DAX: A Widely Distributed Multi-tenant Storage Service for DBMS Hosting

Rui Liu
University of Waterloo
r46liu@uwaterloo.ca

Ashraf Aboulnaga
University of Waterloo
ashraf@uwaterloo.ca

Kenneth Salem
University of Waterloo
kmsalem@uwaterloo.ca

ABSTRACT

Many applications hosted on the cloud have sophisticated data management needs that are best served by a SQL-based relational DBMS. It is not difficult to run a DBMS in the cloud, and in many cases one DBMS instance is enough to support an application's workload. However, a DBMS running in the cloud (or even on a local server) still needs a way to persistently store its data and protect it against failures. One way to achieve this is to provide a scalable and reliable storage service that the DBMS can access over a network. This paper describes such a service, which we call DAX. DAX relies on multi-master replication and Dynamo-style flexible consistency, which enables it to run in multiple data centers and hence be disaster tolerant. Flexible consistency allows DAX to control the consistency level of each read or write operation, choosing between strong consistency at the cost of high latency or weak consistency with low latency. DAX makes this choice for each read or write operation by applying protocols that we designed based on the storage tier usage characteristics of database systems. With these protocols, DAX provides a storage service that can host multiple DBMS tenants, scaling with the number of tenants and the required storage capacity and bandwidth. DAX also provides high availability and disaster tolerance for the DBMS storage tier. Experiments using the TPC-C benchmark show that DAX provides up to a factor of 4 performance improvement over baseline solutions that do not exploit flexible consistency.

1. INTRODUCTION

In this paper we present a storage service that is intended to support cloud-hosted, data-centric applications. There is a wide variety of cloud data management systems that such applications can use to manage their persistent data. At one extreme are so-called NoSQL database systems, such as BigTable [7] and Cassandra [20]. These systems are scalable and highly available, but their functionality is limited. They typically provide simple key-based, record-level, read/write interfaces and limited (or no) support for application-defined transactions. Applications that use such systems directly must either tolerate or work around these limitations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.

Proceedings of the VLDB Endowment, Vol. 6, No. 4

Copyright 2013 VLDB Endowment 2150-8097/13/02... $ 10.00.

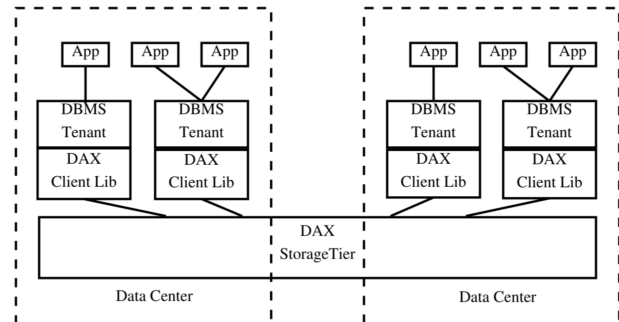

Figure 1: DAX architecture.

At the other extreme, applications can store their data using cloud-hosted relational database management systems (DBMS). For example, clients of infrastructure-as-a-service providers, such as Amazon, can deploy DBMS in virtual machines and use these to provide database management services for their applications. Alternatively, applications can use services such as Amazon's RDS or Microsoft SQL Azure [4] in a similar way. This approach is best suited to applications that can be supported by a single DBMS instance, or that can be sharded across multiple independent DBMS instances. High availability is also an issue, as the DBMS represents a single point of failure – a problem typically addressed using DBMS-level high availability techniques. Despite these limitations, this approach is widely used because it puts all of the well-understood benefits of relational DBMS, such as SQL query processing and transaction support, in service of the application. This is the approach we focus on in this paper.

A cloud-hosted DBMS must have some means of persistently storing its database. One approach is to use a persistent storage service provided within the cloud and accessed over the network by the DBMS. An example of this is Amazon's Elastic Block Service (EBS), which provides network-accessible persistent storage volumes that can be attached to virtual machines.

In this paper, we present a back-end storage service called *DAX*, for Distributed Application-controlled Consistent Store, that is intended to provide network-accessible persistent storage for hosted DBMS tenants. Each DBMS tenant, in turn, supports one or more applications. This architecture is illustrated in Figure 1. We have several ob-

jectives for DAX's design that set it apart from other distributed storage services:

- **scalable tenancy:** DAX must accommodate the aggregate storage demand (space and bandwidth) of all of its DBMS tenants, and it should be able to scale out to accommodate additional tenants or to meet growing demand from existing tenants. Our focus is on scaling out the storage tier to accommodate more tenants, not on scaling out individual DBMS tenants. A substantial amount of previous research has considered techniques for scaling out DBMS [12, 14, 22] and techniques like sharding [15] are widely used in practice. These techniques can also be used to scale individual hosted DBMS running on DAX. Furthermore, while DBMS scale-out is clearly an important issue, current trends in server hardware and software are making it more likely that an application can be supported by a single DBMS instance, with no need for scale out. For example, it is possible at the time of this writing to rent a virtual machine in Amazon's EC2 with 68 GB of main memory and 8 CPU cores, and physical servers with 1 TB of main memory and 32 or more cores are becoming commonplace. While such powerful servers reduce the need for elastic DBMS scale-out, we still need a scalable and resilient storage tier, which we provide with DAX.

- **high availability and disaster tolerance:** We expect the storage service provided by DAX to remain available despite failures of DAX servers (high availability), and even in the face of the loss of entire data centers (disaster tolerance). To achieve this, DAX replicates data across multiple geographically distributed data centers. If a DBMS tenant fails, DAX ensures that the DBMS can be restarted either in the same data center or in a *different data center*. The tenant may be unavailable to its applications while it recovers, but DAX will ensure that no durable updates are lost. Thus, DAX offloads *part* of the work of protecting a tenant DBMS from the effects of failures. It provides highly available and widely distributed access to the stored database, but leaves the task of ensuring that the database *service* remains available to the hosted DBMS.

- **consistency:** DAX should be able to host legacy DBMS tenants, which expect to be provided with a consistent view of the underlying stored database. Thus, DAX must provide sufficiently strong consistency guarantees to its tenants.

- **DBMS specialization:** DAX assumes that its clients have DBMS-like properties. We discuss these assumptions further in Section 2.

A common way to build strongly consistent highly available data stores is through the use of synchronous master-slave replication. However, such systems are difficult to distribute over wide areas. Therefore, we have instead based DAX on multi-master replication and Dynamo-style flexible consistency [16]. Flexible consistency means that clients (in our case, the DBMS tenants) can perform either fast read operations for which the storage system can provide only weak consistency guarantees or slower read operations with strong consistency guarantees. Similarly, clients can perform fast writes with weak durability guarantees, or relatively slow writes with stronger guarantees. As we show in Section 3, these performance/consistency and performance/durability tradeoffs can be substantial, especially in widely distributed systems. DAX *controls these trade-offs automatically* on behalf of its DBMS tenants. Its goal is to approach the performance of the fast weakly consistent (or weakly durable) operations, *while still providing strong guarantees to the tenants*. It does this in part by taking advantage of the specific nature of the workload for which it is targeted, e.g., the fact that each piece of stored data is used by only one tenant.

This paper makes several research contributions. First, we present a technique called *optimistic I/O*, which controls the consistency of the storage operations issued by the DBMS tenants. Optimistic I/O aims to achieve performance approaching that of fast weakly consistent operations while ensuring that the DBMS tenant sees a sufficiently consistent view of storage.

Second, we describe DAX's use of *client-controlled synchronization*. Client-controlled synchronization allows DAX to defer making guarantees about the durability of updates until the DBMS tenant indicates that a guarantee is required. This allows DAX to hide some of the latency associated with updating replicated data. While client-controlled synchronization could potentially be used by other types of tenants, it is a natural fit with DBMS tenants, which carefully control update durability in order to implement database transactions.

Third, we define a consistency model for DAX that accounts for the presence of explicit, DBMS-controlled synchronization points. The model defines the consistency requirements that a storage system needs to guarantee to ensure correct operation when a DBMS fails and is restarted after the failure (possible in a different data center).

Finally, we present an evaluation of the performance of DAX running in Amazon's EC2 cloud. We demonstrate its scalability and availability using TPC-C workloads, and we also measure the effectiveness of optimistic I/O and client-controlled synchronization.

2. SYSTEM OVERVIEW

The system architecture illustrated in Figure 1 includes a client library for each DBMS tenant and a shared storage tier. The DAX storage tier provides a reliable, widely distributed, shared block storage service. Each tenant keeps all of its persistent data, including its database and logs, in DAX. We assume each DBMS tenant views persistent storage as a set of files which are divided into fixed-size blocks. The tenants issue requests to read and write blocks. The DAX client library intercepts these requests and redirects them to the DAX storage tier.

Since the tenants are relational DBMS, there are some constraints on the workload that is expected by DAX. First, since each tenant manages an independent database, the storage tier assumes that, at any time, only a single client can update each stored block. The single writer may change over time. For example, a tenant DBMS may fail and be restarted, perhaps in a different data center. However, at any time, each block is owned by a single tenant. Second, because each DBMS tenant manages its own block buffer cache, there are no concurrent requests for a single block. A DBMS may request different blocks concurrently, but all requests for any particular block are totally ordered.

Since each DBMS tenant reads and writes a set of non-overlapping blocks, one way to implement the DAX storage tier is to use a distributed, replicated key/value store. To

do this, we can use block identifiers (file identifier plus block offset) as unique keys, and the block's contents as the value. In the remainder of this section, we describe a *baseline* implementation of the DAX storage tier based on this idea.

We will use as our baseline a distributed, replicated, multi-master, flexible consistency, key/value system. Examples of such systems include Dynamo [16], Cassandra [20], and Voldemort [21]. The multi-master design of these systems enables them to run in geographically distributed data centers, which we require for disaster tolerance in DAX. Next, we briefly describe how such a system would handle DBMS block read and write requests, using Cassandra as our model.

In Cassandra, each stored value (a block, in our case) is replicated N times in the system, with placement determined by the key (block identifier). A client connects to any Cassandra server and submits a read or write request. The server to which the client connects acts as the coordinator for that request. If the request is a write, the coordinator determines which servers hold copies of the block, sends the new value of the block to those servers, and waits for acknowledgements from at least a *write quorum* W of the servers before sending an acknowledgement to the client. Similarly, if the request is a read, the coordinator sends a request for the block to those servers that have it and waits for each least a *read quorum* R of servers to respond. The coordinator then sends to the client the most recent copy of the requested block, as determined by timestamps (details in the next paragraph). The Cassandra client can balance consistency, durability, and performance by controlling the values of R and W. As we will illustrate in the next section, these tradeoffs can be particularly significant when the copies are distributed across remote data centers, as message latencies may be long.

Since a DBMS normally stores data in a file system or directly on raw storage devices, it expects that when it reads a block it will obtain the most recent version of that block, and not a stale version. In classic quorum-based multi-master systems [17], this is ensured by requiring that $R + W > N$. Global write-ordering can be achieved by requiring that $W > N/2$, with the write order determined by the order in which write operations obtain their quorums. Cassandra takes a slightly different approach to ordering writes: clients are expected to supply a timestamp with each write operation, and these timestamps are stored with the data in Cassandra. The global write-ordering is determined by these client-defined timestamps. That is, the most recent write is defined to be the write with the largest timestamp. Thus, in our baseline DAX implementation, we can ensure that each read sees the most recent version of the block as follows. First, the client library chooses monotonically increasing timestamps for each block write operation issued by its DBMS tenant. This is easy to do since all updates of a given block originate from one tenant and are not concurrent. Second, the client submits read and write requests on behalf of its tenant, choosing any R and W such that $R + W > N$. This ensures that each read of a block will see the most recent preceding update.

There are two potentially significant drawbacks to this baseline. First, it may perform poorly, especially in a geographically distributed setting, because of the requirement that $R + W > N$. Second, failures may impact the performance and availability of such a system. In the remainder of the paper, we describe how DAX addresses these drawbacks.

3. OPTIMISTIC I/O

We conducted some simple experiments to quantify the performance tradeoffs in widely-distributed, multi-master, flexible consistency systems, like the baseline system described in the previous section. We again used Cassandra as a representative system and deployed it in Amazon's EC2 cloud. In each experiment we used a small 6-server Cassandra cluster deployed in one of three different configurations:

1 zone: All six servers are in the same EC2 *availability zone*, which is roughly analogous to a data center.

1 region: The servers are split evenly among three availability zones, all of which are located in the same geographic region (EC2's US East Region, in Virginia).

3 regions: The servers are split evenly among three availability zones, with one zone in each of three geographically distributed EC2 regions: US East, US West-1 (Northern California), and US West-2 (Oregon).

We used the Yahoo! Cloud Serving Benchmark [10] to provide a Cassandra client application which reads or writes values from randomly selected rows in a billion-key Cassandra column family (a collection of tuples), at a rate of 1000 requests/second. The client runs in the US East region. In these experiments, the degree of replication (N) is 3, and we ensure that one copy of each row is located in each zone in the multi-zone configurations. The client can be configured to generate read operations with $R = 1$, $R = N/2 + 1$, or $R = N$, henceforth referred to as `Read(1)`, `Read(QUORUM)`, and `Read(ALL)` respectively. Similarly, writes can be configured with $W = 1$ (`Write(1)`), $W = N/2+1$ (`Write(QUORUM)`), or $W = N$ (`Write(ALL)`). We measured the latency of these operations in each Cassandra configuration.

Figure 2(a) illustrates the read and write latencies for the 1-zone configuration, in which we expect the servers to have low-latency interconnectivity. In this configuration, `Read(ALL)` and `Write(ALL)` requests take roughly 60% longer than `Read(1)` and `Write(1)`, with the latency of `Read(QUORUM)` and `Write(QUORUM)` lying in between. The situation is similar in the 1-region configuration (Figure 2(b)), although the latency penalty for reading or writing a quorum of copies or all copies is higher than in the 1-zone configuration.

In the 3-region configuration (Figure 2(c)), latencies for `Read(ALL)` and `Write(ALL)` operations, as well as `Read(QUORUM)` and `Write(QUORUM)`, are significantly higher, almost ten times higher than latencies in the 1-region configuration. The performance tradeoff between `Read(1)` and `Read(QUORUM)` or `Read(ALL)`, and between `Write(1)` and `Write(QUORUM)` or `Write(ALL)`, is now much steeper – about a factor of ten. Furthermore, note that `Read(1)` and `Write(1)` are almost as fast in this configuration as they were in the two single-region configurations. Although Cassandra sends `Read(1)` and `Write(1)` operations to all replicas, including those in the remote regions, it can respond to the client as soon as the operation has completed at a single replica, which will typically be local.

In summary, operations which require acknowledgements from many replicas are slower than those that require only one acknowledgement, and this "consistency penalty" is significantly larger when the replicas are geographically distributed. In our baseline system, either read operations or write operations (or both) would suffer this penalty.

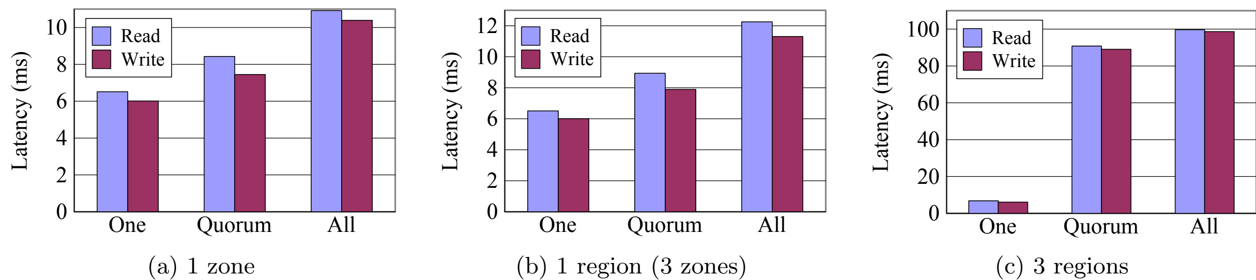

Figure 2: Latency of Read and Write operations in three Cassandra configurations.

3.1 Basic Optimistic I/O

Optimistic I/O is a technique for ensuring consistent reads for DBMS tenants, while avoiding most of the consistency penalty shown in the previous section. In this section we describe an initial, basic version of optimistic I/O, for which we will assume that there are no server failures. In the following sections, we consider the impact of server failures and describe how to refine and extend the basic optimistic I/O technique so that failures can be tolerated.

Optimistic I/O is based on the following observation: using `Read(1)` and `Write(1)` for reading and writing blocks does not *guarantee* consistency, but the `Read(1)` operation will *usually* return the most recent version. This assumes that the storage tier updates *all* replicas of a block in response to a `Write(1)` request, though it only waits for the first replica to acknowledge the update before acknowledging the operation to the client. The remaining replica updates complete asynchronously. Since all replicas are updated, whichever copy the subsequent `Read(1)` returns is likely to be current. Thus, to write data quickly, the storage tier can use fast `Write(1)` operations, and to read data quickly it can optimistically perform fast `Read(1)` operations and hope that they return the latest version. If they do not, then we must fall back to some alternative to obtain the latest version. To the extent that our optimism is justified and `Read(1)` returns the latest value, we will be able to obtain consistent I/O using fast `Read(1)` and `Write(1)` operations. Note that using `Write(1)` provides only a weak durability guarantee, since only one replica is known to have been updated when the write request is acknowledged. We will return to this issue in Section 4.

To implement optimistic I/O, we need a mechanism by which the storage tier can determine whether a `Read(1)` has returned the latest version. DAX does this using per-block version numbers, which are managed by the client library. The library assigns monotonically increasing version numbers to blocks as they are written by the tenant DBMS. The storage tier stores a version number with each replica and uses the version numbers to determine which replica is the most recent. The client library also maintains an in-memory *version list* in which it records the most recent version numbers of different blocks. When the client library writes a block, it records the version number associated with this write in the version list. When a block is read using `Read(1)`, the storage tier returns the first block replica that it is able to obtain, along with that replica's stored version number. The client library compares the returned version number with the most recent version number from the version list to determine whether the returned block replica is stale. Version lists have a configurable maximum size. If a client's version list grows too large, it evicts entries from the list using a least-recently-used (LRU) policy.

Since the version list is maintained in memory by the client library, it does not persist across tenant failures. When a DBMS tenant first starts, or when it restarts after a failure, its version list will be empty. The version list is populated gradually as the DBMS writes blocks. If the DBMS reads a block for which there is no entry in the version list, DAX cannot use `Read(1)`, since it will not be able to check the returned version for staleness. The same situation can occur if a block's latest version number is evicted from the version list by the LRU policy. In these situations, the client library falls back to `Read(ALL)`, which is guaranteed to return the latest version, and it records this latest version in the version list.

When the client library detects a stale block read, it must try again to obtain the latest version. One option is to perform the second read using `Read(ALL)`, which will ensure that the latest version is returned. An alternative, which DAX uses, is to simply retry the `Read(1)` operation. As described in Section 6, DAX uses an asynchronous mechanism to bring stale replicas up to date. Because of this asynchronous activity, a retried `Read(1)` operation will often succeed in returning the latest version even if it accesses the same replica as the original, stale `Read(1)`.

4. HANDLING FAILURES

The basic optimistic I/O protocol in Section 3.1 is not tolerant of failures of DAX servers. Since it uses `Write(1)`, loss of even a single replica may destroy the only copy of an update in the storage tier, compromising durability. (Other copies may have been successfully updated, but there is no guarantee of this.) Conversely, since it uses `Read(ALL)` when Read(1) cannot be used, failure of even a single replica will prevent read operations from completing until that replica is recovered, leading to a loss of availability.

Both of these problems can be resolved by increasing W and decreasing R such that $R + W > N$ is maintained. In particular, we could use `Write(QUORUM)` and `Read(QUORUM)` in the basic protocol in place of `Write(1)` and `Read(ALL)`. The drawback of this simple approach is that `Write(QUORUM)` may be significantly slower than `Write(1)`. This is particularly true, as shown in Figure 2(c), if we insist that the write quorum include replicas in multiple, georgraphically distributed data centers. As discussed in Section 4.2, this is exactly what is needed to tolerate data center failures.

To achieve fault tolerance without the full expense of `Write(QUORUM)`, DAX makes use of *client-controlled synchro-*

nization. The idea is to weaken the durability guarantee slightly, so that the storage tier need not immediately guarantee the durability of every write. Instead, we allow each DBMS tenant to define explicit synchronization points by which preceding updates must be reflected in a quorum (k) of replicas. Until a DBMS has explicitly synchronized an update, it cannot assume that this update is durable.

This approach is very natural because many DBMS are already designed to use explicit update synchronization points. For example, a DBMS that implements the write-ahead logging protocol will need to ensure that the transaction log record describing a database update is safely in the log before the database itself can be updated. To do this, the DBMS will write the log page and then issue a write synchronization operation, such as a POSIX `fsync` call, to obtain a guarantee from the underlying file system that the log record has been forced all the way to the underlying persistent storage, and will therefore be durable. Similarly, a DBMS may write a batch of dirty blocks from its buffer to the storage system and then perform a final synchronization to ensure that the whole batch is durable. Any delay between a write operation and the subsequent synchronization point provides an opportunity for the storage tier to hide some or all of the latency associated with that write.

When DAX receives a write request from a DBMS tenant, it uses a new type of write operation that we have defined, called `Write(CSYNC)`. Like `Write(1)`, `Write(CSYNC)` immediately sends update requests to all replicas and returns to the client after a single replica has acknowledged the update. In addition, `Write(CSYNC)` records the key (i.e., the block identifier) in a per-client `sync_pending` list at the DAX server to which the client library is connected. The key remains in the `sync_pending` list until all remaining replicas have, asynchronously, acknowledged the update, at which point it is removed. `Write(CSYNC)` operations experience about the same latency as `Write(1)`, since both return as soon as one replica acknowledges the update.

When the DBMS needs to ensure that a write is durable (at least k replicas updated) it issues an explicit synchronization request. To implement these requests, DAX provides an operation called `CSync`, which is analogous to the POSIX `fsync`. On a `CSync`, the DAX server to which the client is connected makes a snapshot of the client's `sync_pending` list and blocks until at least k replica update acknowledgements have been received for each update on the list. This ensures that all preceding writes from this DBMS tenant are durable, i.e., replicated at least k times. The synchronization request is then acknowledged to the client.

Figure 3 summarizes the refined version of the optimistic I/O protocol that uses these new operations. We will refer to this version as `CSync`-*aware optimistic I/O* to distinguish it from the basic protocol that was presented in Section 3. In the `CSync`-aware protocol, `Write(CSYNC)` is used in place of `Write(1)` and `Read(QUORUM)` is used in place of `Read(ALL)`. In addition, the client library executes the `CSync` procedure when the DBMS tenant performs a POSIX `fsync` operation.

There are two details of the `CSync`-aware optimistic protocol that are not shown in Figure 3. First, DAX ensures that every block that is on the `sync_pending` list is also on the version list. (The version list replacement policy, implemented by `VersionList.put` in Figure 3, simply skips blocks with unsynchronized updates.) This ensures that the DAX client library always knows the current version number

Read procedure:

```
(1)  proc Read(blockId) ≡
(2)    begin
(3)      if blockId ∈ VersionList
(4)        then
(5)            (data, v) := Read(1);
(6)            comment: v is the stored version number
(7)            while v < VersionList.get(blockId) do
(8)                comment: retry a stale read;
(9)                comment: give up if too many retries
(10)               comment: and return an error
(11)               (data, v) := Read(1); od
(12)       else
(13)           comment: fall back to reading a quorum
(14)           (data, v) := Read(QUORUM);
(15)           VersionList.put(blockId, v); fi
(16)     return data;
(17)   end
```

Write procedure:

```
(1)  proc Write(blockId, data) ≡
(2)    begin
(3)      v := GenerateNewVersionNum();
(4)      comment: write the data using blockId as the key
(5)      Write(CSYNC);
(6)      VersionList.put(blockId, v);
(7)    end
```

Fsync procedure:

```
(1)  proc Fsync() ≡
(2)    begin
(3)      comment: blocks until writes on sync_pending
(4)      comment: list are durable
(5)      Csync();
(6)    end
```

Figure 3: `CSync`-aware optimistic I/O protocol.

for blocks with unsynchronized updates, and need not rely on `Read(QUORUM)` to read the current version. `Read(QUORUM)` cannot be relied on to do so, since the unsynchronized update may not yet be present at a full write quorum of replicas. Second, DAX maintains a copy of the `sync_pending` list at the client library, in case the DAX server to which it is connected should fail. If such a failure occurs, the client library connects to any other DAX server and initializes the `sync_pending` list there using its copy.

Two parameters control the availability and durability of the `CSync`-aware optimistic I/O protocol: N, the total number of copies of each block, and k, the number of copies that must acknowledge a synchronized update. Any *synchronized* update will survive up to $k-1$ concurrent DAX server failures. Unsynchronized updates may not, but the DBMS does not depend on the durability of such updates. DAX will normally be available (for read, write, and `CSync` operations), as long as no more than $\min(k-1, N-k)$ servers have failed. However, client-controlled synchronization does introduce a small risk that a smaller number of failures may compromise read availability. This may occur if all DAX servers holding an unsynchronized update (which is not yet guaranteed to be replicated k times) should fail. Since `Write(CSYNC)` sends updates to all N copies of an object immediately (though it does not wait for acknowledgements), such a situation is

unlikely. Should it occur, it will be manifested as a failed read operation (lines 7-11 in Figure 3), which would require a restart of the DBMS server (see Section 4.1), and hence a temporary loss of DBMS availability. Since the DBMS does not depend on the durability of unsynchronized writes, it will be able to recover committed database updates during restart using its normal recovery procedure (e.g., from the transaction log). We never encountered this type of read failure with our DAX prototype, although we do sometimes need to retry `Read(l)` operations. The maximum number of retries we saw in all of our experiments is 10.

4.1 Failure of a DBMS Tenant

Recovery from a failure of a DBMS tenant is accomplished by starting a fresh instance of that DBMS, on a new server if necessary. The new tenant instance goes through the normal DBMS recovery process using the transaction log and database stored persistently in the DAX storage tier. This approach to recovering failed DBMS instances is also used by existing services like the Amazon Relational Database Service (RDS). However, because DAX can be geographically distributed, a DAX tenant can be restarted, if desired, in a remote data center. The DBMS recovery process may result in some tenant downtime. If downtime cannot be tolerated, a DBMS-level high availability mechanism can be used to reduce or eliminate downtime. However, a discussion of such mechanisms is outside the scope of this paper.

4.2 Loss of a Data Center

If the DAX storage tier spans multiple data centers, it can be used to ensure that stored data survives the loss of all servers in a data center, as can happen, for example, due to a natural disaster. Stored blocks will remain available provided that a quorum of block copies is present at the surviving data center(s). DBMS tenants hosted in the failed data center need to be restarted in a new data center and recovered as discussed in Section 4.1.

To ensure that DAX can survive a data center failure, the only additional mechanism we need is a data-center-aware policy for placement of replicas. The purpose of such a policy is to ensure that the storage tier distributes replicas of every stored block across multiple data centers. Our DAX prototype, which is based on Cassandra, is able to leverage Cassandra's replica placement policies to achieve this.

4.3 Consistency

In this section, we discuss the consistency guarantees made by the `CSync`-aware optimistic I/O protocol shown in Figure 3. We will use the example timeline shown in Figure 4 to illustrate the consistency guarantees. The timeline represents a series of reads and writes of a *single stored block*. Because these requests are generated by a single DBMS, they occur sequentially as disucussed in Section 2. Writes $W0$ and $W1$ in Figure 4 are synchronized writes, because they are followed by a `CSync` operation. The remaining writes are unsynchronized. The timeline is divided into *epochs*. Each failure of the hosted DBMS defines an epoch boundary. Three different epochs are shown in the figure.

The `CSync`-aware optimistic I/O protocol ensures that each read sees the block as written by the most recent preceding write *in its epoch*, if such a write exists. Thus, $R1$ will see the block as written by $W3$ and $R4$ will see the block as written by $W4$. We will refer to this property as *epoch-bounded strong consistency*. It is identical to the usual notion of strong consistency, but it applies only to reads for which there is at least one preceding write (of the same block) in the same epoch.

We refer to reads for which there is no preceding write in the same epoch as *initial reads*. In Figure 4, $R2$, $R3$, and $R5$ are initial reads. For initial reads, the `CSync`-aware optimistic I/O protocol guarantees only that the read will read a version at least as recent as that written by latest preceding *synchronized* write. In our example, this means that $R2$ and $R3$ will see what is written by either $W1$, $W2$, or $W3$, and $R5$ will see one of those versions or the version written by $W4$. This may lead to problematic anomalies. For example, $R2$ may return the version written by $W3$, while $R3$ returns the earlier version written by $W1$. If $R2$ occurs during log-based recovery, the DBMS may decide that the block is up-to-date and thus will not update it from the log. If $R3$ occurs during normal operation of the new DBMS instance, those committed updates will be missing from the block. Thus, the DBMS will have failed to preserve the atomicity or durability of one or more its transactions from the first epoch. In the following section, we present *strong optimistic I/O*, an extension of the protocol shown in Figure 3 that provides stronger guarantees for initial reads.

5. STRONG OPTIMISTIC I/O

One option for handling initial reads is to try to extend the optimistic I/O protocol so that it provides a strong consistency guarantee across epochs as well as within them. However, such a guarantee would be both difficult to enforce and unnecessarily strong. Instead, we argue that the DAX storage tier should behave like a *non-replicated, locally attached* storage device or file system that has an explicit synchronization operation (i.e., like a local disk). Most DBMS are designed to work with such a storage system. In such a system, unsynchronized writes may survive a failure, but they are not guaranteed to do so. However, the key point is that *only one version of each block will survive*. The survival candidates include (a) the version written by the most recent synchronized update preceding the failure and (b) the versions written by any (unsynchronized) updates that occur after the most recent synchronized update. Thus, at $R2$ the storage tier would be free to return either the version written by $W1$ or the version written by $W2$ or the version written by $W3$. However, once it has shown a particular version to $R2$, it must show *the same version* to $R3$, since only one version should survive across a failure, i.e., across an epoch boundary. Similarly, at $R5$ the storage tier can return either the same version that was returned to $R2$ (and $R3$) or the version that was written by $W4$. We can summarize these additional consistency requirements as follows:

- A read operation with no preceding operations (on the same block) in the same epoch (such as $R2$ and $R5$ in Figure 4) must return either (a) the version written by the most recent synchronized update in the preceding epoch (or the version that exists at the beginning of the epoch if there is no synchronized update) or (b) a version written by one of the (unsynchronized) updates that occurs after the most recent synchronized update. We refer to this as *Non-deterministic One-Copy (NOC) consistency*.

- Consecutive read operations, with no intervening writes, must return the same version. Thus, in Figure 4, $R3$ must

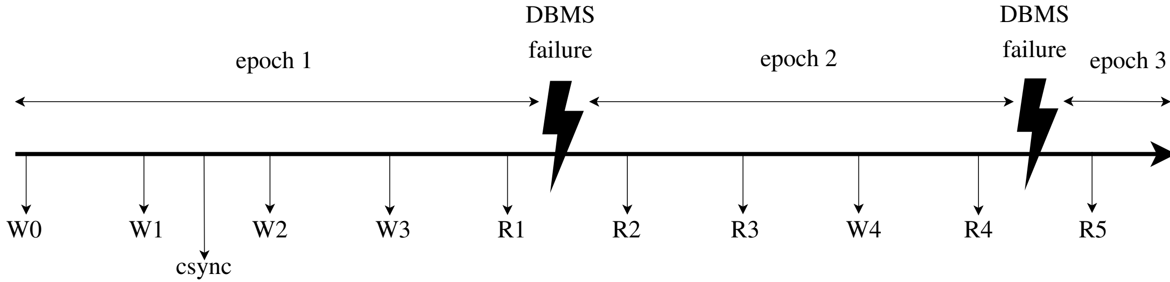

Figure 4: Timeline example showing reads and writes of a single block in the presence of failures.

return the same version as $R2$. We refer to this property as *read stability*.

Next, we present strong optimistic I/O, a refinement of optimistic I/O that provides NOC consistency and read stability, in addition to epoch-bounded strong consistency.

The key additional idea in the strong optimistic I/O protocol is *version promotion*. When a block is initially read in a new epoch, one candidate version of the block has its version number "promoted" from the previous epoch into the current epoch and is returned by the read call. Promotion selects which version will survive from the previous epoch and ensures that other versions from that earlier epoch will never be read. Through careful management of version numbers, the version promotion protocol ensures that only one version of the block can be promoted. In addition, the promotion protocol performs both the read of a block and its promotion using a single round of messages in the storage tier, so promotion does not add significant overhead.

The strong optimistic I/O protocol is identical to the `CSync`-aware protocol presented in Figure 3, except that the `Read(QUORUM)` at line 14 is replaced by a new DAX operation called `ReadPromote` (Figure 5). Note that `ReadPromote` is used only when the current version number of the block is not found in the version list. This will always be the case for an initial read of a block in a new epoch, since the version list does not survive failure of the DBMS tenant.

To implement `ReadPromote`, we rely on two-part composite version numbers $V = (e, t)$, where e is an epoch number and t is a generated timestamp. We assume that each epoch is associated by the client library with an epoch number, and that these epoch numbers increase monotonically with each failover (i.e., with each failure and restart of the DBMS). We also rely on each client library being able to generate monotonically increasing timestamps, even across failures. That is, all version numbers generated during an epoch must have a timestamp component larger than the timestamps in all version numbers generated during *previous* epochs. We say that $V_2 > V_1$ iff $t_2 > t_1$. (Because of our timestamp monotonicity assumption, $t_2 > t_1$ implies $e_2 \geq e_1$.) Whenever the client library needs to generate a new version number, it uses the current epoch number along with a newly generated t that is larger than all previously generated t's.

`ReadPromote` takes a single parameter, which is a version number $V = (e, t)$ generated in the current interval. It contacts a quorum of servers holding copies of the block to be read and waits for them to return a response (lines 3-4). Each contacted server promotes its replica's version into epoch e (the current epoch) and returns both its copy of the block and its original, unpromoted version number. Promoting a version number means replacing its epoch number

ReadPromote procedure:

```
(1)  proc ReadPromote(V) ≡
(2)    begin
(3)      {(data_1, V_1), (data_2, V_2), ..., (data_q, V_q)}
(4)      := ReadAndPromote(QUORUM, epoch(V));
(6)      if ∃1 ≤ i ≤ q : epoch(V_i) = epoch(V)
(7)        then comment: choose the latest V_i
(8)             return(data_i, V_i)
(9)        else if (V_1 = V_2 = ... = V_q)
(10)              then return(data_1, V_1)
(11)              else
(12)                    data := data_i with largest V_i
(13)                    Write(QUORUM, data, V);
(14)                    return(data, V)
(15)             fi
(16)     fi
(18)   end
```

Figure 5: `ReadPromote` operation.

while leaving its timestamp unchanged.

If at least one server returns a version from the current epoch (lines 6-8), the latest such version is returned to the client. Otherwise, all of the returned versions are from previous epochs. If all of the returned versions are identical (lines 9-10), then that version will be returned to the client. That version will already have been promoted into the current epoch by the quorum of servers that received the `ReadPromote` request. Otherwise, `ReadPromote` must choose one of the returned versions to survive from the previous epoch. This version must be made to exist on a quorum of the replicas. `ReadPromote` ensures this by using a `Write(QUORUM)` operation to write the selected version of the block back to the replica servers, using the new version number V (lines 12-13).

In the (hopefully) common case in which the returned versions from the previous epoch are the same, `ReadPromote` requires only a single message exchange, like `Read(QUORUM)`. In the worst case, which can occur, for example, if a write was in progress at the time of the DBMS failover, `ReadPromote` will require a second round of messages for the `Write(QUORUM)`.

THEOREM 1. *The full optimistic I/O protocol, with* `ReadPromote`*, guarantees epoch-bounded strong consistency, NOC consistency, and read stability.*

Proof: Due to space constraints we only provide a sketch of the proof. First, `ReadPromote` is only used for block b

when there are no unsynchronized updates of b in the current epoch, since a block with unsynchronized updates is guaranteed to remain on the version list and be read using `Read(1)`. If there has been a synchronized update in the epoch and the block is subsequently evicted from the version list, `ReadPromote`'s initial quorum read must see it and will return it thereby maintaining intra-epoch strong consistency. Since timestamps increase monotonically even across epochs, updates from the current epoch always have larger version numbers than updates from previous epochs, even if those earlier updates have been promoted. If `ReadPromote` is being used for an initial read in an epoch, it will return a version from a previous epoch and will have ensured that a quorum of replicas with that version exists, either by discovering such a quorum or creating it with `Write(QUORUM)`. Replicas in this quorum will have version numbers larger than any remaining replicas, so this version will be returned by any subsequent reads until a new version is written in the current epoch. ■

6. DAX PROTOTYPE

To build a prototype of DAX, we decided to adapt an existing multi-master, flexible consistency, key/value storage system rather than building from scratch. Starting with an existing system allowed us to avoid re-implementing common aspects of the functionality of such systems. For example, we re-used mechanisms for detecting failures and for bringing failed servers back on-line, and mechanisms for storing and retrieving values at each server.

Candidate starting points include Cassandra and Voldemort, both of which provide open-source implementations of Dynamo-style quorum-based flexible consistency. We chose Cassandra because it allows clients to choose read and write quorums on a per-request basis, as required by the optimistic I/O approach. Voldemort also supports flexible consistency, but not on a per-request basis. Cassandra also expects a client-supplied timestamp to be associated with each write operation, and it uses these timestamps to impose a global total ordering on each key's writes. This also fits well with DAX's optimistic I/O approach, which requires per-request version numbers. In our Cassandra-based prototype, we use these version numbers as Cassandra timestamps. Cassandra also implements data-center-aware replica placement policies, which we take advantage of. Finally, Cassandra provides various mechanisms (e.g., hinted-handoff and read-repair) for asynchronously updating replicas that may have missed writes. Read-repair, for example, is triggered automatically when Cassandra discovers version mismatches on a read. These mechanisms help to improve the performance of optimistic I/O by increasing the likelihood that `Read(1)` will find an up-to-date replica.

To build the DAX prototype, we had to modify and extend Cassandra in several ways. First, we created the DAX client library, which implements optimistic I/O and serves as the interface between the DBMS tenants and the storage tier, controlling the consistency-level of each read and write operation. Second, we added support for the `CSync` operation, which involved implementing the `sync_pending` list at each Cassandra server. Finally, we added support for the `ReadPromote` operation described in Figure 5. This involved the creation of a new Cassandra message type and the addition of version promotion logic at each server.

7. EXPERIMENTAL EVALUATION

We conducted an empirical evaluation of DAX, with the goals of measuring the performance and effectiveness of optimistic I/O and client-controlled synchronization, verifying the system's scalability, and characterizing the system's behavior under various failure scenarios. We conducted all of our experiments in Amazon's Elastic Compute Cloud (EC2). EC2 is widely used to host data-intensive services of the kind that we are interested in supporting, so it is a natural setting for our experiments. Another reason for using EC2 is that it allowed us to run experiments that use multiple data centers. All of our experiments used EC2 large instances (virtual machines), which have 7.5 GB of memory and a single, locally attached 414 GB storage volume holding an ext3 file system. The virtual machines ran Ubuntu Linux with the 2.6.38-11-virtual kernel. Our DAX prototype is based on Cassandra version 1.0.7. MySQL version 5.5.8 with the InnoDB storage manager was used as the DBMS tenant.

All of the experiments ran a TPC-C transaction processing workload generated by the Percona TPC-C toolkit for MySQL [26]. We used a 100-warehouse TPC-C database instance with an initial size of 10 GB, and the MySQL buffer pool size was set to 5 GB. The database grows during a benchmark run due to record insertions. For each DBMS, 50 concurrent TPC-C clients were used to generate the DBMS workload. The performance metric of interest in our experiments is throughput, measured in TPC-C NewOrder transactions per minute (TpmC). The maximum size of the version list maintained by the client library was set to 512K entries, sufficiently large to hold entries for most of the database blocks.

7.1 Performance of Optimistic I/O

In this experiment, we consider three DAX configurations. In each configuration, there is one EC2 server running a single MySQL tenant, and nine EC2 servers implementing the DAX storage tier. (DAX stores its data on those servers' locally attached storage volumes.) Each database block is replicated six times for a total data size of approximately 60 GB across the nine DAX servers. The three configurations are as follows:

- **1 zone:** All ten servers are located in the same EC2 availability zone (roughly analogous to a data center).

- **3 zones:** The nine DAX servers are distributed evenly over three EC2 availability zones, one of which also houses the MySQL tenant server. All three zones are in the same geographic region (US East). Two replicas of each block are placed in each zone.

- **3 regions:** Like the 3-zone configuration, except that the three zones are located in geographically-distributed EC2 regions: US East, US West-1 and US West-2.

In each EC2 configuration, we tested four versions of DAX, for a total of twelve experiments. The DAX versions we tested are as follows:

- **Baseline 1:** In this configuration, DAX performs I/O operations using `Write(ALL)` and `Read(1)`. This ensures that all database and log updates are replicated 6 times before being acknowledged to the DBMS, and all I/O is strongly consistent, i.e., reads are guaranteed to see the most recently written version of the data. Neither optimistic I/O nor client-controlled synchronization is used.

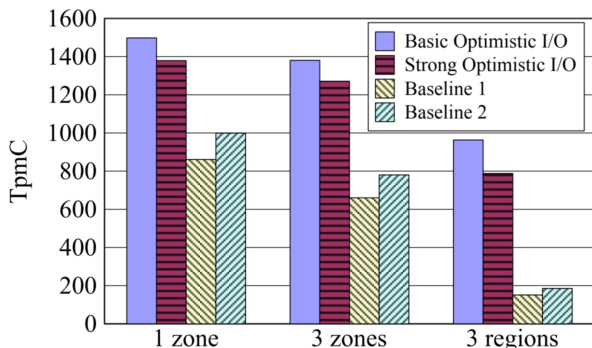

Figure 6: Performance of four DAX variants in three EC2 configurations.

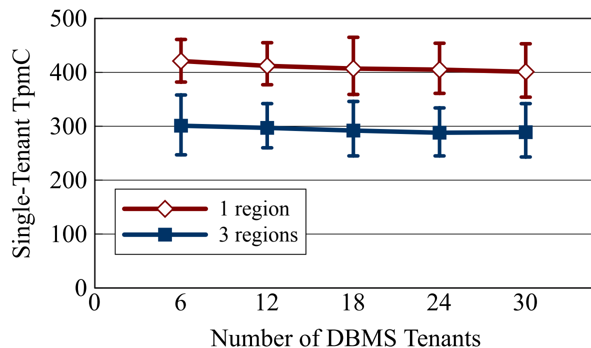

Figure 7: Average, min, and max TpmC per DBMS tenant. The number of DAX servers is 3/2 the number of tenants.

- **Baseline 2:**Like Baseline 1, except that `Write(QUORUM)` and `Read(QUORUM)` are used instead of `Write(ALL)` and `Read(1)`. A quorum consists of N/2+1 replicas, i.e., 4 of 6 replicas. All I/O is strongly consistent, and neither optimistic I/O nor client-controlled synchronization is used.
- **Basic Optimistic I/O:** In this configuration, DAX uses the optimistic I/O technique described in Section 3. This ensures strongly consistent I/O, but only within an epoch. Furthermore, database updates are not guaranteed to be replicated at the time the DBMS write operation completes. The database is, therefore, not safe in the face of failures of DAX servers.
- **Strong Optimistic I/O:** In this configuration, DAX uses the strong optimistic I/O protocol (Section 5), including `CSync` and `ReadPromote`. I/O operations have epoch-bounded strong consistency, NOC consistency, and read stability, and updates are guaranteed to be replicated at the synchronization points chosen by the DBMS.

Results from this experiment are presented in Figure 6. In the two single-region configurations, optimistic I/O provides up to a factor of two improvement in transaction throughput relative to the baseline systems. In the 3-region configuration, the improvement relative to the baseline is approximately a factor of four. Unlike the baselines, optimistic I/O is able to handle most read and write requests using `Read(1)` and `Write(1)`. In contrast, the `Write(ALL)` or `Read(QUORUM)`/`Write(QUORUM)` operations performed by the baseline cases will experience higher latencies. Optimistic I/O is not completely immune to the effects of inter-region latency. The performance of the optimistic I/O variants is lower in the 3-region configuration than in the single-region configurations. However, the drop in throughput is much smaller than for the baselines.

Strong optimistic I/O results in only a small performance penalty relative to basic optimistic I/O, which is not tolerant to failures. Most importantly, the penalty is small even in the 3-region configuration. Thus, strong optimistic I/O is able to guarantee that updates are replicated consistently across three geographically distributed data centers (thereby tolerating failures up to the loss of an entire data center) while achieving only slightly lower throughput than basic optimistic I/O, which makes no such guarantee.

7.2 Scalability

In this experiment, we investigate the tenant scalability of DAX with strong optimistic I/O by varying the number of DBMS tenants from 6 to 30 and increasing the number of DAX servers proportionally, from 9 servers to 45 servers. We use two DAX configurations. In the first, all servers are located in a single availability zone. In the second, the DAX servers and DBMS tenants are evenly distributed in three EC2 regions. Each DBMS tenant manages a separate TPC-C database with 100 warehouses and serves 50 concurrent clients. There are three replicas of each database in the storage tier. Figure 7 shows, for both DAX configurations, the average, minimum, and maximum TpmC per tenant as the number of tenants increases. With the ratio of three DAX storage tier servers to two tenants that we maintained in this experiment, the TPC-C throughput of the tenants is limited by the I/O bandwidth at the DAX servers. We observed nearly linear scale-out up to 30 tenant databases, the largest number we tested. The per-tenant throughput is slightly higher when there are fewer DAX servers. This is because at smaller scales a higher portion of each tenant's database will be located locally at the DAX server to which the tenant connects. As expected, throughput is lower in the 3-region configuration due to the higher latency between DAX servers, but we still observe nearly linear scale-out.

7.3 Fault Tolerance

Next, we present the results of several experiments that illustrate the behavior of DAX under various failure conditions. We present results for DAX with strong optimistic I/O, and for Baseline 2 (`Write(QUORUM)`/`Read(QUORUM)`). Baseline 1 (`Write(ALL)`/`Read(1)`) cannot be used in the presence of failures since `Write(ALL)` requires all replicas to be available. In each case, the experiment begins with the 3-region EC2 configuration: nine DAX servers, three in the US East region (primary), three in the US West-1 region, and three in US West-2 region, with two replicas of each block in each region. A single MySQL tenant runs in the primary region. For each experiment, we measure the tenant's TPC-C throughput (TpmC) as a function of time. Average throughput is reported every 10 seconds.

7.3.1 Failures of DAX Servers

Figure 8 illustrates a scenario in which one DAX server in the primary data center fails at around the 300th second, and then re-joins the storage tier at around the 900th second. After the server fails, it takes the other DAX servers 30 to 90 seconds to detect the failure. During this pe-

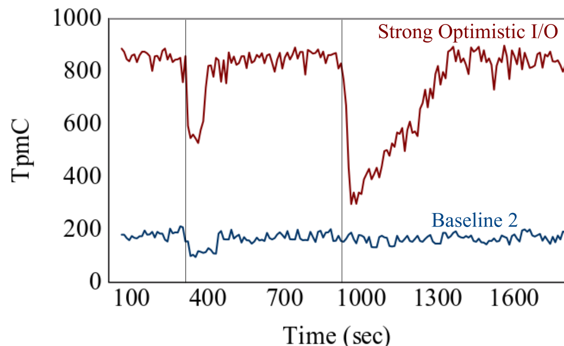

Figure 8: Failure and recovery of a DAX server in the primary data center. Failure occurs at 300 seconds and recovery occurs at 900 seconds.

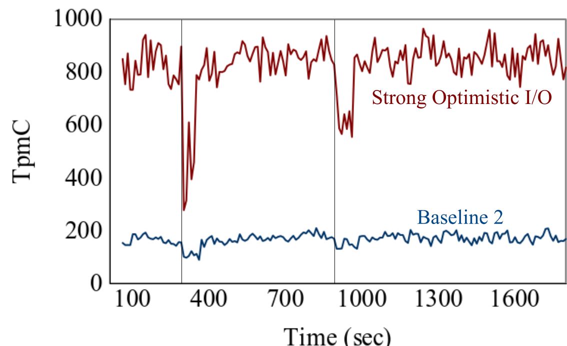

Figure 9: Failure and recovery of a DAX server in a secondary data center. Failure occurs at 300 seconds and recovery occurs at 900 seconds.

riod, read requests that are directed at the failed server will time out and will have to be retried by the tenant. `CSync` requests may also time out. This results in a dip in throughput immediately after the failure for both strong optimistic I/O and Baseline 2. The drop in performance relative to the no-failure performance is about the same for both DAX configurations. Eventually, all DAX servers are made aware of the failure and no longer direct requests to the failed server. The tenant's throughput returns to its original level.

After the failed server re-joins the storage tier, the data stored on that server will be stale. `Read(1)` requests that the strong optimistic I/O protocol directs to the failed server are likely to retrieve stale data and thus need to be retried by the tenant, according to the optimistic I/O protocol. The performance impact of these retries gradually diminishes as the stale data on the re-joined server is brought into synchronization, which Cassandra does automatically. In our test scenario, the system required about 10 minutes after the re-join to return to its full pre-failure performance level. However, the system remains available throughout this recovery period. Furthermore, even when it temporarily dips, optimistic I/O's performance is always better than best performance of Baseline 2.

The (poor) performance of Baseline 2 remains unchanged after the failed server re-joins. That baseline routinely uses relatively heavyweight `Read(QUORUM)` operations, even in the common case of no failure, so it does not experience any additional performance overhead when a failure occurs. Its performance remains low in all cases.

Note that the storage tier has only three servers in the primary region in these experiments, and those servers are heavily utilized. Thus, the loss of one server represents a loss of a third of the storage tier's peak bandwidth at the primary site. In a larger DAX system, or one that is less heavily loaded, we would expect the impact of a single storage server failure to be smaller than what is shown in Figure 8. Specifically, we would expect a shallower performance dip after the server failure for strong optimistic I/O, and a faster performance ramp-up after rejoin.

Figure 9 shows the results of an experiment similar to the previous one, except that the failed DAX server is in one of the secondary data centers. As in the previous scenario, there is a brief dip in performance for both configurations immediately after the failure due to request timeouts. Performance returns to normal once all DAX servers are aware of the failure. This time there is little impact on tenant performance for both configurations when the failed server re-joins the storage tier. Since the re-joined server is remote, `Read(1)` requests issued by the strong optimistic I/O protocol are unlikely to be handled by that server. Instead, they will be handled by servers in the primary region, which are closer to the DBMS tenant. Since the data on those servers is up to date, retries are rarely needed.

In both configurations, the re-joined server will gradually be brought into synchronization in the background by Cassandra, without affecting DBMS performance. As before, the worst performance of strong optimistic I/O is better than the best performance of Baseline 2.

7.3.2 *Loss of a Data Center*

Figure 10 shows the result of an experiment in which the entire primary EC2 data center fails, resulting in the loss of the DBMS tenant and all three DAX servers running there. After the failure, a new MySQL tenant is launched in one of the secondary data centers to take over from the failed primary. The new tenant goes through its log-based transactional recovery procedure using the database and transaction log replicas that it is able to access through the DAX storage servers in the secondary availability zones. Thus, DAX allows the tenant DBMS to restore its availability even after the entire primary data center is lost. No special DBMS-level mechanisms are required to support this.

Figure 10 shows throughput for strong optimistic I/O and Baseline 2. For both configurations, there is an unavailability window of about 270 seconds while the new DBMS instance goes through its recovery procedure. All of the downtime is due to the DBMS recovery procedure. Performance then ramps up gradually to its pre-failure level. The gradual ramp-up under strong optimistic I/O is due to a combination of several factors: the cold start of the new DBMS tenant, the fact that DAX's workload is more read-heavy during restart, and the fact that these reads are more expensive than normal for DAX. The initial read of each database or log block requires `ReadPromote` since the block's identifier will not yet be present in the version list. Baseline 2 incurs the penalty for cold start but not for `ReadPromote`. However, as in previous experiments, the performance of Baseline 2 is always lower than strong optimistic I/O.

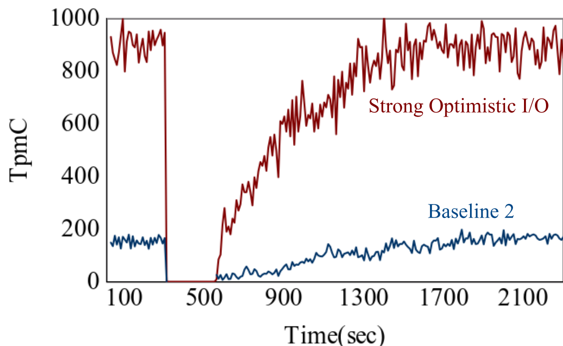

Figure 10: Failure of the primary data center and recovery of MySQL.

8. RELATED WORK

Cloud Storage Systems: There are now many cloud storage systems, all of which sacrifice functionality or consistency (as compared to relational database systems) to achieve scalability and fault tolerance. Different systems make different design choices in the way they manage updates, leading to different scalability, availability, and consistency characteristics. Google's BigTable [7] and its open source cousin, HBase, use master-slave replication and provide strong consistency for reads and writes of single rows. In contrast, systems such as Dynamo [16] and Cassandra [20] use multi-master replication as described in Section 2. PNUTS [9] provides per-record timeline consistency: all replicas of a given record apply all updates to the record in the same order. Spinnaker [27] enables client applications to choose between strong consistency and eventual consistency for every read. The system only runs in one data center, and it uses heavyweight Paxos-based replication, leading to high read and write latencies. All of the systems mentioned thus far provide consistency guarantees at the granularity of one row. Other cloud storage systems offer multi-row consistency guarantees [3, 13, 24, 25, 29], but such multi-row guarantees are not required by DAX.

The importance of consistent transactions on storage systems distributed across data centers is highlighted by recent work in this area. Patterson et al. [24] present a protocol that provides consistent multi-row transactions across multiple data centers, even in the presence of multiple writers for every row. The protocol is fairly heavyweight, and it assumes that the storage system provides strong consistency for reads and writes of individual rows. In contrast, DAX uses a lightweight protocol that takes advantage of the properties of DBMS transactions and the ability of flexibility consistency systems to hide latency across data centers. DAX's latency hiding relies on the assumption that reads in eventual consistency storage systems usually return fresh data. This assumption has been verified and quantified in [2].

Database Systems on Cloud Storage: DAX is not the first cloud storage tier for relational DBMS. ElasTras [12] is perhaps closest to the work presented here. Like DAX, ElasTras is intended to host database systems on a shared storage tier. However, ElasTras assumes that the storage tier provides strongly consistent read and write operations, so it is not concerned with questions of managing request consistency, which is the focus of this paper. Instead, ElasTras is concerned with issues that are orthogonal to this paper, such as allowing co-located hosted database systems to share resources. Commercial services, such as Amazon's EBS, are also used to provide storage services for DBMS tenants. EBS provides strongly consistent reads and writes, but we are not aware of any public information about its implementation. Unlike DAX, these storage services are not intended to support geographically distributed replication.

Brantner et al. [6] considered the question of how to host database systems on Amazon's Simple Storage Service (S3), which provides only weak eventual consistency guarantees. As in our work, they use the underlying storage tier (S3) as a block storage device. However, their task is more challenging because of S3's weak eventual consistency guarantees, and because they allow multiple hosted database systems to access a single stored database. As a result, it is difficult to offer transactional guarantees to the applications built on top of the hosted database systems. In contrast, our work takes advantage of a flexible consistency model to offer transactional guarantees to the hosted DBMS applications without sacrificing performance.

Recent work [23] has looked at hosting relational database services on a storage tier that provides tuple-level access as well as indexing, concurrency control and buffering. The storage tier in that work is based on Sinfonia [1]. Since DAX relies on a storage tier that provides a block-level interface, it resides in a very different part of the design spectrum.

Other Approaches to Database Scalability: DAX scales out to accommodate additional database tenants and additional aggregate demand for storage capacity and bandwidth. However, by itself it does not enable scale-out of individual DBMS tenants. There are other approaches to database scalability that do enable scale-out of individual database systems. One option is to use DBMS-managed (or middleware-managed) database replication. Kemme et al. [19] provide a recent overview of the substantial body of work on this topic. In such systems, updates are typically performed at all replicas, while reads can be performed at any replica. In DAX, each update is executed at only one DBMS, but the effect of the update is replicated in the storage tier. In some replicated DBMS, a master DBMS or middleware server is responsible for determining the order in which update operations will occur at all replicas. Such systems are closest to DAX, in which each hosted DBMS tenant determines the ordering of its write operations and the storage tier adheres to the DBMS-prescribed order.

Database partitioning or sharding [15], which is widely used in practice, is another approach to DBMS scale-out. HStore [18] and its commercial spin-off, VoltDB, store all the data of a database in main memory, partitioned among many servers and replicated for durability. MemcacheSQL [8] is another system that uses main memory, integrating buffer pool management in a relational DBMS with the Memcached distributed caching platform to enable one DBMS instance to use buffer pools that are distributed among many servers. Hyder [5] promises to scale a transactional database across multiple servers by exploiting a fast flash-based shared log. Some of these approaches may be used in conjunction with DAX if there is a need to scale a DBMS tenant beyond one server.

Spanner [11], is a recent multi-tenant database service that supports (a subset of) SQL and runs on a storage system distributed across multiple data centers. As such, Spanner has the same high level goal as DAX. Spanner is a com-

pletely new system built from the ground up, with a new replicated storage layer, transaction manager, and query processor. It is a highly engineered system that supports a rich set of features, some of which (e.g., multiple writers for the same database) are not supported by DAX. However, Spanner relies on a service called TrueTime that minimizes clock skew among data centers and reports the maximum possible clock skew. The implementation of TrueTime is distributed among the data centers in which Spanner runs, and it relies on special hardware. Further, Spanner uses the standard, heavyweight Paxos protocol (with some optimizations) for replication among data centers, which makes its latency high [28]. In contrast, DAX provides a simpler service and focuses on minimizing latency between data centers while maintaining consistency. It can support existing relational database systems such as MySQL and it does not rely on any special hardware.

9. CONCLUSIONS

We have presented DAX, a distributed system intended to support the storage requirements of multiple DBMS tenants. DAX is scalable in the number of tenants it supports, and offers tenants scalable storage capacity and bandwidth. In addition, DAX tolerates failures up to the loss of an entire data center. To provide these benefits, DAX relies on a flexible consistency key/value storage system, and it judiciously manages the consistency level of different read and write operations to enable low-latency, geographically distributed replication while satisfying the consistency requirements of the DBMS tenants. Our experiments with a prototype of DAX based on Cassandra show that it provides up to a factor of 4 performance improvement over baseline solutions.

10. ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by an Amazon Web Services Research Grant.

11. REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, 2007.
[2] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 2012.
[3] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.
[4] P. A. Bernstein et al. Adapting Microsoft SQL Server for cloud computing. In *ICDE*, 2011.
[5] P. A. Bernstein, C. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, 2011.
[6] M. Brantner, D. Florescuy, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, 2008.
[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
[8] Q. Chen, M. Hsu, and R. Wu. MemcacheSQL - a scale-out SQL cache engine. In *BIRTE*, 2011.
[9] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *PVLDB*, 2008.
[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.
[11] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.
[12] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: An elastic transactional data store in the cloud. In *HotCloud*, 2009.
[13] S. Das, D. Agrawal, and A. El Abbadi. G-Store: A scalable data store for transactional multi key access in the cloud. In *SoCC*, 2010.
[14] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 2011.
[15] Database sharding whitepaper. http://www.dbshards.com/articles/database-sharding-whitepapers/.
[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, and A. Lakshman. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.
[17] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.
[18] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.
[19] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.
[20] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. In *LADIS*, 2009.
[21] LinkedIn Data Infrastructure Team. Data infrastructure at LinkedIn. In *ICDE*, 2012.
[22] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson. Elastic scale-out for partition-based database systems. In *SMDB*, 2012.
[23] M. T. Najaran, P. Wijesekera, A. Warfield, and N. B. Hutchinson. Distributed indexing and locking: In search of scalable consistency. In *LADIS*, 2011.
[24] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 2012.
[25] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.
[26] Percona TPC-C toolkit for MySQL. https://code.launchpad.net/ percona-dev/perconatools/tpcc-mysql.
[27] J. Rao, E. J. Shekita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. In *PVLDB*, 2011.
[28] J. Shute et al. F1: the fault-tolerant distributed RDBMS supporting Google's ad business. In *SIGMOD*, 2012.
[29] Y. Sovran et al. Transactional storage for geo-replicated systems. In *SOSP*, 2011.