Spanner : Google's Globally-Distributed Database

James Sedgwick and Kayhan Dursun

Spanner

- A multi-version, globally-distributed, synchronously-replicated database
- First system to
    - Distribute data globally
    - Externally-consistent distributed Xacts.

Introduction

- Spanner ?
    - System that shards data across Paxos machines into data centers all around the world.
    - Designed to scale up to millions of machines and trillions of database rows.

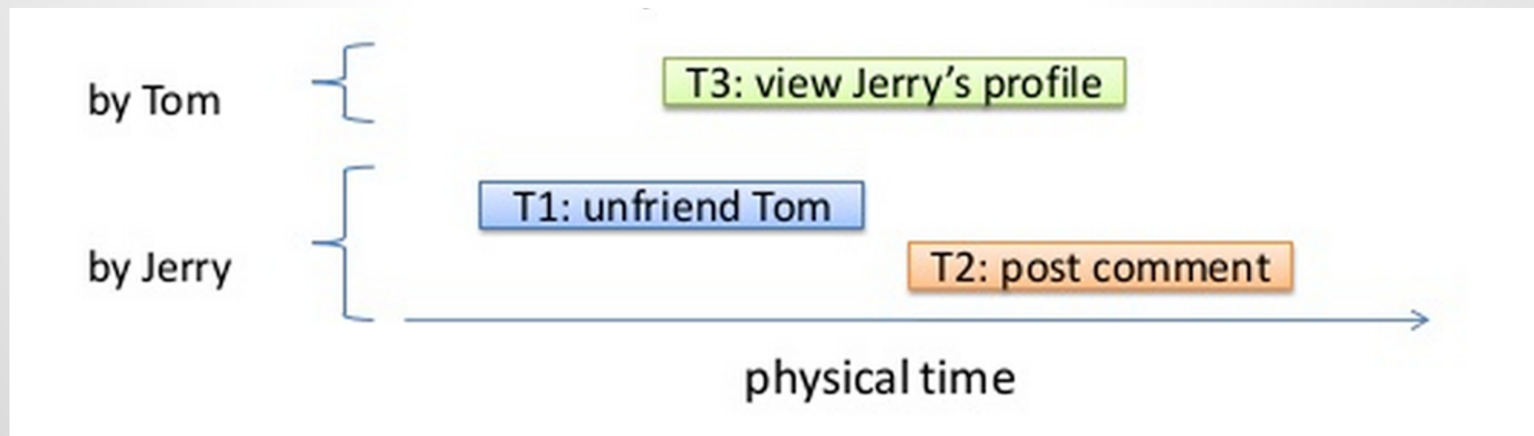
Features

- Dynamic replication configurations
  - Constraints to manage
    - Read latency
    - Write latency
    - Durability, availability
    - Balancing

Features cont.

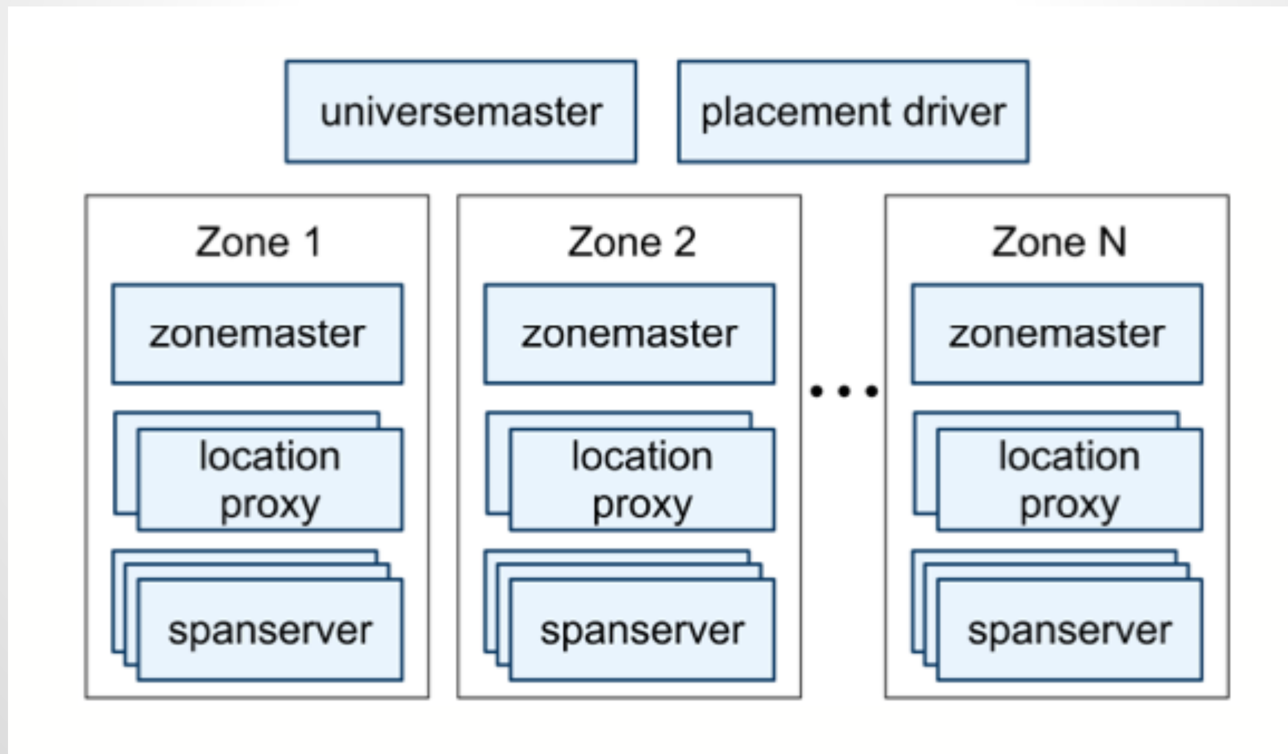
- Externally consistent reads and writes
- Globally consistent reads

Why consistency matters ?



Implementation

- Set of zones = set of locations of dist. data
- Can be more than one zone in a datacenter

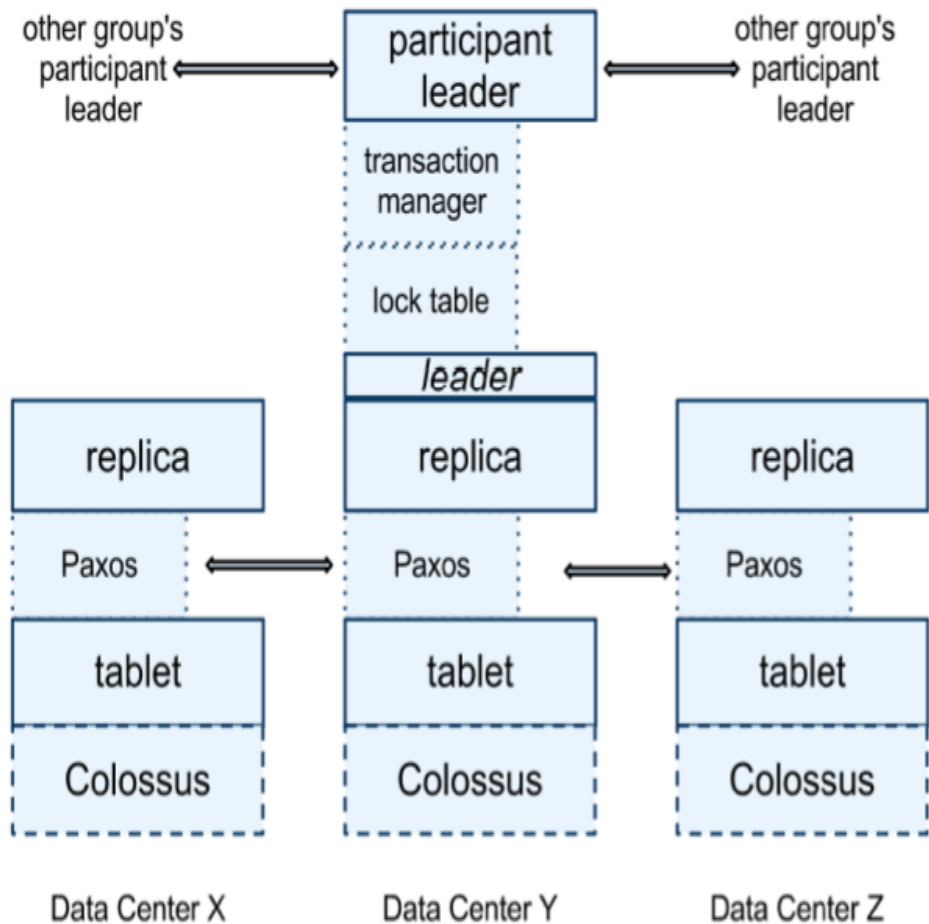


Spanserver Software Stack

Tablet: (key:string, timestamp:int) -> string

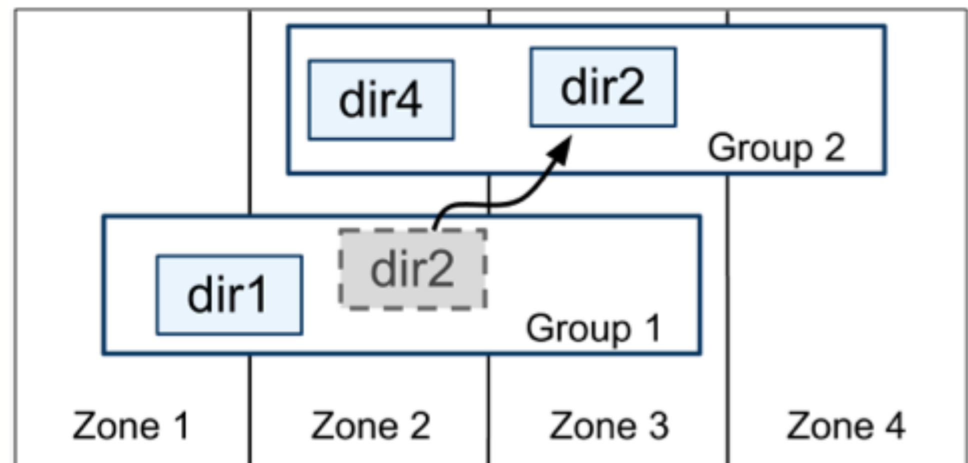
Paxos: Replication sup.

- Writes initiate protocol at leader
- Reads from the tablet directly
- Lock table
- Trans. manager



Directories

- A bucketing abstraction
- Unit of data placement
- Movement
    - Load balancing
    - Access patterns
    - Accessors



Data model

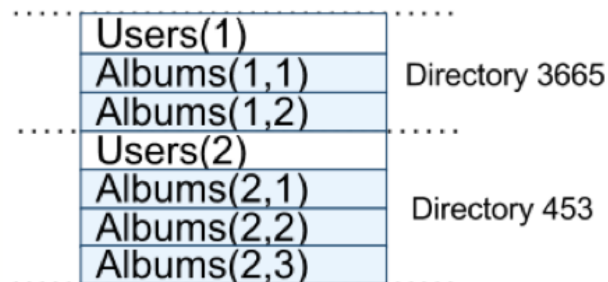
- A data model based on schematized semi-relational tables
  - With popularity of Megastore
- Query language
  - With popularity of Dremel
- General-purpose Xacts.
  - Experienced the lack with BigTable

Data Model cont.

- Not purely relational (rows have names)
- DB must be partitioned into hierarchies

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



TrueTime

Method	Returns
TT.now()	*TTinterval: [earliest, latest]*
TT.after(t)	*true if t has definitely passed*
TT.before(t)	*true if t has definitely not arrived*

- Represents time as intervals with bounded uncertainty
- Let instantaneous error be e (half of interval width)
- Let average error be $\bar{e}$
- Formal guarantee:

  Let $t_{abs}(e)$ be the absolute time of event e

  For tt = *TT.now()*, $tt.earliest <= t_{abs}(e) <= tt.latest$

  where e is the invocation event

TrueTime implementation

- Two underlying time references, used together because they have disjoint failure modes
  - GPS: Antenna/receiver failures, interference, GPS system outage
  - Atomic clock: Drift, etc
- Set of time masters per datacenter (mixed GPS and atomic)
- Each server runs a time daemon
- Masters cross-check time against other masters and rate of local clock
- Masters advertise uncertainty
  - GPS uncertainty near zero, atomic uncertainty grows based on worst case clock drift
- Masters can evict themselves if their uncertainty grows too high

TrueTime implementation, contd.

- Time daemons poll a variety of masters (local and remote GPS masters as well as atomic)
- Use variant of Marzullo's algorithm to detect liars
- Sync local clocks to non-liars
- Between syncs, daemons advertise slowly increasing uncertainty
  - Derived from worst-case local drift, time master uncertainty, and communication delay to masters
- *e* as seen by TrueTime client thus has sawtooth pattern
  - varies from about 1 to 7 ms over each poll interval
- Time master unavailability and overloaded machines/network can cause spikes in *e*

Spanner Operations

- Read-write transactions
  - Standalone writes are a subset
- Read-only transactions
  - Non-snapshot standalone reads are a subset
  - Executed at system-chosen timestamp without locking, such that writes are not blocked.
  - Executed on any replica that is sufficiently up to date w.r.t. chosen timestamp
- Snapshot reads
  - Client provided timestamp or upper time bound

Paxos Invariants

- Spanner's Paxos implementation used timed (10 second) leader leases to make leadership long lived
- Candidate becomes leader after receiving quorum of timed lease votes
- Replicas extend lease votes implicitly on writes. Leader requests a lease extension from a replica if its vote is close to expiration.
- Define a lease interval as starting when a quorum is achieved, and ending when a quorum is lost
- Spanner requires monotonically increasing Paxos write timestamps across leaders in a group, so it is critical that leader lease intervals are disjoint
- To achieve disjointness, a leader could log its interval via Paxos, and subsequent leaders could wait for this interval before taking over.
- Spanner avoids this Paxos communication and preserves disjointness via a TrueTime-based mechanism described in Appendix A.
- It's in an appendix because it's complicated.
- Also: leaders can abdicate, but must wait until *TT.after*(s_{max}) is true, where s_{max} is the maximum timestamp used by a leader, to preserve disjointness

Proof of Externally Consistent RW Transactions

- External consistency: if the start of T_2 occurs after the commit of T_1, then the commit timestamp of T_2 is after the commit timestamp of T_1
- Let start, commit request, and commit events be $e_{i,\ start}$, $e_{i,\ server}$, and $e_{i,\ commit}$
- Thus, formally: if $t_{abs}(e_{1,\ commit}) < t_{abs}(e_{2,\ start})$, then $s_1 < s_2$
- **Start**: Coordinator leader assigns timestamp s_i to transaction T_i s.t. s_i is no less than *TT.now().latest*, computed after $e_{i,\ server}$
- **Commit wait**: Coordinator leader ensures clients can't see effects of T_i before *TT.after(s_i)* is true. That is, $s_i < t_{abs}(e_{i,\ commit})$

$s_1 < t_{abs}(e_{1,\ commit})$ (commit wait)

$t_{abs}(e_{1,\ commit}) < t_{abs}(e_{2,\ start})$ (assumption)

$t_{abs}(e_{2,\ start}) <= t_{abs}(e_{2,\ server})$ (causality)

$t_{abs}(e_{2,\ server}) <= s_2$ (start)

$s_1 < s_2$ (transitivity)

Serving Reads at a Timestamp

- Each replica tracks safe time t_{safe}, which is the maximum timestamp at which it is up to date. Replica can read at t if $t <= t_{safe}$
- $t_{safe} = min(t_{Paxos\text{-}safe}, t_{TM\text{-}safe})$
- $t_{Paxos\text{-}safe}$ is just the timestamp of the highest applied Paxos write on the replica. Paxos write times increase monotonically, so writes will not occur at or below $t_{Paxos\text{-}safe}$ w.r.t. Paxos
- $t_{TM\text{-}safe}$ accounts for uncommitted transactions in the replica's group. Every participant leader (of group g) for transaction T_i assigns prepare timestamp $s_{i,g\text{ - }prepare}$ to its record. This timestamp is propagated to g via Paxos.
- The coordinator leader ensures that commit time s_i of $T_i >= s_{i,g\text{ - }prepare}$ for each participant group g. Thus, $t_{TM\text{-}safe} = min_i(s_{i,g\text{ - }prepare}) - 1$
- Thus, $t_{TM\text{-}safe}$ is guaranteed to be before all prepared but uncommitted transactions in the replica's group

Assigning Timestamps to RO Transactions

- To execute a read-only transaction, pick timestamp s_{read}, then execute as snapshot reads at s_{read} at sufficiently up to date replicas.
- Picking *TT.now().latest* after the transaction start will definitely preserve external consistency, but may block unnecessarily long while waiting for t_{safe} to advance.
- Choose the oldest timestamp that preserves external consistency: *LastTS.*
- Can do better than now if there are no prepared transactions
- If the read's scope is a single Paxos group, simply choose the timestamp of the last committed write at that group.
- If the read's scope encompasses multiple groups, a negotiation could occur among group leaders to determine $max_g(LastTS_g)$
  - Current implementation avoids this communication and simply uses *TT.now().latest*

Details of RW Transactions, pt. 1

- Client issues reads to leader replicas of appropriate groups. These acquire read locks and read the most recent data.
- Once reads are completed and writes are buffered (at the client), client chooses a coordinator leader and sends the identity of the leader along with buffered writes to participant leaders.
- Non-coordinator participant leaders
  - acquire write locks
  - choose a prepare timestamp larger than any previous transaction timestamps
  - log a prepare record in Paxos
  - notify coordinator of chosen timestamp.

Details of RW Transactions, pt. 2

- Coordinator leader
  - acquires locks
  - picks a commit timestamp *s* greater than *TT.now().latest*, greater than or equal to all participant prepare timestamps, and greater than any previous transaction timestamps assigned by the leader
  - logs commit record in Paxos
- Coordinator waits until *TT.after(s)* to allow replicas to commit *T*, to obey commit wait
- Since $s > TT.now().latest$, expected wait is at least 2 * $\bar{e}$
- After commit wait, timestamp is sent to the client and all participant leaders
- Each leader logs commit timestamp via Paxos, and all participants then apply at the same timestamp and release locks

Schema-Change Transactions

- Spanner supports atomic schema changes
- Can't use a standard transaction, since the number of participants (number of groups in the database) could be in the millions
- Use a non-blocking transaction
- Explicitly assign a timestamp *t* in the future to the transaction in the prepare phase
- Reads and writes synchronize around this timestamp
  - If their timestamps precede *t*, proceed
  - If their timestamps are after *t*, block behind schema change

Refinements

- A single prepared transaction blocks $T_{TM\text{-}safe}$ from advancing.
- What if the prepared transactions don't conflict with the read?
- Augment $T_{TM\text{-}safe}$ with mappings from key ranges to prepare timestamps.
- When calculating $T_{TM\text{-}safe}$ as the minimum timestamp of prepared transactions in a group, consult these mappings and only consider transactions which conflict with the read
- Similar problem with *LastTS* - when assigning a timestamp to a read-only transaction, we must wait until after all previous commit timestamps, even if those commits don't conflict with the read.
- Similar solution - maintain mappings of key ranges to commit timestamps, and only consider conflicting commits when calculating a maximum

Refinements

- $t_{Paxos\text{-}safe}$ cannot advance without Paxos writes, so snapshots reads at *t* cannot proceed at groups whose last Paxos write occurred before *t*.
- Paxos leaders instead advance $t_{Paxos\text{-}safe}$ by keeping track of the timestamp above which future Paxos writes will occur.
- Maintain mapping *MinNextTS(n)* from Paxos sequence number *n* to the minimum timestamp that can be assigned to the Paxos write $n + 1$
- Leaders advance *MinNextTS(n)* s.t. it doesn't extend past their lease.
- Advances occur every 8 seconds by default, so in the worst case, replicas can serve reads no more recently than 8 seconds ago
- Advances can occur by a replica's request as well

Evaluation

- Availability
- TrueTime
- Running system F1

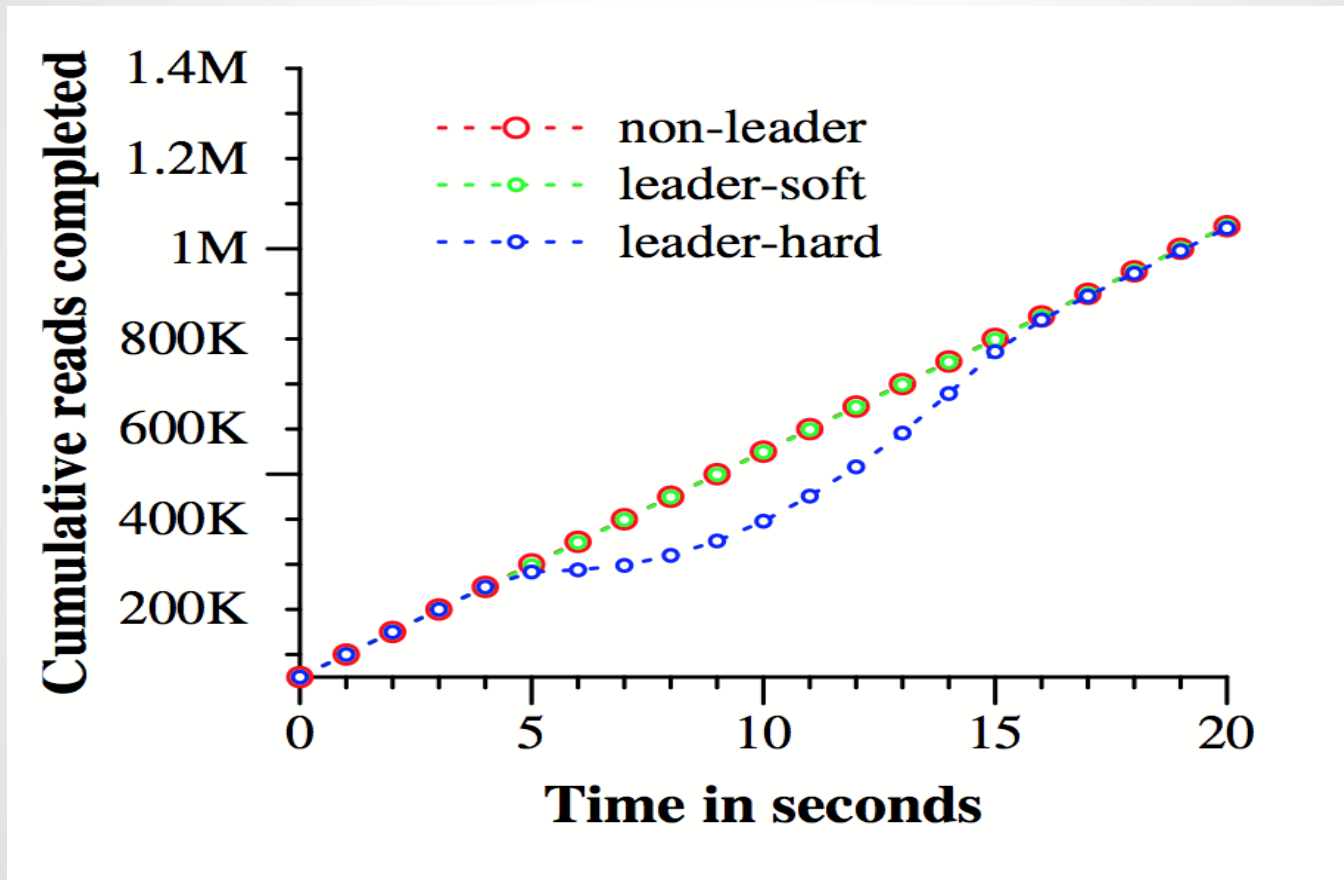
Availability

- Results of 3 experiments in the presence of datacenter failures.

- 5 zones, each has 25 spanservers.

- Data sharded into 1250 paxos groups.

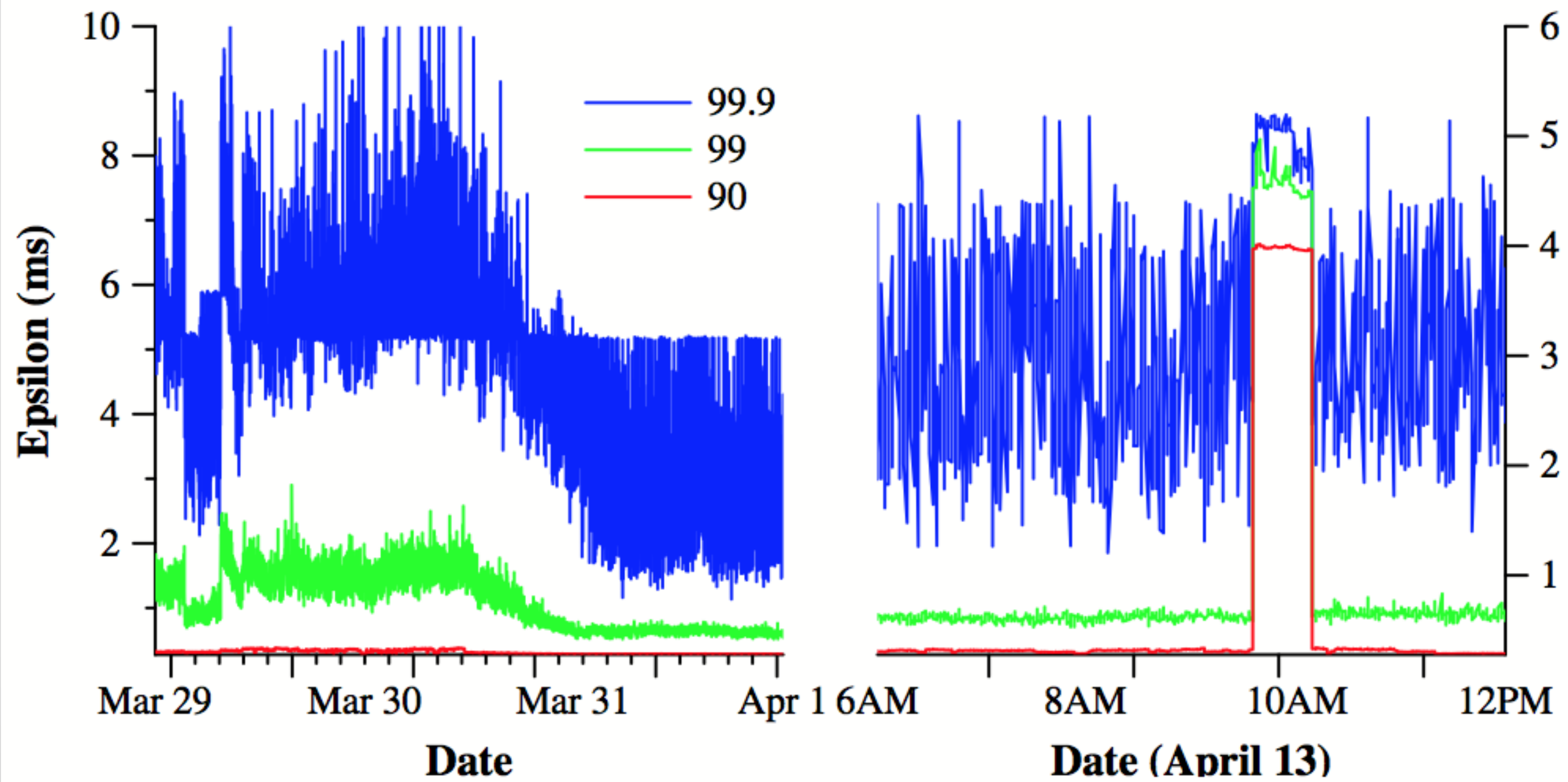
Availability



- leader-hard kill:

10 sec. after killing, throughput is recovered

TrueTime



F1

- First was based on MySql
- Spanner removes the need to manually reshard
- Provides synchronous replication and automatic failover
- F1 requires strong transactional semantics

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

Thank you!