MegaStore: Providing Scalable, Highly Available Storage for Interactive Services

Google, Inc

Presented by Tengda Tang, Chenkai Shao
Motivation

Cloud applications:

1. Scalable
2. Compete for users
3. Responsive
4. Consistent view
5. Highly available

Conflict!

Current:

- Relational database
  - rich set of features for easily building applications but difficult to scale
- Nosql database
  - highly scalable but limited API, loose consistency models
MegaStore

For today’s interactive online services

NoSQL db’s scalability + Relational db’s convenience

- Synchronous replication for availability and consistent view
- Provide RDBMS features only can scale within latency limit and the partition scheme can support
- Use paxos to replicate primary user data across data centers on every write
2. Availability and scale

Our goal:
- global
- reliable
- arbitrarily large in scale
- low latency
- ...

Real hardware:
- geographically confined
- failure-prone
- limited capacity
- ...

Availability: synchronous, fault-tolerant log replicator

Scale: partitioned data into a vast space of small databases
Availability - Replication

Wide Area common replication strategies, Recall from lecture.

- **Asynchronous Master/Slave**
  - Risk data loss. A consensus protocol is required to mediate mastership

- **Synchronous Master/Slave**
  - Master and slave failures need timely detection by an external system. Latency

- **Optimistic Replication**
  - Availability and latency are good. But global mutation ordering is not known at commit time, transaction impossible.

Small latency, no data loss, need ACID transactions!

- **Use extended Paxos!**
  - allow local reads at any up-to-date replica.
  - single round-trip writes
  - multiple replicated logs, each governing a partition of data
Scale - Partitioning and Locality

- Entity Groups partition the datastore
- Each entity group is synchronously replicated across datacenters
- ACID semantics within an entity group
- Looser consistency across entity groups
- Entity group data and replication metadata stored in scalable NoSQL datastores
Operations Across Entity Groups

Most transactions are within a single entity group

Cross entity group transactions supported via Two-Phase Commit

Asynch communication between entity groups supported by Queues

Why do we need 2pc if we already have the asynchronous messaging?
Physical layout

- Underlying data storage in a datacenter - Bigtable
- Application control placement
  - Minimize latency
    - Keep data near users and replicas near each other.
    - Assign entity groups to the region they accessed most.
  - Maximize throughput, cache efficiency
    - Held in contiguous ranges of Bigtable rows
    - Control the placement of hierarchical data
3. Tour of MegaStore - Features

- API
- Data Model
- Transactions and Concurrency Control
- Other features
Megastore API

cost-transparent APIs

● No join in megastore
  ○ High-volume interactive workloads
  ○ Reads dominate writes
  ○ Storing and querying hierarchical data is straightforward in Bigtable
  ○ Queries specify scans against particular tables

● Joins, if required, should be in application code.
  ○ Outer joins: parallel queries.

● Guarantees that features are built with a clear understanding of their performance implications.
Data Model

Abstract tuple of an RDBMS

Concrete row-column storage of NoSQL.

Types: Strings, various numbers or Google’s Protocol buffers

Peroperty: Required, optional or repeated(a list of values)

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);
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);
Data Model - Per-Joining key

- Megastore keys
  - cluster entities that will be read together
  - each entity is mapped into a single Bigtable row

<table>
<thead>
<tr>
<th>Row key</th>
<th>User. name</th>
<th>Photo. time</th>
<th>Photo. tag</th>
<th>Photo._url</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>John</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101,500</td>
<td></td>
<td>12:30:01</td>
<td>Dinner, Paris</td>
<td>...</td>
</tr>
<tr>
<td>101,502</td>
<td></td>
<td>12:15:22</td>
<td>Betty, Paris</td>
<td>...</td>
</tr>
<tr>
<td>102</td>
<td>Mary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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);
Data Model - Indexes

- **Local Index**
  - separate indexes for each entity group
  - find data within an entity group

- **Global index**
  - find entities without knowing entity groups
  - not guaranteed to reflect all recent updates

Additional indexing features

- **Storing Clause**
  - store additional properties in primary table

- **Repeated Indexes**
  - alternative to child tables

- **Inline Indexes**
  - appear as a virtual column in target entity

```sql
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);
```
Transactions and concurrency control

● Same row/column pair with different timestamps
  ○ MVCC (multiversion concurrency control)

● Reads
  ○ current read: Ensures that all previously committed writes are applied
  ○ snapshot read: Picks up the timestamp of the last known fully applied transaction and reads from there
  ○ inconsistent read: ignores the state of the log and reads the latest values directly

● Writes
  ○ Begin with a current read to determine the next available log position
  ○ optimistic concurrency: one win, others follow
Transaction lifecycle

Read: Obtain the timestamp and log position of the last committed transaction

Application logic: Read from Bigtable and gather writes to a log entry

Commit: Use Paxos to achieve consensus for appending that entry to the log

Apply: Write mutations to the entities and indexes in Bigtable

Clean up: Delete data that is no longer required
Transaction - Queues

- Queues
  - cross-group operations
  - batch multiple updates into a single transaction
  - defer work
- Transaction
  - atomically send or receive multiple messages in addition to updating its entities
- Message
  - single sending and receiving entity groups
  - delivery is asynchronous if they differ

Example: a calendar application
Transaction - 2PC

For atomic updates across entity groups.

But higher latency and increase the risk of contention. In favor of queues.

They can be useful in simplifying application code.
Other features

- Integration with Bigtable’s full-text index.
- Strong back up system
- Encryption
Replication

Reads and writes can be initiated from any replica

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
Paxos

Writes: require at least two inter-replica roundtrips - prepare and propose

Reads: require at least one round of prepares to determine the last chosen value

Definitions: accepted, chosen/committed, applied
Master-Based Approaches

Like multi-paxos

Pros
● Reads: no network communication
● Writes: single round of communication
● Batch writes to improve throughput

Cons
● Unbalanced workload
● Bottleneck at the master
● User visible outages
Megastore’s Approach

Goals

For most of the time

- Fast reads: no inter-replica RPCS (assuming write usually succeed on all replicas)
- Fast writes: single round writes
Fast Reads - Coordinator

Each replica has a coordinator server

Tracks a set of entity groups for which its replica has observed all Paxos writes (up-to-date)

Such an entity groups can serve local reads.

The write algorithm needs to manage the coordinator state.

**Conforms to the design choice that favors reads over writes**

Simple and Rarely fail
Fast Writes - Leader

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 first writer to submit a value to the leader wins the right to propose and others fall back to two-phase Paxos.

Minimize writer-leader latency by using the writer of the preceding log position (the closest replica?).
Special Replica Types

Witness replicas

   Essentially an acceptor and not a learner

Read-only replicas

   Not an acceptor, only a learner

   Useful for global-wise data distribution
Architecture

Replication servers periodically scan for incomplete writes and propose no-op values via Paxos to bring them to completion.

Figure 5: Megastore Architecture Example
Replicated Logs

Allow out-of-order proposals - similar to multi-paxos

“Holes” exist

Figure 6: Write Ahead Log
Reads

Reads or writes (need a read before write) need to be processed by a replica that has applied (different from committed) all previous writes.

How do the coordinators find an up-to-date replica?

- ideally the local replica
- if not, find the most up-to-date replica, fix holes and apply mutations
- fix holes: get commited value from peers or run full paxos to commit a value

Figure 7: Timeline for reads with local replica A
Writes

- Ask leader for the right to propose as proposal number zero
- Skip prepare if successful, otherwise run full Paxos
- Propose and accept
- Invalidate the coordinator at the replicas who did not accept
- Apply if possible
Coordinator Availability

- Coordinator failures are not common in practice
  - no external dependencies
  - no persistent storage
  - could still be unavailable during network and host failures

- 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.
Write Throughput

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.

This increases conflicts and latency but Megastore relies on the application to optimize upon conflicts.

Approaches to optimize upon conflicts

- Shard entity groups more finely
- Ensure replicas are placed in the same region
- Batch user operations into fewer Megastore transactions
- Use the fine-grained advisory locks dispensed by coordinator servers
Handle Replica Issues

Responses to an unreliable replica or connectivity

1. Reroute traffic to application servers near other replicas
2. Disable the replica’s coordinator (acceptable impact on latency)
3. Disable the entire replica (rare due to impact on availability)
Production Metrics

Figure 9: Distribution of Availability

Figure 10: Distribution of Average Latencies
Experience

Testing

- assertions and logging, and has thorough unit test coverage
- network simulator - the pseudo-random test framework
- run thousands of simulated hours of operation each night

Achieved local reads and single roundtrip writes for most of the time

Optimize applications in case Megastore doesn’t provide the desired latency
Insights

- Fault tolerance is fault masking
- Chain gang throttling when slowness is tolerated as faults
- Write-ahead log is essential for extenibility
- Application developers need to understand Bigtable NoSQL for optimizations
Thank you!