Database High Availability Using SHADOW Systems

Jaemyung Kim* Kenneth Salem* Khuzaima Daudjee* Ashraf Aboulnaga† Xin Pan*

*University of Waterloo, †Qatar Computing Research Institute

{jaemyung.kim, ken.salem, kdaudjee}@uwaterloo.ca, aaboulnaga@qf.org.qa

Abstract

Hot standby techniques are widely used to implement highly available database systems. These techniques make use of two separate copies of the database, an active copy and a backup that is managed by the standby. The two database copies are stored independently and synchronized by the database systems that manage them. However, database systems deployed in computing clouds often have access to reliable persistent storage that can be shared by multiple servers. In this paper we consider how hot standby techniques can be improved in such settings.

We present SHADOW systems, a novel approach to hot standby high availability. Like other database systems that use shared storage, SHADOW systems push the task of managing database replication out of the database system and into the underlying storage service, simplifying the database system. Unlike other systems, SHADOW systems also provide write offloading, which frees the active database system from the need to update the persistent database. Instead, that responsibility is placed on the standby system. We present the results of a performance evaluation using a SHADOW prototype on Amazon's cloud, showing that write offloading enables SHADOW to outperform traditional hot standby replication and even a standalone DBMS that does not provide high availability.

Categories and Subject Descriptors H.2.2 [*DATABASE MANAGEMENT*]: Physical Design; H.2.4 [*DATABASE MANAGEMENT*]: Systems

General Terms Design, Performance, Availability

1. Introduction

Hot standby systems are widely used to improve the availability of database management systems (DBMS). A hot standby manages a separate, backup copy of the database. In case the active, or primary, DBMS fails, the active system's workload is shifted to the standby so that there is little or no downtime. During normal operation, the active system ships update logs to the standby. The standby re-executes those updates against its copy of the database so that it remains closely synchronized with the active (Figure 1).

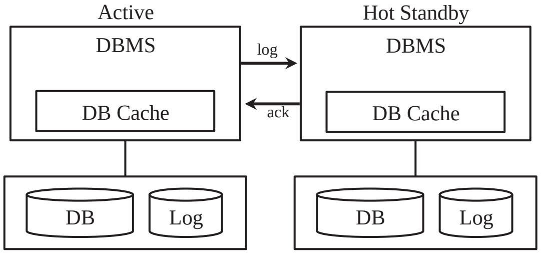


Figure 1. A Hot Standby System

This hot standby approach is well suited to shared-nothing environments because there are two independent copies of the database that can be managed by separate servers. However, database systems deployed in computing clouds often have access to reliable persistent storage that can be shared by multiple servers. For example, Amazon's Elastic Compute Cloud (EC2) includes the Elastic Block Store (EBS), which provides network-attached reliable persistent storage volumes for server instances running in EC2. OpenStack and other systems provide reliable shared storage in private clouds. In some settings, such as EC2, network-attached persistent storage may be the *only* option for persistent storage. Server-attached secondary storage is ephemeral, meaning that it is unrecoverable in the event of a server failure.

It is possible to use existing DBMS hot standby techniques in cloud environments with shared persistent storage. One can simply ignore the data sharing capabilities of the storage system and store the active and standby copies of the database independently in shared storage. This approach is used, for example, in Amazon's RDS managed relational database service [3]. In this paper, we re-visit DBMS hot standby techniques for settings where the active and standby servers are deployed in an environment in which shared persistent storage is available. We address the following ques-

SOCC'15, August 27-29, 2015, Kohala Coast, HI, USA.
Copyright © 2015 ACM 978-1-4503-3651-2/15/08…$15.00.
http://dx.doi.org/10.1145/2806777.2806841

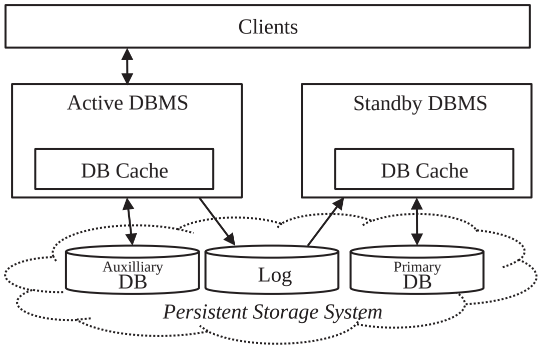


Figure 2. SHADOW System Architecture

tion: how can we exploit shared persistent storage to build better highly available database systems?

We propose SHADOW, a DBMS hot standby technique for computing clouds and other environments in which shared persistent storage is available. A SHADOW system includes active and standby database systems, like other hot standby techniques. However, in a SHADOW system, the division of responsibilities among system components is very different from traditional hot standby techniques. At a high level: (1) The active DBMS interacts with clients and performs database updates on its in-memory cached copy of the database. The active writes update records to the transaction log, but is not responsible for updating the database. (2) The standby DBMS is responsible for updating the database. The standby reads the log records written by the active, taking advantage of the fact that the log is in shared storage, and uses these records to update the database. (3) The storage system replicates the database and the log to ensure that they are durably stored even in case of failure. The storage system also ensures that the active and standby DBMS have shared access to the database and the log. Figure 2 illustrates the SHADOW architecture. More details (such as the need for an auxiliary database) will be presented later.

SHADOW offers several advantages over the traditional hot standby approach to high availability. First, by pushing responsibility for data durability into the storage system, much of the DBMS complexity associated with database replication is eliminated. DBMS hot standby techniques normally make use of a log shipping protocol to synchronize database updates between the active and standby. Such protocols can be complex, especially when various failure and recovery scenarios are considered. In contrast, committing a transaction in a SHADOW system is as easy for the active DBMS as committing a transaction in a standalone setting in which there is no high availability.

Second, by assigning the responsibility for updating the database to the standby, SHADOW systems eliminate all (or most) of the need for the active DBMS to write to the database. Relational database systems regularly *checkpoint* the database, which involves writing database updates from the in-memory buffer cache of the DBMS to the underlying persistent copy of the database. Checkpointing is required to bound the amount of work needed to recover from failure since recovery starts from the latest checkpoint. Checkpointing is also required to bound the amount of history that needs to be kept in the transaction log, since the space occupied by a log record can be reclaimed only after the update represented in this record is checkpointed to the database. Thus, even if the database can completely fit in memory, there would still be write I/O activity to the database due to checkpointing. In modern on-line transaction processing (OLTP) systems, towards which DBMS high availability is targeted, it is very common for the database to fit in memory, so little or no database read I/O is required, and most or all write I/O to the database is due to checkpointing. SHADOW systems *eliminate the need for checkpointing at the active DBMS*, thereby eliminating a significant fraction of the I/O performed by the active. The active DBMS in a SHADOW system writes only log records, and the responsibility for database writes is shifted to the standby DBMS. We call this *write offloading*. Our performance evaluation in Amazon's EC2 cloud shows that a PostgreSQL-based SHADOW prototype has better overall performance, and more stable performance, than PostgreSQL's native synchronous hot standby replication. Indeed, in many settings, a highly-available SHADOW system *can outperform even a standalone PostgreSQL instance, which does not provide high availability*, because of write offloading.

Third, SHADOW systems decouple replication of the database server (the DBMS) from replication of the underlying database and log. In SHADOW, the DBMS is replicated to ensure that the service offered by the DBMS is highly available. Replication of the database and log, for high availability and for durability, is the responsibility of the underlying storage system. This provides flexibility in managing the trade-offs among cost, availability, and durability when deploying a database service. In particular, the degree of replication of the DBMS need not be the same as the degree of replication of the database, as is the case in traditional hot standby systems. For example, in a SHADOW system it is possible to ensure that committed database updates are *3-safe* (i.e., are replicated at least three times) while deploying (and paying for) only two database systems.

Finally, SHADOW systems require less I/O bandwidth between the DBMS and persistent storage than traditional hot standby systems, since data replication occurs *within* the storage system. In environments in which storage system I/O is metered, such as Amazon's EC2, this can reduce the operating cost of a hot standby system.

This paper makes the following research contributions.

- We propose the SHADOW system architecture, and we present algorithms for system management operations, such as launching a standby DBMS or failing over to the standby, in a SHADOW system.

- We present a SHADOW prototype, based on PostgreSQL, which is designed to operate in a cloud setting in which there is a persistent block storage service, such as Amazon's Elastic Block Store (EBS).
- We present a performance evaluation in which SHADOW is compared against a variety of baseline systems under an OLTP workload. Our evaluation shows that when shared storage is available, *a DBMS can be made highly available (using SHADOW's hot standby approach) with no loss in overall system performance*. In contrast, making a DBMS highly available using PostgreSQL's native hot standby replication results in significant performance degradation.

The remainder of the paper is structured as follows. Section 2 presents a brief overview of existing techniques for building highly-available database management systems. Section 3 provides a high-level overview of the SHADOW approach, and contrasts it with existing techniques. Section 4 describes the operation of a SHADOW system in more detail and explains system operations such as establishment of a standby system, and failover from the active system to the standby. Section 5 describes our prototype implementation of SHADOW, which is based on PostgreSQL and which uses Amazon's EBS for persistent storage. Section 6 presents the results of a performance evaluation that compares the prototype against several different baselines.

2. DBMS High Availability

Before presenting the SHADOW approach, we first give an overview of existing techniques for building highly available database systems.

2.1 Shared-Nothing Techniques

Many database high availability techniques are *shared-nothing* [17]. This means that high availability is achieved using two or more DBMS instances, each running on a separate server with its own local storage (no shared network-accessible storage is assumed). DBMS instances communicate by exchanging messages over a network.

The hot standby technique described in Section 1 is a shared-nothing technique. It is widely implemented and used, e.g., in PostgreSQL Replication, MySQL Replication, DB2 [6], Microsoft SQL Server Always On [19], and Oracle Data Guard [12] where it is known as database mirroring [13]. As illustrated in Figure 1, each DBMS instance manages its own copy of the database and transaction log. Clients connect to the active DBMS and execute transactions on the active. The active DBMS sends the database updates to the standby DBMS (e.g., using log shipping), which applies the updates against its own database. Some hot standby systems also allow clients to run queries (but not updates) at the standby system. PNUTS [8] uses a more general version of this approach, in which clusters of servers in different sites manage multiple copies of a database, one copy per site. Each update occurs first at one site, which then ships the update to the other sites.

In hot standby systems, propagation of updates from the active DBMS to the standby may be *synchronous* or *asynchronous*. Typically, synchronous propagation ensures that the active DBMS will not acknowledge a commit request to the client until both the active and standby have persistently logged the transaction, ensuring that the effects of this transaction will survive a failure of the system. However, synchronous replication can hurt the performance of the active DBMS because it must synchronize with the standby on the critical execution path of every transaction commit. By propagating updates lazily, asynchronous hot standby systems avoid this problem, but accept the risk that committed yet unpropagated transactions will be lost as a result of a failure of the active.

While hot standby approaches can use as few as two DBMSs, other shared-nothing systems use quorum-based replication protocols (e.g., Paxos [14]) to synchronously replicate updates among three or more separate DBMS instances. For example, Microsoft's Cloud SQL Server [7] uses a custom replication protocol to synchronize updates across three or more DBMS instances, and transactions are committed only when a majority of the instances acknowledge them. Other examples of such systems include Granola [10], Spanner [9], and Megastore [5], the last two of which are designed to support geographically separated database instances.

Because these systems achieve consensus among DBMS instances about the commitment of each transaction, committed updates are not lost as a result of the failure of a single DBMS. Furthermore, because they can operate as long as quorum of replicas are up, the loss of a single DBMS instance results in little or no downtime. Thus, these systems provide very high availability. However, since at least three replicas are required, they are relatively expensive to operate [7]. In addition, the DBMS-implemented replication protocol introduces performance overhead and latency during normal operation, much like synchronous propagation does in hot standby systems.

2.2 Shared Storage Techniques

An alternative to the shared-nothing approach is to have multiple database systems share access to a single, reliably stored, shared copy of the database. This shared copy can be stored in a cloud shared storage service such as Amazon's EBS, in a cluster file system such as VMFS [20], or in a reliable NAS system implemented using redundant storage devices (RAID [16]) and replicated servers.

Example *shared-storage* systems include Tandem's NonStop SQL [18], IBM DB2 ICE, IBM DB2 pureScale [1], and Oracle RAC [2]. These systems all employ an *active/active* architecture, in which multiple DBMS instances process client transactions against the same shared database. A downside of this approach is that distributed mechanisms for concurrency control and (DBMS) cache coherency are required to coordinate the database accesses made by the different database systems, and these mechanisms may be on the critical path of every transaction. Like SHADOW, these systems rely on the availability of the shared database to maintain overall system availability. To maintain availability despite the loss of the database, the entire shared-storage cluster can be mirrored using hot standby techniques (e.g., Oracle RAC + Data Guard).

Another class of shared-storage systems is *cold standby* systems, which do not have the complexity of active/active systems. A single active DBMS manages a database in shared, network-accessible storage. If the active DBMS fails, a standby DBMS is launched on a different server, and it accesses the same database and log that was used by the active. The standby DBMS must bring the shared database to a consistent state (e.g., by using a log-based recovery protocol such as ARIES [15]) before availability can be restored. Since the cold standby approach requires only a single active DBMS instance at any time, the standby DBMS uses little or no resources (e.g. CPU and memory) during standing by. However, because the system is down while the standby is launched and the database is recovered, the cold standby technique does not provide true high availability. Instead, it reduces downtime by eliminating the need to wait for the active system to be repaired. Cold standby techniques are similar to SHADOW in that only one DBMS instance at a time can update the active database and the shared database log. However, SHADOW standby systems are hot to minimize the downtime that results from a failure of the active.

Because each of these systems has its own advantages (in cost, performance, durability or availability), all are used in practical settings, with the choice made according to application requirements. SHADOW systems, which we introduce in Section 3, belong to the same part of this design space as hot standby systems. However, SHADOW is intended to be a superior alternative to hot standby approaches in cloud environments where shared, network-accessible, persistent storage is available.

3. SHADOW Overview

Figure 2 illustrates a SHADOW system in the normal, protected operational state. Clients connect to the active DBMS, which processes all client transactions. Both the active and the standby DBMS instances have access to a shared reliable persistent storage system. We assume that the shared persistent storage system is itself highly available. For the purposes of this section, we are not concerned with the specific implementation of the shared storage system. Section 5 presents the SHADOW-EBS prototype, which uses Amazon's Elastic Block Store (EBS) for persistent storage.

The SHADOW architecture shown in Figure 2 shows two databases stored in shared storage: a *primary database* managed by the standby DBMS, and an *auxiliary database* managed by the active DBMS. When a SHADOW system starts operation, only the (unprotected) active DBMS is running and there is only one database. A standby DBMS is created by taking a snapshot of the active's database, and that snapshot becomes the primary database managed by the standby. The active retains its database, which now becomes the auxiliary database. Throughout the operation of a SHADOW system, there is one copy of the transaction log that is shared by the active and the standby.

The active DBMS performs transactional updates on its local cached copy of the database, and uses write-ahead logging to push update and transaction commit records to the shared persistent log. A transaction is committed once its log records are safe in the persistent storage system. The standby DBMS reads the log records generated by the active instance from shared storage and replays the logged updates against its local cached copy of the database. In addition, the standby periodically performs database checkpoints, which push database changes from its local database cache to the primary database in persistent storage. The standby's checkpoints also allow older log records to be purged from the shared persistent log. In a SHADOW system, only the standby DBMS reads and updates the primary database. The standby's checkpointing operations ensure at all times that the persistent primary database and log together include the effects of all committed transactions.

The active DBMS does not checkpoint, and writes to the auxiliary database only if the limited capacity of its database cache forces it to "spill" changed pages. We refer to this transfer of checkpointing responsibility from the active DBMS to the standby as *write offloading*. The active DBMS reads from the auxiliary database if it needs to bring database pages into its cache. In the special case that the entire database can fit in the active DBMS's cache (which is a common case in modern OLTP systems), the active DBMS never needs to write database pages, and never needs to read a database page except when this page is touched for the first time. If the database is larger than the active's cache, then there may be some read and write I/O to the auxilliary database. However, we expect the amount of write I/O to be much less than in a normal hot standby system, since there is no checkpointing at the active DBMS.

SHADOW systems are similar to hot standby systems (Figure 1), and this similarity allowed us to build our SHADOW prototype (Section 5) by adapting an existing synchronous hot standby system without too much difficulty. Nonetheless, there are several key differences between SHADOW systems and other hot standby systems which we summarize here, before presenting the operation of the SHADOW system in more detail. First, SHADOW systems require a *single persistent log* that can be shared by both the active and standby DBMS. Second, in a SHADOW system, only one of the two databases (the primary), together with the log, is guaranteed to include the effects of all committed

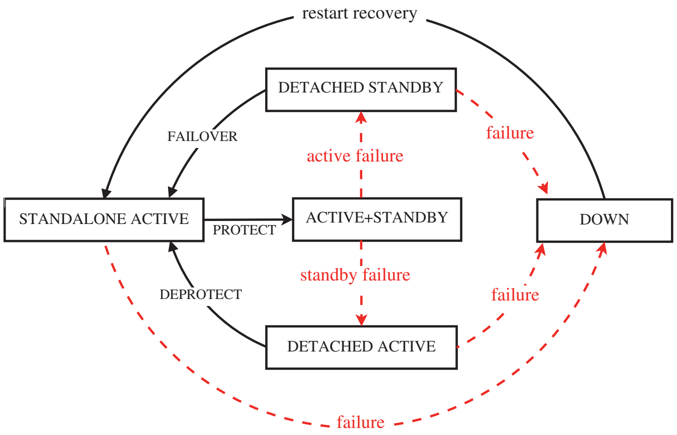


Figure 3. Operational States of a SHADOW System

transactions. Third, in a SHADOW system, the *active DBMS does little or no database I/O*, because of write offloading. Fourth, 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 instances during transaction commit.

4. System Operation

Figure 3 shows the operational states of a SHADOW system, and the transitions among those states. Transitions shown with dashed lines are failure transitions, which occur when either the active instance or the standby instance fails. Transitions shown with 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. However, because a SHADOW system divides responsibility for updating persistent storage between the active and standby DBMS instances, the behavior of a SHADOW system in the various states differs from that of other systems. Similarly, SHADOW system operations, such as PROTECT, differ from those in other active/standby systems.

As described in Section 5, we have implemented our SHADOW prototype using PostgreSQL. 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. Before presenting the SHADOW operations, we begin by describing our assumptions about the behavior of the generic DBMS.

We assume that the DBMS uses physiological [11] or physical write-ahead logging (WAL) to ensure the durability of committed transactions and their updates. Thus, each log record describes an update to a single page, which is identified in the log record. We assume that replaying a logged update 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 assumed to be logically equivalent from the perspective of the DBMS. We also assume that each log record has an associated log sequence number (LSN), and that each database page includes a record of the LSN of the most recent update that has been applied to that page. These techniques are used by well-known logging algorithms, such as ARIES [15], to ensure that updates are applied exactly once to each database page.

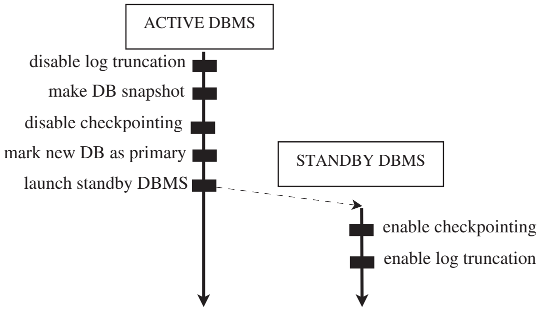


Figure 4. PROTECT Operation Timeline

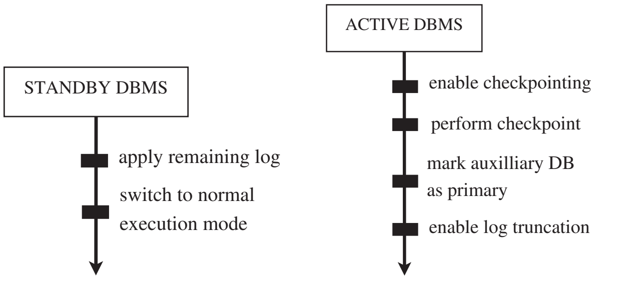
Finally, we assume that the DBMS can perform periodic checkpoint operations, and that as part of a checkpoint operation the DBMS will identify a new *log recovery point*, which is the LSN from which log replay should start in the event of a DBMS failure after the checkpoint. As part of its checkpoint operation, the DBMS is assumed to record the new log recovery point in a *recovery file* in the same persistent storage volume as the log itself. 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 in more detail.

4.1 Protection

The STANDALONE ACTIVE state represents the normal *unprotected* state, in which a single active DBMS handles client requests, and there is no standby. Failure of the active DBMS in this state results in loss of availability. 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 protected operational state of the SHADOW system.

A typical approach to creating a standby instance is to create a snapshot of the standalone database and then deploy the standby instance. The snapshot serves as the initial state of the database managed by the standby instance. As part of the process of creating the snapshot, the active DBMS identifies a *replay start LSN*, from which the standby will begin



(a) FAILOVER (b) DE-PROTECT

Figure 5. The FAILOVER and DE-PROTECT Operations

replaying the log. The reply start LSN is simply the log recovery point of the active system's most recently completed checkpoint. This ensures that any database updates that may not be present in the snapshot database will be replayed from the log.

Figure 4 illustrates how the PROTECT operation is performed in a SHADOW system. It differs in several ways from this typical procedure. First, the active DBMS disables log truncation before creating the database snapshot, to ensure that the shared log will contain all committed database updates that may not be present in the snapshot. Second, the active system switches the role of its copy of the database from *primary* to *auxiliary*, since the new standby system is to manage the primary database. We assume that the SHADOW system records which database is the primary using a file stored in the same persistent volume as the log.

The new 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 instance (starting from the replay start LSN), and re-executes the logged updates against its copy of the database. As the standby DBMS performs checkpoints, it pushes updated database pages from its database cache to persistent storage, and truncates old records from the shared log once the updates they describe are known to be in the persistent copy of the database.

4.2 Failover

In case of a failure of the active instance during protected operation, the standby instance becomes active and takes over the processing of client requests. This FAILOVER operation (Figure 5(a)) in a SHADOW system is very similar to failover in other active/standby systems. The standby instance finishes replaying all log records generated by the active instance before it failed, aborts active transactions, and then switches from log replay mode to normal execution mode in which it can accept new transaction requests from clients.

4.3 Deprotection

When the standby DBMS fails, the DE-PROTECT operation is used to put the active DBMS back into STANDALONE ACTIVE state. This operation, shown in Figure 5(b), involves changing the role of the active system's database from auxiliary to primary. To do this, the DE-PROTECT operation re-enables checkpointing in the active system and forces a checkpoint operation to ensure that all committed updates prior to the checkpoint are present in the persistent auxiliary database. The active DBMS then marks its copy of the database as the primary copy and proceeds as a standalone system. From there, a PROTECT operation can be used to create a new standby DBMS if desired.

4.4 Discussion

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. *At most one DBMS instance at a time is responsible for updating the primary database and for truncating the log.*

Property 1 follows immediately from the PROTECT operation, which disables log truncation in the active DBMS before switching the primary database to the standby and reenabling log truncation there. In all other states there is only a single DBMS. □

PROPERTY 2. *All client transactions see the current state of the database, including the effects of all committed transactions.*

During STANDALONE ACTIVE operation, this is ensured by the normal operation of the active DBMS. During a PROTECT operation, the active assumes management of the auxiliary database rather than the primary. However, the PROTECT operation does not affect the contents of the active DBMS's cache, and the initial state of the auxiliary database is identical to that of the new primary database. Property 2 holds after a FAILOVER operation because the standby DBMS replays all outstanding log records before switching to normal operation and assuming the role of the active database. This failover mechanism is used by other hot standby systems that are based on log shipping. □

PROPERTY 3. *All committed updates are present in persistent storage in the primary database, in the log, or both.*

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. However, one threat to Property 3 is the possibility that log truncation will remove persistent logged updates before they have been applied to the primary copy of the database. Property 1 ensures that only one DBMS at a time is responsible for updating the primary database and for log truncation, so correct operation of that DBMS's checkpointing and log truncation mechanism will preserve Property 3, pro-

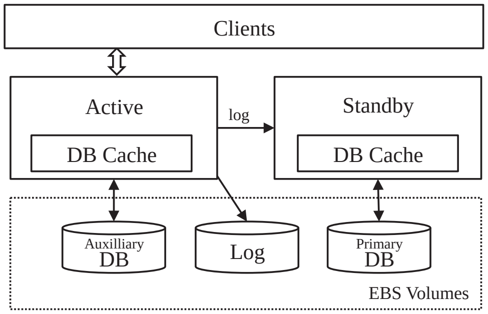


Figure 6. The SHADOW-EBS Prototype

vided that it holds when that DBMS initially takes responsibility for the primary database. In a SHADOW system, the standby DBMS assumes responsibility for the primary database during a PROTECT operation. Because the active DBMS disables log truncation before switching control of the primary database to the standby system (Figure 4), Property 3 will hold when the standby DBMS assumes responsibility for the primary database. The other threat to Property 3 in a SHADOW system occurs when the active system's auxiliary database becomes the primary database during a DE-PROTECT operation. By performing a checkpoint of the auxiliary database before enabling log truncation, the active DBMS ensures that Property 3 will hold in the auxiliary database before it assumes the role of the primary. □

A consequence of Property 3 is that if both active and standby database systems fail, normal DBMS log-based recovery using the primary database and log will correctly restore the database, with no loss of committed transactions.

5. Prototype Implementation

We have implemented a SHADOW prototype, which we refer to as SHADOW-EBS, using PostgreSQL (version 9.3) as the DBMS and Amazon's Elastic Block Store (EBS) for reliable persistent storage. EBS provides raw persistent storage volumes that can be attached in virtual machines running within Amazon's Elastic Compute Cloud (EC2). EBS volumes are replicated to provide both durability and high availability. Currently, Amazon's service level agreement for EBS volumes promises a minimum availability level of 99.95%. EBS volumes can be provisioned with a variety of performance characteristics to suit the application. EBS also provides snapshotting and other volume-level operations that are common in modern storage systems.

The SHADOW-EBS prototype is illustrated in Figure 6. It consists of active and standby instances of PostgreSQL, each running in an EC2 virtual machine, plus three EBS volumes, one each for the log, the primary database, and the auxiliary database. Next, we present details of SHADOW-EBS's use of EBS volumes (Section 5.1) and PostgreSQL (Section 5.2).

5.1 Using EBS for SHADOW

Although an EBS volume can be attached to different EC2 virtual machines over its lifetime, at any particular time an EBS volume can be attached to at most one virtual machine. This is a challenge for a SHADOW system, which expects the active and standby to share access to the log volume. To work around this issue, SHADOW-EBS uses PostgreSQL's native log-shipping facility to ship log records from the active DBMS directly to the standby. During normal operation, the EBS log volume is attached to the active PostgreSQL instance. The active instance commits transactions by writing their commit records to the EBS log volume, and *asynchronously* streams copies of all log records directly to the standby via a network connection. This adds a small amount of overhead to the active DBMS. However, no DBMS-level synchronization is required to commit transactions. Once the active DBMS has written a transaction's commit record to the log, it can immediately acknowledge the commit to the client application, and need not wait for any kind of acknowledgment of receiving log records from the standby system.

If the active DBMS fails, it is possible that some log records will have been written to the log volume but not sent to the standby. As part of the FAILOVER operation, after consuming all streamed log records, the standby DBMS attaches itself to the EBS log volume (which is possible since the active system has failed) and reads and applies any log records that were written by the active but not shipped. This ensures that committed updates will not be lost during the failover operation.

One final challenge related to the log volume is that during normal operation, the SHADOW standby system is responsible for log truncation, which requires access to the log volume. In SHADOW-EBS, the standby DBMS does this indirectly, by sending log truncation requests to a SHADOW-EBS log truncation service that runs on the same virtual machine as the active DBMS. The truncation service simply acts as a proxy for the standby DBMS, truncating the log at the LSN specified by the standby.

SHADOW-EBS's need for log streaming and the log truncation proxy arise directly from EBS's limitation that only a single virtual machine can connect to a volume. Neither would be necessary if SHADOW were implemented on a storage service that supports simultaneous access to the log, e.g., a reliable network-attached file system or a cluster file system.

One positive feature of EBS is that SHADOW-EBS can directly exploit EBS's volume snapshot mechanism during PROTECT operations. In STANDALONE ACTIVE state, there is only one database volume, to which the active DBMS is connected. During a PROTECT operation, SHADOW-EBS uses EBS's native snapshot facility to create a snapshot of the database volume. This snapshot occurs as part of the "make DB snapshot" step shown in Figure 4. This snapshot is used to initialize a second database EBS volume, which is attached to the newly-created standby DBMS server. The new volume holds the primary copy of the database, and the database stored in volume originally attached to the active DBMS becomes the auxiliary copy. In case of a failure of the active DBMS, the auxiliary database's volume, to which the active was attached, can simply be deleted after transaction processing has been failed over to the standby DBMS.

5.2 PostgreSQL Details

SHADOW-EBS makes use of a pair of PostgreSQL instances, with the standby instance operating in continuous recovery mode. This means that it continuously replays log records streamed to it by the active instance.

In a SHADOW system, the standby DBMS is expected to push updates to the persistent primary database by performing checkpoint operations, and the active system does not checkpoint at all. However, a PostgreSQL instance operating in continuous recovery mode cannot checkpoint independently of the active instance. 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.

To address this issue, we modified the active PostgreSQL server so that it will perform periodic *pseudo-checkpoint* operations. During a pseudo-checkpoint, the active DBMS writes a normal checkpoint record to the log, but does not flush any dirty pages to the auxiliary database and does not update its log recovery point. Using the checkpoint log records generated by the active's pseudo-checkpoints, the standby can checkpoint its database as usual. It is necessary for the frequency of pseudo-checkpoints at the active system to be at least as high as the frequency of true checkpoints at the standby. However, the overhead of a pseudo-checkpoint at the active system is very low.

After checkpointing, the standby DBMS can truncate log records that are no longer needed for recovery. As described in Section 5.1, we modified PostgreSQL so that the standby instance will use the active's log truncation service to accomplish this since it cannot access the log volume directly.

Finally, we made a number of changes to PostgreSQL to implement write-offloading by ensuring that the active DBMS will perform as few writes to the auxiliary database as possible during protected operation. PostgreSQL's buffer manager flushes dirty database pages from its buffer cache to persistent storage for three reasons. First, all dirty buffers are flushed when a checkpoint is performed. Second, when there are many dirty pages in the buffer pool, they can be flushed to persistent storage by PostgreSQL's background writer threads. Finally, a dirty page may be flushed to disk during page replacement if the replacement victim is dirty, or when a new page is first allocated.

We added two new parameters to PostgreSQL to control these writes. The first can be used used to enable or disable writes caused by checkpoints. The second can be used to enable or disable PostgreSQL's background writer threads. During a PROTECT operation, these flags are used to disable both types of writes in the active instance (at the "disable checkpointing" step in Figure 4), meaning that PostgreSQL will write to the persistent database only when necessary during page replacement. When the active's buffer cache is large, this occurs very rarely. During a DE-PROTECT operation, these parameters are used to re-enable database writes.

6. Evaluation

In this section we evaluate our SHADOW-EBS prototype, comparing it to other hot standby DBMS configurations, and to standalone DBMS configurations which are not highly available. We evaluate the transaction processing performance of these systems during normal protected operation as well as the time required for failover after failure of the active DBMS.

6.1 Experimental Methodology

We have implemented the SHADOW-EBS prototype system described in Section 5 using PostgreSQL version 9.3. We use this same version of PostgreSQL for the standalone and hot standby baseline systems against which we compare SHADOW-EBS.

For the standalone baseline, the frequency with which the DBMS initiates checkpoints controls a tradeoff between performance during normal operation and recovery time. Checkpointing more frequently reduces recovery time but adds overhead during normal operation because checkpoints consume I/O and may cause buffer cache contention. Thus, we use several different standalone configurations with different checkpointing frequencies in our evaluation:

- SAD: This is the default PostgreSQL configuration, checkpointing every five minutes or when three 1GB log segments fill up (whichever happens earlier). This configuration represents an extreme tradeoff of performance in favor of fast recovery time.
- SA: This configuration checkpoints minimally (every 30 minutes), regardless of the total number of log segments. It represents the opposite extreme on the performance vs. recovery time tradeoff for standalone systems.
- SA10: This configuration checkpoints every 10 minutes. Thus, it represents a balance between the extremes of SA and SAD.

In addition to these standalone configurations, we use PostgreSQL's native hot standby mechanism to provide two highly available baselines (Table 1):

- SR: This baseline includes two database systems, one active and one hot standby, configured to use *synchronous*

system	description	checkpoint interval	highly available?	transaction loss?
SAD	standalone PostgreSQL	5 min	no	n/a
SA10	standalone PostgreSQL	10 min	no	n/a
SA	standalone PostgreSQL	30 min	no	n/a
ASR	PostgreSQL native asynchronous hot standby	30 min	yes	possible
SR	PostgreSQL native synchronous hot standby	30 min	yes	no
SH	SHADOW-EBS	30 min	yes	no

Table 1. Tested Systems. The transaction loss column indicates whether committed transactions might be lost during failover.

replication. Each DBMS manages its own persistent copy of the database and its own log. The active system uses PostgreSQL's log shipping mechanism to stream logged updates to the standby, which replays them against its copy of the database. With synchronous replication, the active system does not acknowledge transaction commits to the client until the standby has received and persistently stored the transaction's commit record. In our SR configuration, both the active and the standby are configured to checkpoint minimally, like the SA baseline.

- ASR: This baseline is identical to SR, except that the active and hot standby systems are configured to use *asynchronous* replication. This means that the active system may acknowledge transaction commits to the client as soon as the transactions are committed locally, without waiting for acknowledgment from the standby [4]. Since the active system does not need to coordinate transaction commits with the standby, some recently committed transactions may be lost during a failover. As in the SR baseline, both the active and the standby are configured to checkpoint minimally.

All of our experiments were run in Amazon's EC2 cloud using c3.4xlarge instances, which have 16 virtual CPUs and 30 GB of memory, for all of our baselines and the SHADOW-EBS prototype. Linux kernel 3.2.0-56-virtual is used for all EC2 instances.

Each standalone baseline used two EBS volumes, one for the log and one for the database. The SR and ASR highly available baselines used four volumes, a database volume and a log volume for the active instance, and a second pair of volumes for the standby. As shown in Figure 6, SHADOW-EBS used a total of three volumes, one for the log and one each for the auxiliary and primary databases. Unless otherwise noted, all EBS volumes used Amazon's provisioned I/O feature to guarantee a minimum performance of 600 I/Os per second (IOPS).

Our experiments used the TPC-C benchmark workload, with clients (terminals) configured to submit transactions without think time. We used 30 TPC-C clients for all experiments, enough to ensure that the tested systems ran at, or near, their peak throughput. Each experiment was run using a 100-warehouse TPC-C database, which had an initial size of approximately 11GB. The original database was restored and the DBMS was restarted prior to each new run. Each experimental run lasted for 100 minutes, during which the database size grew as large as 24GB, depending on system performance. We ignored the first 30 minutes and the last 10 minutes of each run so that our reported measurements represent steady state execution. We performed three independent runs for each configuration tested.

On all DBMS servers, we flushed the operating system's page cache (including cached file data) every ten seconds to minimize the impact of file system caching on our results. We also made two adjustments to the default PostgreSQL configuration for all of our tests. First, we disabled full page logging (PostgreSQL parameter `full_page_writes`), since it led to very high log volumes that severely impacted performance and hindered measurement. Second, we configured PostgreSQL to spread out checkpoint-related I/O over most of the checkpoint interval (parameter `checkpoint_completion_target = 0.9`) to reduce the burstiness of checkpoint I/O. These configuration adjustments were applied to SHADOW-EBS as well as to the baseline configurations.

6.2 Large Memory Scenario

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. Specifically, the PostgreSQL database buffer cache size is set to 26GB, which is large enough to hold the whole database, even after it grows with transaction processing. It is common for database systems that support transactional workloads to be configured with enough memory to hold the entire database, or at least the hot part of the database. However, we also consider a configuration with a smaller buffer pool in Section 6.3.

Figure 7 compares the average TPC-C throughput of the standalone baselines (SA, SA10, SAD) and the hot standby baselines with that of the SHADOW-EBS prototype, which is denoted by SH in the figure. We report throughput in terms of TPC-C New Order transactions per minute (tpmC), which is the standard measure of throughput for the TPC-C benchmark. We make several observations:

- First, SHADOW's throughput is slightly *higher* than that of the standalone SA baseline, which checkpoints infre-

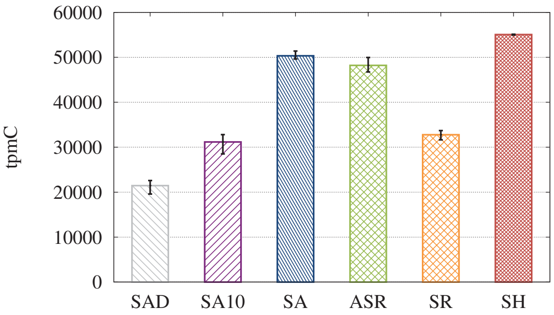


Figure 7. TPC-C Throughput, Large Memory Case. Bar height shows the mean throughput over three independent runs, while the error bars show the range between the fastest and slowest runs.

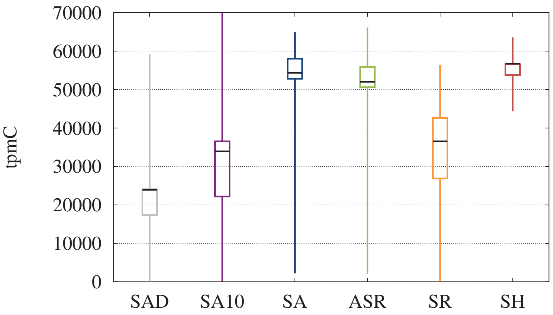


Figure 8. Variability of TPC-C Throughput, Large Memory Case. The ends of the whiskers show the minimum and maximum throughput observed during any 10 second interval, the boxes show the two middle quartiles, and the line in each box represents the mean throughput.

quently, and significantly higher than that of the other two standalone configurations that checkpoint more aggressively. Both SHADOW and SA commit transactions the same way – by writing a commit record to the log. SHADOW achieves slightly better average throughput than SA because of write offloading, which eliminates checkpointing at the active DBMS. Thus, SHADOW can be used to turn a standalone DBMS configuration into a high availability configuration *without negatively impacting performance*. In fact, as the figure shows, write offloading can actually boost performance.

- Second, SHADOW's performance is also slightly higher than that of ASR, which provides high availability but which (unlike SHADOW) may lose committed transactions during failover. ASR's performance is about the same as that of SA, which is not surprising since both work the same way except that the ASR active asynchronously ships log records to the standby. This has little impact on performance.
- Finally, SHADOW's throughput is nearly 70% higher than that of SR, which is the only baseline that provides availability and durability guarantees like SHADOW's.

To explain the performance of SHADOW and the baselines, we first look more closely at transaction throughput. During each run, we measured transaction throughput during each ten-second interval. Figure 8 presents a box-and-whiskers plot that shows the distribution of these throughput measurements over all three runs for each system. As can be seen in the figure, SHADOW normally has performance that is comparable to that of ASR and SA. However, both ASR and SA suffer occasional periods of severe performance degradation, including brief periods during which throughput dropped to less than 1000 tpmC. We have observed that these periods are correlated with checkpointing. In contrast, *SHADOW's performance is much more stable* due to write offloading, which eliminates checkpoint I/O in the active system. SHADOW's throughput remained at 45000 tpmC or above at all times. The performance of the SR baseline is not only lower than that of SHADOW and ASR, but also more variable (bigger box in the plot).

In our experiments, we also measured transaction response times (details omitted for lack of space). SHADOW-EBS and the two high availability baselines had similar median response times, near 10ms. However, the mean response times were 14.2ms for SHADOW-EBS, 16.5ms for ASR, and 22.9ms for SR. ASR and SR have higher mean response times than SHADOW-EBS because a larger fraction of their transactions have a high response time (i.e., their response time distributions have heavier tails than SHADOW's).

6.2.1 Database I/O

In addition to reducing performance variability, SHADOW also reduces the amount of I/O performed by the database systems. Figure 9 shows the total amount of I/O to the database volume(s) for SHADOW and the baselines. For those systems (SHADOW, SR, and ASR) that have two database volumes, Figure 9 shows the total amount of I/O to each volume. In all cases, write I/O dominates, since all of the database systems have sufficient memory to cache the entire database.

Two things are evident from Figure 9. First, the SHADOW active performs much less I/O than the ASR and SR systems, despite the fact that its throughput is higher. The SHADOW active also performs less I/O than any of the standby systems. Almost all of the database writes performed by the SHADOW active are due to the growth of the database over each experimental run since PostgreSQL pushes each newly-created database page to persistent storage (an implementation detail of PostgreSQL). Without database growth, a SHADOW active system with sufficient memory to hold the entire database *will perform no database I/O at all once the database has been loaded into memory*. Second, all of the standby database systems perform a similar amount of I/O. (SR performs slightly less, but only be-

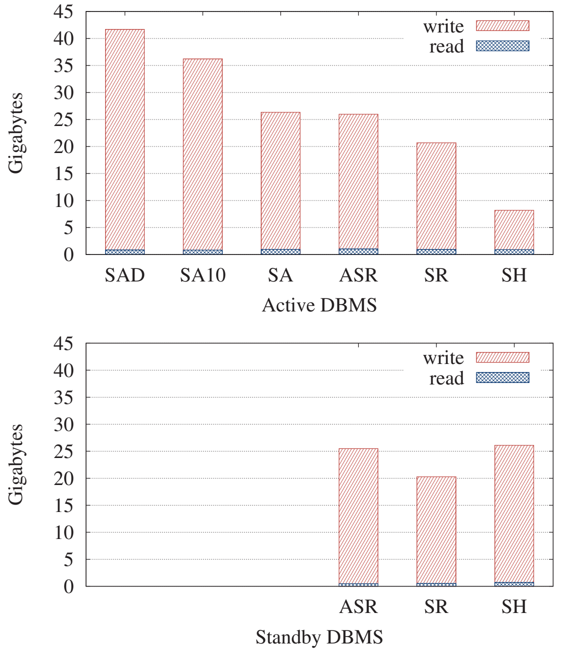


Figure 9. Total Volume of Database I/O

cause its throughput is lower.) The baseline systems (SR and ASR) perform the same amount of I/O at the active and the standby, while SHADOW's I/O pattern is asymmetric, with most of the I/O occurring at the standby.

6.3 Small Memory Scenario

In this section we present the results of an experiment that is identical to the one presented in Section 6.2, except that size of the DBMS buffer cache is reduced from 26GB to 8GB. As noted in Section 6.1, the TPC-C database starts at 11GB in size, and grows significantly larger over the course of the experiment. Because the DBMS buffer cache cannot hold the entire database, we expect that all systems will have to perform more I/O to the database volume(s). Our objective is to see how this change affects the performance of SHADOW relative to the baseline systems.

Figure 10 shows the mean throughput of SHADOW and the baselines in the small memory configuration. Unsurprisingly, all of the configurations achieve lower throughput in the small memory configuration than in the large memory configuration (Figure 7). However, SHADOW-EBS is affected the least by this change. As a result, the throughput of SHADOW-EBS is about 20% better than that of the best standalone baseline (SA). Because of write offloading, SHADOW places less I/O load than the baselines on the database volume of the active DBMS. For example, about 75% of the database I/O bandwidth used by the SA baseline is devoted to writes, leaving only 25% for reads. In contrast, SHADOW is able to devote about 40% of its I/O bandwidth to reads. This number is grows to more than 50% if we ignore writes that are attributable to database growth. Thus, in addition to avoiding checkpointing-induced performance fluctuations, SHADOW's write offloading frees more of the available I/O bandwidth for database reads in scenarios in which the entire database does not fit in memory.

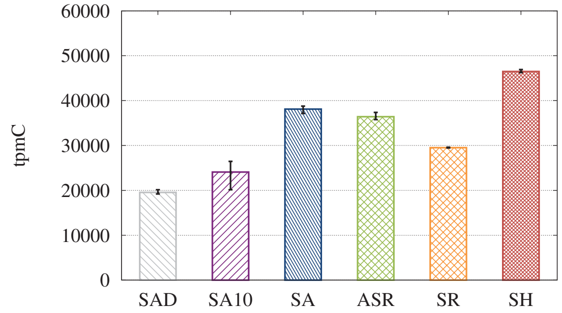


Figure 10. TPC-C Throughput, Small Memory Case

6.4 Direct-attached Storage Scenario

In all of the experiments reported so far, both SHADOW and the baseline systems make use of EBS volumes to hold the database and logs. This is common in cloud settings, and in particular in Amazon's EC2 environment, since storage devices that are directly attached to EC2 virtual machine instances (known as *instance storage*) are ephemeral. This means that if the instance fails or is shut down, the contents of instance storage are permanently lost. Nonetheless, there are other settings in which it is possible for a highly available database system to use directly attached persistent storage, e.g., locally attached disks or SSDs, for database and log volumes. In contrast, SHADOW *must* use reliable, network-attached shared storage for its log and database volumes.

Since directly attached storage may have better performance than network-attached storage, SHADOW's requirement for network-attached storage may place it at a performance disadvantage. In this section, we present an additional experiment that explores how significant this effect is. For this experiment, we configured the SR and ASR baseline systems to use instance storage, rather than EBS volumes, for both the database and the log. We refer to these new configurations as ASR-EP and SR-EP. In the EC2 environment, these new configurations have *no* persistent copy of the database or the log, since instance storage is ephemeral. Nonetheless, we tested these configurations as representatives of a non-cloud setting in which directly attached storage is persistent, rather than ephemeral.

Both the instance storage used by ASR-EP and SR-EP and the EBS volumes used by SHADOW and the original baselines are SSD-based. However, the instance storage offers better I/O performance than the network-attached EBS volumes we used. For example, for a single-threaded sequential write workload, our benchmarking measured over 1200 IOPS for instance storage (about 0.8 ms per request),

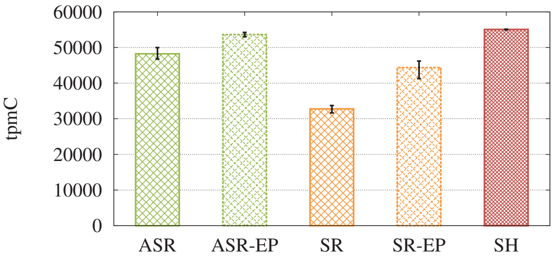


Figure 11. TPC-C Throughput, Ephemeral Volumes

versus approximately 750 IOPS (about 1.3 ms per request) for the EBS volumes.

We ran our TPC-C workload using the new SR-EP and ASR-EP baselines, with all of the database servers configured as in our large memory scenario. Figure 11 shows the throughput achieved by both of these baselines. For comparison, the figure also shows the throughput achieved by SR, ASR, and SHADOW with large memory, from Figure 7.

Switching from ASR to ASR-EP eliminated the small performance gap that existed between SHADOW and ASR. This gap was due primarily to interference from checkpointing, which hurt ASR. By moving the database to instance storage, which offers greater I/O throughput than our provisioned EBS volumes, the performance impact from checkpointing is greatly reduced. Unfortunately, ASR can lose committed transactions during failover.

Comparing SHADOW to SR and SR-EP, we see that SR-EP outperforms SR, but it is still outperformed by SHADOW. The performance gap between SR-EP and SHADOW is interesting, and can be attributed to the way these two systems commit transactions. At first glance, one might expect that committing a transaction would be slower in SHADOW than in SR-EP because SHADOW's network-attached EBS log volume has higher latency than SR-EP's directly attached log volume. However, for SR and SR-EP, committing a transaction involves *two* log writes, one at the active and one at the standby, as well as a network round trip between the two systems. Furthermore, the two log writes happen *sequentially*, i.e., the transaction first commits at the active, and it is then hardened by writing the commit record again at the standby. For SHADOW, committing a transaction involves only a single EBS write. Although EBS writes are slower than writes to instance storage, they are still faster than performing two instance writes in sequence, even if we ignore the network latency between the active and the standby. Thus, transactions have higher commit latencies in SR-EP than in SHADOW, which translates to lower throughput in our closed-loop testing environment.

6.5 Failover

We also compared the time required for failover in a SHADOW-EBS system to the failover time under PostgreSQL's native synchronous replication (SR). In this experiment, we first ran the system under load in ACTIVE+STANDBY state for ten minutes before killing the active DBMS process (active failure) and initiating an immediate failover to the standby. The ten minute warmup time was chosen to be large enough to allow the DBMS caches to warm up in both systems.

The FAILOVER operation in the SR system took about 3 seconds. FAILOVER in the SHADOW system required about 16 seconds. In both cases, replaying the remaining log at the standby required about 2 seconds. However, the SHADOW system required additional time to detach the log volume from the active instance (about 9 seconds) and attach it to the standby (about 4 seconds). Had failover resulted from a failure of the active virtual machine (rather than just the DBMS), re-attaching the log to the standby would be much faster (about 5 seconds total, rather than 13), and so the FAILOVER operation would be faster. In summary, failover times for both SR and SHADOW-EBS are short, but SHADOW-EBS is somewhat slower because of the need to re-attach the SHADOW log volume. We note that this extra delay is a consequence of the EBS limitation that a volume can attach to at most one server instance at a time. With a persistent storage tier that does not have this limitation, we expect this difference to disappear.

7. Conclusions

We have presented SHADOW, a novel hot standby architecture that exploits shared persistent storage, which is commonly available in cloud settings. In SHADOW, the active and standby database systems share access to a single copy of the database and log. 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.

SHADOW is a novel and effective way to build a highly available database service in cloud settings. SHADOW reduces complexity at the DBMS level by pushing responsibility for replication out of the DBMS and into the underlying storage tier. SHADOW also decouples database replication from DBMS replication. Our experiments with TPC-C workloads show that these advantages do not come at the expense of performance. Our SHADOW-EBS prototype significantly outperforms PostgreSQL's native synchronous replication. Because of write offloading, SHADOW-EBS can even outperform a standalone (not highly available) DBMS, even one that is aggressively tuned to favor performance over recovery time.

Acknowledgments

This work was supported by a NetApp Faculty Fellowship, and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Transparent application scaling with IBM DB2 pureScale. IBM white paper, 2009.

[2] Oracle Real Application Clusters 11g release 2. Oracle white paper, 2010.

[3] *Amazon Relational Database Service User Guide*, 2013.

[4] *PostgreSQL 9.3 Documentation*, 2014. URL `http://www.postgresql.org/docs/9.3/`.

[5] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. CIDR*, 2011.

[6] S. Bartkowski et al. High availability and disaster recovery options for DB2 for Linux, UNIX, and Windows. IBM Redbook, 2012.

[7] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kallan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting Microsoft SQL Server for cloud computing. In *Proc. ICDE*, 2011.

[8] B. F. Cooper, R. Ramakrishnan, et al. Pnuts: Yahoo!'s hosted data serving platform. In *Proc. VLDB*, 2008.

[9] J. C. Corbett, et al. Spanner: Google's globally-distributed database. In *Proc. USENIX OSDI*, 2012.

[10] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Proc. USENIX ATC*, 2012.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] M. Hart and S. Jesse. *Oracle Database 10g High Availability with RAC, Flashback, and Data Guard*. McGraw-Hill, 2004.

[13] D. Komo. Microsoft SQL Server 2008 R2 high availability technologies. Microsoft white paper, 2010.

[14] L. Lamport. The part-time parliament. *ACM TODS*, 16(2): 133–169, 1998.

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

[16] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. SIGMOD*, 1988.

[17] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[18] The Tandem Database Group. NonStop SQL, a distributed high performance, high availability implementation of SQL. In *Proc. HPTS*, 1989.

[19] L. Tuttle, Jr. Microsoft SQL Server AlwaysOn solutions guide for high availability and disaster recovery. Microsoft white paper, 2012.

[20] S. B. Vaghani. Virtual machine file system. *ACM SIGOPS OSR*, 44(4):57–70, 2010.