Google BigTable, Spanner
Prof. Chris Clifton
23 April 2021

Google BigTable

• Simple data model

• Tables distributed
  – “row keys”
• Transactional consistency only on a per-row basis
Google Bigtable

- Multi-version
  - Each row timestamped
- Three-level location hierarchy
  - Claim: “B-tree like”

Google Spanner

- SQL-based query language
  - MapReduce based execution
- (Dynamically) replicated data
  - High availability
- Read/write consistency
  - Timestamp based serializability
Google Spanner Data Replication

- Data divided into Zones
  - Replication across zones
  - May be thousands of servers in a zone
  - Placement in a zone dynamic (location proxies)
  - Similar to BigTable (Servers)
- Internally: tablet abstraction
  - Maps (key, timestamp) → string
- Lock Table at each replica

Overview

- Feature: Lock-free distributed read transactions
- Property: External consistency of distributed transactions
  - First system at global scale
- Implementation: Integration of concurrency control, replication, and 2PC
  - Correctness and performance
- Enabling technology: TrueTime
  - Interval-based global time
Concurrency Control

• Three types of transactions
  – Read-write
  – Snapshot Transactions
    • Pre-declared as having no writes
  – Snapshot reads
    • Weak consistency guarantee
    • “sufficiently up to date”
• All data timestamped

Consistency: Read/Write

• Read-write uses strict two-phase locking
  – Locks held until commit
• Timestamp assigned after all locks acquired
  – Timestamps assigned by “leader” at each site
  – All writes have that timestamp
• Replicas track “safe time” – maximum timestamp at which a replica is up-to-date
  – Infinite if no transactions operating on object
  – Otherwise timestamp of first completed (but not committed) transaction
• Serializability is timestamp order
  – If $T_2$ starts after $T_1$ commits, must have later timestamp
Consistency: Reads

- Read transactions assigned a timestamp
  - Only read data written before that timestamp
  - Can’t read data if timestamp > safe time
    \[ T_{\text{write}} < T_{\text{read}} < T_{\text{safe}} \]

- What to assign as a timestamp?
  - Current time means replicas may be past “safe”
  - Can assign “old” timestamps, more replicas are okay
  - Read transactions may serialize before they actually start

Version Management

- Transactions that write use strict 2PL
  - Each transaction \( T \) is assigned a timestamp \( s \)
  - Data written by \( T \) is timestamped with \( s \)

<table>
<thead>
<tr>
<th>Time</th>
<th>&lt;8</th>
<th>8</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>My friends</td>
<td>[X]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>My posts</td>
<td></td>
<td>[P]</td>
<td></td>
</tr>
<tr>
<td>X’s friends</td>
<td>[me]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>
Timestamps, Global Clock

- Strict two-phase locking for write transactions
- Assign timestamp while locks are held

```
Acquired locks
T

Release locks

Pick s = now()
```

Timestamp Invariants

- Timestamp order == commit order
- Timestamp order respects global wall-time order
TrueTime

- "Global wall-clock time" with bounded uncertainty

T

\[ TT.now() \]

earliest

latest

\[ 2^* \epsilon \]

Timestamps and TrueTime

Acquired locks

Release locks

Pick \( s = TT.now().latest \) \quad \text{or} \quad \text{Save until } TT.now().earliest > s

Commit wait

average \( \epsilon \) | average \( \epsilon \)
Some other details…

- Wound-wait used for deadlock prevention of write transactions
  - No deadlocks with read-only transactions (why?)
- Uses 2-phase commit to handle distributed transactions
- Writes only occur at commit
  - Not visible before commit

Commit Wait and Replication

- Start consensus
- Acquire locks
- Pick s
- Achieve consensus
- Notify slaves
- Release locks
- Commit wait done
Commit Wait and 2-Phase Commit

- **Start logging**
- **Done logging**
- **Acquired locks**
- **Release locks**
- **Committed**
- **Notify participants of s**
- **Release locks**
- **Prepare s**
- **Send s**
- **Commit wait done**
- **Compute overall s**
- **Compute s for each**