Spanner Storage insights

COS 518: Advanced Computer Systems
Lecture 6
Michael Freedman

2PL & OCC = strict serialization

- Provides semantics as if only one transaction was running on DB at time, in serial order
  + Real-time guarantees
- 2PL: Pessimistically get all the locks first
- OCC: Optimistically create copies, but then recheck all read + written items before commit

Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp. Allocate correct version to reads.
- Prior example of MVCC:

Multi-version concurrency control diagram:
- Write Request
- Dirty Read
- Clean Read

Diagam showing HEAD and TAIL with replicas and versioning.
Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp. Allocate correct version to reads.
- Unlike 2PL/OCC, reads never rejected
- Occasionally run garbage collection to clean up

MVCC Intuition

- Split transaction into read set and write set
  - All reads execute as if one “snapshot”
  - All writes execute as if one later “snapshot”

- Yields snapshot isolation < serializability

Serializability vs. Snapshot isolation

- Intuition: Bag of marbles: ½ white, ½ black
- Transactions:
  - T1: Change all white marbles to black marbles
  - T2: Change all black marbles to white marbles
- Serializability (2PL, OCC)
  - T1 → T2 or T2 → T1
  - In either case, bag is either ALL white or ALL black
- Snapshot isolation (MVCC)
  - T1 → T2 or T2 → T1 or T1 || T2
  - Bag is ALL white, ALL black, or ½ white ½ black

Timestamps in MVCC

- Transactions are assigned timestamps, which may get assigned to objects those txns read/write
- Every object version $O_V$ has both read and write TS
  - ReadTS: Largest timestamp of txn that reads $O_V$
  - WriteTS: Timestamp of txn that wrote $O_V$
Executing transaction T in MVCC

- Find version of object O to read:
  - # Determine the last version written before read snapshot time
  - Find \( O_v \) s.t. max \{ WriteTS(O_v) | WriteTS(O_v) <= TS(T) \}
  - ReadTS(O_v) = max(TS(T), ReadTS(O_v))
  - Return \( O_v \) to T

- Perform write of object O or abort if conflicting:
  - Find \( O_v \) s.t. max \{ WriteTS(O_v) | WriteTS(O_v) <= TS(T) \}
  - # Abort if another T' exists and has read O after T
  - If ReadTS(O_v) > TS(T)
    - Abort and roll-back T
  - Else
    - Create new version \( O_w \)
    - Set ReadTS(O_w) = WriteTS(O_w) = TS(T)

Digging deeper

- Notation
  - TS = 3
  - TS = 4
  - TS = 5

- Write creates version 1 with WriteTS = 3
- Read of version 1 returns timestamp 3

Digging deeper

- Notation
  - W(1) = 3: Write creates version 1 with WriteTS = 3
  - R(1) = 3: Read of version 1 returns timestamp 3

- Find v such that max WriteTS(v) <= (TS = 4)
  - v = 1 has (WriteTS = 3) <= 4
  - If ReadTS(1) > 4, abort
  - 3 > 4: false
  - Otherwise, write object
Digging deeper

Notation

\[
\begin{align*}
W(1) = 3: & \text{ Write creates version 1 with WriteTS = 3} \\
R(1) = 3: & \text{ Read of version 1 returns timestamp 3}
\end{align*}
\]

\[
\begin{array}{ccc}
\text{txn} & \text{txn} & \text{txn} \\
TS = 3 & TS = 4 & TS = 5
\end{array}
\]

W(1) = 3 \quad W(2) = 5
R(1) = 3 \quad R(2) = 5

Find v such that max WriteTS(v) <= (TS = 4)

\[\Rightarrow v = 1 \text{ has (WriteTS = 3) <= 4}\]

If ReadTS(1) > 4, abort

\[\Rightarrow 3 > 4: \text{ false}\]

Otherwise, write object

\[
\begin{align*}
\text{BEGIN Transaction} \\
tmp = \text{READ}(O) \\
\text{WRITE} (O, \text{tmp} + 1) \\
\text{END Transaction}
\end{align*}
\]

Digging deeper

Notation

\[
\begin{align*}
W(1) = 3: & \text{ Write creates version 1 with WriteTS = 3} \\
R(1) = 3: & \text{ Read of version 1 returns timestamp 3}
\end{align*}
\]

\[
\begin{array}{ccc}
\text{txn} & \text{txn} & \text{txn} \\
TS = 3 & TS = 4 & TS = 5
\end{array}
\]

W(1) = 3 \quad W(2) = 5
R(1) = 3 \quad R(2) = 5

Find v such that max WriteTS(v) <= (TS = 5)

\[\Rightarrow v = 1 \text{ has (WriteTS = 3) <= 5}\]

If ReadTS(1) > 5, abort

\[\Rightarrow 5 > 5: \text{ true}\]

Otherwise, write object

\[
\begin{align*}
\text{BEGIN Transaction} \\
tmp = \text{READ}(O) \\
\text{WRITE} (O, \text{tmp} + 1) \\
\text{END Transaction}
\end{align*}
\]
Digging deeper

Notation

\[ W(1) = 3: \text{Write creates version 1 with WriteTS} = 3 \]
\[ R(1) = 3: \text{Read of version 1 returns timestamp} 3 \]

BEGIN Transaction
\[ \text{tmp = READ(O)} \]
\[ \text{WRITE (P, tmp + 1)} \]
END Transaction

Find \( v \) such that \( \max (\text{WriteTS}(v)) \leq (\text{TS} = 4) \)
\[ \Rightarrow v = 1 \text{ has (WriteTS} = 3) \leq 4 \]
Set \( R(1) = \max (4, R(1)) = 5 \)
Then write on \( P \) succeeds as well

Consider partitioned data over servers

- Why not just use 2PL?
  - Grab locks over entire read and write set
  - Perform writes
  - Release locks (at commit time)

Distributed Transactions

Consider partitioned data over servers

- How do you get serializability?
  - On single machine, single COMMIT op in the WAL
  - In distributed setting, assign global timestamp to \( \text{txn} \)
    (at sometime after lock acquisition and before commit)
    - Centralized \( \text{txn} \) manager
    - Distributed consensus on timestamp (not all ops)
Strawman: Consensus per txn group?

- Single Lamport clock, consensus per group?
  - Linearizability composes!
  - But doesn’t solve concurrent, non-overlapping txn problem

Google’s Setting

- Dozens of zones (datacenters)
- Per zone, 100-1000s of servers
- Per server, 100-1000 partitions (tablets)
- Every tablet replicated for fault-tolerance (e.g., 5x)

Spanner: Google’s Globally-Distributed Database

OSDI 2012

Scale-out vs. fault tolerance

- Every tablet replicated via Paxos (with leader election)
- So every “operation” within transactions across tablets actually a replicated operation within Paxos RSM
- Paxos groups can stretch across datacenters!
  - (COPS took same approach within datacenter)
Disruptive idea:

Do clocks really need to be arbitrarily unsynchronized?

Can you engineer some max divergence?

**TrueTime**

- “Global wall-clock time” with bounded uncertainty

\[
\text{TT.now()} \quad \text{time}
\]

\[
\text{earliest} \quad \text{latest}
\]

\[2^*\epsilon\]

Consider event \(e_{\text{now}}\) which invoked \(tt = \text{TT.new}()\):

Guarantee: \(tt.\text{earliest} \leq t_{\text{abs}}(e_{\text{now}}) \leq tt.\text{latest}\)

---

**Timestamps and TrueTime**

- Acquired locks
- Release locks

- Pick \(s > \text{TT.now().latest}\)
- Wait until \(\text{TT.now().earliest} > s\)
- Commit wait

- Average \(\epsilon\)

---

**Commit Wait and Replication**

- Start consensus
- Achieve consensus
- Notify followers

- Acquired locks
- Release locks

- Pick \(s\)
- Commit wait done
Client-driven transactions

Client:
1. Issues reads to leader of each tablet group, which acquires read locks and returns most recent data
2. Locally performs writes
3. Chooses coordinator from set of leaders, initiates commit
4. Sends commit message to each leader, include identify of coordinator and buffered writes
5. Waits for commit from coordinator

Commit Wait and 2-Phase Commit

• On commit msg from client, leaders acquire local write locks
  – If non-coordinator:
    • Choose prepare ts > previous local timestamps
    • Log prepare record through Paxos
    • Notify coordinator of prepare timestamp
  – If coordinator:
    • Wait until hear from other participants
    • Choose commit timestamp >= prepare ts, > local ts
    • Logs commit record through Paxos
    • Wait commit-wait period
    • Sends commit timestamp to replicas, other leaders, client
• All apply at commit timestamp and release locks

Example

Remove X from friend list
Remove myself from X's friend list
Risky post P

\[
\begin{array}{c|c|c|c}
\text{Time} & <8 & 8 & 15 \\
\hline
\text{My friends} & [X] & \emptyset & \emptyset \\
\text{My posts} & \emptyset & [P] & \emptyset \\
\text{X's friends} & [me] & \emptyset & \emptyset \\
\end{array}
\]
Read-only optimizations

- Given global timestamp, can implement read-only transactions lock-free (snapshot isolation)
- Step 1: Choose timestamp \( s_{\text{read}} = \text{TT.now.latest()} \)
- Step 2: Snapshot read (at \( s_{\text{read}} \)) to each tablet
  - Can be served by any up-to-date replica

Disruptive idea:

Do clocks really need to be arbitrarily unsynchronized?

Can you engineer some max divergence?

TrueTime Architecture

Compute reference [earliest, latest] = now ± \( \epsilon \)

TrueTime implementation

\[
\begin{align*}
\text{now} & = \text{reference now} + \text{local-clock offset} \\
\epsilon & = \text{reference } \epsilon + \text{worst-case local-clock drift} \\
& = 1 \text{ms} + 200 \text{μs/sec}
\end{align*}
\]

- What about faulty clocks?
  - Bad CPUs 6x more likely in 1 year of empirical data
Known unknowns > unknown unknowns

Rethink algorithms to reason about uncertainty

The case for log storage:
Hardware tech affecting software design

Latency Numbers Every Programmer Should Know

<table>
<thead>
<tr>
<th>Operation</th>
<th>HDD Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Read</td>
<td>176 MB/s</td>
</tr>
<tr>
<td>Sequential Write</td>
<td>190 MB/s</td>
</tr>
<tr>
<td>Random Read 4KiB</td>
<td>0.495 MB/s</td>
</tr>
<tr>
<td></td>
<td>121 IOPS</td>
</tr>
<tr>
<td>Random Write 4KiB</td>
<td>0.919 MB/s</td>
</tr>
<tr>
<td></td>
<td>224 IOPS</td>
</tr>
<tr>
<td>DQ Random Read 4KiB</td>
<td>1.188 MB/s</td>
</tr>
<tr>
<td></td>
<td>292 IOPS</td>
</tr>
<tr>
<td>DQ Random Write 4KiB</td>
<td>0.929 MB/s</td>
</tr>
<tr>
<td></td>
<td>227 IOPS</td>
</tr>
</tbody>
</table>

~2016

Seagate ($50)
1TB HDD 7200RPM
Model: STD1000DM003-1SB10C

<table>
<thead>
<tr>
<th>Operation</th>
<th>HDD Performance</th>
<th>SSD Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Read</td>
<td>176 MB/s</td>
<td>2268 MB/s</td>
</tr>
<tr>
<td>Sequential Write</td>
<td>190 MB/s</td>
<td>1696 MB/s</td>
</tr>
<tr>
<td>Random Read 4KiB</td>
<td>0.495 MB/s</td>
<td>44.9 MB/s</td>
</tr>
<tr>
<td></td>
<td>121 IOPS</td>
<td>10,962 IOPS</td>
</tr>
<tr>
<td>Random Write 4KiB</td>
<td>0.919 MB/s</td>
<td>151 MB/s</td>
</tr>
<tr>
<td></td>
<td>224 IOPS</td>
<td>36,865 IOPS</td>
</tr>
<tr>
<td>DQ Random Read 4KiB</td>
<td>1.198 MB/s</td>
<td>348 MB/s</td>
</tr>
<tr>
<td></td>
<td>292 IOPS</td>
<td>84961 IOPS</td>
</tr>
<tr>
<td>DQ Random Write 4KiB</td>
<td>0.929 MB/s</td>
<td>399 MB/s</td>
</tr>
<tr>
<td></td>
<td>227 IOPS</td>
<td>97,412 IOPS</td>
</tr>
</tbody>
</table>


---

The Design and Implementation of a Log-Structured File System

Mendel Rosenblum and John K. Ousterhout

- **Idea:** Traditionally disks laid out with spatial locality due to cost of seeks
- **Observation:** main memory getting bigger → most reads from memory
- **Implication:** Disk workloads now write-heavy → avoid seeks → write log
- **New problem:** Many seeks to read, need to occasionally defragment
- **New tech solution:** SSDs → seeks cheap, erase blocks change defrag