Spanner: Google's Globally Distributed Database

Presenter: Nethaniel Berhane

Outline

- What is Spanner? / Why Spanner?
- Implementation
- External Consistency
- Concurrency Control
- Evaluation of Spanserver
- Latency, Throughput, Avaliability
- Related Work / Future Work
- Pros and Cons / Conclusions
- Questions

What is Spanner?

- Distributed multiversion database.
  - Supports SQL Queries and Transactions.
  - Semi-relation data model.
- Manages shards of replicated data across data centres for high availability.
- Features:
  - Replication can be controlled by clients for load balancing and failure responses.
  - Provides externally consistent reads and writes.
  - Provides globally consistent reads.

Why Spanner ?

Google's Previous Works:

Bigtable (2008): wide-column, key-value NoSQL database service

- Google received complaints for its performance in applications requiring strong consistency guarantees for geo-replicated sites.
- It is also difficult to use for complex, evolving schemas.

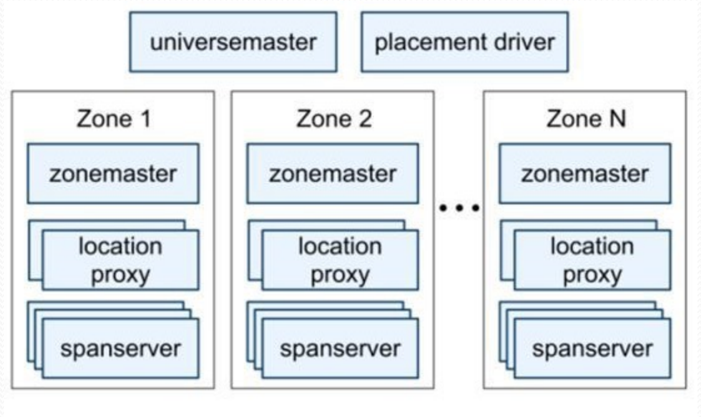
Megastore (2011): storage system developed to meet requirements of today's interactive online services

- It supports synchronous replication by providing a semi-relational data model.
- But it offers low consistency across global data replication and had poor write throughput

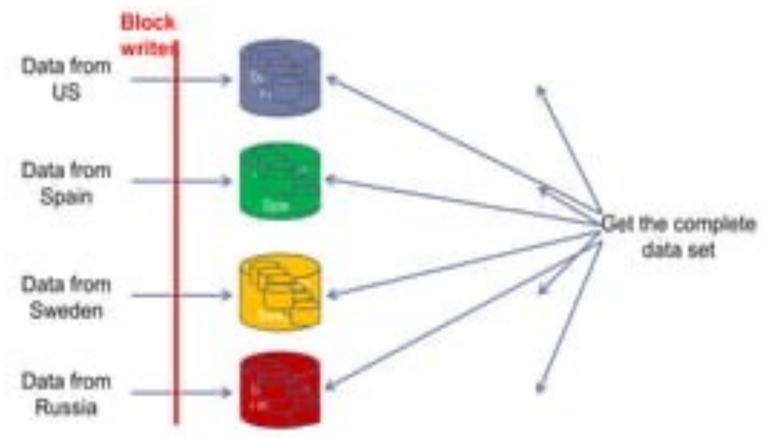
Implementation

Spanner server organization

Universe : Deploys Spanner
Zones : Defines locations across which data can be replicated.
It is unit of physical isolation.
Zone master: Assigns data to spanserver
Spanserver: Serve data to clients
Placement driver: Automatically migrates data.

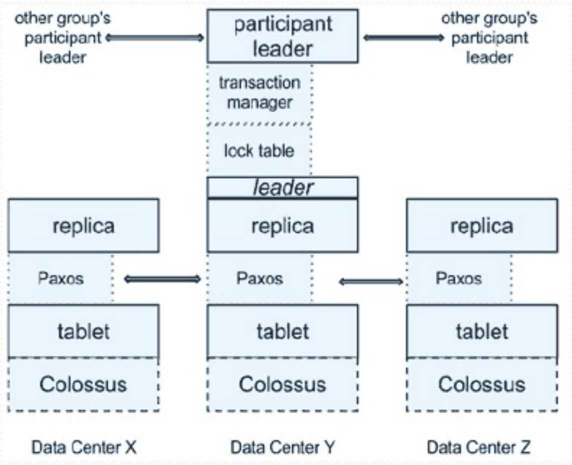


Serving data from multiple datacenters



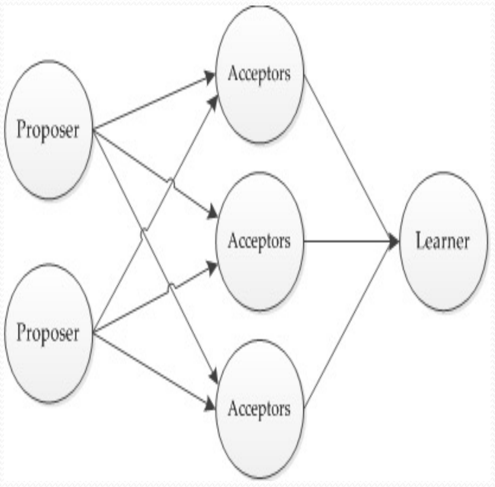
Spanner software stack

Tablet: Each spanserver has 100's of instances of key-value data structure called tablet.
(Key:string,timestamp:int64) ->string
Colossus: The state of the tablet is stored in B-Tree like files
Replication:
1) Paxos state machine is used to maintain consistent replicated mappings within a Paxos group. The Writes will initiate it at its leader and Reads can access any of the tablet.
2) Between Paxos group transaction manager will enforce two phase commit for distributed transaction; with one Paxos leader being the coordinator and other leaders under it.



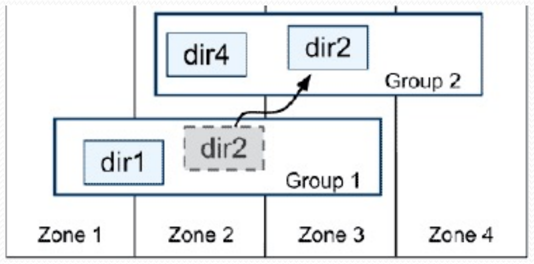
Consensus Protocol

- Paxos:
  - The client sends request to the proposer (leader).
  - Proposer send prepare request to group of acceptors.
  - Acceptors will promise the proposer if it has not received values higher than the proposer.
  - If proposer got enough promise from acceptors it will send accept request
  - If the acceptor accepts as it has not received any higher bids, the learners will notify clients.



Directories

- Directories:
  - Set of contiguous keys that share a common prefix
  - Unit of data placement
  - For load-balancing support for Movedir operation



Data Model

- Semi-relational tables to support Query language and Transactions.

UID	Email
1	XXX

UID	Albums
1	Album1
1	Album2
1	Album3

```
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;
```

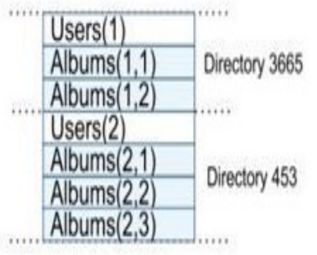


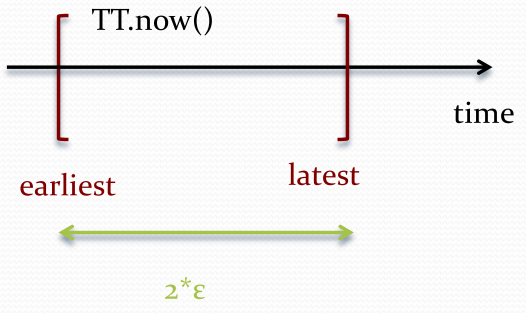
Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by INTERLEAVE IN.

External Consistency

- External consistency is equivalent to linearizability in which the commit order adheres to global clock.
- Property: If a transaction T1 commits before another transaction T2 starts, then T1's timestamp is smaller than T2's.
- Enabler: True time API provides interval based global time to establish the above property.
- Application: This helps to implement lock free distributed read transactions.

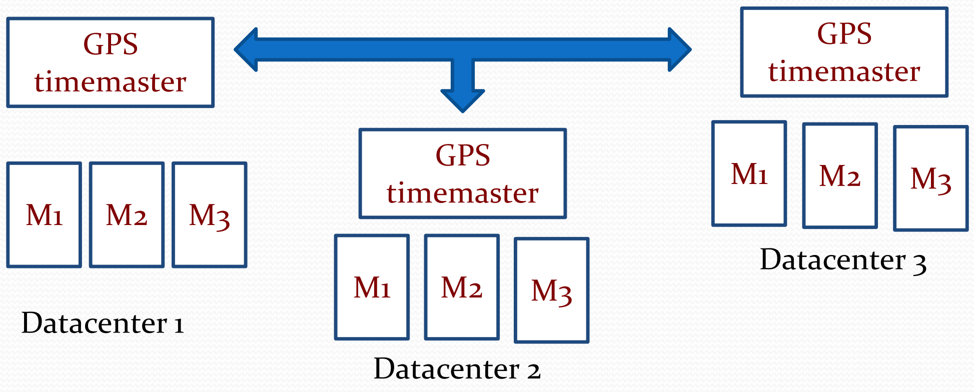
True Time

- Global wall clock time with bounded uncertainty.



1) For any invocation of TT.now(), tt.earliest $< t_{abs}$ (e) < tt.latest.
2) TT.after(t) will return true if t > tt.latest.
3) TT.before(t) will return true if t < tt.earliest.
4) ε is the average error bound.

True Time Architecture



Compute reference [earliest, latest] = now ± ε
now = reference now + local-clock offset
ε = reference ε + worst-case local-clock drift

(the worst case for when a clock does not run at exactly the same rate as a reference clock)

Concurrency Control

Timestamps for RW transaction

- Read-write transaction use two phase commit.
- Let start, commit request, and commit events be e_i^{start}, e_i^{server}, and e_i^{commit} ; and the commit timestamp of transaction Ti by timestamp Si

  Invariant: if $t_{abs}(e_1^{\text{commit}}) < t_{abs}(e_2^{\text{start}})$, then timestamp $S1 < S2$
- Start: Coordinator leader for write Ti assigns timestamp Si no less than TT.now().latest, computed after event e_i^{server}.
- Commit wait: Coordinator leader ensures clients can't see data committed by Ti before TT.after(Si) is true.

Serving reads at a timestamp

- Each replica tracks safe time t_{safe}, which is the maximum timestamp at which it is up to date. Replica can read at t

  if $t <= t_{safe}$
- $t_{safe} = \min(t_{Paxos\text{-}safe}, t_{TM\text{-}safe})$
- $t_{Paxos\text{-}safe}$ = Maximum write timestamp by Paxos. Paxos write times increase monotonically, so writes will not occur at or below $t_{Paxos\text{-}safe}$
- $t_{TM\text{-}safe}$ = Transaction manager safe time. Its value is infinity if there are no prepared transactions. Else it will be the minimum of the lower bound on prepared transaction timestamp.

Timestamp for RO transaction

- To execute a read-only transaction, pick timestamp S_{read}, then execute as snapshot reads at S_{read} at sufficiently up to date replicas.
- Picking TT.now().latest after the transaction start will definitely preserve external consistency, but may block for unnecessarily long amounts of time while waiting for t_{safe} to advance.
- Choose the oldest timestamp that preserves external consistency: LastTS().

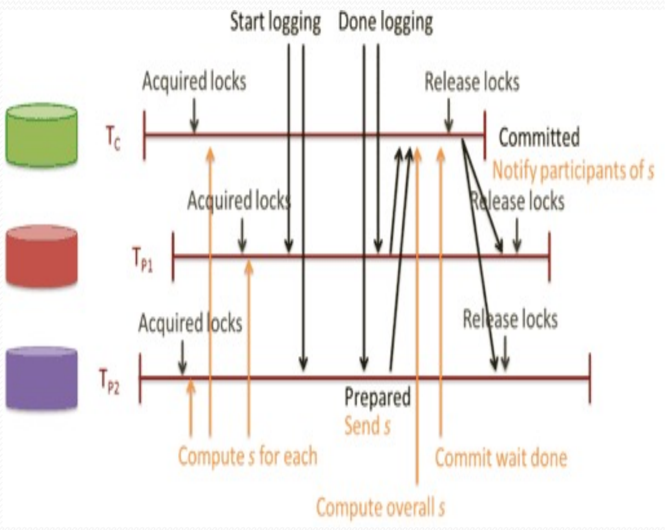
Paxos leader Management

- Spanner's Paxos implementation used timed (10 second) leader leases to make leadership long lived
- Candidate becomes leader after receiving quorum of timed lease votes.
- The current leader will send request for extension before its lease ends.
- Each replica r grants a lease time at e(Grant) which happens after it receives lease request e(Receive). The lease ends at t_e = TT.now().latest + lease_length; it is computed after every lease request e(Receive).
- The replica will not grant another lease until TT.after(t_e) is true.
- So any other potential leaders e(Send) + lease_length < e(Receive) +lease_length
- Shorter lease time will increase lease-renewal traffic.

Read Write transactions

- Client issues reads to leader replicas of appropriate groups. These acquire read locks and read the most recent data.
- Once reads are completed and writes are buffered (at the client), client chooses a coordinator leader and sends the identity of the leader along with buffered writes to participant leaders.
- Non-coordinator participant leaders:
  - ❑ Acquire write locks
  - ❑ Choose a prepare timestamp larger than any previous transaction timestamp's logs and prepare a record in Paxos.
  - ❑ Notify coordinator of chosen timestamp.

- Coordinator leader
  - Acquires locks
  - Picks a commit timestamp s greater than TT.now().latest, greater than or equal to all participant prepared timestamps, and greater than any previous transaction timestamps assigned by the leader.
  - Logs will commit record in Paxos and waits until TT.after(s) to allow replicas to commit T and obeys commit wait.
  - Send timestamp to all participants and they will release the locks.
  - After the required waits, the locally assigned timestamps are guaranteed to agree with the wall-clock (externalized time)



Refinements

- **Problem 1**: A prepared transaction blocks $t_{TM\text{-}safe}$ from advancing. What if the prepared transactions don't conflict with the read?
  Solution: Augment $t_{TM\text{-}safe}$ with mappings from key ranges to these prepared timestamps. The lock table will do a fine-grained block on the transactions which have conflicts with the mapped keys range.

- **Problem 2:** LastTS() also faces similar problem when assigning a timestamp to a read-only
  Solution: - Similar solution of fine grained safe time can be used.
- **Problem3:** $t_{Paxos\text{-}safe}$ cannot advance without Paxos writes, so snapshots reads at t cannot proceed at groups whose last Paxos write occurred before $t_{TM\text{-}safe}$.
  Solution: Paxos leaders instead advance $t_{Paxos\text{-}safe}$ by keeping track of the timestamp above for which future Paxos writes will occur. Advances occur every 8 seconds by default, so in the worst case, replicas can serve reads no more recently than 8 seconds ago.

Evaluation of Spanserver

Spanserver on timeshared machines

- 4GB RAM, 4 cores (AMD Barcelona 2200MHz)
- Clients ran on separate machines
- Clients and zones are very near.

Test database is created with 50 Paxos groups, 2500 directories, operations consisted of 4KB reads and writes.

Latency and Throughput

- Latency depends on slowest quorum machine. As replica increases, the latency increases as there are more replicas to commit.
- Read-only transaction: throughput increases as the number of replicas increases since there are more span servers

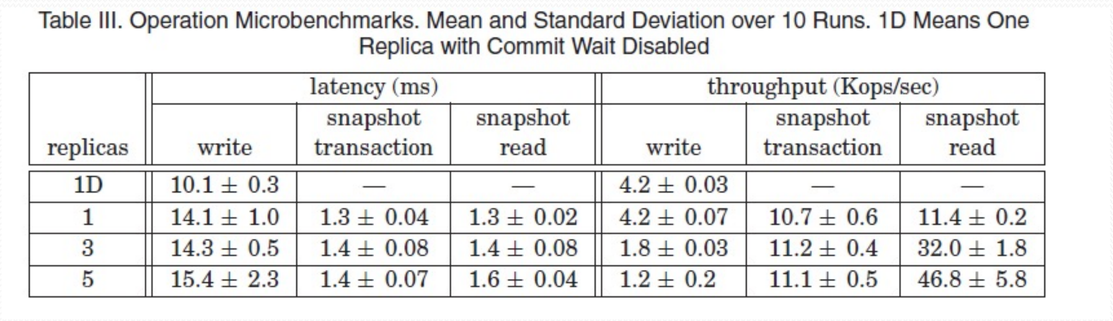


Table III. Operation Microbenchmarks. Mean and Standard Deviation over 10 Runs. 1D Means One Replica with Commit Wait Disabled

	latency (ms)			throughput (Kops/sec)		
replicas	write	snapshot transaction	snapshot read	write	snapshot transaction	snapshot read
1D	10.1 ± 0.3	—	—	4.2 ± 0.03	—	—
1	14.1 ± 1.0	1.3 ± 0.04	1.3 ± 0.02	4.2 ± 0.07	10.7 ± 0.6	11.4 ± 0.2
3	14.3 ± 0.5	1.4 ± 0.08	1.4 ± 0.08	1.8 ± 0.03	11.2 ± 0.4	32.0 ± 1.8
5	15.4 ± 2.3	1.4 ± 0.07	1.6 ± 0.04	1.2 ± 0.2	11.1 ± 0.5	46.8 ± 5.8

Availability

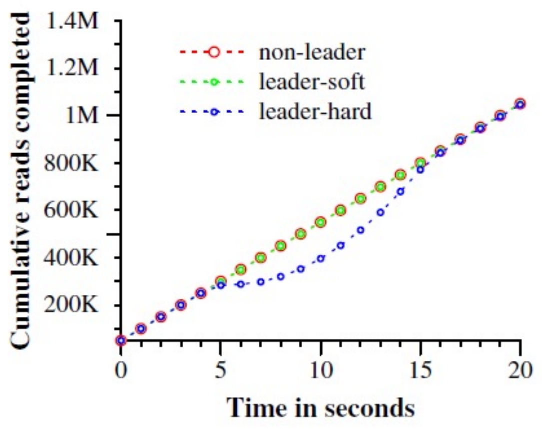


Fig. 5. Effect of killing servers on throughput.

Use of Spanner in F1

F1:

Google advertising backend . This is based on a mySQL database and data are manually sharded.

Spanner was chosen because:

- Automatically shard data
- Provides synchronous replication and strong transactional semantics.

Related Work

Some the related work upon which Spanner is built:

MegaStore: It provides consistent replication across data centres.
DynamoDB: It presents key-value interface, and only replicates within a region
Scatter: It provides Key-value store for P2P systems
Walter: It provides snapshot isolation within data-centres.
Calvin: It collects and orders transactions thus eliminates lock overhead.
Granola: Transaction infrastructure that supports specific types of transactions to avoid locking in snapshot isolation
Farsite: It derives clock uncertainty related to trusted clock reference.

Future Work

- Adapt and enhance the Spanner schema language .
- Automatic load-based resharding to adapt to changes in the rate of data-flowing through stream.
- Increase performance for complex queries as it is based on key-value access.
- Reduce ε below 1ms in the True Time API.

Opinions

Pros:

- Spanner innovates around the area of Time to improve performance in: External consistency/linearizability, Distributed databases, Concurrency control, and Replication.
- True Time API allows to build strong transactional semantics in globally distributed environment.

Cons:

- Alternative to two-phase commit protocol to avoid potential blocking issues and performance overheads?
- The snapshots are not cleaned

Questions?