Spanner: Google’s Globally-Distributed Database

Google, Inc.
OSDI 2012

Presented by: Karen Ouyang
Problem Statement

- Distributed data system with high availability
- Support external consistency!
Key Ideas

• Distributed data system with high availability
• Supports external consistency!
• Enabling technology: TrueTime API
Server Organization

Datacenters have one or more zones.
Server Organization

handles moving data across zones

assigns data to spanservers

used by clients to locate spanservers

serves data to clients

Zone 1
- zonemaster
- location proxy
- spanserver

Zone 2
- zonemaster
- location proxy
- spanserver

Zone N
- zonemaster
- location proxy
- spanserver
Spanserver Stack

between 100 and 1000 instances

(key: string, timestamp: int64) → string
Spanserver Stack

set of replicas: *Paxos group*

writes initiate Paxos protocol at leader; reads from any sufficiently up-to-date replica
supports distributed transactions
contains state for two-phase locking
transactions with 1+ group: two-phase commit
select *coordinator leader* from participant leaders

supports distributed transactions
contains state for two-phase locking
TrueTime API

• Exposes clock uncertainty by expressing time as an interval
• Uses GPS and atomic clocks
• *Time master* machines per datacenter
• Client polls multiple masters to compute time interval
TrueTime API

\texttt{TT.now()} \quad 2\epsilon
Consistency

• Ensure external consistency by ensuring timestamp order
• All transactions are assigned timestamp
• Data written by $T$ is timestamped with $s$
Read-Write Transactions

• Two-phase locking: assign timestamps at any time that locks are held
• Assign timestamps to Paxos writes in increasing order across leaders
  – A leader only assigns timestamps within its leader lease; leader leases are disjoint
Read-Write Transactions

• Transactions: two-phase commit
• Two transactions

\[ T_1 \]
start
commit

\[ T_2 \]
start
commit

• Assign commit timestamps with \( s_1 < s_2 \)
• How?
Read-Write Transactions

Start: commit timestamp is *after* time of commit request at server

• or: $t_{abs}(e_2^{server}) \leq s$
Read-Write Transactions

Commit wait: cannot see data committed by T until s (assigned timestamp) has passed

\[
pick \quad s = TT.now().latest
\]
Read-Write Transactions

Commit wait: cannot see data committed by T until s (assigned timestamp) has passed

- pick $s = \text{TT.now().latest}$
- $s$
- wait until $s < \text{TT.now().earliest}$
Commit wait: cannot see data committed by T until s (assigned timestamp) has passed

- Pick $s = \text{TT.now().latest}$
- Wait until $s < \text{TT.now().earliest}$

commit wait
Read-Write Transactions

Commit wait: cannot see data committed by T until s (assigned timestamp) has passed

pick
s = TT.now().latest

wait until
s < TT.now().earliest

commit wait
Read-Write Transactions

\[ s_1 < t_{\text{abs}}(e_1^{\text{commit}}) \]
\[ t_{\text{abs}}(e_1^{\text{commit}}) < t_{\text{abs}}(e_2^{\text{start}}) \]
\[ t_{\text{abs}}(e_2^{\text{start}}) < t_{\text{abs}}(e_2^{\text{server}}) \]
\[ t_{\text{abs}}(e_2^{\text{server}}) \leq s_2 \]

\[ s_1 < s_2 \]
Read-Write Transactions

Two-phase commit

coordinator
leader

participant

participant
Read-Write Transactions

Two-phase commit: client begins
Read-Write Transactions

Two-phase commit

- Coordinator/Leader
- Participant
- Participant
Read-Write Transactions

Two-phase commit

- Coordinator/Leader
- Participant
- Participant

Choose prepare timestamp
Read-Write Transactions

Two-phase commit

log prepare record in Paxos

choose prepare timestamp

coordinator leader

participant

participant
Read-Write Transactions

Two-phase commit

- The coordinator logs the prepare record in Paxos.
- Participants receive the prepare record and choose a timestamp.
- The coordinator sends a prepare timestamp message to the participants.
- Participants acknowledge receipt of the prepare timestamp.
Read-Write Transactions

Two-phase commit

- Coordinator
  - Leader
  - Participant
  - Participant

- Log prepare record in Paxos
- Send prepare timestamp
- Choose commit timestamp
- Choose prepare timestamp
Read-Write Transactions

Two-phase commit

- log prepare record in Paxos
- log commit in Paxos
- send prepare timestamp
- choose commit timestamp
- choose prepare timestamp
Two-phase commit

- Coordinator/leader:
  - log prepare record in Paxos
  - choose commit timestamp

- Participant:
  - choose prepare timestamp
  - send prepare timestamp
  - choose commit timestamp
  - commit wait done
Read-Write Transactions

Two-phase commit

- log prepare record in Paxos
- log commit in Paxos
- notify
- choose prepare timestamp
- send prepare timestamp
- choose commit timestamp
- commit wait done
- coordinate leader
- participant
- participant
Read-Write Transactions

Two-phase commit

- log prepare record in Paxos
- log commit in Paxos
- choose commit timestamp
- notify
- commit wait done
- choose prepare timestamp
- send prepare timestamp
- coordinator leader
- participant
- participant
Read-Write Transactions

Two-phase commit

- log prepare record in Paxos
- log commit in Paxos
- log outcome in Paxos

coordinator leader

participant

participant

choose prepare timestamp

send prepare timestamp

choose commit timestamp

commit wait done

notify
Read-Write Transactions

Two-phase commit

- log prepare record in Paxos
- log commit in Paxos
- log outcome in Paxos

coordinator
leader

participant

participant

choose prepare timestamp

send prepare timestamp

choose commit timestamp

commit wait done

notify
Read-Only Transactions

• Serving reads at a timestamp
  – Replica tracks safe time $t_{\text{safe}}$: can read $t \leq t_{\text{safe}}$
  – Define $t_{\text{safe}} = \min(t^{\text{Paxos}}, t^{\text{TM}})$

• Assigning timestamps to RO transactions
  – Simplest: assign $s_{\text{read}} = \text{TT.now().latest}$
  – May block; should assign oldest timestamp that preserves external consistency
Microbenchmarks

Two-phase commit scalability

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

Effect of killing servers on throughput
Performance

• TrueTime

• F1, Google’s advertising backend
  – Automatic failover 😊
  – High standard deviation for latency?
Final Thoughts

- Implemented at a large scale (F1)!
- Commit wait is pretty clever
- Very dependent on clocks
- Security?
References


• http://research.google.com/archive/spanner.html