Distributed Systems

21. Spanner

Paul Krzyzanowski
Rutgers University
Fall 2014
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

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 – Used by clients to locate spanservers that hold the data they need
- *Universemaster* – status of all zones
- *Placement driver* – transfers data between zones

---

[Diagram showing the structure of data storage with labeled components: Universemaster, Zonemaster, Location Proxy, Spanserver.]
Data Storage

• Universe holds 1 or more databases
  – Database holds 1 or more tables
  – Table = arbitrary number of rows and columns
    • Table storage may be interleaved
    • All data in a table has version information (timestamp)

• Shards are replicated
  – Synchronous replication via Paxos

• Transactions across shards use 2-phase commit

• Directory = set of continuous keys
  – Unit of data allocation
  – Granularity for data movement between Paxos groups
    • Done in background
Transactions

• ACID properties

• 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 to majority of replicas

4. Do the commit
   – Apply changes locally & to replicas on commit

5. Release locks
2-Phase locking can be slow

Use *read locks* and *write locks*

But
- *read locks* block behind *write locks*
- *write locks* block behind *read locks*

**Multiversion concurrency** to the rescue!
- Take a snapshot of the database for transactions up to a point in time
- You can read old data without getting a lock
  - Great for long-running reads (e.g., searches)
- Because you are reading ≤ a point in time
  - Results are consistent

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

• Not feasible in large systems
  – Pain in HTML (have to embed timestamp in HTTP transaction)
  – Doesn’t work if you introduce things like phone call logs

• Goal: use physical timestamps
  – If $T_1$ commits before $T_2$, $T_1$ must get a smaller timestamp
  – Commit order matches global wall-time order
TrueTime

• 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 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

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

*average worst-case wait is \( \sim 10 \text{ ms} \)*
1. Acquire all locks
   – *do work* –
2. Get a commit timestamp: \( t = \text{TT.now()}.\text{latest} \)
3. (a) Start consensus for replication
   (b) *Commit wait* (in parallel)
4. Commit
5. Release locks
Integrate commit wait with 2-phase commit

• 2-phase commit used across shards

1. Acquire all locks
   – do work –
2. 2PC coordinator gets a commit timestamp: \( t = TT.now().latest \)
3. Use Paxos protocol to commit
   – timestamp included in the Paxos proposal
   – timestamp conveyed to all participants
4. Commit
5. Release locks

November 17, 2014

2014 Paul Krzyzanowski
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