Optimistic concurrency and Linearizability

CS425/ECE428 — SPRING 2019
NIKITA BORISOV, UIUC
Topics

Optimistic concurrency
- Take advantage of the fact that conflicts are rare

Linearizability
- More “usable” than sequential consistency

Spanner
- Global-scale linearizable DB
Lock conflicts are rare

Empirical studies point locks are rarely waited for
  ◦ Many systems run “fine” if all locks are removed!

Well-designed systems **should** limit contention
  ◦ If waiting for a lock is the common case, you get too much overhead
  ◦ Redesign your lock / data access strategies!

Using locks is “pessimistic”
  ◦ Prepare for the unlikely worst case
  ◦ Most 2PL schemes conservative
Optimistic Concurrency

Basic strategy:
- Execute transaction *without locks*
- Check any conflicts at commit time

Example:

T1
read X
write Y

T2
read X
lock Y
write Y
commit
Optimistic Concurrency

Basic strategy:
- Execute transaction *without locks*
- Check any conflicts at commit time

Example:

T1
read X
write X
ABORT

T2
read X
write X
Optimistic Concurrency

Basic strategy:
◦ Execute transaction *without locks*
◦ Check any conflicts at commit time

Example:
T1
read X
write Y
write X

T2
read X
write X

ABORT
Assign each transaction a unique *timestamp* ($ts$)

- Serialize transactions according to timestamps

Keep track of *timestamp* last transaction to read and write an object

Maintain two invariants:

- If T writes O, last read and write timestamp must be lower than T's
- If T reads O, last write timestamp must be lower than T

If T tries to read/write object with higher timestamp, abort and rollback

<table>
<thead>
<tr>
<th>T(1)</th>
<th>U (2)</th>
<th>V (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read X ($X.rts=1$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write Y ($Y.wts=1$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read X ($X.rts=2$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read Y ($Y.rts=3$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write X ($X.wts=3$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read Y ($Y.rts=3$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write X: abort!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multi-Version concurrency

For each object
- A per-transaction version of the object is maintained
  - Marked as tentative versions
  - And a committed version

Write creates a tentative version with current transaction’s timestamp

Read looks at all tentative versions and picks one with latest timestamp <= current transaction
Linearizability

Serial equivalence:
◦ Total effect on system is equivalent to a run that is serial and consistent with each client’s order

Linearizability
◦ Total effect on system is equivalent to a run that is serial and consistent with actual order of events

E.g., buying a movie
◦ Client makes RPC to bank transfers $3.99 to Amazon account
◦ Client requests video from Amazon
◦ Amazon makes RPC to bank, does not see transfer, rejects request!

E.g., lock system
◦ A: lock()->success; A: unlock(); B.lock()->success; B.unlock()
◦ A: lock()->success; **B.lock()**->success; A: unlock(); B.unlock()
Inconsistent Perspectives

Can be solved by causally ordering requests

Adapt recovery mechanism

- Amazon waits for a few seconds for transfer
- Return error to user, allow to retry

In practice, inconsistent perspectives are rare and last for short periods
Spanner: Google’s
Globally-Distributed Database

Wilson Hsieh
representing a host of authors
OSDI 2012
What is Spanner?

• Distributed multiversion database
  • General-purpose transactions (ACID)
  • SQL query language
  • Schematized tables
  • Semi-relational data model

• Running in production
  • Storage for Google’s ad data
  • Replaced a sharded MySQL database
Example: Social Network

User posts
Friend lists

US
x1000

San Francisco
Seattle
Arizona

Spain
x1000

Brazil
x1000

Sao Paulo
Santiago
Buenos Aires

London
Paris
Berlin
Madrid
Lisbon

Russia
x1000

Moscow
Berlin
Krakow

OSDI 2012
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
Read Transactions

• Generate a page of friends’ recent posts
  – Consistent view of friend list and their posts

Why consistency matters

1. Remove untrustworthy person X as friend
2. Post P: “My government is repressive...”
Single Machine

User posts
Friend lists

Block writes

Friend1 post
Friend2 post
...
Friend999 post
Friend1000 post

Generate my page
Multiple Machines

Block writes

Friend1 post
Friend2 post
...
Friend999 post
Friend1000 post

User posts Friend lists

Generate my page

User posts Friend lists

OSDI 2012
Multiple Datacenters

Friend1 post
US

Friend2 post
Spain
...

Friend999 post
Brazil

Friend1000 post
Russia

Generate my page
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></td>
<td>[P]</td>
</tr>
<tr>
<td>X’s friends</td>
<td>[me]</td>
<td>[]</td>
<td></td>
</tr>
</tbody>
</table>
Synchronizing Snapshots

Global wall-clock time

==

External Consistency:
Commit order respects global wall-time order

==

Timestamp order respects global wall-time order
given
timestamp order == commit order
Timestamps, Global Clock

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

Pick $s = \text{now}()$
Timestamp Invariants

- Timestamp order == commit order

- Timestamp order respects global wall-time order
TrueTime

- “Global wall-clock time” with bounded uncertainty

\[ \text{TT.now()} \] earliest \[ \text{latest} \] \[ 2\varepsilon \]
Timestamps and TrueTime

- Acquired locks
- Pick $s = TT.now().latest$
- Commit wait: average $\varepsilon$
- Release locks
- Wait until $TT.now().earliest > s$
- Commit wait: average $\varepsilon$

OSDI 2012
Commit Wait and Replication

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

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

T_C
T_P1
T_P2
Example

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

<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>
What Have We Covered?

- Lock-free read transactions across datacenters
- External consistency
- Timestamp assignment
- TrueTime
  - Uncertainty in time can be waited out
What Haven’t We Covered?

• How to read at the present time
• Atomic schema changes
  – Mostly non-blocking
  – Commit in the future
• Non-blocking reads in the past
  – At any sufficiently up-to-date replica
TrueTime Architecture

Datacenter 1  Datacenter 2  ...  Datacenter n

GPS timemaster  GPS timemaster  GPS timemaster

GPS timemaster  Atomic-clock timemaster  GPS timemaster

Client

Compute reference [earliest, latest] = now ± ε
TrueTime implementation

$$\text{now} = \text{reference now} + \text{local-clock offset}$$

$$\epsilon = \text{reference }\epsilon + \text{worst-case local-clock drift}$$
What If a Clock Goes Rogue?

• Timestamp assignment would violate external consistency
• Empirically unlikely based on 1 year of data
  – Bad CPUs 6 times more likely than bad clocks
Network-Induced Uncertainty

Epsilon (ms)

Date
Mar 29 Mar 30 Mar 31 Apr 1

Date (April 13)
6AM 8AM 10AM 12PM

99.9
99
90

10
9
8
7
6
5
4
3
2
1
0
What’s in the Literature

- External consistency/linearizability
- Distributed databases
- Concurrency control
- Replication
- Time (NTP, Marzullo)
Future Work

• Improving TrueTime
  – Lower $\varepsilon < 1$ ms

• Building out database features
  – Finish implementing basic features
  – Efficiently support rich query patterns
Conclusions

• Reify clock uncertainty in time APIs
  – Known unknowns are better than unknown unknowns
  – Rethink algorithms to make use of uncertainty

• Stronger semantics are achievable
  – Greater scale != weaker semantics
Thanks

• To the Spanner team and customers
• To our shepherd and reviewers
• To lots of Googlers for feedback
• To you for listening!

• Questions?