MVCC and Distributed Txns (Spanner)

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:

Diagram:

- Write Request
- Dirty Read
- Clean Read

HEAD

replica

replica

TAIL

K: V1, V2
K: V2, V2
K: V3
K: V1
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: $\frac{1}{2}$ white, $\frac{1}{2}$ black
- Transactions:
  - T1: Change all white marbles to black marbles
  - T2: Change all black marbles to white marbles
- Serializability (2PL, OCC)
  - T1 $\rightarrow$ T2 or T2 $\rightarrow$ T1
  - In either case, bag is either ALL white or ALL black
- Snapshot isolation (MVCC)
  - T1 $\rightarrow$ T2 or T2 $\rightarrow$ T1 or T1 $\parallel$ T2
  - Bag is ALL white, ALL black, or $\frac{1}{2}$ white $\frac{1}{2}$ 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 oftxn that reads $O_V$
  - WriteTS: Timestamp oftxn 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) \} \( WriteTS(O_v) \leq TS(T) \)
  - 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) \} \( WriteTS(O_v) \leq TS(T) \)
  - # 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**
  - \( W(1) = 3 \): Write creates version 1 with WriteTS = 3
  - \( R(1) = 3 \): Read of version 1 returns timestamp 3

\[ O \]

\[ \text{write(O) by TS=3} \]

Digging deeper

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

\[ O \]

\[ \text{write(O) by TS=4} \]

Digging deeper

- **Notation**
  - \( W(1) = 3 \): Write creates version 1 with WriteTS = 3
  - \( R(2) = 5 \): Write creates version 2 with WriteTS = 5

\[ O \]

\[ \text{write(O) by TS=4} \]
Digging deeper

Notation

<table>
<thead>
<tr>
<th>txn</th>
<th>txn</th>
<th>txn</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(1) = 3: Write creates version 1 with WriteTS = 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(1) = 3: Read of version 1 returns timestamp 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TS = 3  TS = 4  TS = 5

| W(1) = 3 | W(2) = 5 |
| R(1) = 3 | R(2) = 5 |

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

BEGIN Transaction

tmp = READ(O)
WRITE (O, tmp + 1)
END Transaction

Notation

<table>
<thead>
<tr>
<th>txn</th>
<th>txn</th>
<th>txn</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(1) = 3: Write creates version 1 with WriteTS = 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(1) = 3: Read of version 1 returns timestamp 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TS = 3  TS = 4  TS = 5

| W(1) = 3 | W(2) = 5 |
| R(1) = 3 | R(2) = 5 |

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

⇒ v = 1 has (WriteTS = 3) <= 5

Set R(1) = max(5, R(1)) = 5

write(O) by TS = 4

If ReadTS(1) > 4, abort
⇒ 5 > 4: true
Digging deeper

**Notation**

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

- $W(2) = 5$
- $R(2) = 5$

BEGIN Transaction

- $\text{tmp} = \text{READ}(O)$
- $\text{WRITE}(P, \text{tmp} + 1)$

END Transaction

Find $v$ such that $\max\text{WriteTS}(v) \leq (\text{TS} = 4)$

$\Rightarrow v = 1$ has $(\text{WriteTS} = 3) \leq 4$

Set $R(1) = \max(4, R(1)) = 5$

Then write on $P$ succeeds as well

Distributed Transactions

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)

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

Spanner: Google’s Globally-Distributed Database
OSDI 2012

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)

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

\[
TT.now()\quad \text{time} \\
\begin{array}{c}
\text{earliest} \\
2^e \\
\text{latest}
\end{array}
\]

Consider event \( e_{\text{now}} \) which invoked \( tt = TT.new() \):
Guarantee: \( tt.earliest \leq t_{\text{abs}}(e_{\text{now}}) \leq tt.latest \)

**Timestamps and TrueTime**

- Acquired locks
  - \( T \)
  - Pick \( s > TT.now().latest \)
  - Wait until \( TT.now().earliest > s \)

- Release locks

- Commit wait
  - \( \text{average } \epsilon \)

**Commit Wait and Replication**

- Acquired locks
  - \( T \)
  - Pick \( s \)

- Release locks
  - Commit wait done

- Start consensus
  - Acquire consensus
  - Notify followers
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

Commit Wait and 2-Phase Commit

1. Client 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 msg to each leader, incl. identity of coordinator
**Commit Wait and 2-Phase Commit**

5. Client waits for commit from coordinator

---

**Example**

<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>[P]</td>
</tr>
<tr>
<td>My posts</td>
<td>[me]</td>
<td>[]</td>
<td></td>
</tr>
<tr>
<td>X’s friends</td>
<td>[]</td>
<td>[P]</td>
<td></td>
</tr>
</tbody>
</table>

---

**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

Datacenter 1  Datacenter 2  ...  Datacenter n

GPS timemaster  GPS timemaster  GPS timemaster
GPS timemaster  Atomic-clock timemaster  GPS timemaster

GPS timemaster
Client

Compute reference [earliest, latest] = now ± ε

TrueTime implementation

\[
\text{now} = \text{reference now} + \text{local-clock offset}
\]
\[
\epsilon = \text{reference } \epsilon + \text{worst-case local-clock drift}
\]
\[
= 1\text{ms} + 200 \mu\text{sec}/\text{sec}
\]

- 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

Monday lecture
Caching

Project Proposals due
Monday night, 11:59pm