What? - Key Features

- Globally distributed
- Versioned data
- SQL transactions + key-value read/writes
- External consistency
- Automatic data migration across machines (even across datacenters) for load balancing and fault tolerance.
External Consistency

- Equivalent to linearizability
- If a transaction $T_1$ commits before another transaction $T_2$ starts, then $T_1$’s commit timestamp is smaller than $T_2$.
- Any read that sees $T_2$ must see $T_1$.
- The strongest consistency guarantee that can be achieved in practice (Strict consistency is stronger, but not achievable in practice).
Why Spanner?

- **BigTable**
  - Good performance
  - Does not support transaction across rows.
  - Hard to use.

- **Megastore**
  - Support SQL transactions.
  - Many applications: Gmail, Calendar, AppEngine...
  - Poor write throughput.

- Need SQL transactions + high throughput.
Figure: Spanner Server Software Stack
Spanserver Software Stack Cont.

- Spanserver maintains data and serves client requests.
- Data are key-value pairs.
  
  \[(\text{key:}\text{string}, \text{timestamp:}\text{int64}) \rightarrow \text{string}\]
  
- Data is replicated across spanservers (could be in different datacenters) in the unit of tablets.
- A Paxos state machine per tablet per spanserver.
- Paxos group: the set of all replicas of a tablet.
A long lived Paxos leader
  - Timed leases for leader election (more details later).
  - Need only one RTT in failure-free situations.

A lock table for concurrency control
  - Multiple transactions may happen concurrently – need concurrency control.
  - Maintained by Paxos leader.
  - Maps ranges of keys to lock states.
  - Two-phase locking.
  - Wound-wait for dead lock avoidance.
  - Older transactions are aborted for retry if a younger transaction holds the lock (handled internally).

This is the case for most transactions.
Transactions Involving Multiple Paxos Groups

- Participant leader: transaction manager, leader within group.
  - Implemented on Paxos leader.
- Coordinator leader: Chosen among participant leaders involved in the transaction.
  - Initiates two-phase commit for atmoicity.
  - Prepare message is logged as a Paxos action in each Paxos group (via participant leader).
  - Within each group, the commit is dealt with Paxos.
- This logic is bypassed for transactions involving only one Paxos group.
- Running two-phase commit over Paxos mitigates availability problem.
Data Model

- Semi-relational data model.
- The relational part:
  Data organized as tables; support SQL-based query language.
- The non-relational part:
  Each table is required to have an ordered set of primary-key columns.
- Primary-key columns allows applications to control data locality through their choices of keys.
  - Tablets consist of directories.
  - Each directory contains a contiguous range of keys.
  - Directory is the unit of data placement.
TrueTime

- Used to implement major logic in Spanner.

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

- Two kinds of data references: GPS and atomic clocks – different failure causes.
- A set of time master machines per datacenter. Others are daemons.
- Masters synchronize themselves.
- Daemons poll from master periodically.
- Increasing time uncertainty within each poll interval.
Transactions supported by Spanner

<table>
<thead>
<tr>
<th>Operation</th>
<th>Concurrency Control</th>
<th>Replica Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Write Transaction</td>
<td>pessimistic</td>
<td>leader</td>
</tr>
<tr>
<td>Read-Only Transaction</td>
<td>lock-free</td>
<td>leader, any</td>
</tr>
<tr>
<td>Snapshot Read, client-provided timestamp</td>
<td>lock-free</td>
<td>any</td>
</tr>
<tr>
<td>Snapshot Read, client-provided bound</td>
<td>lock-free</td>
<td>any</td>
</tr>
</tbody>
</table>

- Standalone writes are implemented as read-write transactions.
- Standalone reads are implemented as read-only transactions.
Paxos Leader Leases

- A spanserver sends request for timed lease votes.
- Leadership is granted when it receives acknowledgements from a quorum.
- Lease is extended on successful writes.
- Everyone agrees on when the lease expires. No need for fault tolerance master to detect failed leader.
Recall the two types of transactions discussed before.

Invariant #1: timestamps must be assigned in monotonically increasing order.
  - Leader must only assign timestamps within the interval of its leader lease.

Invariant #2: if transaction $T_1$ commits before $T_2$ starts, $T_1$’s timestamp must be greater than $T_2$’s.
Wait-wound for dead lock avoidance of reads.

Clients buffer writes.

Client chooses a coordinate group, which initiates two-phase commit.

A non-coordinator-participant leader chooses a prepare timestamp and logs a prepare record through Paxos and notifies the coordinator.

The coordinator assigns a commit timestamp $s_i$ no less than all prepare timestamps and $TT.now().latest$ (computed when receiving the request).

The coordinator ensures that clients cannot see any data committed by $T_i$ until $TT.after(s_i)$ is true. This is done by commit wait (wait until absolute time passes $s_i$ to commit).
Serving Reads at a Timestamp

- \( t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM}) \). Serves read only if read timestamp no larger than \( t_{safe} \).
- \( t_{safe}^{Paxos} \): the timestamp of highest Paxos write.
- \( t_{safe}^{TM} \): \( \infty \) if there are zero prepared transactions; \( \min_i(s_{i,g}^{\text{prepare}}) - 1 \) if there are prepared transactions.
  - Does not know if the transaction will be eventually committed.
  - Prevents clients from reading it.
- Problem: What if \( t_{safe}^{TM} \) does not advance (no multiple-group transactions)?
Leader assigns a timestamp - obeying external consistency. Then it does a snapshot read on any replica.

External consistency requires the read to see all transactions committed before the read starts - timestamp of the read must be no less than that of any committed writes.

Let $s_{read} = \text{TT.now().latest}$ may cause blocking. Reduce it!

If the read involves only one Paxos group, let $s_{read}$ be the timestamp of last committed write ($\text{LastTS()}$).

If the read involves multiple Paxos group, $s_{read} = \text{TT.now().latest}$ – avoid negotiation.

What if there are no more write transactions? Blocking infinitely?
Refinement #1

- $t_{safe}^{TM}$ may prevent $t_{safe}$ from advancing.
- Solution: lock table maps key ranges to prepared-transaction timestamps.
Refinement #2

- Commit wait causes commits to happen some time after the commit timestamp.
- `LastTS()` causes reads to wait for commit wait.
- Solution: lock table maps key range to commit timestamps. Read timestamp only needs to be the maximum timestamp of conflicting writes.
Refinement #3

- $t^{Paxos}_{safe}$ cannot advance in the absence of Paxos writes. May cause reads to block infinitely.

- Solution: as leader must assign timestamps no less than the starting time of its lease, $t^{Paxos}_{safe}$ can advance as new lease starts.
What does TrueTime Buy You?

- Murat Demirbas: TrueTime benefits snapshot reads the most. Otherwise, there's no easy way to specify an old snapshot.
- TrueTime allows replicas to know expired leadership without a fault tolerance master.
- How would you guarantee timestamp monotonically increase across leaders without TrueTime? New leader needs to figure out the highest timestamp assigned by the old leader.
- Avoid the negotiation round for assigning timestamp for read that involves multiple Paxos groups.
Criticisms

- Same as previous Google papers, poor experiments.
- How is old data cleaned?