Spanner: Google's Globally-Distributed Database

Kocher Daniel

Summarizing Talk
Efficent Algorithms Seminar
University of Salzburg

January 27, 2016

Outline

Motivation

- Distributed data at global scale
- Externally-consistent distributed transactions
- High availability
- Two "competitors": *Bigtable* & *Megastore*

Motivation

- Distributed data at global scale
- Externally-consistent distributed transactions
- High availability
- Two "competitors": *Bigtable* & *Megastore*
- Why not use Bigtable?
  - Difficult to use for applications with complex, evolving schemas
  - Only **eventually-consistent** (no strong consistency)

Motivation

- Distributed data at global scale
- Externally-consistent distributed transactions
- High availability
- Two "competitors": *Bigtable* & *Megastore*
- Why not use Bigtable?
    - Difficult to use for applications with complex, evolving schemas
    - Only **eventually-consistent** (no strong consistency)
- Why not use Megastore
    - Poor write throughput

Characteristics

Spanner evolved from a Bigtable-like versioned key-value store to a **temporal multi-version database**

Characteristics

Spanner evolved from a Bigtable-like versioned key-value store to a **temporal multi-version database**

- Versioned data is stored in schematized semi-relational tables
- Each version is automatically timestamped with its commit time
- Garbage collection & reads at old timestamps
- **General-purpose transactions**
- SQL-like query language

Fundamentals

- Transaction i is denoted T_i
- **Externally-consistent:**
  If T_1 preceeds T_2 (Commit of T_1 happens before Begin of T_2, no overlap), then T_1 is serialized first

Fundamentals

- Transaction i is denoted T_i
- **Externally-consistent:**
  If T_1 preceeds T_2 (Commit of T_1 happens before Begin of T_2, no overlap), then T_1 is serialized first
- **Two-Phase Locking (2PL):**
  Guarantees serializability
  1. *Expanding* phase: locks are aquired, no locks are released
  2. *Shrinking* phase: locks are released, no locks are aquired
- **Two-Phase Commit (2PC):**
  Coordinates processes participating in atomic distributed transaction
  1. *Commit-Request* phase: Request "Yes" (Commit) or "No" (Abort) from every transaction process
  2. *Commit* phase: Commit transaction if all voted "Yes", otherwise abort

Features

As a globally-distributed database, Spanner provides interesting features:

- Dynamically controlled replication configurations at fine grain
- Externally consistent reads & writes
- globally-consistent reads at a timestamp

⇒ enables **atomic schema changes** in presence of ongoing transactions

Features

As a globally-distributed database, Spanner provides interesting features:

- Dynamically controlled replication configurations at fine grain
- Externally consistent reads & writes
- globally-consistent reads at a timestamp

$\Rightarrow$ enables **atomic schema changes** in presence of ongoing transactions

- Timestamps reflect serialization order (linearizability)
- Key enabler: novel TrueTime API exposing clock uncertainty

Implementation (Top-down)

- Spanner deployment is called **universe**
- Organized as a set of **zones**
- Zones are the unit of administrative deployment & physical isolation
- Set of zones is the set of locations for replication

Implementation (Top-down)

- Spanner deployment is called **universe**
- Organized as a set of **zones**
- Zones are the unit of administrative deployment & physical isolation
- Set of zones is the set of locations for replication
- Each zone consists of
  - several **spanservers** to store the data
  - a **zonemaster** to assign data to spanservers
  - a **location proxy** for clients to locate the assigned spanservers

Implementation (Top-down)

- Spanner deployment is called **universe**
- Organized as a set of **zones**
- Zones are the unit of administrative deployment & physical isolation
- Set of zones is the set of locations for replication
- Each zone consists of
  - several **spanservers** to store the data
  - a **zonemaster** to assign data to spanservers
  - a **location proxy** for clients to locate the assigned spanservers
- **universe master:** console to display status information about zones
- **placement driver:**
  - handles automated data movement across zones
  - moves data due to updated replication constraints or load balancing

Implementation (Top-down)

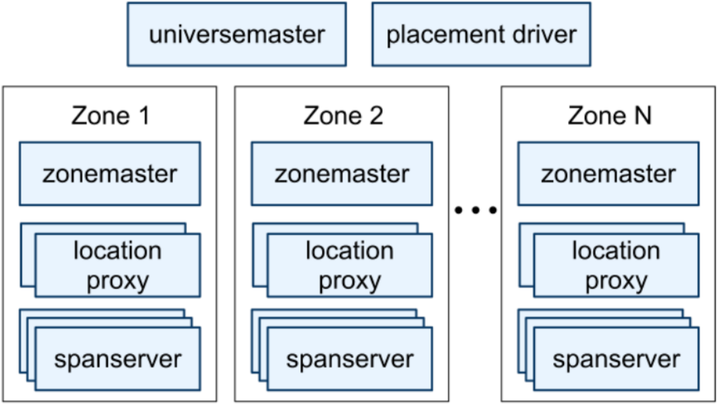

Figure 1 : Spanner server organization

Spanserver Software Stack

- Responsible for 100 - 1000 **tablets**
- A tablet is a bag of key-value mappings
  `(key:string, timestamp:int64)` $\rightarrow$ `string`
- Timestamps are assigned to data $\Rightarrow$ **multi-version database**
- Tablet states are stored on Colossus
- Single Paxos state machine on top of each tablet $\Rightarrow$ **Replication**
- Set of replicas is called **Paxos group**

Spanserver Software Stack

- Every replica which is a leader implements
  - a **lock table** for concurrency control: `(key range)` $\rightarrow$ `lock state`
  - a **transaction manager** (TM) for distributed transactions
- Transaction manager used to implement **participant leader**
- If multiple Paxos groups are involved: Two-Phase Commit
  - One participant group is chosen as **coordinator**
  - Participant leader of this group: **coordinator leader**
- State of each transaction manager is stored in Paxos group

Spanserver Software Stack

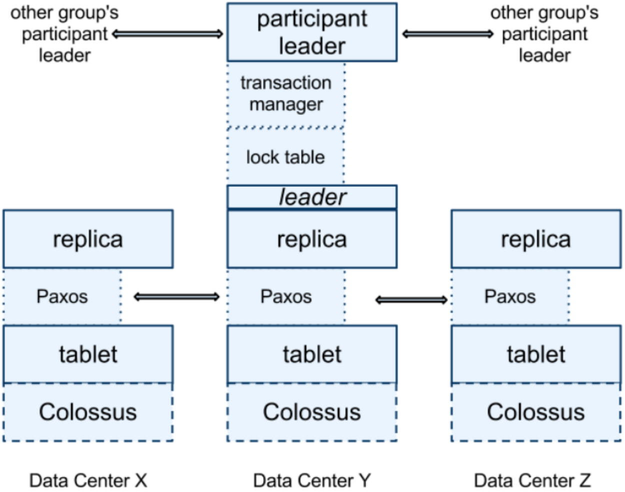

Figure 2 : Spanserver software stack

Directories & Placement

- **Directories:**
  - Bucketing abstraction on top of key-value mappings
  - Contain set of contiguous keys sharing common prefix
  - Control locality of data
  - Unit of data placement (same replication configuration)
  - Smallest unit to specify geographic-replication properties (**placement**)

Directories & Placement

- **Directories:**
  - Bucketing abstraction on top of key-value mappings
  - Contain set of contiguous keys sharing common prefix
  - Control locality of data
  - Unit of data placement (same replication configuration)
  - Smallest unit to specify geographic-replication properties (**placement**)
- Data is moved directory-wise between Paxos groups
- Tablets may have multiple **partitions** of row space
  $\Rightarrow$ co-locate directories frequently accessed together

Directories & Placement

- **Directories:**
  - Bucketing abstraction on top of key-value mappings
  - Contain set of contiguous keys sharing common prefix
  - Control locality of data
  - Unit of data placement (same replication configuration)
  - Smallest unit to specify geographic-replication properties (**placement**)
- Data is moved directory-wise between Paxos groups
- Tablets may have multiple **partitions** of row space
  $\Rightarrow$ co-locate directories frequently accessed together
- Placements can be controlled in two dimensions:
  1. Number & types of replicas
  2. Geographic placement of replicas
- **Movedir:** background task to move directories

Directories & Placement

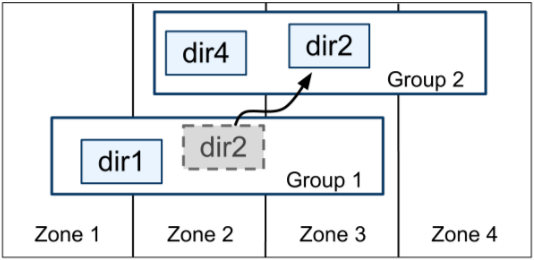

Figure 3 : Directories (the unit of data movement)

Data Model

- Features:
  - Schematized semi-relational tables
  - SQL-like query language
  - General-purpose transactions
- Layered on top of directory-bucketed key-value mappings
- Applications create **databases** in universe
- Database may contain unlimited number of schematized **tables**
- Table contains rows (must have names), columns & versioned values

Data Model

- Databases must be partitioned in one or more hierarchies
- **Directory table**: table at the top
- Directory contains ordered rows starting at a directory table key
- Interleaving tables to form directories is significant
- Locality relationships between multiple tables $\Rightarrow$ good performance

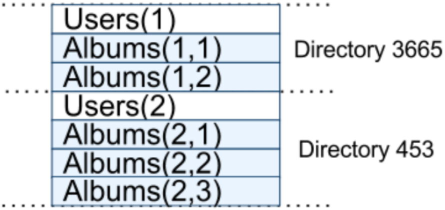

Figure 4 : Interleaving example

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

Table 1 : TrueTime API (t is of type $TTstamp$)

Time is represented as a $TTinterval$ (interval with bounded uncertainty)

Let e be an event and $t_{abs}(e)$ denote the absolute time of e.
For $tt = TT.now()$, the following property holds:

$$tt.earliest \leq t_{abs}(e) \leq tt.latest$$

TrueTime

- Time references: GPS and atomic clocks
- Set of **time master** machines per datacenter
- **Armageddon masters**: masters using atomic clocks
- **Timeslave daemon** on each machine
- Masters compare their time references regularly
- Daemons poll a variety of masters (liar detection)

TrueTime

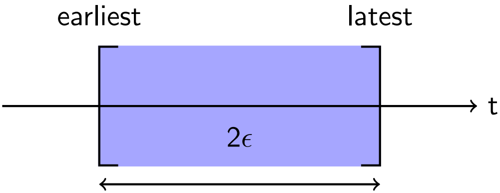

Figure 5 : Visualization of $TT.now()$ and ϵ

Concurrency Control

- TrueTime is used to guarantee correctness properties
- Properties are used to implement
  - Externally-consistent transactions
  - Lock-free read-only transactions
  - Non-blocking reads in the past

Timestamp Management

Operation	CC	Replica required
Read-Write transaction	Pessimistic	leader
Read-Only transaction	Lock-free	leader; any
Snapshot Read, client-provided timestamp	Lock-free	any
Snapshot Read, client-provided bound	Lock-free	any

Table 1 : Supported types of reads & writes

- Standalone write $\Rightarrow$ RW transaction
- Non-snapshot standalone read $\Rightarrow$ RO transaction
- RO transactions
  - must predeclare to not include writes
  - can proceed on any sufficiently up-to-date replica (also snapshot reads)

Paxos Leader Leases

- Timed leases for long-lived leadership (10s by default)
- Quorum of lease votes $\Rightarrow$ leader has a lease
- Successful writes extend lease vote on replica
- **Disjointness invariant:**
  For each Paxos group, each Paxos leader's *lease interval* is disjoint from every other leader's
- Leaders must not abdicate before $TT.after(s_{max})$, where s_{max} denotes the maximum timestamp used by a leader

Timestamps in RW Transactions

- Timestamp assignment only in between the two phases (2PL)
- **Monotonicity variant:**
  Within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders
- Enforced across leaders by disjointness invariant:
  Leader must only assign timestamps within the interval of leader lease
- Timestamp s assigned $\Rightarrow s_{max} = s$
- **External consistency invariant**:

$$t_{abs}\left(e_1^{commit}\right) < t_{abs}\left(e_2^{start}\right) \Rightarrow s_1 < s_2$$

Timestamps in RW Transactions

- Let e_i^{server} denote the arrival event of commit request for T_i
- Two rules guarantee the external consistency invariant:
  1. **Start**
     Coordinator leader assigns commit timestamp
     $s_i \geq TT.now().latest$ after e_i^{server}
  2. **Commit Wait**
     Coordinator leader ensures clients cannot see
     data commited by T_i until $TT.after(s_i)$ is true

Serving Reads at a Timestamp

- Monotonicity invariant allows to determine if a replica's state is **sufficiently up-to-date** to satisfy a read
- **Safe time** t_{safe} at each replica
- Replica satisfies read if $t \leq t_{safe}$
- $t_{safe} = min\left(t_{safe}^{Paxos}, t_{safe}^{TM}\right)$

Timestamps in RO Transactions

- Two phases:
  1. Assign timestamp s_{read}
  2. Execute transaction's reads as snapshot reads at s_{read}
- Spanner assigns the oldest timestamp that preserves external consistency to reduce the changes of blocking

Read-Write Transactions

- RW transactions buffered at client until Commit
- Reads do not see effects of transaction's writes
- Client has completed all reads & buffered all writes $\Rightarrow$ 2PC

Read-Write Transactions

- RW transactions buffered at client until Commit
- Reads do not see effects of transaction's writes
- Client has completed all reads & buffered all writes $\Rightarrow$ 2PC
- Non-coordinator-participant leader
    1. Chooses prepare timestamp according to monotonicity
    2. Logs prepare record through Paxos
    3. Notifies coordinator of its prepare timestamp

Read-Write Transactions

- RW transactions buffered at client until Commit
- Reads do not see effects of transaction's writes
- Client has completed all reads & buffered all writes $\Rightarrow$ 2PC
- Non-coordinator-participant leader
  1. Chooses prepare timestamp according to monotonicity
  2. Logs prepare record through Paxos
  3. Notifies coordinator of its prepare timestamp
- Coordinator leader
  1. Chooses timestamp for whole transaction
  2. Commit timestamp s must be
     - $\geq$ all prepare timestamps
     - $> TT.now().latest$ at the time the Commits were received
     - $>$ any previously assigned timestamp
  3. Logs Commit record through Paxos

Read-Write Transactions

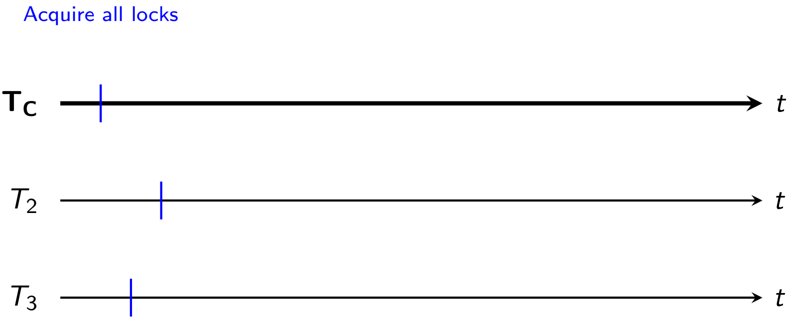

Figure 6 : RW transaction illustration

Read-Write Transactions

Acquire all locks

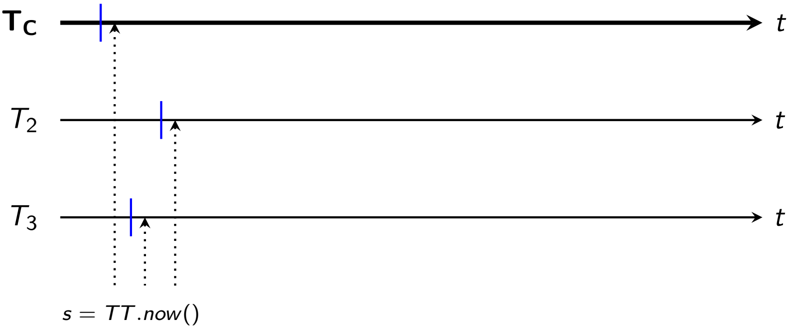

Figure 6 : RW transaction illustration

Read-Write Transactions

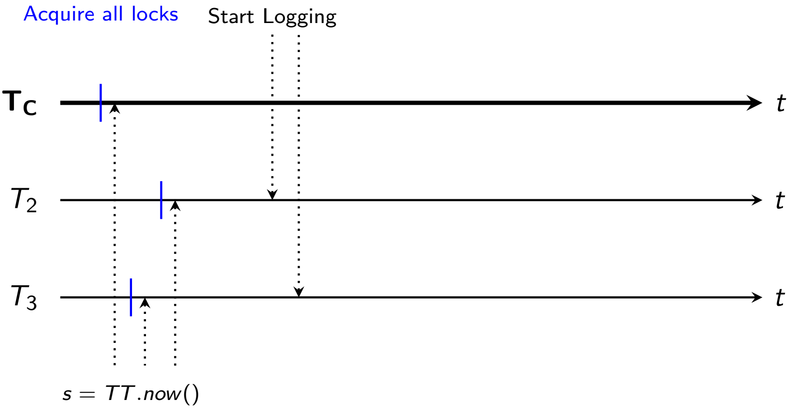

Figure 6 : RW transaction illustration

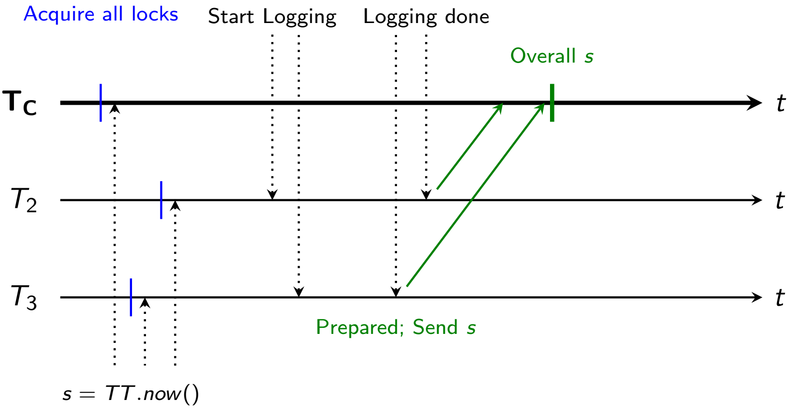

Figure 6 : RW transaction illustration

Read-Write Transactions

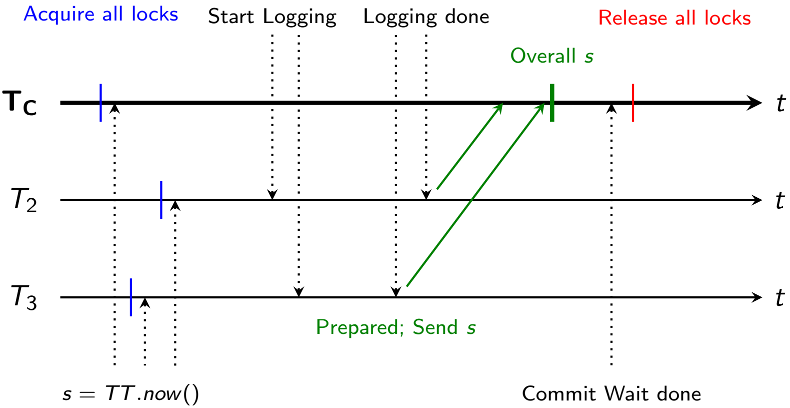

Figure 6 : RW transaction illustration

Read-Write Transactions

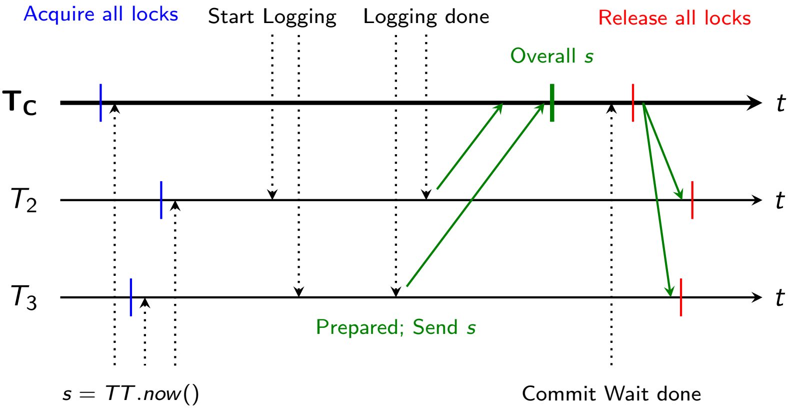

Figure 6 : RW transaction illustration

Read-Only Transactions

- **scope** expression summarizing the keys that will be read
- Served by single Paxos group
  $\Rightarrow$ client issues transaction to group leader
- Served by multiple Paxos groups
  $\Rightarrow$ client executes reads at $s_{read} = TT.now().latest$
- All reads can be sent to replicas that are sufficiently up-to-date

Schema-Change Transactions

- TrueTime enables this feature
- Use of standard transaction infeasible
- Non-blocking variant of standard transaction
  1. Explicit assignment of a future timestamp (Prepare phase)
  2. Reads & writes synchronize with any registered schema-change timestamp at time t
- Without TrueTime $\Rightarrow$ schema change at t would be meaningless

Microbenchmarks

- Setup:
  - 4GB RAM scheduling units per spanserver
  - 4 cores (AMD Barcelona 2200MHz) per spanserver
  - One spanserver per zone
  - Clients & zones in set of datacenters with network distance $<$ 1ms
  - Database created with 50 Paxos groups with 2500 directories
- Operations: standalone reads & writes of 4KB
- One unmeasured round of reads to warm any location caches

Microbenchmarks

Latency [ms] (less is better)

Replicas	Write	RO Transaction	Snapshot Read
1D	9.4 ± 0.6	-	-
1	14.4 ± 1.0	1.4 ± 0.1	1.3 ± 0.1
3	13.9 ± 0.6	1.3 ± 0.1	1.2 ± 0.1
5	14.4 ± 0.4	1.4 ± 0.05	1.3 ± 0.04

Table 2 : Latency experiments

- Sufficiently few operations to avoid queuing
- Increasing number of replicas:
  - Latency stays roughly constant with less standard deviation
  - Latency to achieve quorum less sensitive to stragglers

Microbenchmarks

Throughput [Kops/sec] (more is better)

Replicas	Write	RO Transaction	Snapshot Read
1D	4.0 ± 0.3	-	-
1	4.1 ± 0.05	10.9 ± 0.4	13.5 ± 0.1
3	2.2 ± 0.5	13.8 ± 3.2	38.5 ± 0.3
5	2.8 ± 0.3	25.3 ± 5.2	50.0 ± 1.1

Table 2 : Throughput experiments

- Sufficiently many operations to saturate servers' CPUs
- Increasing number of replicas:
    - Snapshot read throughput increases almost linear

Availability

- Setup:
  - Universe with 5 zones Z_i, each of which has 25 spanservers
  - Database sharded into 1250 Paxos groups
  - 100 clients constantly issued snapshot reads (50K reads/sec)
  - All leaders explicitly place in Z_1
  - 5s into each test, all servers in one zone were killed

Availability

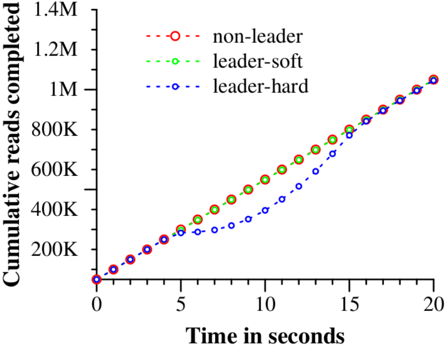

Figure 7 : Effect of killing servers on throughput

- Availability benefits of running Spanner in multiple datacenters

TrueTime

- Two questions:
    - Is ϵ truly a bound on clock uncertainty?
    - How bad does ϵ get?

TrueTime

- Two questions:
  - Is ϵ truly a bound on clock uncertainty?
  - How bad does ϵ get?
- Answers:
  - Clock issues infrequent relative to more serious hardware problems
    $\Rightarrow$ TrueTime as trustworthy as any other piece of software

- Two questions:
  - Is ϵ truly a bound on clock uncertainty?
  - How bad does ϵ get?
- Answers:
  - Clock issues infrequent relative to more serious hardware problems
    $\Rightarrow$ TrueTime as trustworthy as any other piece of software

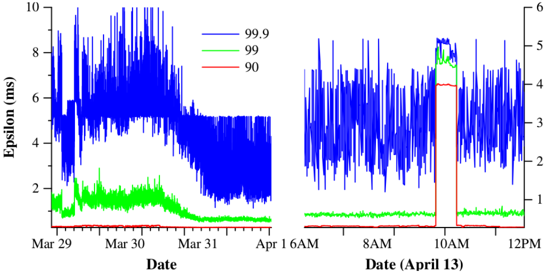

Figure 8 : Distribution of TrueTime ϵ values

Conclusions & Future Work

- 5 years from Spanner's inception to iterate to the current design
- Combines & extends on ideas from database & systems community
- Linchpin: TrueTime
- Build distributed systems with much stronger time semantics
- No longer depend on loosely synchronized clocks & weak time APIs

Conclusions & Future Work

- 5 years from Spanner's inception to iterate to the current design
- Combines & extends on ideas from database & systems community
- Linchpin: TrueTime
- Build distributed systems with much stronger time semantics
- No longer depend on loosely synchronized clocks & weak time APIs
- Future Work:
  - Implement optimistical reads in parallel
  - Support direct changes of Paxos configurations
  - Improve single-node performance
  - Support movement of processes between datacenters

Thank you for your attention!

Questions?