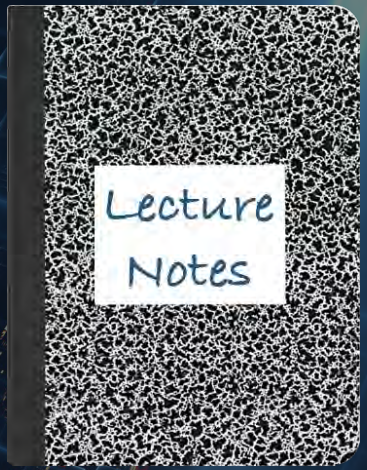
CS 417 – DISTRIBUTED SYSTEMS

Week 9: Distributed Databases

Part 3: Google Spanner



Paul Krzyzanowski

Spanner
Google's successor to Bigtable … (sort of)

Spanner

Take Bigtable and add:

- Familiar SQL-like multi-table, row-column data model
  - One primary key per table
- Synchronous replication (Bigtable was eventually consistent)
- Transactions across arbitrary rows

Spanner

- **Globally distributed multi-version database**
- ACID (general purpose transactions)
- Schematized tables (Semi-relational)
  - Built on top of a key-value based implementation
  - SQL-like queries
- Lock-free distributed read transactions

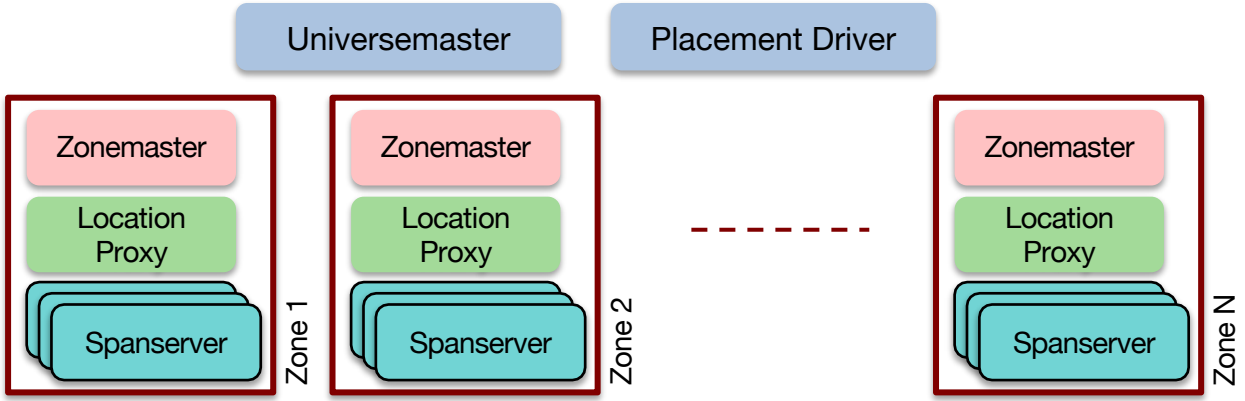
Goal: make it easy for programmers to use

Working with eventual consistency & merging data is hard ⇒ don't make developers deal with it

Data Storage

- Tables sharded across rows into *tablets* (like bigtable)
- Tablets are stored in spanservers
- 1000s of spanservers per zone
  - Collection of servers

- Zonemaster
  Allocates data to spanservers
- Location proxies
  Locate spanservers that have the needed data
- Universemaster
  Tracks status of all zones
- Placement driver
  Transfers data between zones



Data Storage

Universe: holds one or more databases

Database: holds one or more tables

Table: rows & columns

Shards (tablets): pieces of tables
Replicated synchronously via Paxos

Data in table is versioned & has a timestamp

Transactions across shards use two-phase commit

Directory: "bucket" – set of contigious keys with a common prefix
Unit of data movement

Transactions

- ACID properties
  - Elected transaction manager for distributed transactions
  - **Two-phase commit protocol** used outside of a group of replicas
- Transactions are serialized: strict 2-phase locking used

1. Acquire all locks

   – do work –

2. **Get a commit timestamp**
3. Log the commit timestamp via Paxos consensus to majority of replicas
4. Do the commit
   - Apply changes locally & to replicas
5. Release locks

Even 2-Phase locking can be slow

Read-write transactions

Spanner uses strict two-phase locking with *read locks* and *write locks*

- *Writes in read/write transactions*
  ⇒ ***two-phase locking***
- *Reads in read/write transactions*
  ⇒ ***wound-wait algorithm***
  prevents deadlocks

Read-only transactions & Snapshot reads

Multiversion concurrency

- **Snapshot isolation:**
  provide a view of the database up to a point in time
- No locking needed – great for long-running reads (e.g., searches)
  - Snapshot reads = read versions < user-chosen time
  - Read-only transactions: read versions of data < transaction start time
- Because ***you are reading the version of data before a specific point in time***, results are consistent

We need ***commit timestamps*** that will enable meaningful snapshots

Getting good commit timestamps

- Vector clocks work
  - Pass along the current server's notion of time with each message
  - Receiver updates its concept of time (if necessary)

- But are not feasible in large systems
  - Pain in HTML (have to embed a large vector timestamp in the HTTP transaction)
  - Doesn't work if you introduce things like phone call logs

- **Spanner: use physical timestamps**
  - If T_1 commits before T_2 then T_1 *must* get a smaller timestamp
  - Commit order matches global wall-time order

External consistency

If a transaction T_1 commits before another transaction T_2 starts, then T_1's commit timestamp must be smaller than that of T_2. If the results of T_2 are visible to the user, then the results of T_1 must also be visible, even if the transactions did not conflict.

TrueTime API

Remember: we can't know global time across servers!

Global wall-clock time = time + interval of uncertainty

`TT.now().earliest` = time guaranteed to be ≤ current time

`TT.now().latest` = time guaranteed to be ≥ current time

Each data center has a GPS receiver & atomic clock

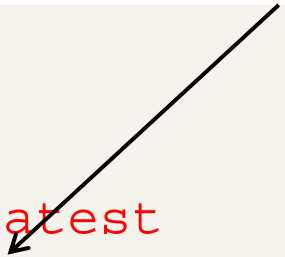
- Atomic clock synchronized with GPS receivers
  - Validates data from GPS receivers
- Spanservers periodically synchronize with time servers
  - Know uncertainty based on interval
  - Synchronize ~ every 30 seconds: clock uncertainty < 10 ms

Commit Wait

We don't know the *exact* time

… but we can wait out the uncertainty and finish the commit when the commit timestamp is definitely in the past

average worst-case wait is ~10 ms



1. Acquire all locks

   – do work –

2. Get a commit timestamp: `t = TT.now().latest`
3. **Commit wait**: **wait until** `TT.now().earliest > t`
4. Commit
5. Release locks

Integrate replication with concurrency control

1. Acquire all locks
   – *do work* –
2. Get a commit timestamp: `t = TT.now().latest`
3. (a) Start consensus for replication
   (b) Commit wait (in parallel)

   } **Make the replicas & wait for all to finish**
4. Commit
5. Release locks

Spanner Summary

Features

- Semi-relational database of tables
  - Supports externally consistent distributed transactions
  - No need for users to deal with eventual consistency
- Multi-version database
- Synchronous replication
- Scales to millions of machines in hundreds of data centers
- SQL-based query language

Deployments

- Used in F1, the system behind Google's Adwords platform
- Likely used in YouTube, Drive, and Gmail
- Available as a public service via Cloud Spanner

Are we breaking the rules?

- **Global ordering of transactions**
  - ***Systems cannot have globally synchronized clocks***
  - But we can synchronize closely enough that we can have a transaction wait until a specific time has passed
- **CAP theorem**
  - ***We cannot offer Consistency + Availability + Partition tolerance*** (CAP Theorem)
  - Spanner is a CP system – if there is a partition, Spanner chooses C over A
  - In practice, partitions are rare: ~8% of all failures of Spanner
    - Spanner uses Google's private global network, not the Internet
    - Each data center has at least three independent fiber connections
  - In practice, users can feel they have a CA system – high availability AND consistency!

https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45855.pdf

Spanner Conclusion

- ACID semantics not sacrificed
  - Life gets easy for programmers
  - Programmers don't need to deal with eventual consistency
- Wide-area distributed transactions built-in
  - Bigtable did not support atomic multi-table or multi-row transactions
  - Programmers had to write their own, which could be buggy
  - Easier if programmers don't have to get 2PC right
- Clock uncertainty is known
  - The system can wait it out
  - Users get **external consistency** – transaction order = real time order

The End