Spanner

A recent Google system designed for multi-datacenter replicated data. Deltas from megastore:

- fully transactional; no user-visible notion of entity groups
- externally consistent (i.e., linearizable) transactions; commit order reflects real time
- lock-based concurrency control; conflicting transactions are slowed down, rather than aborted, and conflicts are based on actual row-level conflict
- no details in paper, but declarative SQL-like language

Data model

SQL-like set of tables

- table must have a primary key
- that key becomes a row name inside a bigtable-like underlying table

Can have a hierarchy of tables

- a "directory" contains a collection of related rows across the hierarchy
- directory ends up being the unit of data placement / replication

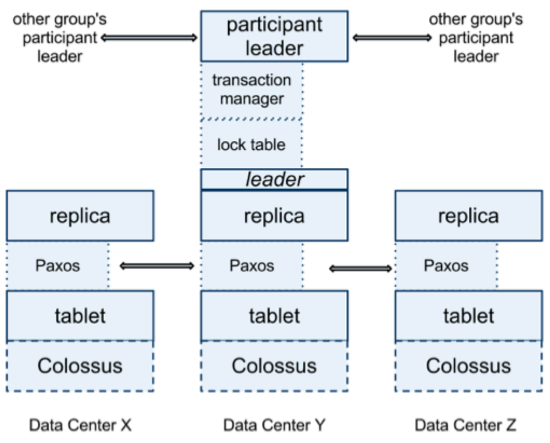
Software structure

Tablet is like a BigTable tablet: a sequence of rows

- like BigTable, tablet state stored in a set of SSTables and a WAL, within GFS' successor Colossus

Each tablet is replicated across multiple data centers using Paxos

- each paxos state machine has a relatively long-lived leader
- call the set of replicas, plus the leader, a "paxos group"



Transactions can span paxos groups

- implemented as two-phase commit across the groups
    - i.e., have 2PC running on top of paxos!
- the leader of each paxos group manages a lock table for concurrency control
    - one paxos leader ends up being the 2PC coordinator, others the slaves
- "some authors have claimed that two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos mitigates the availability problems."

TrueTime

Synchronized clocks across Google's entire infrastructure

- time master: a time server; set per data center
  - GPS receivers with dedicated antenna, or,
  - atomic clock
- timeslave daemon: a time client on a workstation
  - selects multiple masters, from multiple data centers
  - Marzullo's algorithm to sync clock and detect/reject liars
  - between synchronizations, advertise slowly increasing uncertainty
    - about 200 microseconds/second drift rate applied

Interface

- TTInterval tt = TT.now();
  - tt.latest – tt.earliest = e, the instantaneous error bound
  - in practice, e sawtooths between 1ms and 6ms

Time and transactions

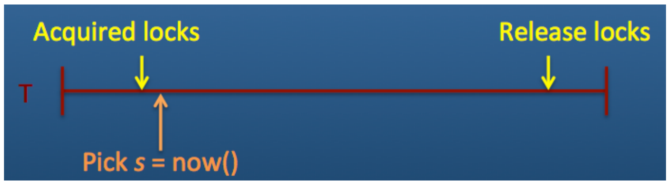
Want to be able to do two things:

- assign a meaningful timestamp to a transaction
- order transactions meaningfully

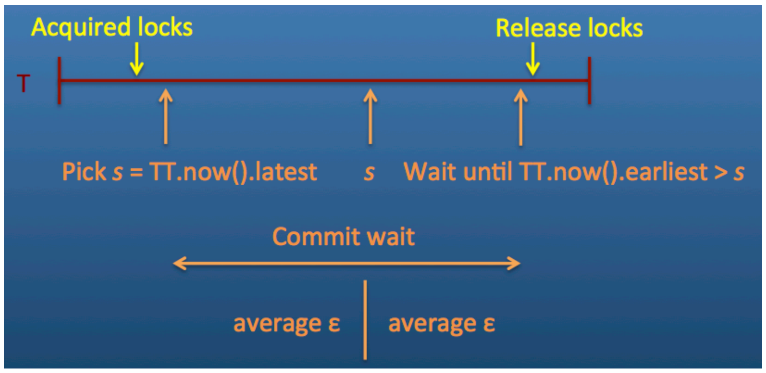
<u>Assigning a timestamp to a transaction</u>

Have strict two-phase locking for write transactions:
- transaction begins
- a period where you gather locks
- a period where locks are held
- a period where you release locks
- transaction ends

Can assign a timestamp anytime between (all locks held) and (any locks released). Q: Why?



Once you've picked this timestamp, need to ensure that the transaction timestamp is consistent with global time. In other words, nobody else should be able to see the transaction side-effects until after s.latest. To do this, need to introduce a "commit wait" period before the locks are released.


Acquired locks
Release locks
T
Pick *s* = TT.now().latest
s
Wait until TT.now().earliest > *s*
Commit wait
average ε
average ε

Commit wait also enforces external consistency (same as linearizability): if the start of T2 occurs after the commit of T1, then the commit timestamp of T2 > commit timestamp of T1. Proof is in the paper.

<u>Picking a timestamp during 2-Phase commit</u>

The 2PC coordinator gathers a number of TrueTime timestamps

- the prepare timestamps from non-coordinators
- the timestamp that the coordinator received the commit message from the client, TTcommit

Chooses overall transaction timestamp to be greater than the prepare timestamps, greater than TTcommit.latest, and greater than any timestamps assigned to earlier transactions. Does commit wait based on this maximum.

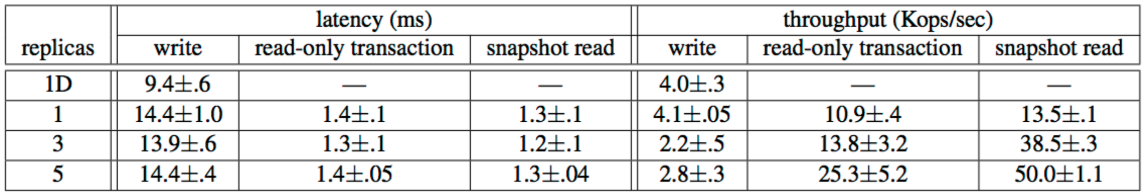
Implications of commit wait

- the larger the uncertainty bound from TrueTime, the longer commit wait period you get
- commit wait will slow down dependent transactions, since locks are held during commit wait
- so, as time gets less certain, Spanner gets slower (!!)

Attacking TrueTime

- you can cause very long commit wait periods – slow the system down
- you can break the ordering guarantees; no longer have external consistency, but cannot cause inconsistency (because still doing two-phase locking)
- go read Winstanley's board posting for full details.

Performance



replicas	latency (ms)			throughput (Kops/sec)		
	write	read-only transaction	snapshot read	write	read-only transaction	snapshot read
1D	9.4±.6	—	—	4.0±.3	—	—
1	14.4±1.0	1.4±.1	1.3±.1	4.1±.05	10.9±.4	13.5±.1
3	13.9±.6	1.3±.1	1.2±.1	2.2±.5	13.8±3.2	38.5±.3
5	14.4±.4	1.4±.05	1.3±.04	2.8±.3	25.3±5.2	50.0±1.1

This is for a single spanserver per zone; enough load to saturate spanserver CPU, and all data is served out of memory to measure overhead of Spanner stack. Get about 2,500 transactional writes/s per CPU, and about 15,000 transactional reads/s per CPU.