Spanner: Google's Globally-Distributed Database*

Kocher Daniel

Summarizing Talk
Efficent Algorithms Seminar
University of Salzburg

January 29, 2016

* James C. Corbett et al. (26 authors), OSDI 2012.

Outline

Motivation

- Distributed data at global scale
- Externally-consistent distributed transactions
- High availability

Motivation

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

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

- **External consistency:**
  If T_1 preceeds T_2, then T_1 is serialized first
- **Paxos (Replication):**
  - Solves problem of resilient replication of data
  - Data eventually propagates to all nodes
  - Different nodes always see the same data
  - Majority of nodes up $\Rightarrow$ Writes/Reads processed correctly
  - Single node is elected as leader and initiates consensus
  - Guarantees **consistency**

Fundamentals

- **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

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

- Timestamps reflect serialization order
- Key enabler: novel **TrueTime** API **exposing clock uncertainty**

Implementation

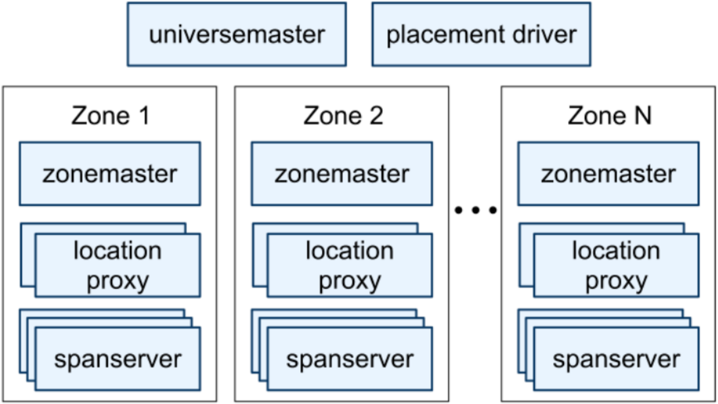

Figure 1 : Spanner server organization

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: state for 2PL
  - 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 group**
  - Participant leader of this group: **coordinator leader**
- State of each transaction manager is stored in Paxos group

Spanserver Software Stack

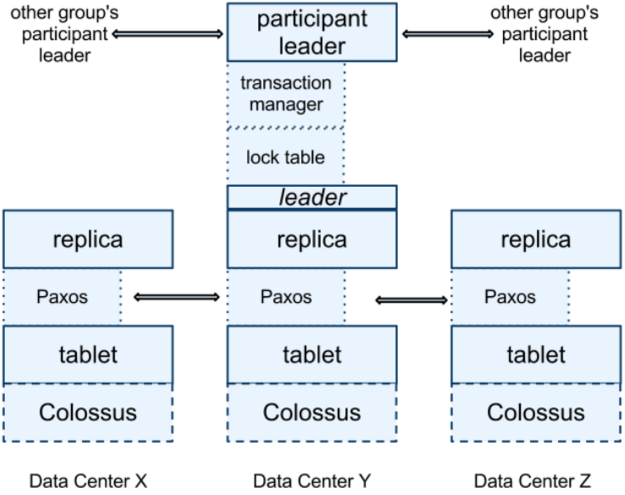

Figure 2 : Spanserver software stack

Directories & Placement

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

Directories & Placement

- **Directories:**
  - Bucketing abstraction on top of tablet
  - Contain set of contiguous keys sharing common prefix
  - Control locality of data (pyhsical)
  - 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 directories
  $\Rightarrow$ co-locate data frequently accessed together
- Placements can be controlled in two dimensions:
  1. Number & types of replicas
  2. Geographic placement of replicas

Directories & Placement

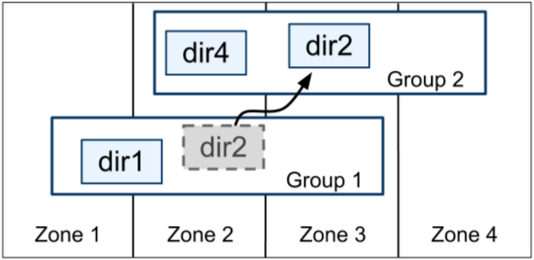

Figure 3 : Directories (the unit of data movement)

Data Model

- Layered on top of directory-bucketed key-value mappings
- Applications create **databases**
- Database may contain any number of schematized **tables**
- Table contains rows, columns & versioned values

Data Model

- Databases must be partitioned in one or more hierarchies
- **Directory table**: table at the top
- Directory contains hierarchically ordered rows
- Interleaving tables ⇒ locality relationships of multiple tables
  ⇒ significant for performance

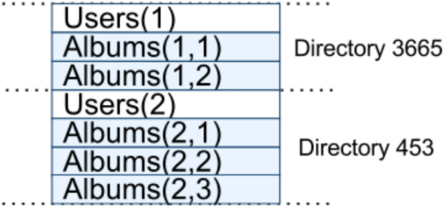

Figure 4 : Interleaving example

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

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

Guarantee: $TT.now().earliest \leq t_{abs}(e) \leq TT.now().latest$

TrueTime

- Time references: GPS and atomic clocks
- Set of **time master** machines per datacenter
- **Timeslave daemon** on each machine
- Masters compare time references regularly
- Daemons poll a variety of masters
- ϵ is the **instantaneous error bound** (typically 1 - 7ms)
- $\bar{\epsilon}$ is the average error bound (typically 4ms)
- ϵ is derived from applied worst-case local clock drift ($200\mu s/sec$)

TrueTime

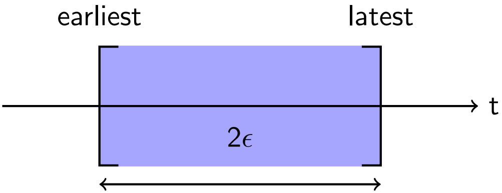

Figure 5 : Visualization of $TT.now()$ and ϵ

Concurrency Control

- TrueTime used to guarantee correctness properties to implement:
  - Externally-consistent transactions
  - Lock-free read-only transactions
  - Non-blocking reads in the past (snapshot reads)

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

- Standalone write ⇒ Read-Write (RW) transaction
- Non-snapshot standalone read ⇒ Read-Only (RO) transaction
- RO transactions must predeclare to not include writes
- RO transactions & snapshot reads proceed on any **sufficiently up-to-date** replica

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 $TT.after(s_{max})$
- s_{max} ... maximum timestamp used by a leader

Read-Write Transactions

- Uses 2PL: Timestamp assignment in between the two phases
- **Monotonicity invariant:**
  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 interval of leader lease
- Timestamp s assigned $\Rightarrow s_{max} = s$
- **External consistency invariant**:

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

Read-Write Transactions

- Writes buffered at client until commit
- Client has completed all reads & buffered all writes $\Rightarrow$ 2PC
- 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

Read-Write Transactions

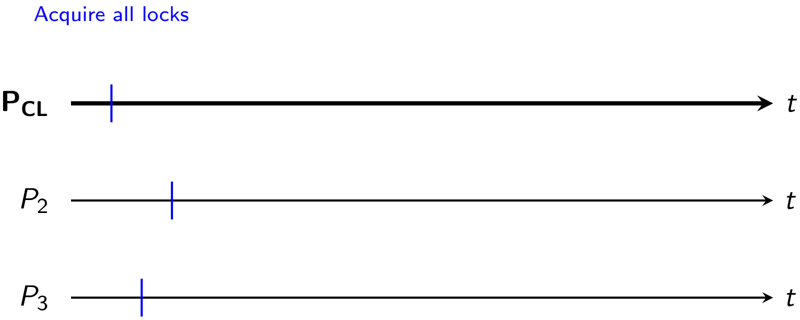

Figure 6 : RW transaction illustration

Read-Write Transactions

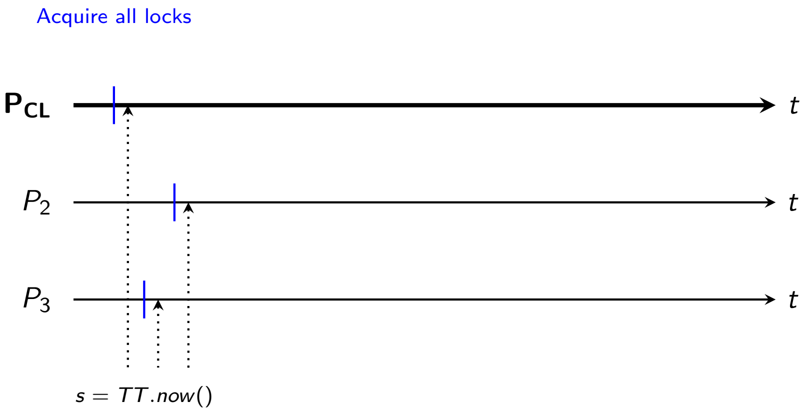

Figure 6 : RW transaction illustration

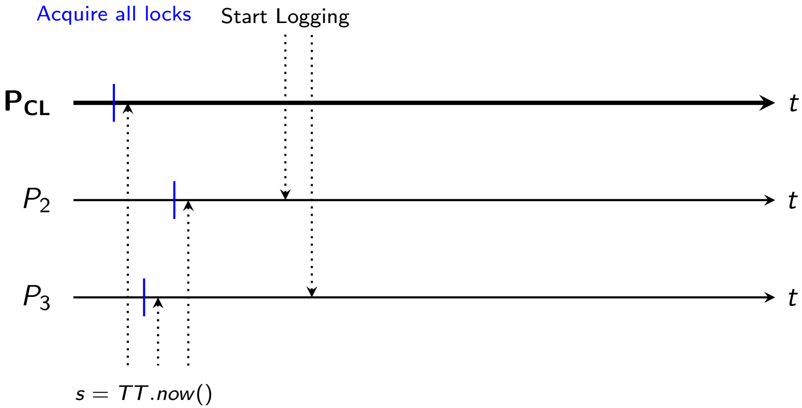

Figure 6 : RW transaction illustration

Read-Write Transactions

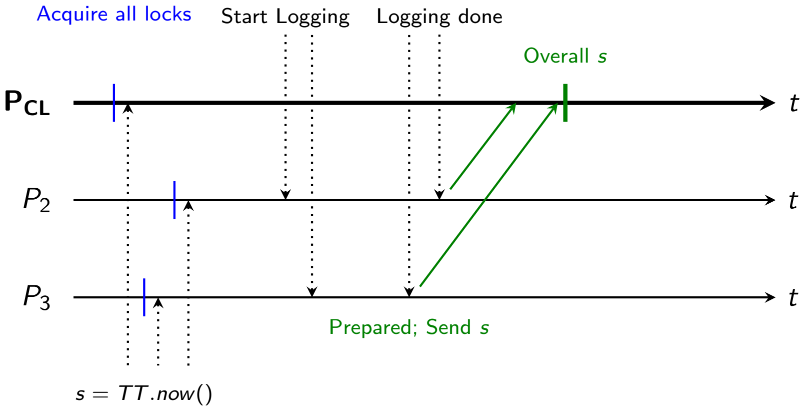

Figure 6 : RW transaction illustration

Read-Write Transactions

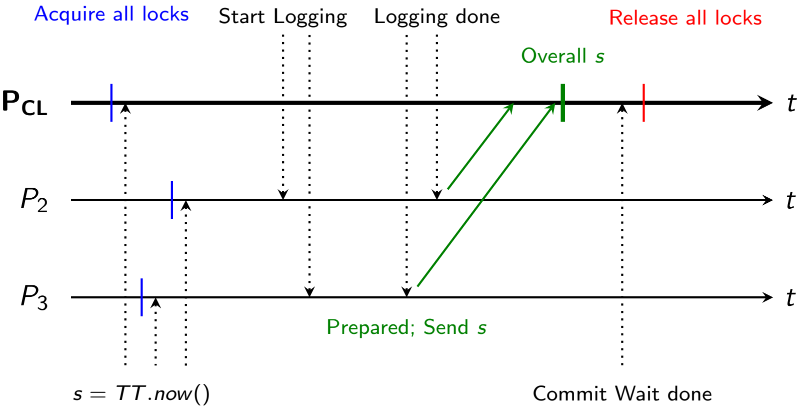

Figure 6 : RW transaction illustration

Read-Write Transactions

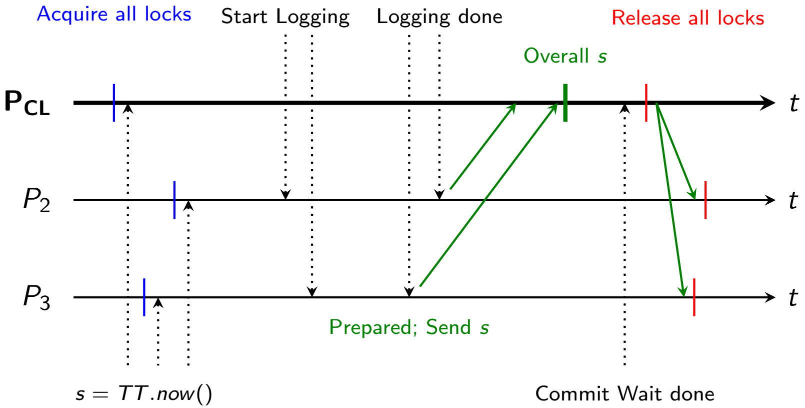

Figure 6 : RW transaction illustration

Read-Write Transactions

Operation	CC	Replica required
Read-Write transaction	Pessimistic	leader

Serving Reads at a Timestamp

Operation	CC	Replica required
Snapshot Read, client-provided timestamp	Lock-free	any
Snapshot Read, client-provided bound	Lock-free	any

- 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 at timestamp t if $t \leq t_{safe}$
- Performance benefits

Read-Only Transactions

Operation	CC	Replica required
Read-Only transaction	Lock-free	leader; any

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

Read-Only Transactions

- Negotiation phase between all Paxos groups involved
- **scope** expression summarizing the keys that will be read
- Scope's values served by single Paxos group
  - Client issues RO transaction to group leader
  - Leader assigns s_{read} = timestamp of last committed write
- Scope's values served by multiple Paxos groups
  - Negotiation between leaders possible
  - Simpler: reads execute at $s_{read} = TT.now().latest$
- All reads can be sent to replicas that are sufficiently up-to-date

Schema-Change Transactions

- Enabled by TrueTime
- Non-blocking variant of standard transaction
  1. Explicit assignment of a future timestamp
  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

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

Availability

- Setup:
  - Universe with 5 zones Z_i, each of which has 25 spanservers
  - All leaders explicitly placed in Z_1
  - 5s into each test, all servers in one zone were killed

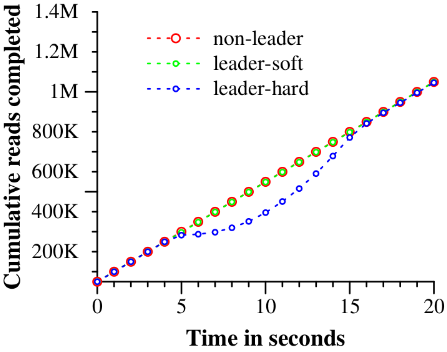

Figure 6 : Effect of killing servers on throughput

TrueTime

1. Is ϵ truly a bound on clock uncertainty?
    - Local clock drifts $> 200\mu s/sec$ would break assumptions
    - Clock issues infrequent relative to more serious hardware problems
      $\Rightarrow$ TrueTime as trustworthy as any other piece of software

TrueTime

1. Is ϵ truly a bound on clock uncertainty?
   - Local clock drifts $> 200\mu$s/sec would break assumptions
   - Clock issues infrequent relative to more serious hardware problems
     $\Rightarrow$ TrueTime as trustworthy as any other piece of software
2. How bad does ϵ get?

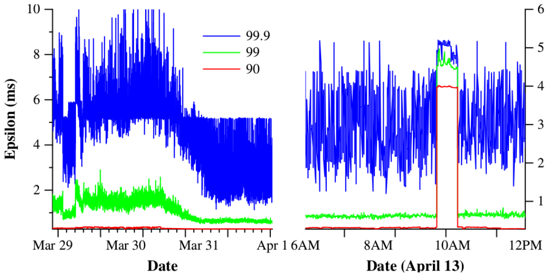

Figure 7 : Distribution of TrueTime ϵ values

Conclusions & Future Work

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

Thank you for your attention!

Questions?