Megastore: Providing Scalable, Highly Available Storage for Interactive Services

&

Spanner: Google’s Globally-Distributed Database.

Presented by Kewei Li
The Problem

**db**
- complex
- legacy
- tuning
- expensive
- ...

**noSQL**
- simple
- clean
- just enough
- ...

As apps become more complex as apps need to be more scalable
## The Problem

<table>
<thead>
<tr>
<th></th>
<th>Guarantees</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>db</strong></td>
<td>Supports transactions (ACID)</td>
<td>Hard to scale beyond a few machines</td>
</tr>
<tr>
<td><strong>noSQL</strong></td>
<td>Eventual consistency</td>
<td>Highly scalable &amp; available</td>
</tr>
</tbody>
</table>
The Problem

Guarantees

- Supports transactions (ACID)

Scalability

- Hard to scale beyond a few machines

db

noSQL

Eventual consistency

newSQL

Highly scalable & available
MegaStore: Key Features

- Partition data into **entity groups** (chosen by user)
- **ACID semantics** within each entity group
- Availability is achieved by **replication**
MegaStore: Key Features

• Partition data into **entity groups** (chosen by user)

• **ACID semantics** within each entity group

• Availability is achieved by **replication**

• **ACID semantics across groups** is expensive (asynchronous queues are recommended)
MegaStore: Key Features

• **Google BigTable** (NoSQL) used as backend data store within each data center

• **Cost-transparent** API (implement your own joins)

• Some control over **physical layout**
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);
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);
write-ahead log for each entity group is stored here

<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>

**Committed**: in the log  
**Applied**: written to underlying data
MegaStore: Read Transactions

• Current
  • Apply all previously committed writes (in the log), then read
  • Can only be done within an entity group

• Snapshot
  • Look for timestamp of latest applied write and read from there

• Inconsistent
  • Read the latest values from applied writes
MegaStore: Write within Entity Group

1. **Current 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.
MegaStore: Write across Entity Groups

- Use asynchronous queues (message sending)
- Two-phase commit
MegaStore: Replication and Paxos

- **Coordinator** keeps track of which entity groups (within a replica) have observed all writes. These entity groups can respond immediately to reads.
- Writes are agreed upon by **Paxos** (one for each write).
- Replicas can also be witnesses or read-only.
MegaStore: Replication and Paxos

<table>
<thead>
<tr>
<th>Replica A</th>
<th>Replica B</th>
<th>Replica C</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td></td>
<td></td>
<td>All Scavenged.</td>
</tr>
<tr>
<td>100</td>
<td>α</td>
<td>α</td>
<td>Some scavenged (α)</td>
</tr>
<tr>
<td>101</td>
<td>β</td>
<td>β</td>
<td>Uniform consensus (β)</td>
</tr>
<tr>
<td>102</td>
<td>γ</td>
<td>δ</td>
<td>Disputed consensus (γ)</td>
</tr>
<tr>
<td>103</td>
<td>ε</td>
<td></td>
<td>Consensus, Missing Vote (ε)</td>
</tr>
<tr>
<td>104</td>
<td>η</td>
<td>θ</td>
<td>Hole Log, No Consensus</td>
</tr>
</tbody>
</table>
MegaStore: Read Algorithm

Client

Check Coordinator

Find Pos

Catchup

Validate

Query Data

Coordinator A

Replica A

Replica B

Replica C

Optional Majority Read

Get Logs

Apply Logs
MegaStore: Write Algorithm

Client → Replica A → Replica B → Replica C → Coordinator C

- Accept Leader
- Optional Prepare Messages
- Accept
- X
- Optional Invalidate Message
- Apply
MegaStore: Failures & Debugging

• Coordinators are only valid if they are holding Chubby locks. If a coordinator fails
  • They (conservatively) lose Chubby locks if they fail
  • Reads are processed by other replicas
  • Writes wait to be sure that the coordinator has failed

• If a replica fails,
  • Reroute traffic
  • Invalidate its unhealthy coordinator if it is still holding Chubby locks
  • Disable replica entirely (impacts availability)

• Debug synchronization issues through pseudo-random test framework
MegaStore: Deployment

![Bar Chart and Line Graph](Images emploi 2022-08-01 16-00-41.png)
MegaStore: Weaknesses

• Can only support a few writes per second per entity group due to Paxos conflicts
• Users need to carefully partition data into entity groups
• Global ACID semantics are expensive due to two-phase commits
• “Slow”
Spanner: Key Features

- Supports **complicated transaction** but less cost transparent
- **Global ACID** semantics
- **Availability** through replication
- **Rebalancing** of data across spanservers
Replicas provide availability.

Leader chosen by Paxos

Coordinates two phase commit across Paxos groups

Provides synchronization within a Paxos group

Btree/distributed file system

Replicas provide availability
Spanner: Directories

- Administrators can specify regions and number & type of replicas
- Data is grouped into directories, which can be assigned to regions by applications
- Movedir is a background process that can move directories across Paxos groups
Spanner: Data Model

CREATE TABLE Users {
    uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
    uid INT64 NOT NULL, aid INT64 NOT NULL, name STRING
} PRIMARY KEY (uid, aid), INTERLEAVE IN PARENT Users ON DELETE CASCADE;
Spanner: True Time

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT.now()</td>
<td>TTinterval: [earliest, latest]</td>
</tr>
<tr>
<td>TT.after(t)</td>
<td>true if $t$ has definitely passed</td>
</tr>
<tr>
<td>TT.before(t)</td>
<td>true if $t$ has definitely not arrived</td>
</tr>
</tbody>
</table>

• Invocation of TT.now() is guaranteed to fall within returned interval
• Implemented using GPS synchronized and atomic clocks (with algorithms to detect outliers)
• Error in time typically rise from 1ms to 7ms between synchronizations (every 30s), though overloading/network conditions can cause spikes
Spanner: Time Stamping Writes

• Spanner always fulfill these time stamping invariants for writes:
  • **Start**: the time stamp for a write is after the write has arrived at the server
  • **Commit Wait**: the time stamp for a commit is before the actual commit

• Within a Paxos group, the leader can use locks to ensure these invariants (leaders are leased every 10s; there can only be one leader)

• Across Paxos groups
  • all participants send prepare messages to the coordinator leader. The leader chooses a timestamp that is later than all received timestamps (to fulfill Start)
  • The leader chooses a commit time stamp, waits until the time stamp has passed, then allows participants to commit (to fulfill Commit Wait)
Spanner: Time Stamping Reads

• Every replica has a $t_{safe}$, which is the time that it is up to date

• $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$

• $t_{safe}^{Paxos}$ is the time of the last applied Paxos write

• $t_{safe}^{TM}$ is $\infty$ if there are no prepared but not committed transactions (phase one of a two phase write). If there are, $t_{safe}^{TM}$ is the minimum of prepare timestamps from all the transaction managers

• Reads can be served from any replica with a sufficiently high $t_{safe}$. You can request a time stamp (snapshot) or get one assigned
  • If the read can be served by one Paxos group, assigned time stamp is latest commit. If read is across groups, then it is TT.now().latest
Spanner: Other Time Stamping Refinements

• $t_{\text{safe}}^{TM}$ can be augmented with finer locking to allow reads to proceed with a two phase write happening

• $t_{\text{safe}}^{Paxos}$ is advanced every 8 seconds by the Paxos leader

• Schema changes can be time stamped and completed in the background
Spanner: Performance Testing

- Two phase commit scalability (over 10 runs)

![Graph showing latency over participants]

- Mean
- 99th Percentile
Spanner: Performance Testing

• Availability: killing a server
Spanner: Performance Testing

• Availability: killing a server
Spanner: Performance Testing

- Availability: true time error
Spanner: Performance Testing

- F1 deployment: distribution of fragments and latency measurements

<table>
<thead>
<tr>
<th># fragments</th>
<th># directories</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&gt;100M</td>
</tr>
<tr>
<td>2–4</td>
<td>341</td>
</tr>
<tr>
<td>5–9</td>
<td>5336</td>
</tr>
<tr>
<td>10–14</td>
<td>232</td>
</tr>
<tr>
<td>15–99</td>
<td>34</td>
</tr>
<tr>
<td>100–500</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operation</th>
<th>latency (ms)</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>std dev</td>
</tr>
<tr>
<td>all reads</td>
<td>8.7</td>
<td>376.4</td>
</tr>
<tr>
<td>single-site commit</td>
<td>72.3</td>
<td>112.8</td>
</tr>
<tr>
<td>multi-site commit</td>
<td>103.0</td>
<td>52.2</td>
</tr>
</tbody>
</table>
Spanner: Performance Testing

• F1 benefits over sharded MySQL
  • No need for manual resharding
  • Better failover management
  • (F1 needs transactions so NoSQL won’t work)

• Future works
  • True time improvements (better hardware)
  • “scale-up” in each server