Bigtable in retrospect

- Definitely a useful, scalable system!
- Still in use at Google, motivated lots of NoSQL DBs
- Biggest mistake in design (per Jeff Dean, Google): not supporting distributed transactions!
  - became really important w/ incremental updates
  - users wanted them, implemented themselves, often incorrectly!
  - at least 3 papers later fixed this

Two-phase commit

- keys partitioned over different hosts; one coordinator per transaction
- acquire locks on all data read/written; release after commit
- to commit, coordinator first sends prepare message to all shards; they respond prepare_ok or abort
  - if prepare_ok, they must be able to commit transaction
  - if all prepare_ok, coordinator sends commit to all; they write commit record and release locks

Is this the end of the story?

- Availability: what do we do if either some shard or the coordinator fails?
  - generally: 2PC is a blocking protocol, can’t make progress until it comes back up
- Performance: can we really afford to take locks and hold them for the entire commit process?

Spanner

- Backend for the F1 database, which runs the ad system
- Basic model: 2PC over Paxos
- Uses physical clocks for performance

Example: social network

- simple schema: user posts, and friends lists
- but sharded across thousands of machines
- each replicated across multiple continents
Example: social network
- example: generate page of friends’ recent posts
- what if I remove friend X, post mean comment?
  - maybe he sees old version of friends list, new version of my posts?
- How can we solve this with locking?
  - acquire read locks on friends list, and on each friend’s posts
  - prevents them from being modified concurrently
  - but potentially really slow?

Spanner architecture
- Each shard is stored in a Paxos group
  - replicated across data centers
  - has a (relatively long-lived) leader
- Transactions span Paxos groups using 2PC
  - use 2PC for transactions
  - leader of each Paxos group tracks locks
  - one group leader becomes the 2PC coordinator, others participants

Basic 2PC/Paxos approach
- during execution, read and write objects
  - contact the appropriate Paxos group leader, acquire locks
  - client decides to commit, notifies the coordinator
    - coordinator contacts all shards, sends PREPARE message
    - they Paxos-replicate a prepare log entry (including locks), vote either ok or abort
    - if all shards vote OK, coordinator sends commit message
      - each shard Paxos-replicates commit entry
      - leader releases locks
Basic 2PC/Paxos approach

- Note that this is really the same as basic 2PC from before
- Just replaced writes to a log on disk with writes to a Paxos replicated log!
- It is linearizable (= strict serializable = externally consistent)

- So what's left?
  - Lock-free read-only transactions

Lock-free r/o transactions

- Key idea: assign meaningful timestamp to transaction
- such that timestamps are enough to order transactions meaningfully
- Keep a history of versions around on each node
- Then, reasonable to say: r/o transaction X reads at timestamp 10

TrueTime

- Common misconception: the magic here is fancy hardware (atomic clocks, GPS receivers)
  - this is actually relatively standard (see NTP)

- Actual key idea: expose the uncertainty in the clock value

TrueTime API

- interval = TT.now()
- “correct” time is “guaranteed” to be between interval.latest and interval.earliest

- What does this actually mean?

Implementing TrueTime

- time masters (GPS, atomic clocks) in each data center
- NTP or similar protocol syncs with multiple masters, rejects outliers
- TrueTime returns the local clock value, plus uncertainty
  - uncertainty = time since last sync * 200 usec/sec

Assigning transaction timestamps

- When in the basic protocol should we assign a transaction its timestamp?
- What timestamp should we give it?
- How do we know that timestamp is consistent with global time?
Assigning transaction timestamps

- When in the basic protocol should we assign a transaction its timestamp?
  - any time between when all locks acquired and first lock released
- What timestamp should we give it?
  - a time TrueTime thinks is in the future, TT.latest
- How do we know that timestamp is consistent with global time?
  - commit wait until TrueTime knows timestamp is in the past
  - thus: we know that when that timestamp was correct, we’re holding the locks

Spanner pt 2

1. Distributed transactions, in detail
   - On one Paxos group
   - Between Paxos groups
2. Fast read-only transactions with TrueTime
3. Discussion
x = read(checking_bal)
if (x > 100) {
    y = read(savings_bal)
    write(checking_bal, x - 100)
    write(savings_bal, y + 100)
}

LOG:
1. A: Lock(R, checking_bal)
2.
3.
4.
5.
6.
7.
\[ x = \text{read}(\text{checking\_bal}) \]
\[ \text{if} \ (x > 100) \ {\} \]
\[ y = \text{read}(\text{savings\_bal}) \]
\[ \text{write}(\text{checking\_bal}, x - 100) \]
\[ \text{write}(\text{savings\_bal}, y + 100) \]

**LOG:**
1. A: Lock(R, checking\_bal)
2. A: Lock(R, savings\_bal)
3. A: Lock(W, checking\_bal)
4. A: Write(checking\_bal, 100)
5. 
6. 
7.
Leader

Follower

Follower

Client

x = read(checking_bal)
if (x > 100) {
y = read(savings_bal)
write(checking_bal, x - 100)
write(savings_bal, y + 100)
}

LOG:
1. A: Lock(R, checking_bal)
2. A: Lock(R, savings_bal)
3. A: Lock(W, checking_bal)
4. A: Write(checking_bal, 100)
5. A: Lock(W, savings_bal)
6. A: Write(savings_bal, 100)
7.

Leader

Follower

Follower

Client

x = read(checking_bal)
if (x > 100) {
y = read(savings_bal)
write(checking_bal, x - 100)
write(savings_bal, y + 100)
}

LOG:
1. A: Lock(R, checking_bal)
2. A: Lock(R, savings_bal)
3. A: Lock(W, checking_bal)
4. A: Write(checking_bal, 100)
5. A: Lock(W, savings_bal)
6. A: Write(savings_bal, 100)
7.

Leader

Follower

Follower

Client

x = read(checking_bal)
if (x > 100) {
y = read(savings_bal)
write(checking_bal, x - 100)
write(savings_bal, y + 100)
}

LOG:
1. A: Lock(R, checking_bal)
2. A: Lock(R, savings_bal)
3. A: Lock(W, checking_bal)
4. A: Write(checking_bal, 100)
5. A: Lock(W, savings_bal)
6. A: Write(savings_bal, 100)
7. A: Commit
8. A: Unlock(checking_bal)
9. A: Unlock(savings_bal)

Leader 1

Leader 2

LOG:

LOG:
x = read(checking_bal)
if (x > 100) {
    y = read(savings_bal)
    write(checking_bal, x - 100)
    write(savings_bal, y + 100)
}

LOG:
1. A: Lock(R, checking_bal)
2. A: Lock(W, checking_bal)
3. A: Write(checking_bal, 100)
Leader 1
Client
Leader 2

\[ x = \text{read}(\text{checking\_bal}) \]
if \((x > 100)\) {
    \[ y = \text{read}(\text{savings\_bal}) \]
    \[ \text{write}(\text{checking\_bal}, x - 100) \]
    \[ \text{write}(\text{savings\_bal}, y + 100) \]
}

LOG:
1. A: Lock(R, checking\_bal)
2. A: Lock(W, checking\_bal)
3. A: Write(checking\_bal, 100)

Leader 2
LOG:
1. A: Lock(R, savings\_bal)
2. A: Lock(W, savings\_bal)
3. A: Write(savings\_bal, 100)

Leader 1
Client
Leader 2

commit(1)

Leader 2
LOG:
1. A: Lock(R, savings\_bal)
2. A: Lock(W, savings\_bal)
3. A: Prepare
4. A: OK
How can we get fast reads?

R/W transactions are complicated!
- And slow
Can we do fast, lock-free reads?
- Real time to the rescue

TrueTime

API that exposes real time, with uncertainty
{earliest: e, latest: l} = TT.now()

"Real time" is between earliest and latest

Time is an illusion; lunchtime, doubly so
If I call TT.now() on two nodes simultaneously, intervals guaranteed to overlap!
If intervals don’t overlap, the later one happened later!

TrueTime usage

Assign a timestamp to each transaction
- At each Paxos group, timestamp increases monotonically
- Globally, if T1 returns before T2 starts, timestamp(T1) < timestamp(T2)
TrueTime usage

Timestamp for an RW transaction chosen by coordinator leader

Timestamps for R/W transactions is max of:
- Local time (when client request reached coord.)
- Prepare timestamps at every participant
- Timestamp of any previous local transaction

Commit wait

Need to ensure that all future transactions will get a higher timestamp
Therefore, need to wait until
\[ \text{TT.now()} > \text{transaction timestamp} \]
And only then release locks

Leader 1
Client
Leader 2

\[
\begin{align*}
x &= \text{read(checking_bal)} \\
\text{if } (x > 100) \{ \\
y &= \text{read(savings_bal)} \\
\text{write(checking_bal, } x - 100) \\
\text{write(savings_bal, } y + 100) \\
\}
\end{align*}
\]

LOG:
1. A: Lock(R, checking_bal)
2. A: Lock(W, checking_bal)
3. A: Write(checking_bal, 100)

Leader 1
Client
Leader 2

\[
\begin{align*}
x &= \text{read(checking_bal)} \\
\text{if } (x > 100) \{ \\
y &= \text{read(savings_bal)} \\
\text{write(checking_bal, } x - 100) \\
\text{write(savings_bal, } y + 100) \\
\}
\end{align*}
\]

LOG:
1. A: Lock(R, checking_bal)
2. A: Lock(W, checking_bal)
3. A: Write(checking_bal, 100)
Leader 1

x = read(checking_bal)
if (x > 100) {
    y = read(savings_bal)
    write(checking_bal, x - 100)
    write(savings_bal, y + 100)
}

LOG:
1. A: Lock(R, checking_bal)
2. A: Lock(W, checking_bal)
3. A: Write(checking_bal, 100)
4. A: Prepare@t1
5. A: Unlock(checking_bal)

Leader 2

x = read(checking_bal)
if (x > 100) {
    y = read(savings_bal)
    write(checking_bal, x - 100)
    write(savings_bal, y + 100)
}

LOG:
1. A: Lock(R, checking_bal)
2. A: Lock(W, checking_bal)
3. A: Write(checking_bal, 100)
4. A: Prepare@t1

Waiting…
**TrueTime usage**

Timestamps for RO transactions
- Client can choose a timestamp
- For external consistency, always safe to choose
  
  `TT.now().latest`

A response from a replica is a promise: will never have a new transaction that commits before timestamp

Can read from any replica that has seen a Paxos write after timestamp

**TrueTime implementation**

GPS, atomic clocks

Armageddon masters and timeslave daemons

All local clocks synced with masters, and expose uncertainty to local apps

Assumptions made about local clock drift

**Commit wait**

- What does this mean for performance?
  - Larger TrueTime uncertainty bound  
    => longer commit wait
  - Longer commit wait => locks held longer  
    => can’t process conflicting transactions  
    => lower throughput
  - i.e., if time is less certain, Spanner is slower!

**What does this buy us?**

- Can now do a read-only transaction at a particular timestamp, have it be meaningful
  - Example: pick a timestamp T in the past, read version w/ timestamp T from all shards
  - since T is in the past, they will never accept a transaction with timestamp < T
  - don’t need locks while we do this!
  - What if we want the current time?
What if TrueTime fails?

• Google argument: picked using engineering considerations, less likely than a total CPU failure
• But what if it went wrong anyway?
  • can cause very long commit wait periods
  • can break ordering guarantees, no longer externally consistent
  • but system will always be serializable: gathering many timestamps and taking the max is a Lamport clock

Conclusions

What's cool about Spanner?
- Distributed transactions with decent performance
- What makes that possible?
- Read-only transactions with great performance
- What makes that possible?

Clocks are a form of communication!

Discussion

CPU errors and bad clocks
  - When does Google discover errors?
Disaster preparedness
Is Spanner's complexity scary?
What happens if there is high clock skew?
  - If we know about it
  - If we don't?

Timestamp details

• Coordinator actually collects several TrueTime timestamps:
  • timestamps from when each shard executed prepare
  • timestamp that coordinator received commit request from client
  • highest timestamp of any previous transaction
  • Actually picks a timestamp greater than max of these, waits for it to be in the past