6.824 2020 Lecture 13: Spanner

Why this paper (Google Spanner, OSDI 2012)?
  A rare example of wide-area distributed transactions.
    Very desirable.
    But two-phase commit viewed as too slow and prone to blocking.
  A rare example of wide-area synchronous replication.
  Neat ideas:
    Two-phase commit over Paxos.
    Synchronized time for fast r/o transactions.
  Used a lot inside Google.

What was the motivating use case?
  Google F1 advertising database (Section 5.4).
  Previously sharded over many MySQL and BigTable DBs; awkward.
  Needed:
    Better (synchronous) replication.
    More flexible sharding.
    Cross-shard transactions.
  Workload is dominated by read-only transactions (Table 6).
  Strong consistency is required.
    External consistency / linearizability / serializability.

The basic organization:
  Datacenter A:
    "clients" are web servers e.g. for gmail
    data is sharded over multiple servers:
      a-m
      n-z
  Datacenter B:
    has its own local clients
    and its own copy of the data shards
      a-m
      n-z
  Datacenter C:
    same setup

Replication managed by Paxos; one Paxos group per shard.
  Replicas are in different data centers.
  Similar to Raft -- each Paxos group has a leader.
  As in the labs, Paxos replicates a log of operations.

Why this arrangement?
  Sharding allows huge total throughput via parallelism.
  Datacenters fail independently -- different cities.
  Clients can read local replica -- fast!
  Can place replicas near relevant customers.
  Paxos requires only a majority -- tolerate slow/distant replicas.

What are the challenges?
  Read of local replica must yield fresh data.
    But local replica may not reflect latest Paxos writes!
  A transaction may involve multiple shards -> multiple Paxos groups.
  Transactions that read multiple records must be serializable.
    But local shards may reflect different subsets of committed transactions!

Spanner treats read/write and read/only transactions differently.

First, read/write transactions.

Example read/write transaction (bank transfer):
  BEGIN
    x = x + 1
    y = y - 1
  END
We don't want any read or write of x or y sneaking between our two ops.
After commit, all reads should see our updates.

Summary: two-phase commit (2pc) with Paxos-replicated participants.
  (Omitting timestamps for now.)
  (This is for r/w transactions, not r/o.)
  Client picks a unique transaction id (TID).
  Client sends each read to Paxos leader of relevant shard (2.1).
    Each shard first acquires a lock on the relevant record.
      May have to wait.
    Separate lock table per shard, in shard leader.
    Read locks are not replicated via Paxos, so leader failure -> abort.
  Client keeps writes private until commit.
  When client commits (4.2.1):
    Chooses a Paxos group to act as 2pc Transaction Coordinator (TC).
    Sends writes to relevant shard leaders.
    Each written shard leader:
      Acquires lock(s) on the written record(s).
      Log a "prepare" record via Paxos, to replicate lock and new value.
      Tell TC it is prepared.
      Or tell TC "no" if crashed and thus lost lock table.
  Transaction Coordinator:
    Decides commit or abort.
    Logs the decision to its group via Paxos.
    Tell participant leaders and client the result.
  Each participant leader:
    Log the TC's decision via Paxos.
    Release the transaction's locks.

Some points about the design so far.
  Locking (two-phase locking) ensures serializability.
  2pc widely hated b/c it blocks with locks held if TC fails.
    Replicating the TC with Paxos solves this problem!
  r/w transactions take a long time.
    Many inter-data-center messages.
    Table 6 suggests about 100 ms for cross-USA r/w transaction.
    Much less for cross-city (Table 3).
  But lots of parallelism: many clients, many shards.
    So total throughput could be high if busy.

From now on I'll mostly view each Paxos group as a single entity.
  Replicates shard data.
  Replicates two-phase commit state.

Now for read-only (r/o) transactions.
  These involve multiple reads, perhaps from multiple shards.
  We'd like r/o xactions to be much faster than r/w xactions!

Spanner eliminates two big costs for r/o transactions:
  Read from local replicas, to avoid Paxos and cross-datacenter msgs.

But note local replica may not be up to date!
No locks, no two-phase commit, no transaction manager.
Again to avoid cross-data center msg to Paxos leader.
And to avoid slowing down r/w transactions.
Tables 3 and 6 show a 10x latency improvement as a result!
How to square this with correctness?

Correctness constraints on r/o transactions:
Serializable:
Same results as if transactions executed one-by-one.
Even though they may actually execute concurrently.
I.e. an r/o xaction must essentially fit between r/w xactions.
See all writes from prior transactions, nothing from subsequent.
Externally consistent:
If T1 completes before T2 starts, T2 must see T1's writes.
"Before" refers to real (wall-clock) time.
Similar to linearizable.
Rules out reading stale data.

Why not have r/o transactions just read the latest committed values?
Suppose we have two bank transfers, and a transaction that reads both.

```
T1:  Wx  Wy  C
T2:            Wx  Wy  C
T3:        Rx          Ry
```

The results won't match any serial order!
Not T1, T2, T3.
Not T1, T3, T2.
We want T3 to see both of T2's writes, or none.
We want T3's reads to *all* occur at the *same* point relative to T1/T2.

Idea: Snapshot Isolation (SI):
Synchronize all computers' clocks (to real wall-clock time).
Assign every transaction a time-stamp.
r/w: commit time.
r/o: start time.
Execute as if one-at-a-time in time-stamp order.
Even if actual reads occur in different order.
Each replica stores multiple time-stamped versions of each record.
All of a r/w transactions's writes get the same time-stamp.
An r/o transaction's reads see version as of xaction's time-stamp.
The record version with the highest time-stamp less than the xaction's.
Called Snapshot Isolation.

Our example with Snapshot Isolation:

```
            x@10=9      x@20=8
            y@10=11     y@20=12
T1 @ 10:  Wx  Wy  C
T2 @ 20:            Wx  Wy  C
T3 @ 15:      Rx          Ry
```

"@ 10" indicates the time-stamp.
Now T3's reads will both be served from the @10 versions.
T3 won't see T2's write even though T3's read of y occurs after T2.
Now the results are serializable: T1 T2 T3
The serial order is the same as time-stamp order.

Why OK for T3 to read the *old* value of y even though there's a newer value?
T2 and T3 are concurrent, so external consistency allows either order.

Remember: r/o transactions need to read values
as of their timestamp, and *not* see later writes.

Problem: what if T3 reads x from a replica that hasn't seen T1's write?
Because the replica wasn't in the Paxos majority?

Solution: replica "safe time".
Paxos leaders send writes in timestamp order.
Before serving a read at time 20, replica must see Paxos write for time > 20.
So it knows it has seen all writes < 20.
Must also delay if prepared but uncommitted transactions (Section 4.1.3).
Thus: r/o transactions can read from local replica -- usually fast.

Problem: what if clocks are not perfectly synchronized?

What goes wrong if clocks aren't synchronized exactly?
No problem for r/w transactions, which use locks.
If an r/o transaction's TS is too large:
Its TS will be higher than replica safe times, and reads will block.
Correct but slow -- delay increased by amount of clock error.
If an r/o transaction's TS is too small:
It will miss writes that committed before the r/o xaction started.
Since its low TS will cause it to use old versions of records.
This violates external consistency.

Example of problem if r/o xaction's TS is too small:
r/w T0 @ 0: Wx1 C
r/w T1 @ 10: Wx2 C
r/o T2 @ 5: Rx?
(C for commit)
This would cause T2 to read the version of x at time 0, which was 1.
But T2 started after T1 committed (in real time),
so external consistency requires that T2 see x=2.
So there must be a solution to the possibility of incorrect clocks!

Can we synchronize computer clocks perfectly?
So that every computer, in every data center, has the same time?
I.e. we all agree it's 2:00pm on Tuesday April 7 2020.
Not in practice, not perfectly.
Time is defined by clocks at a collection of government labs.
And distributed by various protocols, e.g. GPS, WWV, NTP.
Distribution delays are variable and hard to predict.
So there's always uncertainty.
Plus the distribution systems may be faulty, without warning.

Google's time reference system (Section 5.3)
[UTC, GPS satellites, masters, servers, TTinterval]
A few time master servers per data center.
Each time master has either a GPS receiver or an "atomic clock".
GPS receivers are typically accurate to better than a microsecond.
The paper doesn't say what it means by an atomic clock.
Probably synced to GPS, but accurate for a while w/o GPS.
If coasting, error accumulates, maybe microseconds per week.
Other servers talk to a few nearby time masters.
Uncertainty due to network delays, drift between checks.

TrueTime

Time service yields a TTinterval = [earliest, latest].
The correct time is guaranteed to be somewhere in the interval.
Interval width computed from measured network delays,
clock hardware specifications.
Figure 6: intervals are usually microseconds, but sometimes 10+ milliseconds.
So: server clocks aren't exactly synchronized, but TrueTime
provides guaranteed bounds on how wrong a server's clock can be.

How Spanner ensures that if r/w T1 finishes before r/o T2 starts, TS1 < TS2.
I.e. that r/o transaction timestamps are not too small.
Two rules (4.1.2):
Start rule:
xaction TS = TT.now().latest
for r/o, at start time
for r/w, when commit begins
Commit wait, for r/w xaction:
Before commit, delay until TS < TS.now().earliest
Guarantees that TS has passed.

Example updated with intervals and commit wait:
The scenario is T1 commits, then T2 starts, T2 must see T1's writes.
I.e. we need TS1 < TS2.

```
r/w T0 @  0: Wx1 C
             |1----------10| |11--------------20|
r/w T1 @ 10:       Wx2 P          C
                   |10--------12|
r/o T2 @ 12:                  Rx?
```

(P for prepare)
C guaranteed to occur after its TS (10) due to commit wait.
Rx occurs after C by assumption, and thus after time 10.
T2 chooses TT.now().latest, which is after current time, which is after 10.
So TS2 > TS1.

Why this provides external consistency:
Commit wait means r/w TS is guaranteed to be in the past.
r/o TS = TT.now().latest is guaranteed to be >= correct time
thus >= TS of any previous committed transaction (due to its commit wait)

More generally:
Snapshot Isolation gives you serializable r/o transactions.
Timestamps set an order.
Snapshot versions (and safe time) implement consistent reads at a timestamp.
Xaction sees all writes from lower-TS xactions, none from higher.
Any number will do for TS if you don't care about external consistency.
Synchronized timestamps yield external consistency.
Even among transactions at different data centers.
Even though reading from local replicas that might lag.

Why is all this useful?
Fast r/o transactions:
Read from replica in client's datacenter.
No locking, no two-phase commit.
Thus the 10x latency improvement in Tables 3 and 6.
Although:
r/o transaction reads may block due to safe time, to catch up.
r/w transaction commits may block in Commit Wait.
Accurate (small interval) time minimizes these delays.

Summary:
Rare to see deployed systems offer distributed transactions
over geographically distributed data.
Spanner was a surprising demonstration that it can be practical.
Timestamping scheme is the most interesting aspect.
Widely used within Google; a commercial Google service; influential.