Week 9: Distributed Databases
Part 3: Google Spanner
Spanner
Google’s successor to Bigtable … (sort of)
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

Goal: make it easy for programmers to use
Working with eventual consistency & merging is hard ⇒ don't make developers deal with it
Data Storage

- Tables sharded across rows into *tablets* (like Bigtable)
- Tablets stored in *spanservers*
- 1000s of spanservers per zone
  - Collection of servers – can be run independently

- **Zonemaster**: Allocates data to spanservers
- **Location proxies**: Locate spanservers with needed data
- **Universemaster**: Tracks status of all zones
- **Placement driver**: Transfers data between zones
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 contiguous 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
Read-write transactions
Spanner uses two-phase locking with *read locks* and *write locks*

- **Writes in read/write transactions** ⇒ *two-phase locking*
- **Reads in read/write transactions** ⇒ *wound-wait concurrency control*

Read-only transactions
Reads versions of data < current time

Snapshot reads

*Multiversion concurrency*

- **Snapshot isolation:**
  provide a view of the database for transactions up to a point in time
  - Read old versions of data at a chosen past time without getting a lock
    - Great for long-running reads (e.g., searches)
  - Because you are reading before a specific point in time
  - Results are consistent

We need *commit timestamps* that will enable meaningful snapshots

Even 2-Phase locking can be slow
Getting good commit timestamps

• Vector clocks work
  – Pass along 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 vector timestamp in 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
Remember: we can’t know global time across servers!

- **Global wall-clock time** = time + interval of uncertainty
  - `TT.now().earliest` = time guaranteed to be $\leq$ current time
  - `TT.now().latest` = time guaranteed to be $\geq$ current time

- Each data center has a GPS receiver & atomic clock
- Atomic clock synchronized with GPS receivers
  - Validates 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

average worst-case wait is ~10 ms

1. Acquire all locks
   – do work –
2. Get a commit timestamp: \( t = \text{TT.now().latest} \)
3. **Commit wait**: wait until \( \text{TT.now().earliest} > t \)
4. Commit
5. Release locks
Integrate replication with concurrency control

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

Make the replicas & wait for all to finish
Deadlock

Spanner uses wound-wait to deal with deadlock

• Old transaction wants a lock held by a younger transaction
  − Old process aborts the younger transaction
• Young transaction wants a lock held by an older transaction
  − Young transaction waits

Only permit younger transactions to wait on resources held by older transactions.
Spanner Summary

• Semi-relational database of tables
  – Supports externally consistent distributed transactions
  – No need for users to try deal with eventual consistency

• Multi-version database

• Synchronous replication

• Scales to millions of machines in hundreds of data centers

• SQL-based query language

• Used in F1, the system behind Google’s Adwords platform

• May be used in Gmail & Google search and others…
Are we breaking the rules?

• **Global ordering of transactions**
  – *Systems cannot have globally synchronized clocks*
  – But we can synchronize closely enough that we can wait until we are sure a specific time has passed

• **CAP theorem**
  – *We cannot offer Consistency + Availability + Partition tolerance*
  – 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

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 distributed transactions
  – Programmers had to write their own
  – Easier if programmers don’t have to get 2PC right

• Clock uncertainty is known to programmers
  – You can wait it out
The End