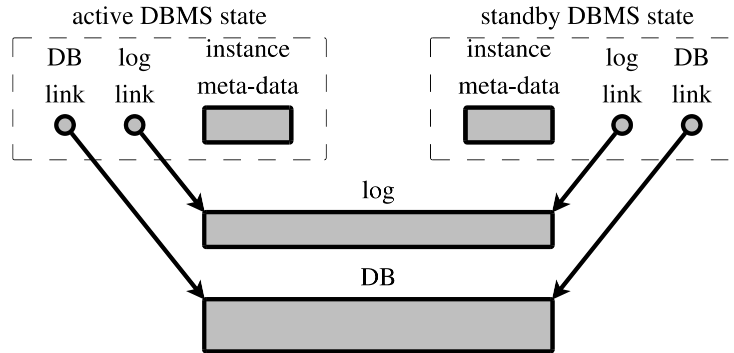

Figure 3.3: DBMS Persistent State After Linked Backup

process as *linked backup*. As a result, the active and standby systems will have private copies of the instance specific meta-data, but will share a single copy of the log and a single copy of the database. Linked backup is much faster than the normal backup process because there is no need to copy the database.

Second, PROTECT transfers responsibility for updating the shared persistent copy of the database from the active DBMS to the standby. As shown in Figure 3.2, the standby DBMS requests that the active DBMS disable its database updates before the standby commences log replay. This ensures that, at any time, at most one of the two systems is updating the shared database.

Third, PROTECT transfers responsibility for checkpointing and log truncation from the active system to the standby. By disabling log truncation before performing the linked backup, active DBMS ensures that the shared persistent log will contain all of the log records identified by the backup operation as necessary for replay at the standby.

3.4 Active + Standby State

In the ACTIVE+STANDBY state, the standby DBMS operates in *log replay mode*, as is the case in other log shipping hot standby systems. This means that the standby continuously reads log entries written by the active system, and re-executes the logged updates. The standby system also pushes changed database pages from its local database cache to the shared, persistent copy of the database. The standby DBMS also performs periodic checkpoint operations and truncates the shared, persistent database log.

Since we assume in this section that database cache at the active DBMS is large enough to hold the whole database, the active can always get up-to-date pages from its database cache as long as the page has been read once. We will discuss potential problems when the database cache cannot hold the whole database, and how to solve to problem, in Section 4.

3.5 Failover

In case of a failure of the active system in the ACTIVE+STANDBY state, the system switches to DETACHED STANDBY state, in which only the standy system is running. The duration of this state is minimized by performing a failover operation immediately. During failover, the standby is promoted and becomes a standalone active system.

SHADOW shares most failover procedures with traditional HADBs. The standby finishes processing all log records generated by the active system before it failed, and then switches from log replay mode to normal execution mode, in which it can accept new transaction requests from clients.

There is one subtle difference between failover in SHADOW and traditional failover. In the traditional ACTIVE+STANDBY state, the standby stores the logs shipped from the active in its private storage. Hence, during traditional failover, the standby only replays all the logs that it has fetched and stored in its private storage. Getting the logs that have not been shipped by the failed active requires some special effort. In contrast, the SHADOW logs are shared and available to both the active and standby. Hence, the standby can fetch the latest logs generated by the active from the shared persistent storage, even if the active has failed. When the standby becomes active and starts generating logs, the logs are written to the shared storage.

3.6 Re-protect

In case of a failure of the standby system in the ACTIVE+STANDBY state, the system switches to DETACHED ACTIVE state.

In the DETACHED ACTIVE state, the active DBMS keeps processing transactions using its local cached copy of the database. Thus, the SHADOW system is still available, and transactions are still durably committed because the active DBMS writes its log records to shared persistent storage. However, since there is no standby DBMS to replay the log, the amount of un-replayed log will grow and the persistent copy of the database will

become staler. This will not affect the performance of the active DBMS, provided that it has enough local storage to hold the entire database. However, it may affect the active's performance if the active has less memory. We discuss this situation further in Section 4.

From the DETACHED ACTIVE state, a RE-PROTECT operation can be used to move the system back into ACTIVE+STANDBY state. Traditionally, there are normally two ways to RE-PROTECT. The first method is to launch a new standby using the database and logs left over by the failed standby. The first method has two requirements. First, the database and logs of the failed standby must still be available. This requirement can be met by storing database and logs in shared persistent storage. Second, the standby must be able to find the correct log restart point. The SHADOW standby uses the ARIES recovery algorithm, which can handle repeated failures correctly. When the standby restarts, it will restart log replay from the latest checkpoint using the logs in shared storage. Since the active delegates the log truncation responsiblity to the standby, the standby can always obtain all log records necessary for RE-PROTECT.

The second option is to create a new database copy and launch the standby using the new database copy. This option is available to most HADB since the procedure is exactly the same as PROTECT. However, this option is normally more expensive than the first option since another copy of database needs to be created.

SHADOW only uses the first way to implement RE-PROTECT. The second way is not available to SHADOW since the active has surrendered its write ability during the PROTECT operation. The active is unable to create a new copy of database.

3.7 DBMS Recovery

The SHADOW system can reach the DOWN state in two ways: First, the Standalone node may fail when it is running in the STANDALONE ACTIVE state. It has never entered the ACTIVE+STANDBY state before. Second, both the active and standby are down. Before that the systems could probably run in DETACHED ACTIVE or DETACHED STANDBY or ACTIVE+STANDBY state. DBMS recovery for the first case is easy. We can simply restart the standalone node. However, recovering the DBMS in the second case is more complex.

To recover the system in the second condition, SHADOW reuses the oprations of RE-PROTECT and FAILOVER. First, the RE-PROTECT recovers the failed standby. The standby will replays all the logs left over by the failed active in shared persistent storage. After that, the system is in the DETACHED STANDBY state. Then a FAILOVER operation is performed

to promote the standby as the new active. At this point the system is in the STANDALONE ACTIVE state and is able to process user workloads.

Another possible solution is to directly restart the active. However, this requires that the active system to find out the last checkpoint performed by the standby. To understand the reason, we need to briely describe about the checkpoint mechanism in SHADOW here. More details are covered in Section 5. In SHADOW active+standby state, the active system does not perform database writes and checkpoints. It delegates the checkpoint responsibility to the standby. The active only writes the checkpoint record to the logs. The standby then replays the checkpoint record and performs the checkpoint. Hence, the last checkpoint generated by the active might not be replayed. In order to correctly restart the system, the active needs to find out which checkpoint was replayed by the standby using the standby's checkpoint meta data.

3.8 Correctness of SHADOW

In this section we present a set of invariant properties that are preserved during the operation of a SHADOW system. For each property, we present an informal argument to explain why the property is preserved.

Property 1 *All committed updates are present in persistent storage, either in the database, in the log, or in both. If both active and standby fail, normal database recovery using the persistent database and log will correctly restore the database, with no loss of committed transactions.*

In a SHADOW system, write-ahead logging by the active DBMS ensures that the updates of committed transactions are in the persistent log before a transaction commits. The only threat to Property 1 is the possibility that log truncation will remove persistent logged updates before they have been applied to the persistent copy of the database. In the STANDALONE ACTIVE state, the normal checkpointing and log truncation mechanism of the active DBMS ensures that log records will not be truncated prematurely. During PROTECT, disabling log truncation at the active prior to the linked backup operation ensures that all logged updates that may not have been applied to the database are available to the standby for replay. In the ACTIVE+STANDBY and DETACHED STANDBY states, the standby's normal checkpointing and log truncation mechanism prevents log records from being truncated too soon.

Property 2 *At most one DBMS at a time is responsible for updating the database, for checkpointing, and for truncating the log.*

Property 2 follows immediately from the PROTECT operation, which disables updates, log truncation and checkpointing at the active before enabling them at the standby.

Property 3 *Suppose that the standby system has replayed log records up through LSN L. All database pages in the standby's local database cache are current up through L, i.e., they include all updates with log sequence numbers less than or equal to L. Furthermore, all database pages that are* not *in the standby's local database cache are current up through L in the persistent copy of the database.*

Property 3 holds when the standby system is first deployed because the active system's linked backup operation (Figure 3.2) ensures that all updates prior the replay start LSN are already in the persistent database. During operation of the standby in the ACTIVE+STANDBY and DETACHED STANDBY states, the property is maintained as it replays each log entry onto a page in its local database cache. Since the standby is assumed to have enough local storage to hold the database, it will not evict pages from its local database cache. However, even if it were to do so, Property 3 would be maintained by the usual procedure of pushing dirty pages to persistent storage before evicting them. If the standby fails and is restarted, the property is maintained because the standby's current LSN will revert back to the replay LSN recorded in the standby's most recent checkpoint.

Property 4 *When the active system reads page P from persistent storage, it reads the current version of P.*

Property 4 holds when active first starts due to normal log-based recovery process. In a SHADOW system, the threat to Property 4 is that the active DBMS will update a page locally, evict it, and then attempt to re-read that page from the persistent storage system. Since active does not write to the persistent database when the page is evicted, the page may be stale when re-read. However, since the active DBMS is assumed to have sufficient memory to cache the entire database locally, this scenario cannot occur. Each database page will be read at most once from persistent storage by the active DBMS. Property 4 also holds initially when the standby DBMS becomes the active as a result of a FAILOVER operation. As part of the FAILOVER operation, the standby replays all outstanding log records before becoming the active. Thus, because of Property 3, all pages, whether initially cached in the new active system or not, will be current as of the point of failover.

Chapter 4

Large Databases

So far, we have assumed that each DBMS has enough local cache to cache a complete copy of the database. In this section, we relax this assumption. When the local cache is large, the active DBMS will read a page from persistent storage at most once, since the page can remain indefinitely in the local cache once it is read. If the active DBMS has less local cache, it may have to read a page more than once from persistent storage. In particular, the active may read a page P, later evict P from its local cache, and later still read P again from persistent storage. At this point, the active DBMS faces a race condition. If the active reloads P after the standby has applied the logged updates to P and flushed P to persistent storage, then the active will read the current version of P. Otherwise, the version of P in persistent storage will be stale, and the active DBMS will read a stale version of P. This is a violation of Property 4 (Section 3.8).

To address this problem, we introduce a new mechanism in the active DBMS to allow it to detect stale reads if they occur. We also introduce mechanisms in the active and standby database systems to allow the active system to obtain a current version of the page if it detects a stale read. In Section 4.1 we describe how stale reads are detected, and in Section 4.2 we describe how they are corrected when they occur.

4.1 Stale Read Detection

When the active DBMS evicts a dirty page from its local database cache, it records the page ID and page LSN of the evicted page into an in-memory hash table called the *page version table*, keyed by page ID. An entry for page P in the page version table indicates that subsequent reads of P should return the version with the recorded page LSN.

When the active DBMS reads a page from the persistent storage system, it compares the LSN of the returned page with page's LSN from the page version table. There are three possible cases:

1. The read page may have a smaller LSN than is recorded in the page version table. In this case, the active has read a stale page from the persistent storage system, which is handled as described in Section 4.2.
2. The read page's LSN may match the one found in the page version table. In this case, the active has read the latest version of the page. The active uses the page, and removes the page's entry from the page version table.
3. There may not be an entry for this page in the page version table. In this case, the page read from the persistent storage system must be fresh.

This behavior ensures that the page version table includes entries for *all pages for which there is a risk of a stale read* from persistent storage. The period of risk begins when the active DBMS evicts a dirty page, and continues until the active system confirms (during a subsequent read) that the persistent copy of the page is up to date. Thus, in case 3 above, the lack of an entry in the page version table indicates that the persistent copy of the page is current.

4.2 Correcting Stale Reads

When the active DBMS reads a stale page from persistent storage, one option it has is to retry the read from persistent storage until it obtains the latest version. As the standby DBMS processes log records and pushes updated database pages to persistent storage, the persistent copy of the database grows fresher. In general, however, the active DBMS may have to wait a long time, depending on how quickly the standby DBMS flushes updated pages to persistent storage.

Instead, SHADOW implements *Page Fetching* to address the problem of stale reads. Our SHADOW prototypes take advantage of the fact that the local database copy cached at the standby will be fresher than the persistent copy. If the active DBMS reads a stale page from the persistent storage system, it next tries to obtain the current version of the page directly from the standby DBMS. To support this, we modified the standby system so that it can act as a database page server. To read a page from the standby, the active

```
1: function READ(p)
2:     // get required page LSN from page version table
3:     LSN_req ← PAGEVERSIONTABLELOOKUP(p)
4:     // fetch p from persistent DB, returning page LSN
5:     LSN_read ← PERSISTENTSTOREFETCH(p)
6:     if LSN_req = NULL then
7:         // version from persistent DB is current
8:         return OK
9:     else if LSN_req = LSN_read then
10:        // version from persistent DB is current
11:        PAGEVERSIONTABLEREMOVE(p)
12:        return OK
13:    else
14:        // stale read from persistent DB
15:        // try to get page from the standby
16:        if STANDBYFETCH(p, LSN_req) = OK then
17:            // standby had the current version
18:            return OK
19:        else
20:            // standby did not have the current version
21:            return error
22:        end if
23:    end if
24: end function
```

Figure 4.1: Reading a Page at the Active DBMS

DBMS sends a request specifying the ID of the required page and the expected page LSN for that page, which the active obtains from its page version table.

In response to such a request, the standby first searches for the page in its local cache. If the page is not in cache, the standby tries to read it from persistent storage. If the standby cannot find the page in cache and persistent storage (the log records have not been replayed yet), it returns a negative response. After finding the page, the standby compares the page LSN from the page it just found with the page LSN specified in the request. If the two LSNs are the same, the standby returns the page. Otherwise, the standby returns a negative response.

If the active receives a negative response, the active DBMS could potentially get the

fresh page by retrying its request one or more times until the standby's copy is no longer stale. However, in our current prototype implementation, the active treats a negative response from the standby as a failed read attempt, and immediately aborts the transaction that required the stale page. In general, we expect the standby's local copy of the database to be almost current, so we expect such aborts to be rare. Figure 4.1 summarizes the read procedure used by the active DBMS.

4.3 Discussion

In order to correct stale reads, the active relies on the standby system. If the standby is down, the active is unable to fetch pages from the standby, and therefore some transactions are aborted. We discuss some potential alternative solutions to adress this problem.

4.3.1 Alternative Stale Read Correcting Methods

We propose two solutions to reduce the active's dependence on the standby. The first solution tries to avoid the condition that the active reads back a page immediately after evicting it. The second solution eliminates the dependency. Hence, the active is able to keep processing transactions when the standby is down.

- Change buffering. The idea comes from InnoDB. By buffering the change logs of recent updates, the active is able to reconstruct a fresh page. The change logs could be discarded in the FIFO manner. When the active reads a stale page from persistent storage, it first tries to find change logs for the page. If the change logs for the page have been discarded, which means the change was done some time ago, the standby is likely to have the fresh page.
- Re-scan redo logs. If the standby has not made a fresh page persistent in shared storage, the logs describing the changes of the page must be available in the persistent storage. Hence, the active is able to get the redo logs for the stale page and bring the page up-to-date by replaying those redo logs. The position of the latest redo logs relevant the stale page can be calculated using the LSN cached in the page version table. A backtrace is performed from the log record with the latest LSN until the oldest log record that has not been applied to the stale page is found. This re-scan method might cost more time than simply retrying page fetching since the standby is normally very close to the active. However, it can avoid aborting transaction when the standby is down.

Chapter 5

Prototype Implementation

We have implemented a SHADOW prototype called SHADOW-NFS. Figure 5.1 illustrates the SHADOW-NFS prototype, which we have deployed for evaluation purposes in Amazon's EC2 cloud computing environment. It consists of two major components, described below.

The first major component is the DBMS. We implemented the DBMS of SHADOW-NFS based on PostgreSQL (version 9.2). PostgreSQL is an open source relational database system widely used in both academia and industry. It provides built-in DBMS-level replication. Since SHADOW-NFS uses storage-level replication for high availability, some modifications to the PostgreSQL source code were necessary. We describe the modifications in Section 5.1. More details about the PostgreSQL built-in DBSM-level replication are described in the Related Work chapter (Chapter 7).

The second major component is the shared storage. We uses a shared file system [8], NFS (version 4), as our shared storage. Shared file systems are a widely used type of shared storage. Shared filesystems are often referred to as Network Attached Storage, or network file services such as NFS or CIFS. Shared file systems can be accessed simultaneously by multiple servers. Database systems are deployed using shared file systems because they provides flexibility and manageability for persistent storage. Shared file systems also provide a simple way to implement failover to a cold standby system in case of a failure of the active DBMS, since persistent storage is network-accessible, rather than locally attached to the active DBMS. Shared file systems can be made reliable, fault tolerant and highly available through the use of redundant storage of file data (e.g., mirroring or other RAID configurations) and redundant servers [21]. The implemenatation details of the shared storage (i.e. NFS) for SHADOW-NFS are described in Section 5.2.

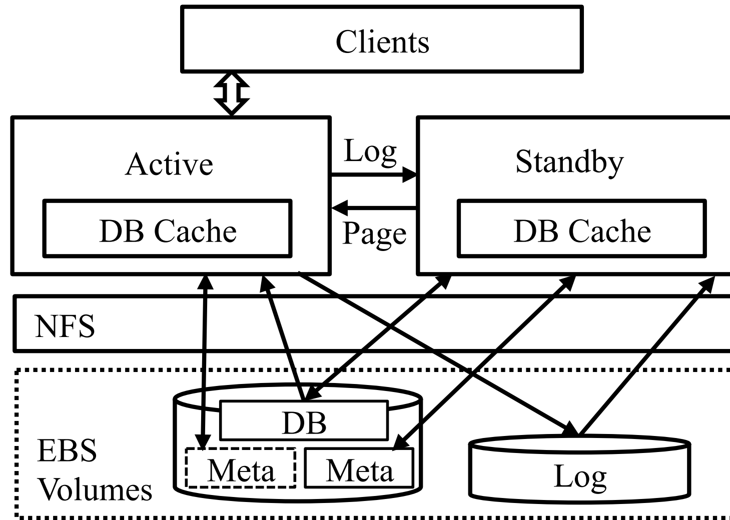

Figure 5.1: The SHADOW-NFS Prototype

5.1 DBMS for SHADOW-NFS

As mentioned previously, the DBMS component for SHADOW-NFS is implemented based on PostgreSQL (version 9.2). We describe the major modifications in PostgreSQL in this section.

New Pages and Database File Sizes. When PostgreSQL is changing the size of a database file, e.g., to add additional pages to the database, it relies on the file system to determine the current size of the file. In a SHADOW system, since the active DBMS does not actually update the database files, it may see temporarily incorrect file sizes until updates are replayed by the standby. To avoid this problem, we added an in-memory *last block table* to PostgreSQL to record database file lengths. PostgreSQL uses this table, instead of the file system, to determine file sizes.

To create new database pages, PostgreSQL will extend a database file to create room for the new pages and then load the (uninitialized) page into memory. In a SHADOW system, this page load operation may fail since the file may not yet have been extended. We added a small refinement to the active DBMS page reading procedure (Figure 4.1) to handle this situation. When the active DBMS creates new pages, it adds entries to the page version table for those pages. These entries associate the new page IDs with a special LSN that indicates that the page is new. When PostgreSQL tries to read these pages, the

page version table lookup performed by the page reading algorithm indicates that the page is new, and the active simply creates the uninitialized page in its buffer without attempting to read it from persistent storage (or the standby's page server). If the page is eventually updated and evicted from the active's local storage, the page's special LSN will be replaced in the page version table with the page's pageLSN, as usual.

Checkpointing. A peculiarity of checkpointing in PostgreSQL hot standby configurations is that the standby DBMS cannot checkpoint more frequently than the active DBMS checkpoints. This is because checkpointing requires creation of a checkpoint record in the log. Since a PostgreSQL standby system replays log records but does not generate log records of its own, the standby system essentially re-uses the active's checkpoint records to implement its own checkpoints.

In a SHADOW system, it is not necessary for the active to checkpoint at all. However, our SHADOW-NFS prototype needs the active's checkpoint records to enable the standby system to checkpoint. To do this, in the ACTIVE+STANDBY state we replace checkpointing at the active DBMS with *pseudo-checkpointing*. With pseudo-checkpointing, the active goes through its normal checkpointing procedure, including creation of a checkpoint record in the log, but *without writing any database pages from local cache to the persistent database*. Thus, a pseudo-checkpoint is much faster, and results in much less overhead, than a normal checkpoint. The standby DBMS can then use the active's logged checkpoint records to implement its own checkpoints. In our prototypes, the active DBMS must be configured to pseudo-checkpoint at least as frequently as the standby will checkpoint, so that the standby has a sufficient number of logged checkpoint records.

Disabling Database Writes In ACTIVE+STANDBY state, a SHADOW active system does not write to the DBMS. We implemented this by disabling writes at a low-level PostgreSQL internal I/O interface. On each write attempt, PostgreSQL checks for the existence of a special control file that indicates that writes should be blocked. If the file exists, PostgreSQL simply skips the file system write operation that it would otherwise have performed. This is a simple way to eliminate page writing without making major changes to the PostgreSQL implementation. However, a disadvantage of this approach is that PostgreSQL still experiences the internal overhead associated with page writes, e.g., it still tracks which pages are clean and dirty, and it still maintains the internal state necessary to flush dirty pages to the persistent database. Further improvements in SHADOW system performance could potentially be obtained by eliminating these overheads from the active DBMS.

Log Streaming PostgreSQL implements a mechanism for streaming log pages directly

from the active system to the standby in high-availability configurations. As described in Sections 5.2, we have taken advantage of this mechanism in our SHADOW prototype. We were able to use this mechanism as is, without making any changes.

Page Fetching Page Fetching is used to fetch fresh pages from standby system in case of a stale read at the active DBMS. Page Fetching in SHADOW has gone through several versions. In the first version, the active appended page requests to a shared file, which was polled by the standby. When the standby read a page request, it flushed the requested page to the shared storage. The active kept reading from the shared storage until a fresh page was read. Then the active cleared the entry in page version table. The advantage of writing page requests to a file is easier debugging. However, it is too slow when page requests happen very frequently. The advantage of flushing the page to shared storage is that the active may be able to read the fresh page from shared storage next time.

The second implementation of page fetching used a client/server model based on a TCP connection. The active established a TCP connection with the standby and sent the page requests via the connection. The standby returned the page via the connection if the fresh page was available. The active retried page fetches when the fresh page was not available. There are two problems with this second approach. Establishing a TCP connection for each page fetch is a potential overhead. TCP is based on byte streams. Hence, extra effort is needed to ensure that a complete page is fetched. Another problem is that the retry mechanism is not very effective. Retrying a few times might not eliminate failed page fetching. However, retrying too many times reduces performance.

The final implementation of page fetching uses UDP without retry. Each page fits in a UDP packet easily. SHADOW can check the page validity with a checksum. No connection needs to be established with the UDP protocol. As shown in Section 6, the page fetching failure ratio is very low even if the database cache is relatively small.

Page Service Another implmentation problem is when to start the page service at the standby. At the beginning, the page service was started after the standby started log streaming. However, the PostgreSQL standby does not start log streaming until it has replayed all locally stored logs. The locally stored logs could be huge if the standby has not checkpointed for a long time and then fails. When the standby restarts, it needs to replay all logs since the last checkpoint. However, we want standby to start page service as soon as possible since the active relies on it. Thus, standby starts page service as soon as it starts log replaying.

5.2 Shared Storage for SHADOW-NFS

The shared storage service used by SHADOW-NFS is NFS (version 4). We describe the NFS configuration used by our SHADOW-NFS prototype in this section.

Since the SHADOW-NFS is tested in the Amazon EC2 environment, the NFS shared file system mounts Elastic Block Service (EBS) volumes as its block volumes. SHADOW-NFS uses two EBS volumes, one for the database and one for the log. Both volumes are mounted by an NFS server, which provides database and log access to the active and standby database system through a pair of NFS mount points. In our prototype, the NFS server is a threat to system availability, since it represents a single point of failure. However, in practice this problem is easily avoided through the use of a standby NFS server.

In SHADOW-NFS, the standby DBMS can read log records directly from the NFS-mounted log volume, as described in Section 2. However, as an optimization, we also use PostgreSQL's existing asynchronous log streaming mechanism (Section 5.1) to stream log records directly from active DBMS to the standby, as illustrated in Figure 5.1. In practice, this reduces the log replay delay at the standby. However, this is only an optimization; It is still the case that transactions are durably committed in a SHADOW system if and only if the transaction's commit record is present in the log volume. During a FAILOVER operation in SHADOW-NFS, the standby DBMS checks the shared log for any remaining log records that may not have been streamed before the failure and replays them.

5.3 Alternative Shared Reliable Storage

In this section, we look at some other shared reliable storage services. They could potentially be used as the shared persistent storage for SHADOW.

5.3.1 Dynamo

Dynamo [9] is a highly available, scalable key-value store implemented by Amazon. Data is distributed using consistent hashing [13]. Multiple clients can write and read any value concurrently. Conflict resolution is implemented to guarantee eventual consistency. Each value is replicated on N different physical nodes. A write operation writes to all replicas and succeeds as long as W replicas are updated. A read operation reads from all replicas and succeeds as long as R replicas return. $W + R > N$ is used to guarantee the read of latest vesion. $W > N/2$ is used to guarantee global write orders.

Dynamo could potentially be used to store SHADOW logs. LSN is used as key and the log record is used as value. Since the active only appends new logs to the shared storage and each log record is written at most once, no conflict resolution is required. W should be configured large enough to achieve the expected reliability. The standby reads logs from the shared storage. R can be set to one since each log record is written only once. The read log record must be the up-to-date one. There is no need to guarantee $W + R > N$ and $W > N/2$.

Some effort is required to store database pages in Dynamo, since the standby updates and reads the shared database. Strongly consistent write (e.g. write to all replicas) can be adopted by standby since the standby performance does not directly influence the active performance. Since each update is written to all replicas, the active can finish a read as soon as one replica returns.

We just described the potential solution to run SHADOW on Dynamo when it is in ACTIVE+STANDBY state. However, it is challenging to run SHADOW on Dynamo when it is in STANDALONE ACTIVE state. In ACTIVE+STANDBY state, only the standby reads/writes the database pages. Hence, the read/write could be slow. However, in STANDALONE ACTIVE state, the standalone needs to read/write the database efficiently. This performance problem could be solved by DAX [18], another shared storage service.

5.3.2 DAX

DAX [18] is a shared storage service based on Dynamo. It is designed to support multiple DBMS tenants, each of which is a standalone active system. When a DBMS tenanent fails, it can be restarted immediately since DAX replicates its database and logs within and across data centers. Optimistic I/O, a complex data version management protocol, is implemented to optimize read/write performance of the standalone active system, while ensuring strong consistency from the DBMS perspective.

In previous subsection, we described the performance problem of SHADOW on Dynamo when it is running in STANDALONE ACTIVE state. DAX is able to provide good performance when DBMS is running in STANDALONE ACTIVE state and guarantees correctness. A detailed design of using DAX as the storage service for SHADOW is beyond the scope of this thesis.

Chapter 6

Evaluation

In this section, we present our evaluation of the SHADOW-NFS system. Our objectives are (i) to measure the performance of SHADOW-NFS system in terms of throughput, latency, time to enable protection and time to handle failures, (ii) to compare the performance of SHADOW-NFS systems against a standalone (SA) database system, which does not provide high availability, and (iii) to compare the performance of SHADOW-NFS system against a system that provides synchronous replication (SR), which provides high-availability, but without shared storage and write offloading.

6.1 Experiment Methodology

We have implemented the SHADOW-NFS prototype system using PostgreSQL (version 9.2) and NFS (version 4). We use this same version of PostgreSQL and NFS for our SA and SR baseline systems. For the SA baseline, the frequency with which the DBMS checkpoints controls a tradeoff between performance during normal operation and recovery time. Thus, we used several different SA baselines, with different checkpointing frequencies, to reflect this tradeoff. The configurations of the baseline systems are as follows:

- SA-D: This standalone baseline uses the default PostgreSQL configurations, which checkpointed every 7 to 29 seconds in our experiments. This configuration represents an extreme tradeoff of normal performance in favor of fast recovery time.
- SA: This standalone baseline is identical to SA-D except that it is set to checkpoint minimally, i.e., once every hour. This configuration represents the opposite extreme on the performance vs. recovery time tradeoff for standalone systems.

- SA-10: This standalone baseline identical to SA-D except that the time between checkpoints is set to 10 minutes. It represents a balance between the extremes of SA and SA-D.
- SR: This baseline uses PostgreSQL's native synchronous replication to provide high availability. It includes two database systems, one active and one standby. Each server instance manages its own persistent copy of the database and its own log. The active system streams logged updates to the standby, which replays them against its copy of the database. Synchronous replication ensures that a transaction commits only after it is committed at the active system and its log records are persistent at the standby. Both the active and the standby are configured to checkpoint minimally, like the SA baseline.

We compared these baselines against the SHADOW-NFS prototype described in Section 5. When comparing the baselines to SHADOW-NFS, we used NFS for the baselines as well. The log and the database were always stored in separate standard EBS volumes. For the SA, SA-10, and SA-D baselines, the DBMS mounted these volumes through a single NFS server (like SHADOW-NFS). The SR baseline, which uses two database volumes and two log volumes, used two NFS servers, one for the active DBMS and one for the standby.

All of our experiments were run in the Amazon's Elastic Compute Cloud (EC2) environment[1]. c1.xlarge instances, which have 8 virtual CPUs and 7GB of memory, were used for the active, standby, and standalone in most experiments. We also tested a large dataset configuration using c3.4xlarge instances for active, standby and standalone systems. c3.4xlarge instances have 16 virtual CPUs and 30 GB of memory. m1.large instances, with 2 virtual CPUs and 7.5GB of memory, were used for the NFS servers. We selected instance types that were sufficiently powerful that the CPU was not the performance bottlneck in our experiments.

Our experiments are run using the TPC-C benchmark. It submits transactions only to the active. No queries are sent to the standby. Each experiment is run with 10 warehouses (except the large dataset experiment, which is described in 6.4), with an initial database size of approximately 1.1GB. The database size grows to around 1.5GB after 5 minutes, and to over 3GB after 30 minutes. We used 30 TPC-C terminals (clients) for all experiments. Each TPC-C experiment was repeated three times, and we report average of the throughput obtained on the three runs.

[1]Resource specifications of these instances are detailed at http://aws.amazon.com/ec2/instance-types/#instance-details.

	SA	SA-D	SA-10	SR	SHADOW
Large Memory	477	4,591	1,125	397	5
Small Memory	1,790	4,250	1,753	1,101	10
Large Dataset	3,554	5,731	5,192	2,967	6

Figure 6.1: Database Writes (KB/s) between Active and NFS server

Percentile Latency	**50%-tile**	**99%-tile**
SR	24	758
SA-10	21	686
SA	23	634
SHADOW	22	530

Figure 6.2: 50th and 99th Percentile NewOrder Transaction Latency (ms)

6.2 Large Memory Case

Our first set of experiments considers system performance during normal operation, in a setting in which there is sufficient memory to hold the entire database locally in each DBMS buffer pool. This is the setting for which SHADOW systems are targeted. Specifically, we set the PostgreSQL database buffer cache size to 4GB, which is large enough to hold the whole database, even after it grows.

Figure 6.3 shows the TPC-C throughput for SHADOW-NFS and the baselines. The SHADOW-NFS prototype provides higher performance than PostgreSQL's native high availability technique (SR). This is because the SHADOW-NFS architecture eliminates the need for commit coordination between the active and standby databases, since it manages only a single logical copy of the database.

SHADOW-NFS also provides throughput that is comparable to that of the SA baseline, which is not highly available and which would have a very long recovery time because of its infrequent checkpointing. The SHADOW-NFS system outperforms SA-10 and substantially outperforms SA-D in both settings, despite the fact that the SHADOW-NFS system provides high availability while SA-10 and SA-D do not. These performance gains come from write offloading. The first row of Figure 6.1 shows the database write bandwidth between the database server and NFS server for SHADOW-NFS and the baseline systems in the large memory configuration. These shows that SA-10 generates about 1.1 MB/s of database writes I/O, which is eliminated by the SHADOW-NFS system. The performance advantage of SHADOW-NFS is not very significant since the write I/O is not very intense.

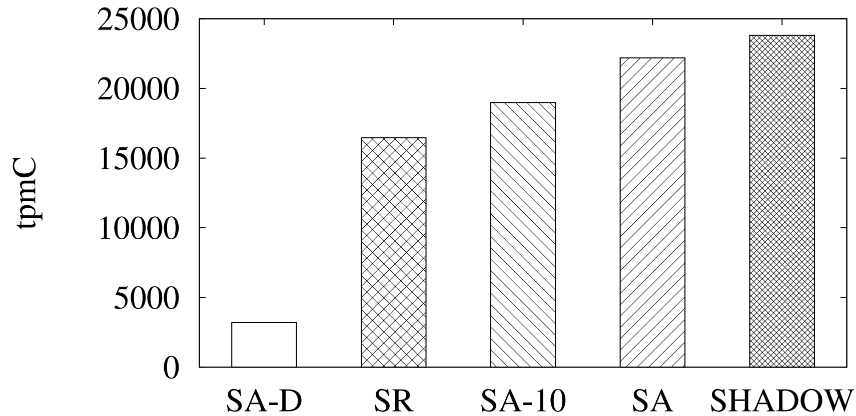

Figure 6.3: TPC-C Throughput, Large Memory

We will further show the performance advange of SHADOW-NFS in Section 6.4, using a larger dataset.

Because of the relatively large memory size, neither the baseline system nor the SHADOW-NFS system generates a significant amount of read traffic to the persistent database. Thus, the SHADOW-NFS system is able to eliminate almost all I/O to the persistent database, and is limited in its performance only by the local processing capacity of the active server and by the write bandwidth that is available to the persistent log.

We also measured the latencies of TPC-C NewOrder transactions. These are shown, for all configurations except SA-D, in Figure 6.2. All systems show similar 50th-percentile latencies except for SR, which has slightly higher latencies because of the extra commit coordination it requires. At the 99th percentile, differences among the systems are more exaggerated. I/O activity triggered by checkpointing can result in temporary latency spikes, which hurts SA-10 and, to a lesser extent, SA. SR suffers from its additional coordination latency when transactions commit. The SHADOW-NFS system, which has almost no database writes and no coordination overhead, avoids these latency spikes.

6.3 Small Memory Case

Our next set of experiments considers system performance in a scenario in which the DBMS buffer cache is not large enough to hold the entire database. For these experiments,

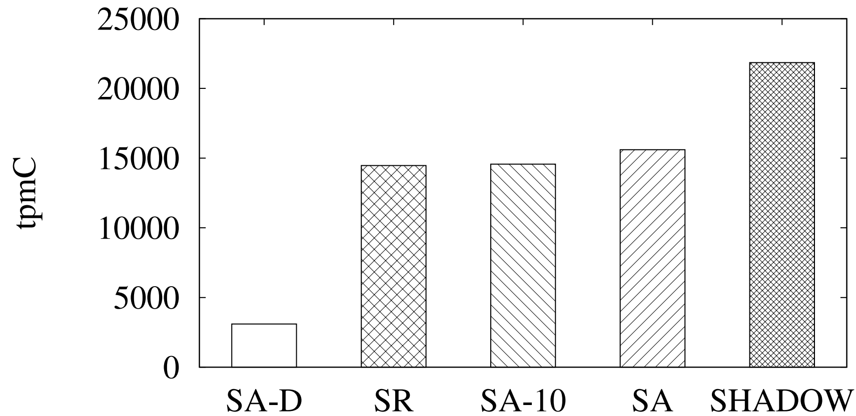

Figure 6.4: TPC-C Throughput, Small Memory

the DBMS page buffer size is set to 1GB. During the 30 minute TPC-C experiment, the database size grows from 1GB to around 3GB.

Because of file system caching at the DBMS servers and at the NFS server, we do not expect to see a significant performance impact from read traffic in these experiments. However, we do expect to see increases in database write traffic between the database server and the NFS server. This is because of the smaller DBMS buffer size, and therefore more intensive dirty page eviction from the DBMS page buffer. In addition, since the DBMS page buffer cannot hold all of the database pages, SHADOW-NFS may experience stale reads. As described in Section 4.1, SHADOW-NFS performs page fetches from the standby DBMS to correct stale reads.

Our objective is to see how these changes affect the performance of the SHADOW-NFS system relative to the baselines.

Figure 6.4 shows the throughput of the SHADOW-NFS system and baselines in the small memory configuration. As shown in Figure 6.4, SA suffered a significant drop in throughput relative to the large memory scenario, while the other systems, including SHADOW-NFS, suffered only minor drops.

As shown in Figure 6.1, the database write rate for the SA system is substantially higher in the small memory scenario than it was when memory was larger. The more intensive write traffic of SA is mostly caused by more frequent write system calls from the DBMS. The DBMS needs to evict dirty pages to make room for newly read pages. This puts more pressure on the OS to write out the dirty pages to the NFS server. In contrast, SA-10 and

SA-D have minor performnace drop since they have no significant increase in write I/O, as shown in Figure 6.1. They don't have significant increase in write I/O because their checkpointing operations have already reduced the ratio of dirty pages in the DBMS cache. SA-D and SA-10 are more likely to read in pages from persistent storage by dropping the clean pages in DBMS cache without write system calls. Like SA, SHADOW-NFS does not perform checkpointing during the experiment and has high dirty page ratio in DBMS cache. Thanks to write-offloading, SHADOW-NFS is able to drop dirty pages when reading new pages from the persistent storage. Hence, as shown in Figure 6.1, it has trivial amount of write I/O, giving it performance advantages.

The active DBMS in the SHADOW-NFS system must sometimes rely on the standby DBMS to obtain the current version of a page that it needs to read (Section 4). We found that SHADOW-NFS generated an average of 247 page read requests per second to the standby's page server over the course of the experiment, resulting in a network data transfer rate of about 2 MB per second between the standby DBMS and the active DBMS. Generally, the active DBMS succeeded in obtaining up-to-date page copies from the standby's page server. On average, only 0.037% of NewOrder transactions were aborted in our SHADOW-NFS experiment due to failure to obtain a current page version from the standby page server. There was little effect on the SHADOW-NFS transaction throughput as a result of this activity. Thus, it appears that the SHADOW-NFS standby page server is an effective way to deliver current versions of database pages to the active DBMS, at least in the current experiment. As the database size gets larger relative to active DBMS's buffer pool size, the request rate to the standby page server is likely to increase. Thus, the page server may eventually become a bottleneck.

One surprising result of our small memory experiments was that the SHADOW-NFS system generated substantially more read I/O traffic between the active DBMS and the NFS server than did any of the baselines: about 3 MB/second, versus about 1 MB/second for the SA baseline. This is partially due to the higher performance of SHADOW-NFS. However, we believe that additional traffic results from maintaining consistency between the NFS client caches at the active and standby systems, for the shared database file system.

6.4 Large Dataset Case

We also evaluated the performance of SHADOW-NFS and the baselines with a large dataset. The experiment methodology was changed because of the larger dataset and

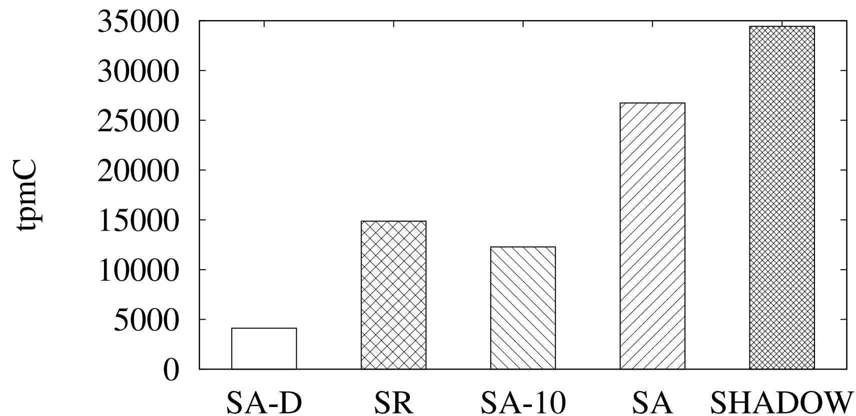

Figure 6.5: TPC-C Throughput, Large Dataset

the use of more powerful EC2 instances (c3.4xlarge). Each experiment runs for 90 minutes. The first 30 minutes is a warm up period and not taken into account. The dataset size grows from 10GB to nearly 20GB during the experiments. In order to hold the dataset in memory, the database server shared buffer size was set to 26GB. The checkpointing period was set to 30 minutes. Hence, SHADOW-NFS and all other baselines include at least one checkpoint during the experiments. We also disabled full-page writes in all of the systems. Full-page writes are an optional feature of PostgreSQL which causes the DBMS to write all database pages into the logs during each checkpoint. We disabled it because it saturates the log volume bandwidth and seriously damages the throughput. We also set the checkpoint_completion_target to 0.9, which means that the checkpointing length will be 90% of the checkpointing interval. It helps distribute the checkpointing overhead evenly during the checkpointing interval. We also write 1 into the drop_caches file in the Linux proc filesystem at 5 second interval. It avoids extensive caching of database pages by the OS.

The average throughput is shown in Figure 6.5. It does not include the first 30 minutes warm up period. SHADOW-NFS outperforms the other configurations significantly. Due to the larger dataset, the benefit of eliminating the write I/O at the active becomes more obvious. SHADOW-NFS is over twice as fast as SR. It also outperforms SA more significantly compared with small dataset case (10 warehouses). In addition, SHADOW-NFS is about three times as fast as SA-10.

Figure 6.6 shows the throughput of all congurations as a function of time. SA-D is omitted here. The vertical lines point to the 30 minute and 60 minute time. SR's warm

up period is much faster than the other configurations since we needed to add an extra reboot to switch it from asynchronous replication mode to synchronous replication mode. The reboot happens to make the warm up much faster. It wouldn't influence the overall result since the experiment length is sufficiently long. SA-10 throughput goes up and down due to the 10-minute checkpointing interval. SA, SR and SHADOW-NFS experience a performance drop after 30 minutes due to the checkpointing operation. SHADOW-NFS has much smaller performance drop compared with SA. This is because SHADOW-NFS does not need to write any database pages during the checkpoint. The small performance drop of SHADOW-NFS is because of the internal overhead during each checkpointing operation. We also observed that SHADOW-NFS's performance is more stable than the other configurations.

The performance shown above can be further explained with the help of Figure 6.7. It shows the write I/O between the active DBMS and the NFS server. The y-axis shows the write bandwitdh and the x-axis shows the time. SHADOW-NFS has almost no write I/O during the experiment. SA and SR experienced two write I/O bursts at 30 minute and 60 minute. They correspond to the two checkpointing operations. SA-10 experienced much higher write I/O overhead due to its higher checkpointing frequency. It needs to flush all dirty pages to the persistent storage within a shorter period of time compared with SA and SR.

6.5 Protection and Failure Handling

In this section, we present a series of experiments designed to compare the behavior of SHADOW-NFS to that of the SR baseline during PROTECT and FAILOVER operations. We first measure the time taken to enable protection. We also present an experiment that characterizes the behavior of SHADOW-NFS in the DETACHED ACTIVE state, when the standby system has failed. We used the small database (10 warehouses) with large memory (4GB) for all of these experiments.

6.5.1 Protection

Our first experiment compares the time required for a PROTECT operation in SHADOW-NFS to that of PostgreSQL native synchronous replication (SR). For this experiment, a single DBMS was started in STANDALONE ACTIVE mode and run for five minutes, at which point a PROTECT operation was initiated to launch a standby DBMS. The PROTECT

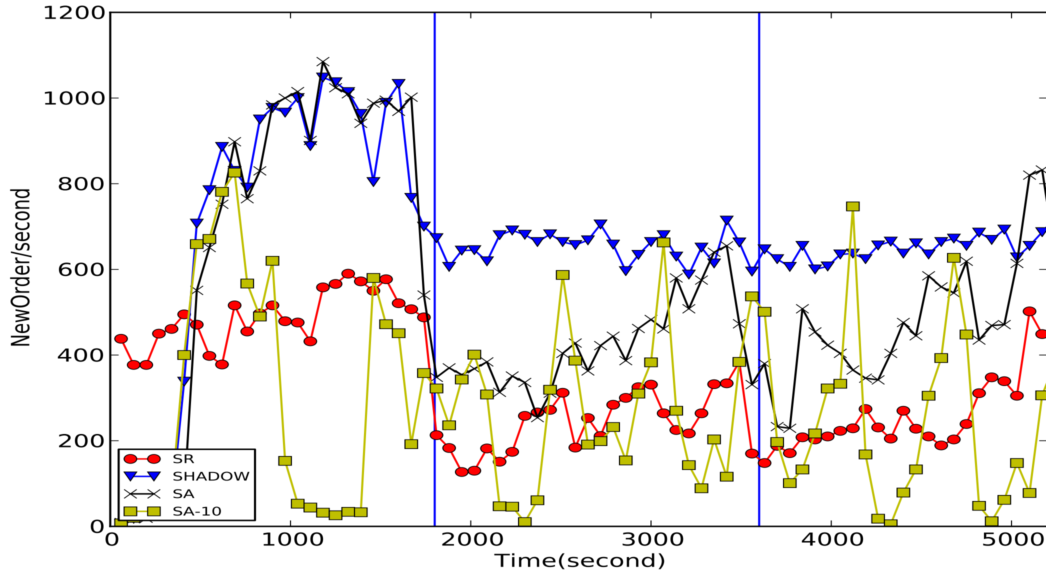

Figure 6.6: TPC-C Throughput, Large Dataset

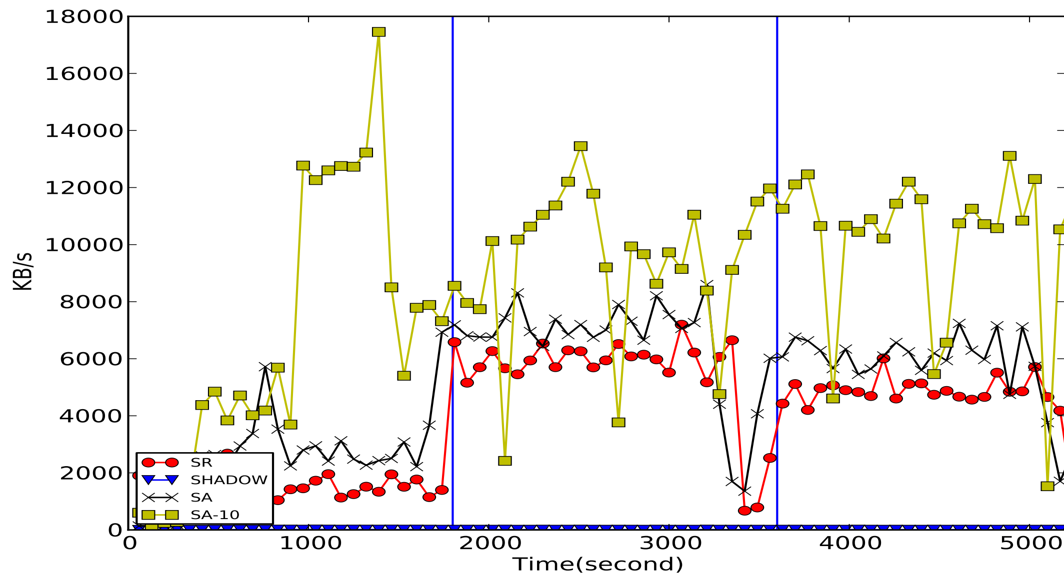

Figure 6.7: Write I/O between the Active and the NFS, Large Dataset

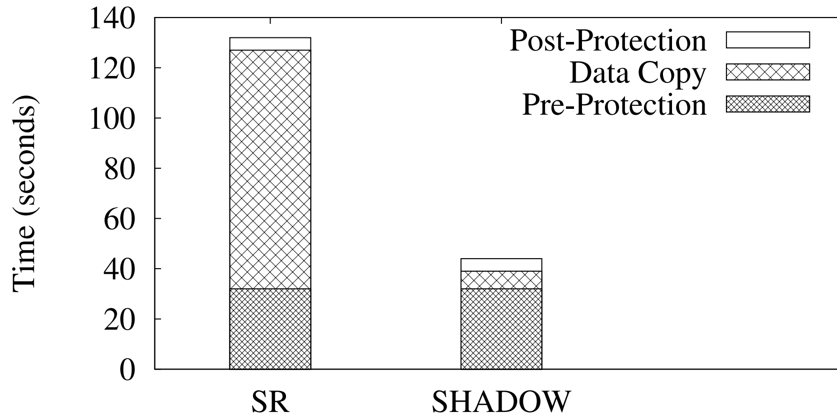

Figure 6.8: Protection Time in SHADOW-NFS and SR

operation can be broken down into three general steps, and we measured the time required for each of these steps. The first step is Pre-Protection, which includes all preparation before duplicating data. The most time consuming part of Pre-Protection is a checkpoint operation, which is performed by PostgreSQL as part of its backup procedure. The second step is Data Copy. For SR, it is necessary to duplicate the database to create a backup copy. In contrast, SHADOW-NFS uses linked backup, which copies only instance metadata, as described in Section 3.3. The third step is Post-Protection, which lasts until the standby starts and is ready to replay logs from active.

Figure 6.8 shows the results of this experiment. PROTECT is about three times faster in SHADOW-NFS than in SR in our setting, largely because of the time SR requires to copy the database. For larger databases, we expect this time difference to increase.

6.5.2 Failover

Our next experiment compared failover times in SHADOW-NFS and SR. For each system, we first ran the system under load in ACTIVE+STANDBY state for five minutes before triggering a failure of the active DBMS and an immediate failover to the standby. The five minute warmup time was chosen to be large enough to allow the DBMS caches to warm up in both systems, but short enough that the database sizes of the two systems did not diverge significantly prior to failover. The result is shown in Figure 6.9. The failover contains 3 major procedures: 1) *Log Replay.* The standby replays all the logs left over by the failed active systems. 2) *Other.* The standby cleans up some resources

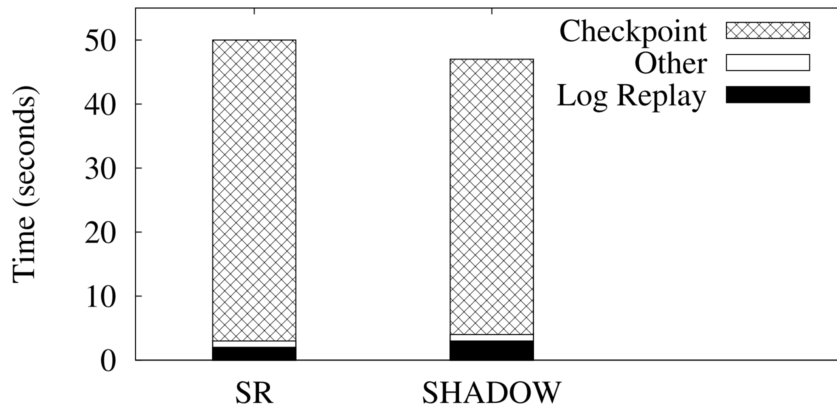

Figure 6.9: Failover Time in SHADOW-NFS and SR

and estabilshes some state before becoming the new active. 3) *Checkpoint.* The standby performs a consistent checkpoint to flush dirty pages to persistent storage and truncate logs. *Log Replay* takes only 2 to 3 seconds. The *Other* operations are also very fast and consume around 1 second. The dominating time consumption is *Checkpoint* [2]. The time depends of the size of the database. This checkpoint operation is the default in PostgreSQL (version 9.2). The SHADOW-NFS system also performs a this checkpoint. The overall failover time of SHADOW-NFS and SR are very similar.

6.5.3 Re-protect

Finally, we consider the performance of the SHADOW-NFS system in the DETACHED ACTIVE state, when the standby has failed. We tested SHADOW-NFS in both large memory (4GB) and small memory (1GB) configurations, which we refer to here as *SHADOW 4G* and *SHADOW 1G*. In each experiment, we run the system under test for 5 minutes in ACTIVE+STANDBY mode, before triggering a failure of the standby DBMS. A RE-PROTECT operation, which restarts the standby, is initiated 10 seconds after the standby's failure. During the 5 minute pre-failure warm-up period, the size of the database grows to approximately 1.5GB. We also ran a second version of the small memory experiment in which we did not restart the standby DBMS after the failure. We refer to that experiment as *SHADOW 1G NoRestart*.

[2] As of PostgreSQL version 9.3, this consistent checkpoint is optional.

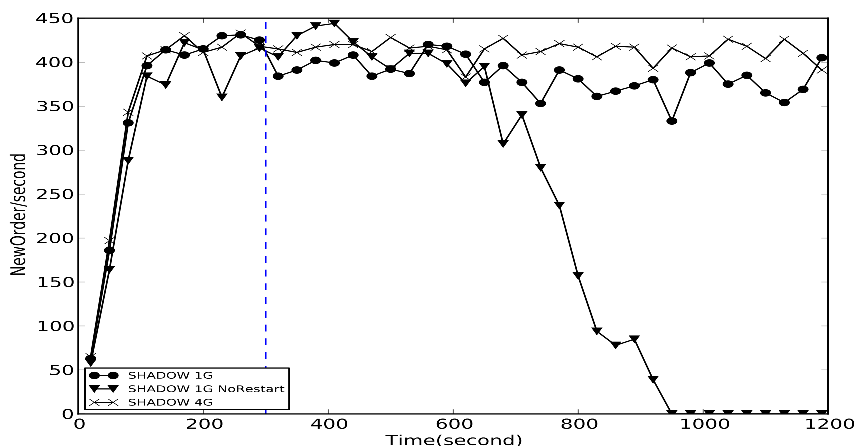

Figure 6.10: TPC-C Throughput During a Standby Failure

Figure 6.10 shows a timeline of the TPC-C New Order transaction throughput for each of the three scenarios we tested. For *SHADOW 4G*, the active DBMS continues to process transactions unaffected since it can cache the whole database and does not need to fetch pages from the standby. For *SHADOW 1G*, the active is unable to hold the complete database in memory, and attempts to request some pages from the standby. These requests fail while the standby is down, resulting in transaction aborts. In the *SHADOW 1G* experiment, the standby required about 400 seconds to fully recover. Although the standby is able to serve some page requests during recovery, approximately 9100 standby page requests failed. However, the resulting transaction aborts had only a small impact on system throughput.

The *SHADOW 1G NoRestart* scenario shows that the performance of SHADOW-NFS will eventually suffer if the failed standby remains unavailable for too long before recovering, and if the active DBMS is not able to store the entire database locally. As the active DBMS continues to process client transactions while the standby is down, the persistent copy of the database gradually becomes staler. Eventually, this increases the likelihood that the active DBMS will have to request pages from the standby, which leads to transaction aborts if the standby is down. As shown in Figure 6.10, in our setting the active DBMS is able to maintain its throughput for about five minutes after the standby fails, after which high abort rates quickly become a problem. Thus, in a SHADOW system, it is important that the standby be recovered within this “grace period”. The length of the grace period will depend on how much of the database the active DBMS can cache locally.

Chapter 7

Related Works

HADB has been studied in both the academia and industry for several decades. We first provide an overview of the HADB technques in the research literature. Then some representative products are reviewed in more detail.

7.1 Overview

Replication is a commonly used technique to achieve database system high availability. Data is maintained redundantly with several replicas. The system is able to tolerate the failure of one or more replicas and remains available.

In this thesis, we focus on HADB based on replication. We categorize and discuss the techniques along the following two dimensions:

- Synchronous/Asynchronous. Depending on *when* the updates are replicated, replication can be categorized as synchronous (eager) replication or asynchronous (lazy) replication [15, 12, 11]. Synchronous replication normally requires that the update commits at multiple sites atomically and durably. It could be implemented using by distributed consensus algorithms, such as Two Phase Comit [24] and Paxos [17]. However, these algorithms are known to have high overhead. In contrast, asynchronous replication commits updates locally. The updates are then propagated to other sites asynchronously. Asynchronous replication normally has lower overhead. However, the consistency and durability of data are not guaranteed.

- Active-Standby/Active-Active. Depending on *where* the updates take place, replication can be categorized as Active-Standby (a.k.a. Primary Copy) and Active-Active (a.k.a. Update Anywhere). Active-Standby configurations allow updates to happen in a single active replica. The standbys replays the operations that happen at the active. The standby replicas can be configured to serve read-only queries. Such a configuration is simple and is able to scale out read-only workloads. However, the single active system represents a bottleneck of the overall system, especially when updates happen frequently. Active-Active allows updates to happen anywhere. For some datasets and workloads, for example, when the dataset and workload can be partitioned well, update operations can be scaled out. However, when the datasets and workloads get more complex, Active-Active requires extensive high-overhead coordination among different replicas to guarantee data consistency.

	Active-Standby	Active-Active
Synchronous	Pros: • Consistency. • Not need to coordinate concurrent updates. Cons: • Long response time.	Pros: • Consistency. • Better update performance in some circumstances. (e.g. good partition) Cons: • Long response time. • Update coordination required
Asynchronous	Pros: • Short response time. • Not need to coordinate concurrent updates. Cons: • Local copy is not up-to-date	Pros: • Short response time. • Not need for coordination Cons: • Data inconsistency.

Figure 7.1: Replication design dimensions

Different combinations of replication techniques are shown in Figure 7.1. Active-

standby systems with asynchronous replication is a relatively simple and widely-used solution [2, 4]. Such systems are also known as 1-safe systems [12, 22]. Transaction results can be returned to the user once the transaction commits at the active system. Updates are shipped to the standby asynchronously. Normally, it has low overhead and is easy to implement. However, it risks data loss and inconsistency under various failure scenarios. Active-standby systems with synchronous replication avoids such drawbacks. However they normally have high overhead since the transaction commit needs to be coordinated between the active and the standby. Such systems are also known as 2-safe system. Normally, active-standby systems are also shared-nothing systems. The active and standby do not share data and state. Updates are replicated to standby via log shipping [2, 4], VM replication [19] or other techniques.

SHADOW belongs to the active-standby category. Unlike most other active-standby systems, SHADOW is a shared-storage system. The database and logs are shared by the active and the standby systems. Thanks to the shared-storage design, SHADOW is able to support the write-offloading technique, which eliminates database writes at the active system. Another significant difference between SHADOW and other active-standby system is that, typically, the active system does not rely on the standby system. Instead, SHADOW fetches pages from standby, effectively using the buffer cache at the standby.

Active-active is normally a more complex high availability solution. Updates can happen at multiple active systems concurrently. Updates are also propagated to each active system. Various techniques, such as those based on quorum consensus [10, 25] or on the availability of an underlying atomic broadcast mechanism [14], are used ensure the consistency among the active systems. Oracle RAC [1] and MySQL Cluster with NDB [3] belongs to this category. Like SHADOW, Oracle RAC is shared-storage system. In contrast, MySQL/NDB is shared-nothing system.

7.2 MySQL

We first describe MySQL (version 5.7) with InnoDB [2], which was the latest version when this thesis was written. By default, MySQL uses asynchronous replication. Each standby (i.e. slave) has an thread that fetches the logs from active (i.e. master). The standby also has other threads that replay the logs read from the active. When the active system receives a log fetch request from the standby, a thread at active will read the logs locally and ship them to the standby. Transactions could be lost during failover. As of version 5.7, MySQL also provides semi-synchronous replication to reduce the gap between transaciton commit and log shipping. Semi-synchronous replication requires that commit logs are shipped to

the standby before a transaction's completion is acknowledged to a client. Two variants of semi-synchronous replication exist: 1) *After Sync* (Default). The transaction operations are first shipped to the standby. Then the transaction log is committed locally. Finally, the active returns an acknowledgement to the transaction session. 2) *After Commit*. The transaction log is first committed locally. Then the transaction operations are shipped to the standby. Finally, the transaction is acknowledged. There are subtle differences between these two variants. For the first option, no other client can see a transaction's effect until the transaction is persistent at both active and standby. However, some special caution is required if the active failed to commit the transaction locally (e.g. disk failure) after the transaction operations have been shipped to slave. For the second option, since the transaction commits locally before the commit log is shipped, it is possible that logs shipping fails after the transaction commits locally. In this case, some clients could see the effect of the transaction at the active and find it missing after a failover.

Another interesting thing worth mentioning is that MySQL uses binary logs for replication, rather than transactional logs. In old versions of MySQL, the binary log has no concept of transaction. The standby needs to keep track of the log position that it has fethced and replayed. Hence, at the time of failover, the standby has no knowledge of which transaction are fully replayed. MySQL (version 5.7) uses a technique called Global Transaction ID (GTID) to add the concept of transaction. Binary logs describing the same transactions are grouped and assigned GTID number, which is unique among transactions. With GTID, failover can be more automatic since standbys have a globally unique transaction ID (log position) to agree upon when one standby is promoted as the new active. We will discuss transactional log-based replication at PostgreSQL, which seems to be a simpler and cleaner solution.

MySQL Cluster with NDB (version 7.3) [3] supports in-cluster replication and cluster-wise replication. Within a cluster, data is partitioned into node groups. Each partion in a node group contains primary replica and backup replica. The data is replicated using two phase commit. A MySQL cluster can also replicate its operations to an other MySQL cluster with aynchronous replication.

7.3 PostgreSQL

We focus on the PostgreSQL (version 9.3) [4], which was the latest version when the thesis was written. Like MySQL (version 5.7), PostgreSQL also supports asynchronous and synchronous replication. The thread model between MySQL replication and PostgreSQL is also very similar. Hence, we don't describe it in detail.

One major difference between PostgreSQL and MySQL replication is that PostgreSQL uses its transactional log for replication while MySQL uses a binary log. Recall that binary log is not born to have transaction in mind. Hence, extra effort is required during failover (e.g. including GTID). In contrast, with transactional logs, PostgreSQL is able to unify the traditional failure recovery and replication. Traditional failure recovery (not failover) of both MySQL with InnoDB and PostgreSQL uses transactional logs. Failure recovery can always find the recovery point under repeated failure and recover correctly. Since both replication and failure recovery are implemented based on log replay, PostgreSQL reuses the algorithms of failure recovery for replication. The standby fetches transactional logs from active and replays the logs in failure recovery mode as if it was a standalone node that was doing failure recovery. As a result, the standby can always recover correctly under repeated failure. Failover can be done automatically as long as standby is told that the active is down.

7.4 Oracle

We focus on Oracle 12c [1], the latest version when the thesis was written. Oracle supports an optional active (a.k.a. Primary) and standby (a.k.a. Data Guard) configuration. The standby can be hot standby (Active Data Guard) and support read-only queries. Oracle supports two types of logs. Based on the description in Oracle's white paper [1], we infer that one is similar to physical (and physiological) logs. The other is similar to logical logs. The database system can switch between these two types of logs. For instance, during normal replication, physical (or physiolological) log is used. In this way, when the active finds a corrupted page, it fetches a correct one from the standby. When the standby finds a corrupted page, it fetches a correct one from the active. However, during system upgrades, the logical log (based on SQL) is used to achieve compatibility between different versions.

As in MySQL and PostgreSQL, replication can be synchronous or asynchronous. Synchronous replication has a performance impact on the active system. According to the description in Oracle's white paper [1], the synchronous replication mechanism is very similar to those in MySQL and PostgreSQL. Oracle claims that it guarantees zero data loss. Oracle also supports a feature called Far Sync. It is similar to chain/cascade replication in MySQL and PostgreSQL. The active first synchronously ships logs to a geographically nearby proxy. The proxy then asynchronously ships the logs to a geographically different site. The proxy could optionally do some data compression.

Oracle 12c supports an advanced feature through which all in-flight transactions are protected and therefore not aborted during failover. This is supported by a middle-tier

that replays the transactions aborted by the database system during failover. Hence, from the users perspective, no transaction is aborted. Only a small number of transactions have longer response time than normal due to the replay.

Oracle RAC is different from traditional active-standby systems in that the database is shared, as in SHADOW. However, Oracle RAC is an active-active system. Multiple read/write transactions can happen at different application servers concurrently. A more advanced high availability product called GoldenGate is also part of Oracle 12c. It is able to replace both active and standby. GoldenGate can support relication in an active-active configuration.

7.5 RemusDB

RemusDB [19] is a novel HADB solution that is based on VM replication. It pushes the replication task from the DBMS to the virtualization layer. Changes are continuously copied from active to standby. Several optimizations are implemented to reduce the overhead of VM-based repliaction. RemusDB achieves synchronous replication by buffering results until changes at the active have been replicated to the standby. Once the changes are safely replicated, the results are returned to clients. Since the orginal virtualization layer is unaware of transactions, the DBMS and virtualization layer are modified to synchronize the transaction commit with result buffering. The transaction result is bufferred until the transaction-caused changes have been replicated to the standby. During failover, all in-flight transactions are aborted at the new active.

7.6 Spanner

Spanner [7] is not a traditional relational DBMS. However, it supports OLTP workloads and guarantees ACID requirements. Besides, it is scalable and highly available. Hence, we consider it as a related work and describe it here.

Spanner is a scalable, multi-version, globally distributed, and synchronously replicated database implemented by Google. Each Spanner deployment is a universe, which contains a set of Zones. Zone is the unit of physical isolation. Each zone, in turn, include hundreds or thousands of spanservers that store tablets. Hence, data in Spanner is partitioned into tablets. Each tablet is replicated.

Traditional DBMS that provides strong ACID does not scale out well. Distributed consensus algorithms, such as Paxos and two phase commit have high overhead, especially when used across wide area network. Besdies, transaction concurrency is very difficult in distributed environments. Spanner introduces two smart designs that make itself possible for many types of datasets and workloads. The first is fine-grained data management. With fine-grained partitions and live migration, many carefully designed applications can reduce the quantity and geographic span of distributed transacitons. The second is TrueTime. Thanks to Google's efficient and customized data center, the clock of different nodes can be synchronized within a guaranteed bound. With the synchronized time, Spanner implements transaction concurrency in a distributed environment.

Each tablet and its replicas form a replication group. Unlike the synchronous replication of MySQL and PostgreSQL, data within each replication group is replicated using Paxos. Hence, updates are guaranteed to be replicated atomically and durably. The replication group is available as long as a quorum of replicas are available. Update to a replication group is coordinated by a group leader. However, update can be proposed (i.e. initiated) by any replica within the replication group. Hence, Spanner is an Active-Active system.

Since data is partitioned, two phase commit is used to support transactions that span several replication groups. The group leader of one replication group is selected as the transaction manager and coordinates the distributed transaction.

Chapter 8

Conclusion

We have presented SHADOW, a novel architecture for building highly available database systems. SHADOW systems incorporate two key ideas. First, the active and standby database systems share access to a single copy of the database and log. Second, the active DBMS writes to the log to commit transactions, but does not update the database. Instead, database updates are the responsibility of the standby DBMS. The SHADOW architecture targets settings in which the active and standby database systems can cache most or all of the database.

Our experiments with TPC-C workloads show that SHADOW systems outperform traditional hot standby systems, largely by eliminating the need to synchronize the active and standby database systems when committing a transaction. Our experiments also show that the active DBMS in a SHADOW system can outperform a standalone DBMS, because database writes are offloaded from the active to the standby in a SHADOW system. The less tolerant the standalone DBMS is of downtime, and hence the more aggressively it pushes changes to the database, the greater the performance advantage of the SHADOW system.

References

[1] Maximize Availability with Oracle Database 12c. http://www.oracle.com/technetwork/database/availability/maximum-availability-wp-12c-1896116.pdf.

[2] MySQL 5.7: Chapter 16. Replication. http://dev.mysql.com/doc/refman/5.7/en/replication.html.

[3] MySQL Cluster Replication. http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-replication.html.

[4] PostgreSQL 9.3: Chapter 25. High Availability, Load Balancing, and Replication. http://www.postgresql.org/docs/9.3/static/high-availability.html.

[5] 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 *CIDR*, volume 11, pages 223–234, 2011.

[6] S. Bartkowski and et al. High Availability and Disaster Recovery Options for DB2 for Linux, UNIX, and Windows. Technical report, IBM Redbooks, 2012.

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

[8] Stephen Daniel. Running Database Applications On NAS: How and Why? http://www.snia.org/.

[9] 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. In *SOSP*, volume 7, pages 205–220, 2007.

[10] David K Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.

[11] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proc. SIGMOD*, pages 173–182, 1996.

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

[13] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[14] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB*, pages 134–143, 2000.

[15] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Management of a remote backup copy for disaster recovery. *ACM Trans. Database Syst.*, 16(2):338–368, 1991.

[16] D. Komo. Microsoft SQL Server 2008 R2 High Availability Technologies Whitepaper. Technical report, Microsoft, 2010.

[17] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[18] Rui Liu, Ashraf Aboulnaga, and Kenneth Salem. DAX: A widely distributed multi-tenant storage service for DBMS hosting. *Proc. VLDB Endowment*, 6, 2013.

[19] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent high availability for database systems. *PVLDB*, 4(11):738–748, 2011.

[20] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[21] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, pages 109–116, 1988.

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

[23] Krishna P.N. Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. Frugal storage for cloud file systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 71–84, New York, NY, USA, 2012. ACM.

[24] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM, 1981.

[25] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[26] Satyam B. Vaghani. Virtual machine file system. *SIGOPS Oper. Syst. Rev.*, 44(4):57–70, December 2010.

[27] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.