MEGASTORE:
Providing Scalable, Highly Available Storage for Interactive Services

Based on: https://research.google.com/pubs/pub36971.html

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Adam Erdreich

236826 - Seminar in Databases, 21-05-2017
Google’s online applications, such as Gmail, Google Docs, Google Photos, etc. pose high requirements for a database, which might even seem conflicting:

- Huge audience ⇒ must be highly scalable
- Strong competition ⇒ must enable rapid development
- Responsive service ⇒ must impose low latency
- Collaborative ⇒ must provide a consistent view of data
- Critical ⇒ must have 24/7 availability
ENTITY GROUPS: “MIDDLE GROUND” BETWEEN RDMBS & NoSQL

- Data will be divided into fine-grained “Entity Groups”, defined by each application as the developer sees fit
  - Each group is independently geo-replicated (overview of replication comes soon).
- Mutations within an entity group support single-phase ACID transactions.
- Write to entities from different group provide much weaker consistency
  - Transactions across groups are supported using costly 2-phase commits.
- Meaning:
  - Too fine-grained partitioning will not benefit from ACID capabilities
  - Grouping unrelated data hurts performance
- Example: Email App - Accounts are natural entity groups
  - When a user receives a message, it is immediately and persistently observable to him.
    - also other actions: labeling, deleting...
  - Communications between accounts are 2-phase-committed.
PHYSICAL LAYOUT

- **Google’s BigTable**
  - NoSQL Column-family store

- **Datacenters**
- **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
  - *Google’s BigTable*
  - (NoSQL Column-family store)
REPLICATION STRATEGIES: COMMON PRACTICES

- **Asynchronous Master/Slave:**
  - Pretty fast
  - Supports fast ACID transactions in the master replica
  - .... but might suffer data-loss on failover to a slave
  - Requires Master Election

- **Synchronous Master/Slave**
  - No data-loss (for an acknowledged write)
  - Full support for transactions
  - Suffers WAN latency
  - Requires Master Election

- **Optimistic Replication**
  - Excellent availability and latency
  - No transactions possible, within any granularity
REPLICATION STRATEGIES: ENTER PAXOS!

- Proven, Fault-Tolerant
- Allows any member to initiate a write

In many systems, Paxos is used to distinguish a master, which will then be used to coordinate all accesses to a piece of data, creating a possible hotspot [the master needs to be able to handle the full system load].

Megastore wants to avoid this, by using Paxos for the actual data path. For every entity group, we keep a separate replicated logs, in which for every write to the group we add an entry to the log at a quorum of replicas.

Paxos has its downsides, when discussing geo-distributed systems, the need to read from a quorum might cause high latency. Megastore has interesting ways to overcome these downsides.
We want to enable *local reads*, that require reading only a single replica. It means that we need a way to tell who’s holding up-to-date data at any given point in time.

So we introduce a “Coordinator” service to run on each replica, which keeps track of the entity groups that are up-to-date at that replica.

- It is extremely simplistic, holds no persistent state, and barely uses I/O
- Thus it is much more stable than a BigTable instance
- “Coordinator” failures can cause write-outages, but are so rare that the risk is justified for the benefit of fast local reads

But... how can a replica know *for sure* it’s up-to-date?

- Must be told by an external entity when misses a write
PAXOS: MEGASTORE’S ADDITIONS

Faster Reads

Diagram showing the process of faster reads in PAXOS.
PAXOS: MEGASTORE’S ADDITIONS

Faster Writes

- We don’t want the 2-round-trip cost of Paxos for every write, but don’t want to use multi-Paxos to avoid having a distinguished master.

- The approach we take is in-between: for each write we set a leader.
  - It arbitrates between writers who want to propose values, allowing only the first one to use a single-round-trip Paxos “Accept”, while the other fall back to 2-phase.

- Each write also nominates the next leader for this entity group.
  - Chosen by a very simple heuristic - the closest replica to the writer.
PAXOS: MEGASTORE’S ADDITIONS

Faster Writes
Megastore’s data model is based on *strongly typed schemas* (somewhat SQL-like)

Each schema is composed of *tables* (which represent entity types)

Each table holds *properties*

Defined like a Protobuf “Message”

Let’s go through the example, it is surprisingly clear

```
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);
```

https://developers.google.com/protocol-buffers/
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);
MEGASTORE’S DATA MODEL

Representation in BigTable

<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>
PERFORMANCE METRICS

- Majority of apps have “five-nines” success for both writes and reads.
- noted: some measured apps are in very early development stages

- ~60% of reads have sub-100ms latency
- ~80% of writes have MORE than 100ms latency
QUESTIONS?