Spanner: Google's Globally-Distributed Database

Presented by Maciej Swiech
What is Spanner?

"...Google's scalable, multi-version, globally-distributed, and synchronously replicated database."
What is Spanner?

- Distributed Multiversion Database
  - General-purpose transactions (ACID)
  - SQL syntax
  - Schematized tables
  - Semi-relational data model
- TrueTime API
  - Versioning / ordering
Organization

- **Universes**
  - test, dev, production

- **Zones**
  - unit of administrative deployment

- **Spanservers**
  - data served by zonemaster
Organization - spanserver

- **Tablet**
  - (key, timestamp) --> value
  - Collusus storage
  - stores replica state
- **Paxos state machine**
  - consensus solving
Organization - spanserver

- Participant leader
  - tx manager
  - lock table
  - long-lived (10s)
spanserver leader selection

- Potential leader sends requests for lease votes
- If it receives a quorum, it is leader
- Successful write extends leadership
- Votes are timestamped
  - upon expiration, no longer count towards quorum
- Invariant: Each leader's lease interval is disjoint
Organization - directories

- Sharded buckets generally located in one zone
- Spatial locality - often accessed directories are colocated
- Replication specified by application / location
- \textit{Movedir} directive takes advantage of sharding to move a directory
Data Model

- SQL, schematized table structure
  - move away from BigTable / NoSQL
- Semi-relational
  - all tables must have a PK set -- a "name"
- Two phase commits
  - run through Paxos state machine
- Schema allows for locality and replica description
Data Model - syntax

- Mostly SQL
- **INTERLEAVE IN**
- Schematize tables allow for more complex lookups

```sql
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;
```
Synchronization

- Transactional reads/writes must acquire a lock in the lock table
- Other operations can read directly from the replica
How to solve ordering?

We have seen:

- Versioning
- Vector clocks
- Matrix clocks

Spanner's answer...
TrueTime

- GPS and Atomic clock driven
- Known inaccuracy
- Common worst-case scenario: 10 ms delay

<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>
TrueTime

- GPS
  - more precise, more likely to fail
- Atomic clock
  - predictable inaccuracy, less likely to fail
- Server hierarchy to provide time to nodes
- If accuracy is too low, system will slow to ensure ordering is preserved

- Synchronization every 30 seconds
- Sawtooth pattern of inaccuracy, max = 7ms
Concurrency - transactions

- Spanner internally retries transactions (how?)
- Transactions are served by checking timestamps on replicas
- Transactions store in log and garbage collected (how?)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Timestamp Discussion</th>
<th>Concurrency Control</th>
<th>Replica Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Write Transaction</td>
<td>§ 4.1.2</td>
<td>pessimistic</td>
<td>leader</td>
</tr>
<tr>
<td>Read-Only Transaction</td>
<td>§ 4.1.4</td>
<td>lock-free</td>
<td>leader for timestamp; any for read, subject to § 4.1.3</td>
</tr>
<tr>
<td>Snapshot Read, client-provided timestamp</td>
<td>—</td>
<td>lock-free</td>
<td>any, subject to § 4.1.3</td>
</tr>
<tr>
<td>Snapshot Read, client-provided bound</td>
<td>§ 4.1.3</td>
<td>lock-free</td>
<td>any, subject to § 4.1.3</td>
</tr>
</tbody>
</table>
Concurrency - leader leases

- Remember, leader leases must be disjoint!
- Leader maintains $s_{max}$ - its maximum timestamp
- $s_{max}$ increases with transactions
- Leader must wait for $TT.after(s_{max})$ to abdicate
Concurrency - RW

- Two-phased locking
  - timestamp at any locked time
  - Paxos timestamp used
- Timestamps issued are monotonically increasing
- $s_{\text{max}}$ incremented
Concurrency - RW

- Event $e_{i^{start}}$ occurs that starts transaction
- Leader for $T_i$ assigns commit timestamp $s_i$ no less than $TT.now().latest$ (eval at $e_{i^{server}}$)
- Commit wait: no other client can see data until $TT.after(s_{i^{commit}})$, where $s_{i^{commit}}$ is time of commit
- Ordering is guaranteed
Concurrency - RW

\[ s_1 < t_{abs}(e_1^{commit}) \]  (commit wait)

\[ t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \]  (assumption)

\[ t_{abs}(e_2^{start}) \leq t_{abs}(e_2^{server}) \]  (causality)

\[ t_{abs}(e_2^{server}) \leq s_2 \]  (start)

\[ s_1 < s_2 \]  (transitivity)
Concurrency - reads

- Each replica maintains $t_{\text{safe}}$
  - maximum up-to-date timestamp
- $t_{\text{safe}} = \min(t_{\text{safe}}^{\text{Paxos}}, t_{\text{safe}}^{\text{TM}})$
- $t_{\text{safe}}^{\text{Paxos}} = \text{highest applied write timestamp}$
- $t_{\text{safe}}^{\text{TM}}$ depends on number of pending tx
- If $t_{\text{read}} < t_{\text{safe}}$, read is allowed through
Concurrency - RO

- Assign $s_{read} = TT.now().latest$ to transaction
- May block if $t_{safe}$ is not advanced enough
- Reads are done at any up-to-date replica
  - inferred from transaction "scope"
- $s_{max}$ increased
Overview

- Lock-free distributed read transactions
- External consistency of distributed transactions
- Integration of concurrency control, replication, and two-phase commits
- TrueTime global time system
Evaluation

- Somewhat lacking...
- Throughput
  - throughput increases with number of replicas, optimal occurs at 3-5 replicas
- Latency
  - number of replicas does not affect latency much
  - commit wait: 5ms
  - Paxos latency: 9ms
Evaluation - scalability

- 3 zones, 25 spanservers each
- 50 participant optimal scale

<table>
<thead>
<tr>
<th>participants</th>
<th>latency (ms)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>99th percentile</td>
</tr>
<tr>
<td>1</td>
<td>17.0 ±1.4</td>
<td>75.0 ±34.9</td>
</tr>
<tr>
<td>2</td>
<td>24.5 ±2.5</td>
<td>87.6 ±35.9</td>
</tr>
<tr>
<td>5</td>
<td>31.5 ±6.2</td>
<td>104.5 ±52.2</td>
</tr>
<tr>
<td>10</td>
<td>30.0 ±3.7</td>
<td>95.6 ±25.4</td>
</tr>
<tr>
<td>25</td>
<td>35.5 ±5.6</td>
<td>100.4 ±42.7</td>
</tr>
<tr>
<td>50</td>
<td>42.7 ±4.1</td>
<td>93.7 ±22.9</td>
</tr>
<tr>
<td>100</td>
<td>71.4 ±7.6</td>
<td>131.2 ±17.6</td>
</tr>
<tr>
<td>200</td>
<td>150.5 ±11.0</td>
<td>320.3 ±35.1</td>
</tr>
</tbody>
</table>
Evaluation - availability

- 5 zones, 25 spanservers each
- All leaders in $Z_1$
- Soft kill notifies of necessary re-election
- CAVEAT - spare capacity
Evaluation - TrueTime

- CPU failure 6x more likely than clock failure
- March 30th improvement due to network
- Inaccuracy is low enough to guarantee invariants Spanner depends on for ordering
• Advertising backend - sharded MySQL DB
  ○ sharded data stored in BigTables

• Why switch?
  ○ dynamic sharding (remember directories?)
  ○ synchronous replication and automatic failover
  ○ strong transactional semantics
Evaluation - F1

- Replicas on both US coasts
- Database mainetnance and upkeep greatly simplified over previous model
- Cluster failure mitigated by schema re-definition

<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>all reads</td>
<td>8.7</td>
<td>21.5B</td>
</tr>
<tr>
<td>single-site commit</td>
<td>72.3</td>
<td>31.2M</td>
</tr>
<tr>
<td>multi-site commit</td>
<td>103.0</td>
<td>32.1M</td>
</tr>
</tbody>
</table>
Questions?