Spanner

Iztok Savnik, FAMNIT

Januar, 2023.

Literature

James C. Corbett, Et.al., Spanner: Google’s Globally-Distributed Database, OSDI 2012.

Robert Morris, Lecture: Spanner, MIT 6.824, Distributed Systems, 2020.

Outline

- Introduction
- Software stack
- Data model
- TrueTime
- RW transactions
- RO transactions
- Snapshot reads
- Overview with examples

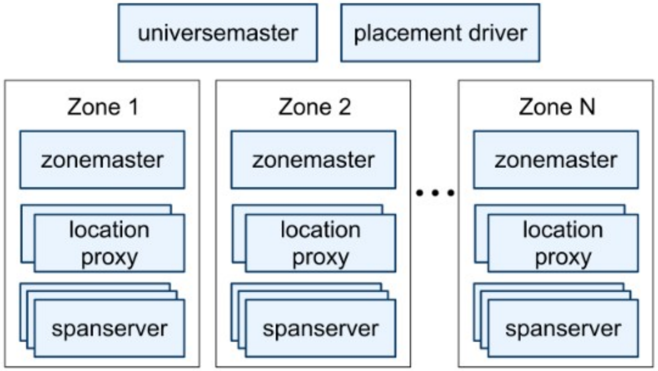
Introduction

- Spanner is Google’s scalable, multi-version, globally-distributed, and synchronously-replicated database.
  - First system to distribute data at global scale and
  - Support externally-consistent distributed transactions.
- A novel time API that exposes clock uncertainty is critical to provide:
  - External consistency:
    - If T1 commits before T2 starts, then ts(T1)<ts(T2), and T2 must see T1's writes, globally.
  - Non-blocking reads in the past,
  - Lock-free read-only transactions, and
  - Atomic schema changes.

Introduction

- Shards data across many sets of Paxos state machines in data-centers spread globally.
  - Replication is used for global availability and geographic locality;
    - Clients automatically failover between replicas.
    - Managing cross-datacenter replication is main focus.
  - Spanner automatically:
    - Reshards data across machines on the changed amount of data or number of servers.
    - Migrates data across machines to balance load and in response to failures.
  - Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database.

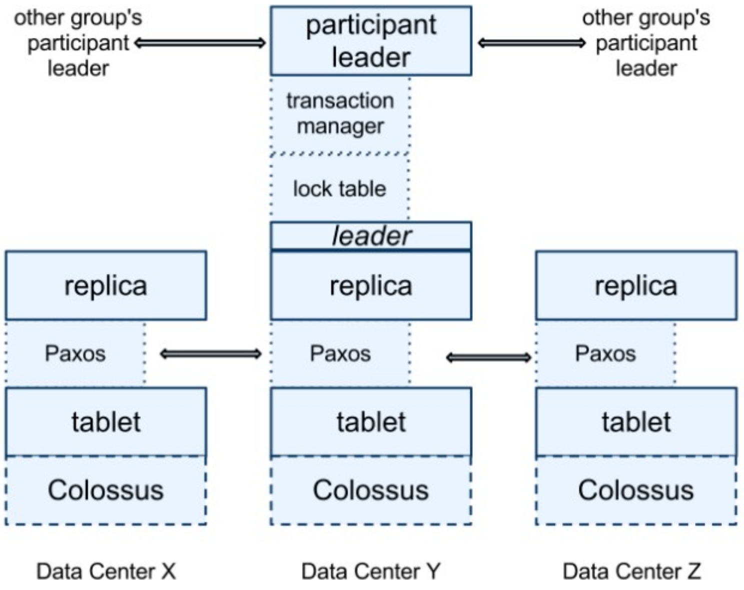
Implementation



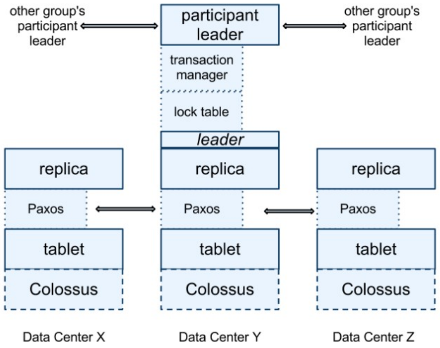
- Spanner deployment is called *universe* (there are only a few universes)
- Spanner is organized as a set of *zones*.
  - Analog of a deployment of Bigtable servers.
  - Unit of administrative deployment.
  - Locations across which data can be replicated.
  - Unit of physical isolation: one or more zones in a DC.
  - 1 zonemaster – [100,1000*n] spanservers, n~10
    - The former assigns data to spanservers;
    - The latter serve data to clients.
  - *Universemaster*: console displaying status of zones; debugging.
  - *Placement driver*: automated movement of data across zones on the timescale of minutes.

Spanserver Software Stack

- How replication and distributed Xacts are layered?
  - Onto BigT-based storage manager.
- Each sserver responsible for 100-1000 tablets
- Tablet = A bag of mappings:
  - (key:string, TS:int64) → string
  - Similar to BigT tablet
  - Multi-version database (not KV)
  - Table is stored
    - B-tree-like files and a WAL (log)
- For replication, each sserver
  - Implements single Paxos state machine on each tablet

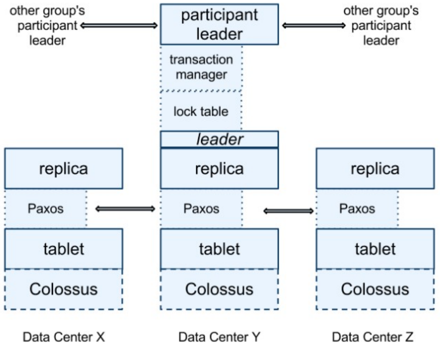


Spanserver Software Stack



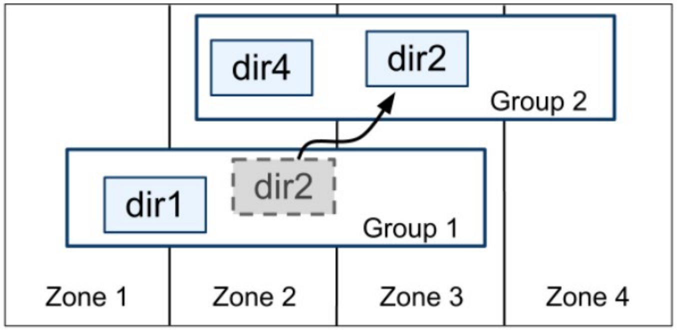
- Paxos implementation:
  - Long-lived leaders with time-based leader leases (10s)
  - Logs every Paxos write twice (tablet's and Paxos log)
  - Writes are applied by Paxos in a timestamp order (see later!)
- Paxos implements consistently replicated bag of mappings
  - KV mapping state of ∀ replica is stored in corresponding tablet.
  - Writes must initiate the Paxos protocol at the *participant leader*.
    - Other participants are slaves.
  - Reads access state directly from the tablet at any replica.
  - Set of replicas is collectively a *Paxos group*.

Spanserver Software Stack



- A leader spanserver
  - Uses a *lock table* to implement concurrency control.
  - Implem. a transaction manager to support distributed Xacts.
  - If Xact involves only one Paxos group, it can bypass TM.
    - Lock tables provide transactionality
  - If Xact involves more than one Paxos group
    - Groups' leaders coordinate to perform 2PC
    - One of the participant groups is chosen as *coordinator leader.*
    - Slaves in that group are called *coordinator slaves*.

Directories and Placement



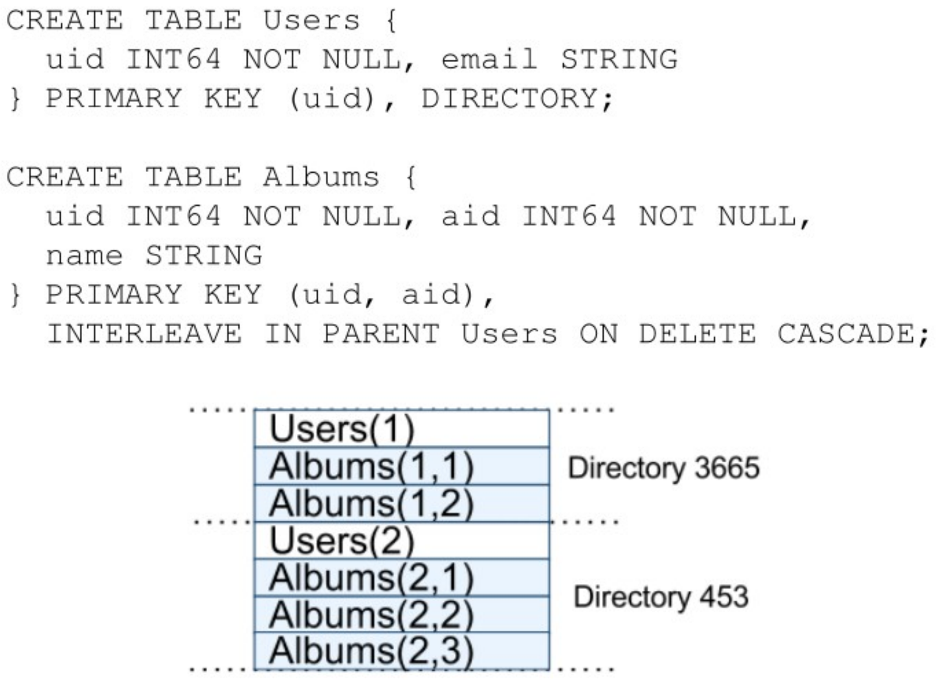
- Bucketing abstraction called a *directory*
  - Set of contiguous keys that share common prefix (~50MB).
  - A directory is the unit of *data placement*.
  - A Paxos group is a set of directories.
  - Movement between Paxos groups in directories
    - to shed load from a Paxos group;
    - to put dirs frequently accessed together into the same group; or
    - to move a directory into a group that is closer to its accessors.
  - Spanner tablet is different from BigT tablet
    - Includes different ranges of KV pairs.
    - To colocate multiple directories that are freq accessed together.
    - Moves the data in the background; not a single Xact.

Spanner Data Model

- Spanner exposes to applications:
  - Semi-relational tables & syncronous replication
    - Lead by the popularity of Megastore (300 apps)
  - SQL-like query language
    - Popularity of Dremel (an interactive data-analysis tool)
  - General-purpose transactions.
    - Lead by lack of cross-row transactions in BigT.
    - 2PC too expensive? Performance or availability problems?
    - Better that apps programmers deal with performance problems.
- Spanner’s data model is not purely relational
  - Every row is named: with ordered set of primary-key columns.
    - This requirement is where Spanner still looks like a key-value store

Spanner Data Model

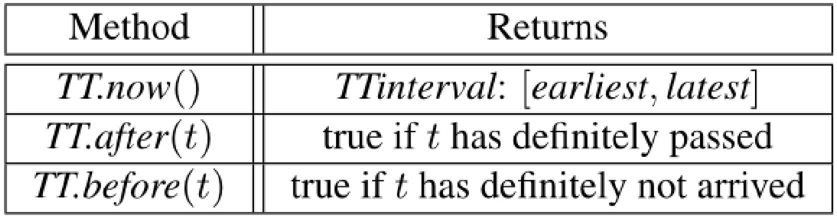
- Example schema:
  - Photo metadata on per-user, per-album basis.
    - Schema language is similar to Megastore’s.
  - Every database must be partitioned by clients into one or more hierarchies of tables.
    - INTERLEAVE IN
    - ON DELETE CASCADE
  - This allows clients to describe the locality relationships that exist between multiple tables.
    - Necessary for good performance in a sharded, distributed database.



```
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;
```

TrueTime



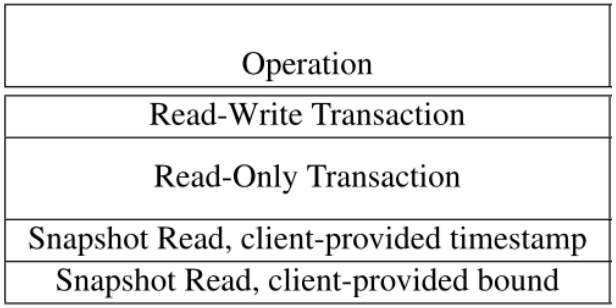
Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(t)	true if t has definitely passed
TT.before(t)	true if t has definitely not arrived

- TrueTime represents time as a *TTinterval*
  - Interval with bounded time uncertainty!
    - Endpoints of a *TTinterval* are of type *TTstamp.*
  - Define the instantaneous error bound as ε.
    - Half of the *TTinterval* width; the average error bound as $\bar{\varepsilon}$.
  - Guaranteed:
    - $tt = TT.now() \Rightarrow tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$
  - Time references: GPS and atomic clocks.
    - Synchronisation among clocks every 30s
    - ε is varies from 1ms to 7ms; $\bar{\varepsilon}$ is about 4ms.
    - Current applied drift rate is set at 200 µs/s (micros).

Concurrency Control

- TrueTime is used to guarantee the correctness properties around concurrency control.
- Those properties are used to implement features:
  1) externally consistent transactions,
  2) lock-free read-only transactions, and
  3) non-blocking reads in the past.
- We will distinguish writes
  - as seen by Paxos from
  - Spanner client writes.

Timestamp Management



Operation
Read-Write Transaction
Read-Only Transaction
Snapshot Read, client-provided timestamp
Snapshot Read, client-provided bound

- Read/Write transaction
  - Uses Paxos and 2PC
- Read-only Xact has performance benefits of snapshot isolation
  - It must be predeclared as not having any writes.
  - Reads execute without locking, at a system-chosen timestamp, so that incoming writes are not blocked.
- A snapshot read is a read in the past
  - Executes without locking.
  - A client specifies a timestamp, or provide an upper bound on TS’s staleness.
  - Read proceeds at any replica that is sufficiently up-to-date.

Paxos Leader Leases

- Paxos uses timed leases to make leadership long-lived (10s)
  - Potential leader sends requests for timed lease votes.
  - When receiving a quorum of votes, leader has a lease.
  - Lease is extended on a successful write.
  - Leader requests lease extensions if near expiration.
  - Disjointness invariant:
    - For each Paxos group, each Paxos leader’s lease interval is disjoint from every other leader’s.

Assigning TS to RW Transactions

- Transact. reads and writes use two-phase locking.
  - TS can be assigned after all locks acquired, but before any locks have been released.
  - Spanner assigns TS to Xact that Paxos assigns to the Paxos write for the Xact commit.
- Spanner depends on the <u>monotonicity invariant</u>:
  - Within each Paxos group, Spanner assigns TS to Paxos writes in monotonically increasing order, even across leaders.
  - This invariant is enforced across leaders by making use of the *disjointness invariant*:
    - Leader must only assign TS within the interval of its leader lease

Assigning TS to RW Transactions

- External-consistency invariant:
  - If the start of T_2 occurs after the commit of T_1, then the commit TS of T_2 must be greater than the commit TS of T_1 .
    - $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$, s_1=TS(T1), s_2=TS(T2), e_i event of T_i
- Commit request at the coordinator leader (abbr. CL)
  - Arrival of commit request for a write T_i is the event e_i^{server}.
  - **start** CL for a write Ti assigns a commit TS s_i no less than the value of TT.now().latest, computed after e_i^{server}
  - **commit wait** CL ensures that clients cannot see any data committed by T_i until TT.after(s_i) is true.
    - Commit wait ensures $s_i < t_{abs}(e_i^{commit})$.

Serving Reads at a Timestamp

- Is replica’s state sufficiently up-to-date to read?
  - To determine this Spanner uses *monotononicity invariant*.
  - Every replica tracks a value at t_{safe} = max TS up-to-date.
  - Replica can satisfy a read at a timestamp t if t <= t_{safe}.
    - Define $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$
    - T_{safe}^{Paxos} = TS of highest-applied Paxos write
      - TS inrease monotonically + writes applied in order => writes will no longer occur at or below T_{safe}^{Paxos}.
    - $T_{safe}^{TM} = \infty$, if no prepared Xacts (Xacts in between 2PC)
    - $T_{safe}^{TM} = \min_i (s_{i,g}^{prepare})-1$, if there are any prepared Xacts
      - State affected by prepared Xacts is indeterminate.
      - Participant leaders (for a group g) for a Xact T_i assigns a prepare TS $s_{i,g}^{prepare}$ to its prepare record.
      - Coordinator leader ensures: Commit TS $s_i >= s_{i,g}^{prepare}$ for all g.

Assigning TS to RO Transactions

- A read-only Xact executes in two phases:
  - Assign a timestamp s_{read} to Xact, and
  - Execute the Xact's reads as snapshot reads at s_{read}.
    - Snapshot reads execute at any replicas sufficiently up-to-date.
- Simple assignment of s_{read} = TT.now().latest
  - Assign at any time after a transaction starts.
  - Preserves external consistency by an argument analogous to that presented for writes.
  - Xact may block at s_{read}, if t_{safe} has not advanced sufficiently.
  - To reduce the chances of blocking, Spanner should assign the oldest TS that preserves external consistency.

Overview with examples

- We now overview the problems and solutions presented previously
  - RW transactions use 2PC gouided by Paxos.
    - Every Paxos write is replicated to sservers in Paxos group.
    - Sservers in a group are in different data centers.
    - Locking guaratees serializability regardless of TS-s.
    - **commit wait** assures monotonicity of TS despite of time drifts.
  - RO transactions use snapshot isolation
    - No locks, no 2PC, no Paxos: reads from the local replica.
    - *Safe time* solution uses <u>monotonicity invariant</u>.
      - RO T2 starts after RW T1, assumed.
      - T1 has to wait until TS(T2) < s_{safe} (maintained by replica).

RO Xact: Overview

- Spanner eliminates two overheads for RO Xact
  - Read from local replicas (avoid Paxos among DC-s).
    - But note local replica may not be up to date!
  - No locks, no 2PC, no transaction manager.
    - Again to avoid cross-DC msgs (Paxos).
    - And to avoid slowing down r/w transactions.
  - Tables 3 and 6 show a 10x latency improvement
    - This is a big deal.
  - How to square this with correctness?
- Let’s see now examples.

RO Xact: Correctness constraints

- Serializable
  - Same results as if Xacts executed one-by-one.
    - Even though they may actually execute concurrently.
- RO Xact must essentially fit between RW Xacts.
  - See writes from prior transactions, not from subsequent.
  - Even though *concurrent* with RW Xacts! And not locking!
- Externally consistent
  - T1 completes before T2 starts, T2 must see T1’s writes.
  - "Before" refers to real (wall-clock) time.
  - Similar to linearizable.
  - Rules out reading stale data.

RO Xact: Why not just read?

- Suppose: two bank transfers, and Xact that reads both.
  - T1: Wx Wy C
  - T2: Wx Wy C
  - T3: Rx Ry
- The results won't match any serial order!
  - Not T1, T2, T3.
  - Not T1, T3, T2.
- We want T3 to see all of T2's writes, or none.
- We want T3's reads to **all** occur at the **same** point relative to T1/T2.

- RO Xact: Snapshot Isolation (SI)

- Synchronize all computers' clocks (to real time).
- Assign every transaction a time-stamp.
  - RW: commit time.
  - RO: start time.
- We want results as if one-at-a-time in TS order.
  - Even if actual reads occur in different order.
- Replica stores multiple TS-ed versions of each record.
  - All of a RW Xact's writes get the same time-stamp.
- An RO Xact’s reads see version as of Xact’s TS.
  - The record version with the highest TS less than Xact’s.

RO Xact: Example with SI

```
                     x@10=9         x@20=8
                     y@10=11        y@20=12
T1 @ 10:  Wx  Wy  C
T2 @ 20:                  Wx  Wy  C
T3 @ 15:             Rx             Ry
```

- Now T3's reads will both be served from the @10 versions.
  - T3 won't see T2's write even though T3's read of y occurs after T2.
- Now the results are serializable: T1 T3 T2.
- The serial order is the same as TS order!
  - Why is it OK for T3 to read the old value of y even though there's a newer value?

RO Xact: Local replica up-to-date?

- Problem:
  - What if T3 reads x from replica that hasn't seen T1's write?
    - Because the replica wasn't in the Paxos majority?
- Solution:
  - Replica "safe time".
  - Paxos leaders send writes in TS order.
  - Before serving a read at time 20, replica must see Paxos write for time > 20.
    - So it knows it has seen all writes < 20.
  - Must also delay if prepared but uncommitted Xacts.
- RO Xacts can read from local replica, usually fast.

RO Xact: Clocks of of sync?

- Problem:
  - What if clocks are not perfectly synchronized?
- Solution:
  - If RW T1 finishes before RO T2 starts, TS1 < TS2.
  - **start** rule:
    - xaction TS = TT.now().latest
    - for RO, at start time
    - for RW, when commit begins
  - **commit wait**, for RW Xact:
    - Before completing commit, delay until TS < TS.now().earliest
    - Guarantees that TS has passed.

RO Xact: Example of clock problem

RW T0 @ 0: Wx1 C
RW T1 @ 10: Wx2 C
RO T2 @ 5: Rx?
(C for commit)

- Problem if RO Xact's TS is too small.
  - T2 reads the version of x at time 0, which was 1.
- But T2 started after T1 committed (in real time).
  - External consistency requires that T2 see x=2.
- So we need a way to deal with incorrect clocks!

RO Xact: Example of clock problem

```
RW T0 @  1: Wx1 C
                         |1-------------10| |11----------------20|
RW T1 @ 10:              Wx2 P         C
                                       |10-----------12|
RO T2 @ 12:                                     Rx?
```

- Scenario: T1 commits, T2 starts, T2 must see T1's writes.
  - We need TS1 < TS2.
  - (P for T1's Prepare, C for T1 finishing Commit)
  - At P, T1 chooses TS1 = TT.now().latest = 10
  - **commit wait** forces C to occur after TS1.
  - T2 starts after C by assumption, and thus after time 10.
  - TS2 = TT.now().latest, which is after current time, which is after 10.
  - So TS2 > TS1 and T2's Rx sees T1's Wx.