Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.

{jasonbaker,chrisbond,jcorbett,jfurman,akhorlin,jimlarson,jm,yaweili,alloyd,vadimy}@google.com

ABSTRACT

Megastore is a storage system developed to meet the requirements of today's interactive online services. Megastore blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS in a novel way, and provides both strong consistency guarantees and high availability. We provide fully serializable ACID semantics within fine-grained partitions of data. This partitioning allows us to synchronously replicate each write across a wide area network with reasonable latency and support seamless failover between datacenters. This paper describes Megastore's semantics and replication algorithm. It also describes our experience supporting a wide range of Google production services built with Megastore.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases; H.2.4 [**Database Management**]: Systems—*concurrency, distributed databases*

General Terms

Algorithms, Design, Performance, Reliability

Keywords

Large databases, Distributed transactions, Bigtable, Paxos

1. INTRODUCTION

Interactive online services are forcing the storage community to meet new demands as desktop applications migrate to the cloud. Services like email, collaborative documents, and social networking have been growing exponentially and are testing the limits of existing infrastructure. Meeting these services' storage demands is challenging due to a number of conflicting requirements.

First, the Internet brings a huge audience of potential users, so the applications must be *highly scalable*. A service can be built rapidly using MySQL [10] as its datastore, but scaling the service to millions of users requires a complete redesign of its storage infrastructure. Second, services must compete for users. This requires *rapid development* of features and fast time-to-market. Third, the service must be responsive; hence, the storage system must have *low latency*. Fourth, the service should provide the user with a *consistent view of the data*—the result of an update should be visible immediately and durably. Seeing edits to a cloud-hosted spreadsheet vanish, however briefly, is a poor user experience. Finally, users have come to expect Internet services to be up 24/7, so the service must be *highly available*. The service must be resilient to many kinds of faults ranging from the failure of individual disks, machines, or routers all the way up to large-scale outages affecting entire datacenters.

These requirements are in conflict. Relational databases provide a rich set of features for easily building applications, but they are difficult to scale to hundreds of millions of users. NoSQL datastores such as Google's Bigtable [15], Apache Hadoop's HBase [1], or Facebook's Cassandra [6] are highly scalable, but their limited API and loose consistency models complicate application development. Replicating data across distant datacenters while providing low latency is challenging, as is guaranteeing a consistent view of replicated data, especially during faults.

Megastore is a storage system developed to meet the storage requirements of today's interactive online services. It is novel in that it blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS. It uses synchronous replication to achieve high availability and a consistent view of the data. In brief, it provides fully serializable ACID semantics over distant replicas with low enough latencies to support interactive applications.

We accomplish this by taking a middle ground in the RDBMS vs. NoSQL design space: we partition the datastore and replicate each partition separately, providing full ACID semantics within partitions, but only limited consistency guarantees across them. We provide traditional database features, such as secondary indexes, but only those features that can scale within user-tolerable latency limits, and only with the semantics that our partitioning scheme can support. We contend that the data for most Internet services can be suitably partitioned (e.g., by user) to make this approach viable, and that a small, but not spartan, set of features can substantially ease the burden of developing cloud applications.

Contrary to conventional wisdom [24, 28], we were able to use Paxos [27] to build a highly available system that pro-

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

vides reasonable latencies for interactive applications while synchronously replicating writes across geographically distributed datacenters. While many systems use Paxos solely for locking, master election, or replication of metadata and configurations, we believe that Megastore is the largest system deployed that uses Paxos to replicate primary user data across datacenters on every write.

Megastore has been widely deployed within Google for several years [20]. It handles more than three billion write and 20 billion read transactions daily and stores nearly a petabyte of primary data across many global datacenters.

The key contributions of this paper are:

1. the design of a data model and storage system that allows rapid development of interactive applications where high availability and scalability are built-in from the start;
2. an implementation of the Paxos replication and consensus algorithm optimized for low-latency operation across geographically distributed datacenters to provide high availability for the system;
3. a report on our experience with a large-scale deployment of Megastore at Google.

The paper is organized as follows. Section 2 describes how Megastore provides availability and scalability using partitioning and also justifies the sufficiency of our design for many interactive Internet applications. Section 3 provides an overview of Megastore's data model and features. Section 4 explains the replication algorithms in detail and gives some measurements on how they perform in practice. Section 5 summarizes our experience developing the system. We review related work in Section 6. Section 7 concludes.

2. TOWARD AVAILABILITY AND SCALE

In contrast to our need for a storage platform that is global, reliable, and arbitrarily large in scale, our hardware building blocks are geographically confined, failure-prone, and suffer limited capacity. We must bind these components into a unified ensemble offering greater throughput and reliability.

To do so, we have taken a two-pronged approach:

- for availability, we implemented a synchronous, fault-tolerant log replicator optimized for long distance-links;
- for scale, we partitioned data into a vast space of small databases, each with its own replicated log stored in a per-replica NoSQL datastore.

2.1 Replication

Replicating data across hosts within a single datacenter improves availability by overcoming host-specific failures, but with diminishing returns. We still must confront the networks that connect them to the outside world and the infrastructure that powers, cools, and houses them. Economically constructed sites risk some level of facility-wide outages [25] and are vulnerable to regional disasters. For cloud storage to meet availability demands, service providers must replicate data over a wide geographic area.

2.1.1 Strategies

We evaluated common strategies for wide-area replication:

Asynchronous Master/Slave A master node replicates write-ahead log entries to at least one slave. Log appends are acknowledged at the master in parallel with transmission to slaves. The master can support fast ACID transactions but risks downtime or data loss during failover to a slave. A consensus protocol is required to mediate mastership.

Synchronous Master/Slave A master waits for changes to be mirrored to slaves before acknowledging them, allowing failover without data loss. Master and slave failures need timely detection by an external system.

Optimistic Replication Any member of a homogeneous replica group can accept mutations [23], which are asynchronously propagated through the group. Availability and latency are excellent. However, the global mutation ordering is not known at commit time, so transactions are impossible.

We avoided strategies which could lose data on failures, which are common in large-scale systems. We also discarded strategies that do not permit ACID transactions. Despite the operational advantages of eventually consistent systems, it is currently too difficult to give up the read-modify-write idiom in rapid application development.

We also discarded options with a heavyweight master. Failover requires a series of high-latency stages often causing a user-visible outage, and there is still a huge amount of complexity. Why build a fault-tolerant system to arbitrate mastership and failover workflows if we could avoid distinguished masters altogether?

2.1.2 Enter Paxos

We decided to use Paxos, a proven, optimal, fault-tolerant consensus algorithm with no requirement for a distinguished master [14, 27]. We replicate a write-ahead log over a group of symmetric peers. Any node can initiate reads and writes. Each log append blocks on acknowledgments from a majority of replicas, and replicas in the minority catch up as they are able—the algorithm's inherent fault tolerance eliminates the need for a distinguished "failed" state. A novel extension to Paxos, detailed in Section 4.4.1, allows local reads at any up-to-date replica. Another extension permits single-roundtrip writes.

Even with fault tolerance from Paxos, there are limitations to using a single log. With replicas spread over a wide area, communication latencies limit overall throughput. Moreover, progress is impeded when no replica is current or a majority fail to acknowledge writes. In a traditional SQL database hosting thousands or millions of users, using a synchronously replicated log would risk interruptions of widespread impact [11]. So to improve availability and throughput we use multiple replicated logs, each governing its own partition of the data set.

2.2 Partitioning and Locality

To scale our replication scheme and maximize performance of the underlying datastore, we give applications fine-grained control over their data's partitioning and locality.

2.2.1 Entity Groups

To scale throughput and localize outages, we partition our data into a collection of *entity groups* [24], each independently and synchronously replicated over a wide area. The underlying data is stored in a scalable NoSQL datastore in each datacenter (see Figure 1).

Entities within an entity group are mutated with single-phase ACID transactions (for which the commit record is

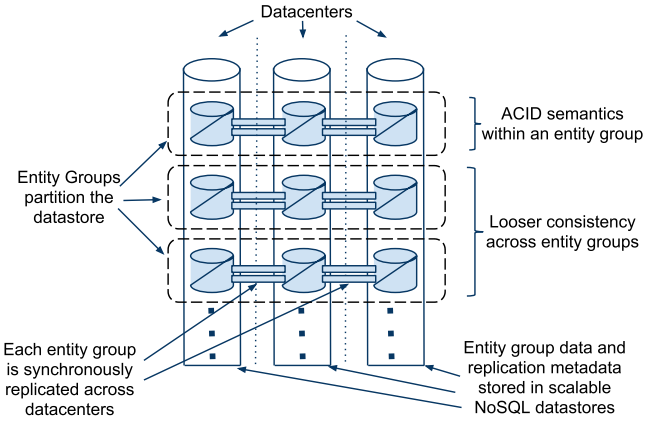

Figure 1: Scalable Replication

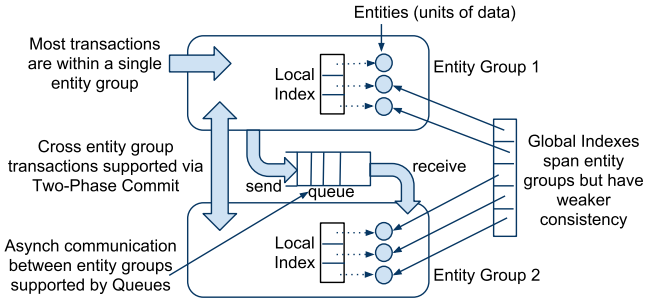

Figure 2: Operations Across Entity Groups

replicated via Paxos). Operations across entity groups could rely on expensive two-phase commits, but typically leverage Megastore's efficient asynchronous messaging. A transaction in a sending entity group places one or more messages in a queue; transactions in receiving entity groups atomically consume those messages and apply ensuing mutations.

Note that we use asynchronous messaging between logically distant entity groups, not physically distant replicas. All network traffic between datacenters is from replicated operations, which are synchronous and consistent.

Indexes local to an entity group obey ACID semantics; those across entity groups have looser consistency. See Figure 2 for the various operations on and between entity groups.

2.2.2 *Selecting Entity Group Boundaries*

The entity group defines the *a priori* grouping of data for fast operations. Boundaries that are too fine-grained force excessive cross-group operations, but placing too much unrelated data in a single group serializes unrelated writes, which degrades throughput.

The following examples show ways applications can work within these constraints:

Email Each email account forms a natural entity group. Operations within an account are transactional and consistent: a user who sends or labels a message is guaranteed to observe the change despite possible failover to another replica. External mail routers handle communication between accounts.

Blogs A blogging application would be modeled with multiple classes of entity groups. Each user has a profile, which is naturally its own entity group. However, blogs are collaborative and have no single permanent owner. We create a second class of entity groups to hold the posts and metadata for each blog. A third class keys off the unique name claimed by each blog. The application relies on asynchronous messaging when a single user operation affects both blogs and profiles. For a lower-traffic operation like creating a new blog and claiming its unique name, two-phase commit is more convenient and performs adequately.

Maps Geographic data has no natural granularity of any consistent or convenient size. A mapping application can create entity groups by dividing the globe into non-overlapping patches. For mutations that span patches, the application uses two-phase commit to make them atomic. Patches must be large enough that two-phase transactions are uncommon, but small enough that each patch requires only a small write throughput. Unlike the previous examples, the number of entity groups does not grow with increased usage, so enough patches must be created initially for sufficient aggregate throughput at later scale.

Nearly all applications built on Megastore have found natural ways to draw entity group boundaries.

2.2.3 *Physical Layout*

We use Google's Bigtable [15] for scalable fault-tolerant storage within a single datacenter, allowing us to support arbitrary read and write throughput by spreading operations across multiple rows.

We minimize latency and maximize throughput by letting applications control the placement of data: through the selection of Bigtable instances and specification of locality within an instance.

To minimize latency, applications try to keep data near users and replicas near each other. They assign each entity group to the region or continent from which it is accessed most. Within that region they assign a triplet or quintuplet of replicas to datacenters with isolated failure domains.

For low latency, cache efficiency, and throughput, the data for an entity group are held in contiguous ranges of Bigtable rows. Our schema language lets applications control the placement of hierarchical data, storing data that is accessed together in nearby rows or denormalized into the same row.

3. A TOUR OF MEGASTORE

Megastore maps this architecture onto a feature set carefully chosen to encourage rapid development of scalable applications. This section motivates the tradeoffs and describes the developer-facing features that result.

3.1 API Design Philosophy

ACID transactions simplify reasoning about correctness, but it is equally important to be able to reason about performance. Megastore emphasizes *cost-transparent* APIs with runtime costs that match application developers' intuitions.

Normalized relational schemas rely on joins at query time to service user operations. This is not the right model for Megastore applications for several reasons:

- High-volume interactive workloads benefit more from predictable performance than from an expressive query language.

- Reads dominate writes in our target applications, so it pays to move work from read time to write time.
- Storing and querying hierarchical data is straightforward in key-value stores like Bigtable.

With this in mind, we designed a data model and schema language to offer fine-grained control over physical locality. Hierarchical layouts and declarative denormalization help eliminate the need for most joins. Queries specify scans or lookups against particular tables and indexes.

Joins, when required, are implemented in application code. We provide an implementation of the merge phase of the merge join algorithm, in which the user provides multiple queries that return primary keys for the same table in the same order; we then return the intersection of keys for all the provided queries.

We also have applications that implement outer joins with parallel queries. This typically involves an index lookup followed by parallel index lookups using the results of the initial lookup. We have found that when the secondary index lookups are done in parallel and the number of results from the first lookup is reasonably small, this provides an effective stand-in for SQL-style joins.

While schema changes require corresponding modifications to the query implementation code, this system guarantees that features are built with a clear understanding of their performance implications. For example, when users (who may not have a background in databases) find themselves writing something that resembles a nested-loop join algorithm, they quickly realize that it's better to add an index and follow the index-based join approach above.

3.2 Data Model

Megastore defines a data model that lies between the abstract tuples of an RDBMS and the concrete row-column storage of NoSQL. As in an RDBMS, the data model is declared in a *schema* and is strongly typed. Each schema has a set of *tables*, each containing a set of *entities*, which in turn contain a set of *properties*. Properties are named and typed values. The types can be strings, various flavors of numbers, or Google's Protocol Buffers [9]. They can be required, optional, or repeated (allowing a list of values in a single property). All entities in a table have the same set of allowable properties. A sequence of properties is used to form the primary key of the entity, and the primary keys must be unique within the table. Figure 3 shows an example schema for a simple photo storage application.

Megastore tables are either *entity group root* tables or *child* tables. Each child table must declare a single distinguished foreign key referencing a root table, illustrated by the `ENTITY GROUP KEY` annotation in Figure 3. Thus each child entity references a particular entity in its root table (called the *root entity*). An entity group consists of a root entity along with all entities in child tables that reference it. A Megastore instance can have several root tables, resulting in different classes of entity groups.

In the example schema of Figure 3, each user's photo collection is a separate entity group. The root entity is the `User`, and the `Photo`s are child entities. Note the `Photo.tag` field is repeated, allowing multiple tags per `Photo` without the need for a sub-table.

3.2.1 Pre-Joining with Keys

While traditional relational modeling recommends that

```
CREATE SCHEMA PhotoApp;

CREATE TABLE User {
 required int64 user_id;
 required string name;
} PRIMARY KEY(user_id), ENTITY GROUP ROOT;

CREATE TABLE Photo {
 required int64 user_id;
 required int32 photo_id;
 required int64 time;
 required string full_url;
 optional string thumbnail_url;
 repeated string tag;
} PRIMARY KEY(user_id, photo_id),
  IN TABLE User,
  ENTITY GROUP KEY(user_id) REFERENCES User;

CREATE LOCAL INDEX PhotosByTime
  ON Photo(user_id, time);

CREATE GLOBAL INDEX PhotosByTag
  ON Photo(tag) STORING (thumbnail_url);
```

Figure 3: Sample Schema for Photo Sharing Service

all primary keys take surrogate values, Megastore keys are chosen to cluster entities that will be read together. Each entity is mapped into a single Bigtable row; the primary key values are concatenated to form the Bigtable row key, and each remaining property occupies its own Bigtable column.

Note how the `Photo` and `User` tables in Figure 3 share a common `user_id` key prefix. The `IN TABLE User` directive instructs Megastore to colocate these two tables into the same Bigtable, and the key ordering ensures that `Photo` entities are stored adjacent to the corresponding `User`. This mechanism can be applied recursively to speed queries along arbitrary join depths. Thus, users can force hierarchical layout by manipulating the key order.

Schemas declare keys to be sorted ascending or descending, or to avert sorting altogether: the `SCATTER` attribute instructs Megastore to prepend a two-byte hash to each key. Encoding monotonically increasing keys this way prevents hotspots in large data sets that span Bigtable servers.

3.2.2 Indexes

Secondary indexes can be declared on any list of entity properties, as well as fields within protocol buffers. We distinguish between two high-level classes of indexes: *local* and *global* (see Figure 2). A local index is treated as separate indexes for each entity group. It is used to find data within an entity group. In Figure 3, `PhotosByTime` is an example of a local index. The index entries are stored in the entity group and are updated atomically and consistently with the primary entity data.

A global index spans entity groups. It is used to find entities without knowing in advance the entity groups that contain them. The `PhotosByTag` index in Figure 3 is global and enables discovery of photos marked with a given tag, regardless of owner. Global index scans can read data owned by many entity groups but are not guaranteed to reflect all recent updates.

Megastore offers additional indexing features:

3.2.2.1 Storing Clause.

Accessing entity data through indexes is normally a two-step process: first the index is read to find matching primary keys, then these keys are used to fetch entities. We provide a way to denormalize portions of entity data directly into index entries. By adding the `STORING` clause to an index, applications can store additional properties from the primary table for faster access at read time. For example, the `PhotosByTag` index stores the photo thumbnail URL for faster retrieval without the need for an additional lookup.

3.2.2.2 Repeated Indexes.

Megastore provides the ability to index repeated properties and protocol buffer sub-fields. Repeated indexes are a efficient alternative to child tables. `PhotosByTag` is a repeated index: each unique entry in the `tag` property causes one index entry to be created on behalf of the `Photo`.

3.2.2.3 Inline Indexes.

Inline indexes provide a way to denormalize data from source entities into a related target entity: index entries from the source entities appear as a virtual repeated column in the target entry. An inline index can be created on any table that has a foreign key referencing another table by using the first primary key of the target entity as the first components of the index, and physically locating the data in the same Bigtable as the target.

Inline indexes are useful for extracting slices of information from child entities and storing the data in the parent for fast access. Coupled with repeated indexes, they can also be used to implement many-to-many relationships more efficiently than by maintaining a many-to-many link table.

The `PhotosByTime` index could have been implemented as an inline index into the parent `User` table. This would make the data accessible as a normal index or as a virtual repeated property on `User`, with a time-ordered entry for each contained `Photo`.

3.2.3 Mapping to Bigtable

The Bigtable column name is a concatenation of the Megastore table name and the property name, allowing entities from different Megastore tables to be mapped into the same Bigtable row without collision. Figure 4 shows how data from the example photo application might look in Bigtable.

Within the Bigtable row for a root entity, we store the transaction and replication metadata for the entity group, including the transaction log. Storing all metadata in a single Bigtable row allows us to update it atomically through a single Bigtable transaction.

Each index entry is represented as a single Bigtable row; the row key of the cell is constructed using the indexed property values concatenated with the primary key of the indexed entity. For example, the `PhotosByTime` index row keys would be the tuple (*user_id*, *time*, *primary key*) for each photo. Indexing repeated fields produces one index entry per repeated element. For example, the primary key for a photo with three tags would appear in the `PhotosByTag` index thrice.

3.3 Transactions and Concurrency Control

Each Megastore entity group functions as a mini-database

Row key	User. name	Photo. time	Photo. tag	Photo. _url
101	John			
101,500		12:30:01	Dinner, Paris	...
101,502		12:15:22	Betty, Paris	...
102	Mary			

Figure 4: Sample Data Layout in Bigtable

that provides serializable ACID semantics. A transaction writes its mutations into the entity group's write-ahead log, then the mutations are applied to the data.

Bigtable provides the ability to store multiple values in the same row/column pair with different timestamps. We use this feature to implement multiversion concurrency control (MVCC): when mutations within a transaction are applied, the values are written at the timestamp of their transaction. Readers use the timestamp of the last fully applied transaction to avoid seeing partial updates. Readers and writers don't block each other, and reads are isolated from writes for the duration of a transaction.

Megastore provides *current*, *snapshot*, and *inconsistent* reads. Current and snapshot reads are always done within the scope of a single entity group. When starting a current read, the transaction system first ensures that all previously committed writes are applied; then the application reads at the timestamp of the latest committed transaction. For a snapshot read, the system picks up the timestamp of the last known fully applied transaction and reads from there, even if some committed transactions have not yet been applied. Megastore also provides inconsistent reads, which ignore the state of the log and read the latest values directly. This is useful for operations that have more aggressive latency requirements and can tolerate stale or partially applied data.

A write transaction always begins with a current read to determine the next available log position. The commit operation gathers mutations into a log entry, assigns it a timestamp higher than any previous one, and appends it to the log using Paxos. The protocol uses optimistic concurrency: though multiple writers might be attempting to write to the same log position, only one will win. The rest will notice the victorious write, abort, and retry their operations. Advisory locking is available to reduce the effects of contention. Batching writes through session affinity to a particular front-end server can avoid contention altogether.

The complete transaction lifecycle is as follows:

1. **Read**: Obtain the timestamp and log position of the last committed transaction.
2. **Application logic**: Read from Bigtable and gather writes into a log entry.
3. **Commit**: Use Paxos to achieve consensus for appending that entry to the log.
4. **Apply**: Write mutations to the entities and indexes in Bigtable.
5. **Clean up**: Delete data that is no longer required.

The write operation can return to the client at any point after Commit, though it makes a best-effort attempt to wait for the nearest replica to apply.

3.3.1 Queues

Queues provide transactional messaging between entity groups. They can be used for cross-group operations, to

batch multiple updates into a single transaction, or to defer work. A transaction on an entity group can atomically send or receive multiple messages in addition to updating its entities. Each message has a single sending and receiving entity group; if they differ, delivery is asynchronous. (See Figure 2.)

Queues offer a way to perform operations that affect many entity groups. For example, consider a calendar application in which each calendar has a distinct entity group, and we want to send an invitation to a group of calendars. A single transaction can atomically send invitation queue messages to many distinct calendars. Each calendar receiving the message will process the invitation in its own transaction which updates the invitee's state and deletes the message.

There is a long history of message queues in full-featured RDBMSs. Our support is notable for its scale: declaring a queue automatically creates an inbox on each entity group, giving us millions of endpoints.

3.3.2 Two-Phase Commit

Megastore supports two-phase commit for atomic updates across entity groups. Since these transactions have much higher latency and increase the risk of contention, we generally discourage applications from using the feature in favor of queues. Nevertheless, they can be useful in simplifying application code for unique secondary key enforcement.

3.4 Other Features

We have built a tight integration with Bigtable's full-text index in which updates and searches participate in Megastore's transactions and multiversion concurrency. A full-text index declared in a Megastore schema can index a table's text or other application-generated attributes.

Synchronous replication is sufficient defense against the most common corruptions and accidents, but backups can be invaluable in cases of programmer or operator error. Megastore's integrated backup system supports periodic full snapshots as well as incremental backup of transaction logs. The restore process can bring back an entity group's state to any point in time, optionally omitting selected log entries (as after accidental deletes). The backup system complies with legal and common sense principles for expiring deleted data.

Applications have the option of encrypting data at rest, including the transaction logs. Encryption uses a distinct key per entity group. We avoid granting the same operators access to both the encryption keys and the encrypted data.

4. REPLICATION

This section details the heart of our synchronous replication scheme: a low-latency implementation of Paxos. We discuss operational details and present some measurements of our production service.

4.1 Overview

Megastore's replication system provides a single, consistent view of the data stored in its underlying replicas. Reads and writes can be initiated from any replica, and ACID semantics are preserved regardless of what replica a client starts from. Replication is done per entity group by synchronously replicating the group's transaction log to a quorum of replicas. Writes typically require one round of inter-datacenter communication, and healthy-case reads run locally. Current reads have the following guarantees:

- A read always observes the last-acknowledged write.
- After a write has been observed, all future reads observe that write. (A write might be observed before it is acknowledged.)

4.2 Brief Summary of Paxos

The Paxos algorithm is a way to reach consensus among a group of replicas on a single value. It tolerates delayed or reordered messages and replicas that fail by stopping. A majority of replicas must be active and reachable for the algorithm to make progress—that is, it allows up to F faults with $2F + 1$ replicas. Once a value is *chosen* by a majority, all future attempts to read or write the value will reach the same outcome.

The ability to determine the outcome of a single value by itself is not of much use to a database. Databases typically use Paxos to replicate a transaction log, where a separate instance of Paxos is used for each position in the log. New values are written to the log at the position following the last chosen position.

The original Paxos algorithm [27] is ill-suited for high-latency network links because it demands multiple rounds of communication. Writes require at least two inter-replica roundtrips before consensus is achieved: a round of *prepares*, which reserves the right for a subsequent round of *accepts*. Reads require at least one round of prepares to determine the last chosen value. Real world systems built on Paxos reduce the number of roundtrips required to make it a practical algorithm. We will first review how master-based systems use Paxos, and then explain how we make Paxos efficient.

4.3 Master-Based Approaches

To minimize latency, many systems use a dedicated master to which all reads and writes are directed. The master participates in all writes, so its state is always up-to-date. It can serve reads of the current consensus state without any network communication. Writes are reduced to a single round of communication by piggybacking a prepare for the next write on each accept [14]. The master can batch writes together to improve throughput.

Reliance on a master limits flexibility for reading and writing. Transaction processing must be done near the master replica to avoid accumulating latency from sequential reads. Any potential master replica must have adequate resources for the system's full workload; slave replicas waste resources until the moment they become master. Master failover can require a complicated state machine, and a series of timers must elapse before service is restored. It is difficult to avoid user-visible outages.

4.4 Megastore's Approach

In this section we discuss the optimizations and innovations that make Paxos practical for our system.

4.4.1 Fast Reads

We set an early requirement that current reads should usually execute on any replica without inter-replica RPCs. Since writes usually succeed on all replicas, it was realistic to allow local reads everywhere. These *local reads* give us better utilization, low latencies in all regions, fine-grained read failover, and a simpler programming experience.

We designed a service called the *Coordinator*, with servers in each replica's datacenter. A coordinator server tracks a set of entity groups for which its replica has observed all Paxos writes. For entity groups in that set, the replica has sufficient state to serve local reads.

It is the responsibility of the write algorithm to keep coordinator state conservative. If a write fails on a replica's Bigtable, it cannot be considered committed until the group's key has been evicted from that replica's coordinator.

Since coordinators are simple, they respond more reliably and quickly than Bigtable. Handling of rare failure cases or network partitions is described in Section 4.7.

4.4.2 Fast Writes

To achieve fast single-roundtrip writes, Megastore adapts the pre-preparing optimization used by master-based approaches. In a master-based system, each successful write includes an implied prepare message granting the master the right to issue accept messages for the next log position. If the write succeeds, the prepares are honored, and the next write skips directly to the accept phase. Megastore does not use dedicated masters, but instead uses *leaders*.

We run an independent instance of the Paxos algorithm for each log position. The leader for each log position is a distinguished replica chosen alongside the preceding log position's consensus value. The leader arbitrates which value may use proposal number zero. The first writer to submit a value to the leader wins the right to ask all replicas to accept that value as proposal number zero. All other writers must fall back on two-phase Paxos.

Since a writer must communicate with the leader before submitting the value to other replicas, we minimize writer-leader latency. We designed our policy for selecting the next write's leader around the observation that most applications submit writes from the same region repeatedly. This leads to a simple but effective heuristic: use the closest replica.

4.4.3 Replica Types

So far all replicas have been *full* replicas, meaning they contain all the entity and index data and are able to service current reads. We also support the notion of a *witness* replica. Witnesses vote in Paxos rounds and store the write-ahead log, but do not apply the log and do not store entity data or indexes, so they have lower storage costs. They are effectively tie breakers and are used when there are not enough full replicas to form a quorum. Because they do not have a coordinator, they do not force an additional roundtrip when they fail to acknowledge a write.

Read-only replicas are the inverse of witnesses: they are non-voting replicas that contain full snapshots of the data. Reads at these replicas reflect a consistent view of some point in the recent past. For reads that can tolerate this staleness, read-only replicas help disseminate data over a wide geographic area without impacting write latency.

4.5 Architecture

Figure 5 shows the key components of Megastore for an instance with two full replicas and one witness replica.

Megastore is deployed through a client library and auxiliary servers. Applications link to the client library, which implements Paxos and other algorithms: selecting a replica for read, catching up a lagging replica, and so on.

Each application server has a designated *local replica*. The client library makes Paxos operations on that replica durable by submitting transactions directly to the local Bigtable. To minimize wide-area roundtrips, the library submits remote Paxos operations to stateless intermediary *replication servers* communicating with their local Bigtables.

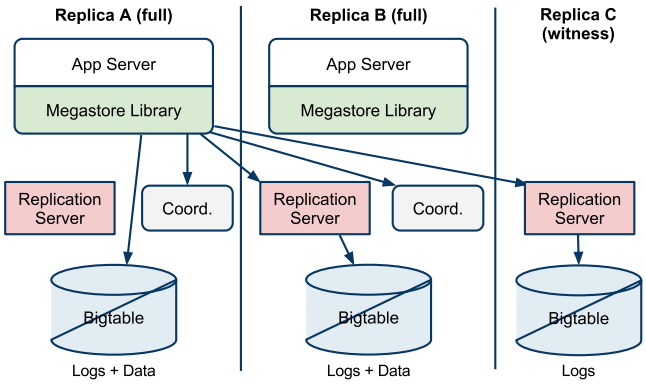

Figure 5: Megastore Architecture Example

Client, network, or Bigtable failures may leave a write abandoned in an indeterminate state. Replication servers periodically scan for incomplete writes and propose no-op values via Paxos to bring them to completion.

4.6 Data Structures and Algorithms

This section details data structures and algorithms required to make the leap from consensus on a single value to a functioning replicated log.

4.6.1 Replicated Logs

Each replica stores mutations and metadata for the log entries known to the group. To ensure that a replica can participate in a write quorum even as it recovers from previous outages, we permit replicas to accept out-of-order proposals. We store log entries as independent cells in Bigtable.

We refer to a log replica as having "holes" when it contains an incomplete prefix of the log. Figure 6 demonstrates this scenario with some representative log replicas for a single Megastore entity group. Log positions 0-99 have been fully scavenged and position 100 is partially scavenged, because each replica has been informed that the other replicas will never request a copy. Log position 101 was accepted by all replicas. Log position 102 found a bare quorum in A and C. Position 103 is noteworthy for having been accepted by A and C, leaving B with a hole at 103. A conflicting write attempt has occurred at position 104 on replica A and B preventing consensus.

4.6.2 Reads

In preparation for a current read (as well as before a write), at least one replica must be brought up to date: all mutations previously committed to the log must be copied to and applied on that replica. We call this process *catchup*.

Omitting some deadline management, the algorithm for a current read (shown in Figure 7) is as follows:

1. **Query Local**: Query the local replica's coordinator to determine if the entity group is up-to-date locally.
2. **Find Position**: Determine the highest possibly-committed log position, and select a replica that has ap-

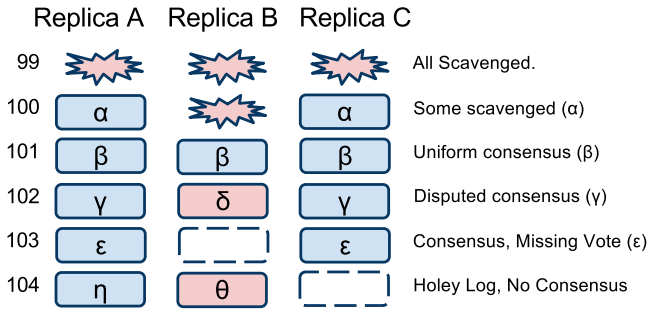

Figure 6: Write Ahead Log

plied through that log position.

(a) (*Local read*) If step 1 indicates that the local replica is up-to-date, read the highest accepted log position and timestamp from the local replica.

(b) (*Majority read*) If the local replica is not up-to-date (or if step 1 or step 2a times out), read from a majority of replicas to find the maximum log position that any replica has seen, and pick a replica to read from. We select the most responsive or up-to-date replica, not always the local replica.

3. **Catchup**: As soon as a replica is selected, catch it up to the maximum known log position as follows:

   (a) For each log position in which the selected replica does not know the consensus value, read the value from another replica. For any log positions without a known-committed value available, invoke Paxos to propose a no-op write. Paxos will drive a majority of replicas to converge on a single value—either the no-op or a previously proposed write.

   (b) Sequentially apply the consensus value of all unapplied log positions to advance the replica's state to the distributed consensus state.

   In the event of failure, retry on another replica.

4. **Validate**: If the local replica was selected and was not previously up-to-date, send the coordinator a *validate* message asserting that the (*entity group*, *replica*) pair reflects all committed writes. Do not wait for a reply—if the request fails, the next read will retry.

5. **Query Data**: Read the selected replica using the timestamp of the selected log position. If the selected replica becomes unavailable, pick an alternate replica, perform catchup, and read from it instead. The results of a single large query may be assembled transparently from multiple replicas.

Note that in practice 1 and 2a are executed in parallel.

4.6.3 Writes

Having completed the read algorithm, Megastore observes the next unused log position, the timestamp of the last write, and the next leader replica. At commit time all pending changes to the state are packaged and proposed, with a timestamp and next leader nominee, as the consensus value for the next log position. If this value wins the distributed

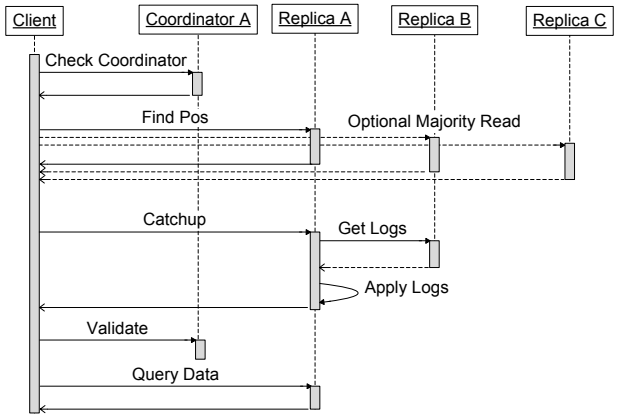

Figure 7: Timeline for reads with local replica A

consensus, it is applied to the state at all full replicas; otherwise the entire transaction is aborted and must be retried from the beginning of the read phase.

As described above, coordinators keep track of the entity groups that are up-to-date in their replica. If a write is not accepted on a replica, we must remove the entity group's key from that replica's coordinator. This process is called *invalidation*. Before a write is considered committed and ready to apply, all full replicas must have accepted or had their coordinator invalidated for that entity group.

The write algorithm (shown in Figure 8) is as follows:

1. **Accept Leader**: Ask the leader to accept the value as proposal number zero. If successful, skip to step 3.
2. **Prepare**: Run the Paxos Prepare phase at all replicas with a higher proposal number than any seen so far at this log position. Replace the value being written with the highest-numbered proposal discovered, if any.
3. **Accept**: Ask remaining replicas to accept the value. If this fails on a majority of replicas, return to step 2 after a randomized backoff.
4. **Invalidate**: Invalidate the coordinator at all full replicas that did not accept the value. Fault handling at this step is described in Section 4.7 below.
5. **Apply**: Apply the value's mutations at as many replicas as possible. If the chosen value differs from that originally proposed, return a conflict error.

Step 1 implements the "fast writes" of Section 4.4.2. Writers using single-phase Paxos skip Prepare messages by sending an Accept command at proposal number zero. The next leader replica selected at log position n arbitrates the value used for proposal zero at $n + 1$. Since multiple proposers may submit values with proposal number zero, serializing at this replica ensures only one value corresponds with that proposal number for a particular log position.

In a traditional database system, the *commit point* (when the change is durable) is the same as the *visibility point* (when reads can see a change and when a writer can be notified of success). In our write algorithm, the commit point is after step 3 when the write has won the Paxos round, but the visibility point is after step 4. Only after all full replicas have accepted or had their coordinators invalidated can the write be acknowledged and the changes applied. Acknowledging before step 4 could violate our consistency guarantees: a current read at a replica whose invalidation was skipped might

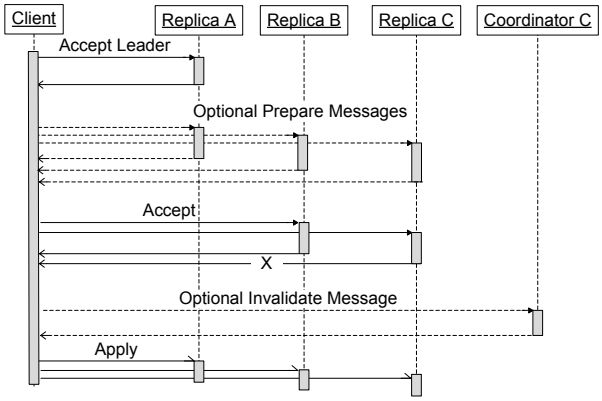

Figure 8: Timeline for writes

fail to observe the acknowledged write.

4.7 Coordinator Availability

Coordinator processes run in each datacenter and keep state only about their local replica. In the write algorithm above, each full replica must either accept or have its coordinator invalidated, so it might appear that any single replica failure (Bigtable and coordinator) will cause unavailability.

In practice this is not a common problem. The coordinator is a simple process with no external dependencies and no persistent storage, so it tends to be much more stable than a Bigtable server. Nevertheless, network and host failures can still make the coordinator unavailable.

4.7.1 Failure Detection

To address network partitions, coordinators use an out-of-band protocol to identify when other coordinators are up, healthy, and generally reachable.

We use Google's Chubby lock service [13]: coordinators obtain specific Chubby locks in remote datacenters at startup. To process requests, a coordinator must hold a majority of its locks. If it ever loses a majority of its locks from a crash or network partition, it will revert its state to a conservative default, considering all entity groups in its purview to be out-of-date. Subsequent reads at the replica must query the log position from a majority of replicas until the locks are regained and its coordinator entries are revalidated.

Writers are insulated from coordinator failure by testing whether a coordinator has lost its locks: in that scenario, a writer knows that the coordinator will consider itself invalidated upon regaining them.

This algorithm risks a brief (tens of seconds) write outage when a datacenter containing live coordinators suddenly becomes unavailable—all writers must wait for the coordinator's Chubby locks to expire before writes can complete (much like waiting for a master failover to trigger). Unlike after a master failover, reads and writes can proceed smoothly while the coordinator's state is reconstructed. This brief and rare outage risk is more than justified by the steady state of fast local reads it allows.

The coordinator liveness protocol is vulnerable to asymmetric network partitions. If a coordinator can maintain the leases on its Chubby locks, but has otherwise lost contact with proposers, then affected entity groups will experience a write outage. In this scenario an operator performs a manual step to disable the partially isolated coordinator. We have faced this condition only a handful of times.

4.7.2 Validation Races

In addition to availability issues, protocols for reading and writing to the coordinator must contend with a variety of race conditions. Invalidate messages are always safe, but validate messages must be handled with care. Races between validates for earlier writes and invalidates for later writes are protected in the coordinator by always sending the log position associated with the action. Higher numbered invalidates always trump lower numbered validates. There are also races associated with a crash between an invalidate by a writer at position n and a validate at some position $m < n$. We detect crashes using a unique epoch number for each incarnation of the coordinator: validates are only allowed to modify the coordinator state if the epoch remains unchanged since the most recent read of the coordinator.

In summary, using coordinators to allow fast local reads from any datacenter is not free in terms of the impact to availability. But in practice most of the problems with running the coordinator are mitigated by the following factors:

- Coordinators are much simpler processes than Bigtable servers, have many fewer dependencies, and are thus naturally more available.
- Coordinators' simple, homogeneous workload makes them cheap and predictable to provision.
- Coordinators' light network traffic allows using a high network QoS with reliable connectivity.
- Operators can centrally disable coordinators for maintenance or unhealthy periods. This is automatic for certain monitoring signals.
- A quorum of Chubby locks detects most network partitions and node unavailability.

4.8 Write Throughput

Our implementation of Paxos has interesting tradeoffs in system behavior. Application servers in multiple datacenters may initiate writes to the same entity group and log position simultaneously. All but one of them will fail and need to retry their transactions. The increased latency imposed by synchronous replication increases the likelihood of conflicts for a given per-entity-group commit rate.

Limiting that rate to a few writes per second per entity group yields insignificant conflict rates. For apps whose entities are manipulated by a small number of users at a time, this limitation is generally not a concern. Most of our target customers scale write throughput by sharding entity groups more finely or by ensuring replicas are placed in the same region, decreasing both latency and conflict rate.

Applications with some server "stickiness" are well positioned to batch user operations into fewer Megastore transactions. Bulk processing of Megastore queue messages is a common batching technique, reducing the conflict rate and increasing aggregate throughput.

For groups that must regularly exceed a few writes per second, applications can use the fine-grained advisory locks dispensed by coordinator servers. Sequencing transactions back-to-back avoids the delays associated with retries and the reversion to two-phase Paxos when a conflict is detected.

4.9 Operational Issues

When a particular full replica becomes unreliable or loses connectivity, Megastore's performance can degrade. We have

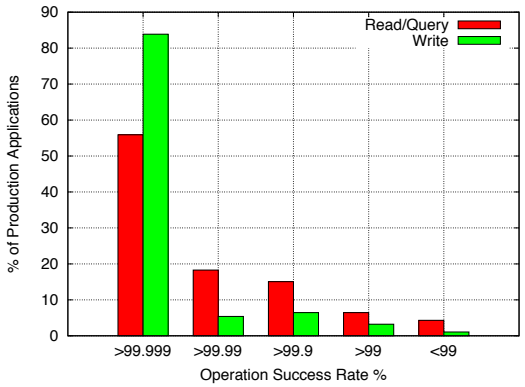

Figure 9: Distribution of Availability

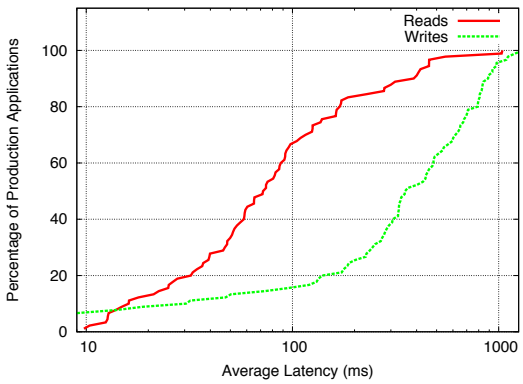

Figure 10: Distribution of Average Latencies

a number of ways to respond to these failures, including: routing users away from the problematic replica, disabling its coordinators, or disabling it entirely. In practice we rely on a combination of techniques, each with its own tradeoffs.

The first and most important response to an outage is to disable Megastore clients at the affected replica by rerouting traffic to application servers near other replicas. These clients typically experience the same outage impacting the storage stack below them, and might be unreachable from the outside world.

Rerouting traffic alone is insufficient if unhealthy coordinator servers might continue to hold their Chubby locks. The next response is to disable the replica's coordinators, ensuring that the problem has a minimal impact on write latency. (Section 4.7 described this process in more detail.) Once writers are absolved of invalidating the replica's coordinators, an unhealthy replica's impact on write latency is limited. Only the initial "accept leader" step in the write algorithm depends on the replica, and we maintain a tight deadline before falling back on two-phase Paxos and nominating a healthier leader for the next write.

A more draconian and rarely used action is to disable the replica entirely: neither clients nor replication servers will attempt to communicate with it. While sequestering the replica can seem appealing, the primary impact is a hit to availability: one less replica is eligible to help writers form a quorum. The valid use case is when attempted operations might cause harm—e.g. when the underlying Bigtable is severely overloaded.

4.10 Production Metrics

Megastore has been deployed within Google for several years; more than 100 production applications use it as their storage service. In this section, we report some measurements of its scale, availability, and performance.

Figure 9 shows the distribution of availability, measured on a per-application, per-operation basis. Most of our customers see extremely high levels of availability (at least five nines) despite a steady stream of machine failures, network hiccups, datacenter outages, and other faults. The bottom end of our sample includes some pre-production applications that are still being tested and batch processing applications with higher failure tolerances.

Average read latencies are tens of milliseconds, depending on the amount of data, showing that most reads are local. Most users see average write latencies of 100–400 milliseconds, depending on the distance between datacenters, the size of the data being written, and the number of full replicas. Figure 10 shows the distribution of average latency for read and commit operations.

5. EXPERIENCE

Development of the system was aided by a strong emphasis on testability. The code is instrumented with numerous (but cheap) assertions and logging, and has thorough unit test coverage. But the most effective bug-finding tool was our network simulator: the *pseudo-random test* framework. It is capable of exploring the space of all possible orderings and delays of communications between simulated nodes or threads, and deterministically reproducing the same behavior given the same seed. Bugs were exposed by finding a problematic sequence of events triggering an assertion failure (or incorrect result), often with enough log and trace information to diagnose the problem, which was then added to the suite of unit tests. While an exhaustive search of the scheduling state space is impossible, the pseudo-random simulation explores more than is practical by other means. Through running thousands of simulated hours of operation each night, the tests have found many surprising problems.

In real-world deployments we have observed the expected performance: our replication protocol optimizations indeed provide local reads most of the time, and writes with about the overhead of a single WAN roundtrip. Most applications have found the latency tolerable. Some applications are designed to hide write latency from users, and a few must choose entity group boundaries carefully to maximize their write throughput. This effort yields major operational advantages: Megastore's latency tail is significantly shorter than that of the underlying layers, and most applications can withstand planned and unplanned outages with little or no manual intervention.

Most applications use the Megastore schema language to model their data. Some have implemented their own *entity-attribute-value* model within the Megastore schema language, then used their own application logic to model their data (most notably, Google App Engine [8]). Some use a hybrid of the two approaches. Having the dynamic schema built on top of the static schema, rather than the other way around, allows most applications to enjoy the performance, usability,

and integrity benefits of the static schema, while still giving the option of a dynamic schema to those who need it.

The term "high availability" usually signifies the ability to mask faults to make a collection of systems more reliable than the individual systems. While fault tolerance is a highly desired goal, it comes with it its own pitfalls: it often hides persistent underlying problems. We have a saying in the group: "Fault tolerance is fault masking". Too often, the resilience of our system coupled with insufficient vigilance in tracking the underlying faults leads to unexpected problems: small transient errors on top of persistent uncorrected problems cause significantly larger problems.

Another issue is flow control. An algorithm that tolerates faulty participants can be heedless of slow ones. Ideally a collection of disparate machines would make progress only as fast as the least capable member. If slowness is interpreted as a fault, and tolerated, the fastest majority of machines will process requests at their own pace, reaching equilibrium only when slowed down by the load of the laggers struggling to catch up. We call this anomaly *chain gang throttling*, evoking the image of a group of escaping convicts making progress only as quickly as they can drag the stragglers.

A benefit of Megastore's write-ahead log has been the ease of integrating external systems. Any idempotent operation can be made a step in applying a log entry.

Achieving good performance for more complex queries requires attention to the physical data layout in Bigtable. When queries are slow, developers need to examine Bigtable traces to understand why their query performs below their expectations. Megastore does not enforce specific policies on block sizes, compression, table splitting, locality group, nor other tuning controls provided by Bigtable. Instead, we expose these controls, providing application developers with the ability (and burden) of optimizing performance.

6. RELATED WORK

Recently, there has been increasing interest in NoSQL data storage systems to meet the demand of large web applications. Representative work includes Bigtable [15], Cassandra [6], and Yahoo PNUTS [16]. In these systems, scalability is achieved by sacrificing one or more properties of traditional RDBMS systems, e.g., transactions, schema support, query capability [12, 33]. These systems often reduce the scope of transactions to the granularity of single key access and thus place a significant hurdle to building applications [18, 32]. Some systems extend the scope of transactions to multiple rows within a single table, for example the Amazon SimpleDB [5] uses the concept of *domain* as the transactional unit. Yet such efforts are still limited because transactions cannot cross tables or scale arbitrarily. Moreover, most current scalable data storage systems lack the rich data model of an RDBMS, which increases the burden on developers. Combining the merits from both database and scalable data stores, Megastore provides transactional ACID guarantees within a entity group and provides a flexible data model with user-defined schema, database-style and full-text indexes, and queues.

Data replication across geographically distributed datacenters is an indispensable means of improving availability in state-of-the-art storage systems. Most prevailing data storage systems use asynchronous replication schemes with a weaker consistency model. For example, Cassandra [6], HBase [1], CouchDB [7], and Dynamo [19] use an eventual consistency model and PNUTS uses "timeline" consistency [16]. By comparison, synchronous replication guarantees strong transactional semantics over wide-area networks and improves the performance of current reads.

Synchronous replication for traditional RDBMS systems presents a performance challenge and is difficult to scale [21]. Some proposed workarounds allow for strong consistency via asynchronous replication. One approach lets updates complete before their effects are replicated, passing the synchronization delay on to transactions that need to read the updated state [26]. Another approach routes writes to a single master while distributing read-only transactions among a set of replicas [29]. The updates are asynchronously propagated to the remaining replicas, and reads are either delayed or sent to replicas that have already been synchronized. A recent proposal for efficient synchronous replication introduces an ordering preprocessor that schedules incoming transactions deterministically, so that they can be independently applied at multiple replicas with identical results [31]. The synchronization burden is shifted to the preprocessor, which itself would have to be made scalable.

Until recently, few have used Paxos to achieve synchronous replication. SCALARIS is one example that uses the Paxos commit protocol [22] to implement replication for a distributed hash table [30]. Keyspace [2] also uses Paxos to implement replication on a generic key-value store. However the scalability and performance of these systems is not publicly known. Megastore is perhaps the first large-scale storage systems to implement Paxos-based replication across datacenters while satisfying the scalability and performance requirements of scalable web applications in the cloud.

Conventional database systems provide mature and sophisticated data management features, but have difficulties in serving large-scale interactive services targeted by this paper [33]. Open source database systems such as MySQL [10] do not scale up to the levels we require [17], while expensive commercial database systems like Oracle [4] significantly increase the total cost of ownership in large deployments in the cloud. Furthermore, neither of them offer fault-tolerant synchronous replication mechanism [3, 11], which is a key piece to build interactive services in the cloud.

7. CONCLUSION

In this paper we present Megastore, a scalable, highly available datastore designed to meet the storage requirements of interactive Internet services. We use Paxos for synchronous wide area replication, providing lightweight and fast failover of individual operations. The latency penalty of synchronous replication across widely distributed replicas is more than offset by the convenience of a single system image and the operational benefits of carrier-grade availability. We use Bigtable as our scalable datastore while adding richer primitives such as ACID transactions, indexes, and queues. Partitioning the database into entity group sub-databases provides familiar transactional features for most operations while allowing scalability of storage and throughput.

Megastore has over 100 applications in production, facing both internal and external users, and providing infrastructure for higher levels. The number and diversity of these applications is evidence of Megastore's ease of use, generality, and power. We hope that Megastore demonstrates the viability of a middle ground in feature set and replication consistency for today's scalable storage systems.

8. ACKNOWLEDGMENTS

Steve Newman, Jonas Karlsson, Philip Zeyliger, Alex Dingle, and Peter Stout all made substantial contributions to Megastore. We also thank Tushar Chandra, Mike Burrows, and the Bigtable team for technical advice, and Hector Gonzales, Jayant Madhavan, Ruth Wang, and Kavita Guliani for assistance with the paper. Special thanks to Adi Ofer for providing the spark to make this paper happen.

9. REFERENCES

[1] Apache HBase. http://hbase.apache.org/, 2008.

[2] Keyspace: A consistently replicated, highly-available key-value store. http://scalien.com/whitepapers/.

[3] MySQL Cluster. http://dev.mysql.com/tech-resources/articles/mysql_clustering_ch5.html, 2010.

[4] Oracle Database. http://www.oracle.com/us/-products/database/index.html, 2007.

[5] Amazon SimpleDB. http://aws.amazon.com/simpledb/, 2007.

[6] Apache Cassandra. http://incubator.apache.org/cassandra/, 2008.

[7] Apache CouchDB. http://couchdb.apache.org/, 2008.

[8] Google App Engine. http://code.google.com/appengine/, 2008.

[9] Google Protocol Buffers: Google's data interchange format. http://code.google.com/p/protobuf/, 2008.

[10] MySQL. http://www.mysql.com, 2009.

[11] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. On the performance of wide-area synchronous database replication. Technical Report CNDS-2002-4, Johns Hopkins University, 2002.

[12] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. Scads: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[13] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[14] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[15] 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. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[16] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, New York, NY, USA, 2010. ACM.

[18] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174, New York, NY, USA, 2010. ACM.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[20] J. Furman, J. S. Karlsson, J.-M. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A scalable data system for user facing applications. In *ACM SIGMOD/PODS Conference*, 2008.

[21] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[22] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[23] S. Gustavsson and S. F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 105–107, New York, NY, USA, 2002. ACM.

[24] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.

[25] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[26] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson. Strongly consistent replication for a bargain. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 52 –63, 2010.

[27] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[28] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. Technical Report MSR-TR-2009-63, Microsoft Research, 2009.

[29] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[30] F. Schintke, A. Reinefeld, S. e. Haridi, and T. Schutt. Enhanced paxos commit for transactions on dhts. In *10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, pages 448–454, 2010.

[31] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

[32] S. Wu, D. Jiang, B. C. Ooi, and K. L. W. Towards elastic transactional cloud storage with range query support. In *Int'l Conference on Very Large Data Bases (VLDB)*, 2010.

[33] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009.