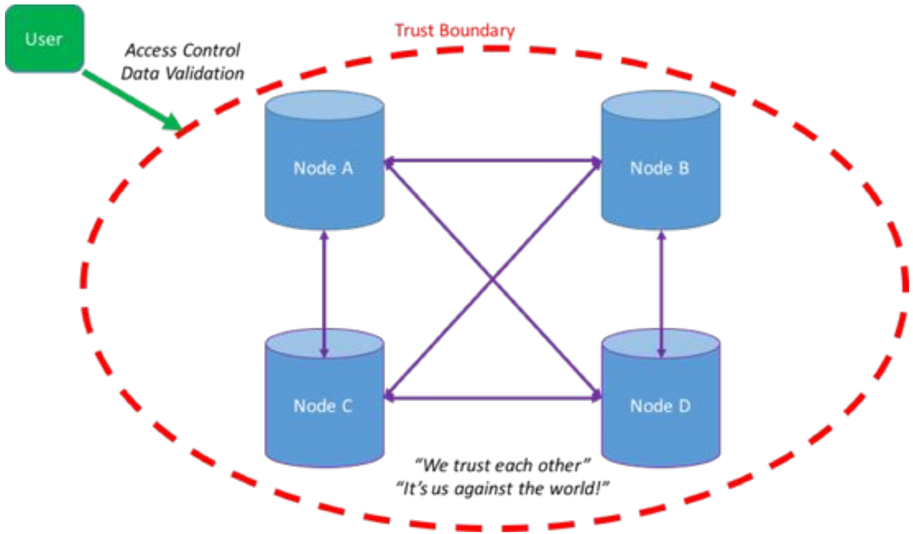
Spanner: Google's Globally-Distributed Database

Google, Inc.

Nikhilesh Murugavel
04/09/2019

What is a Distributed Database?

Two or more files located in different sites either on the same network or on entirely different networks.



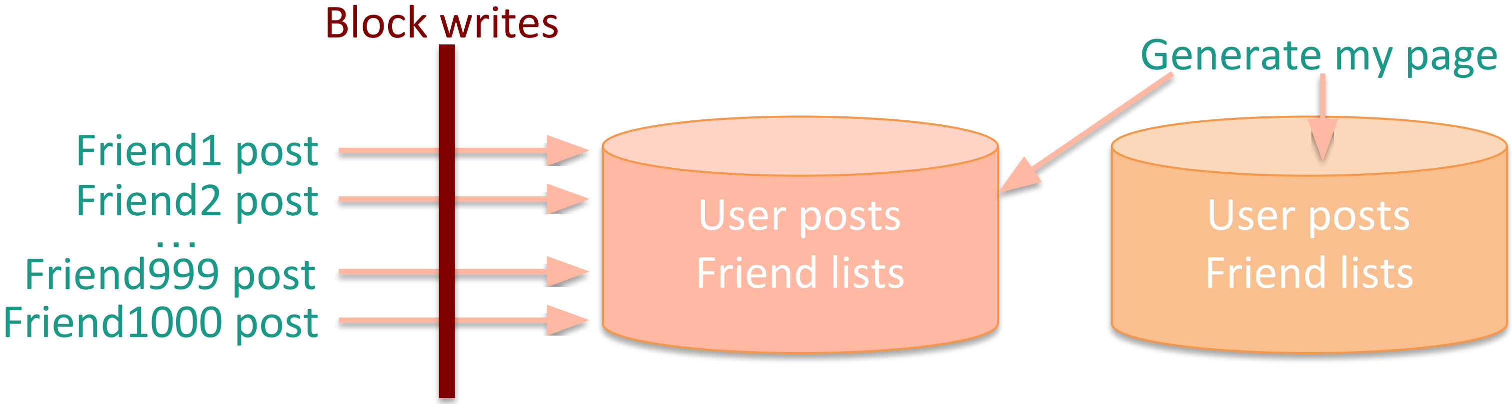
Problem?

Consistency..

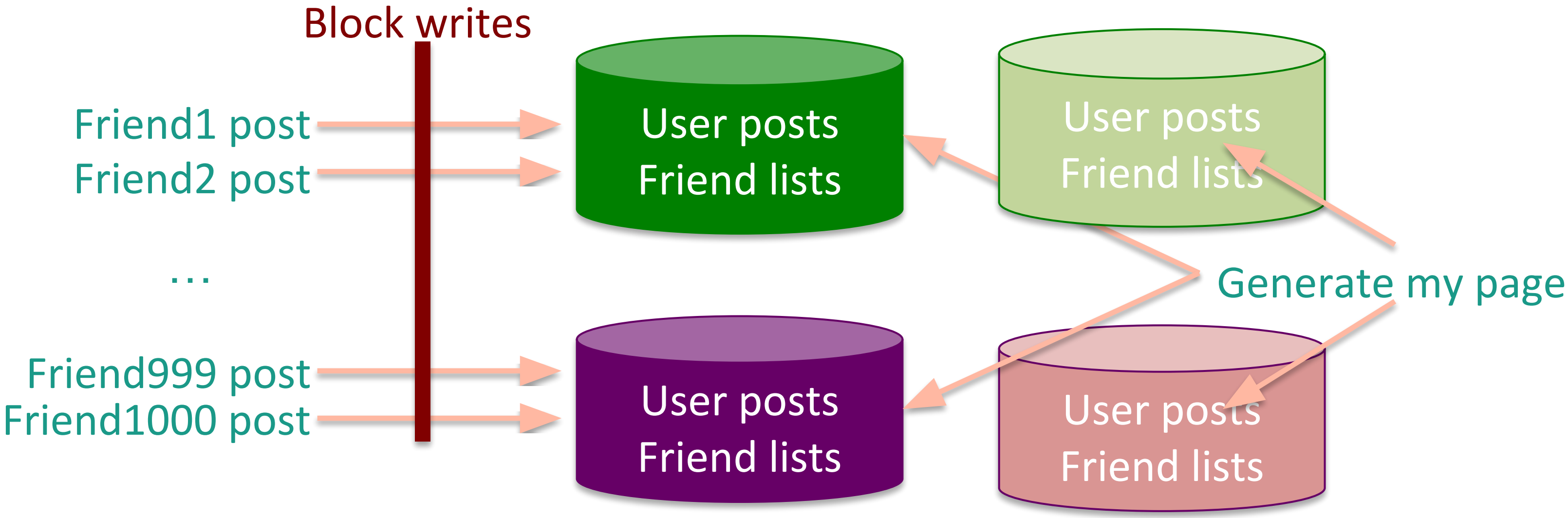
Why consistency matters?

- Generate a page of friends’ recent posts
    - Consistent view of friend list and their posts

Single Machine



Multiple Machines



Block writes

Friend1 post

Friend2 post

…

Friend999 post

Friend1000 post

User posts
Friend lists

User posts
Friend lists

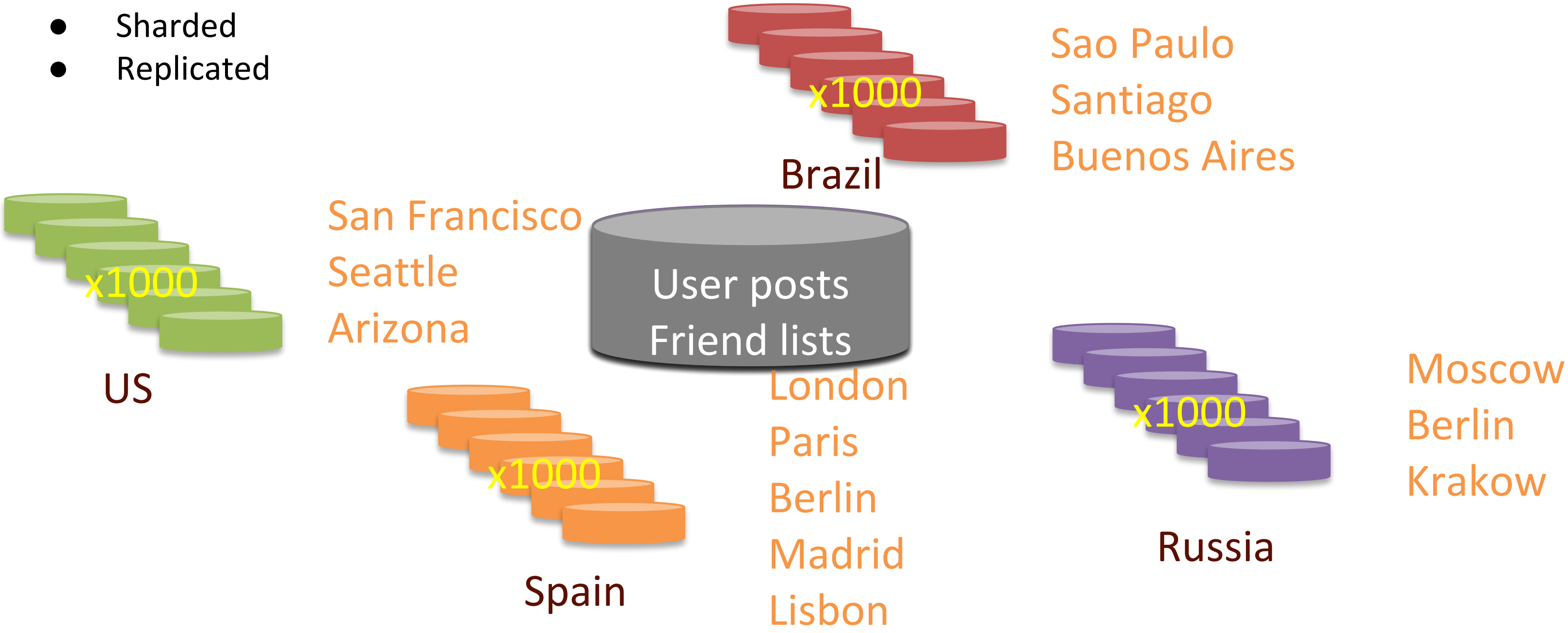
User posts
Friend lists

User posts
Friend lists

Generate my page

Example

- Distributed
- Sharded
- Replicated



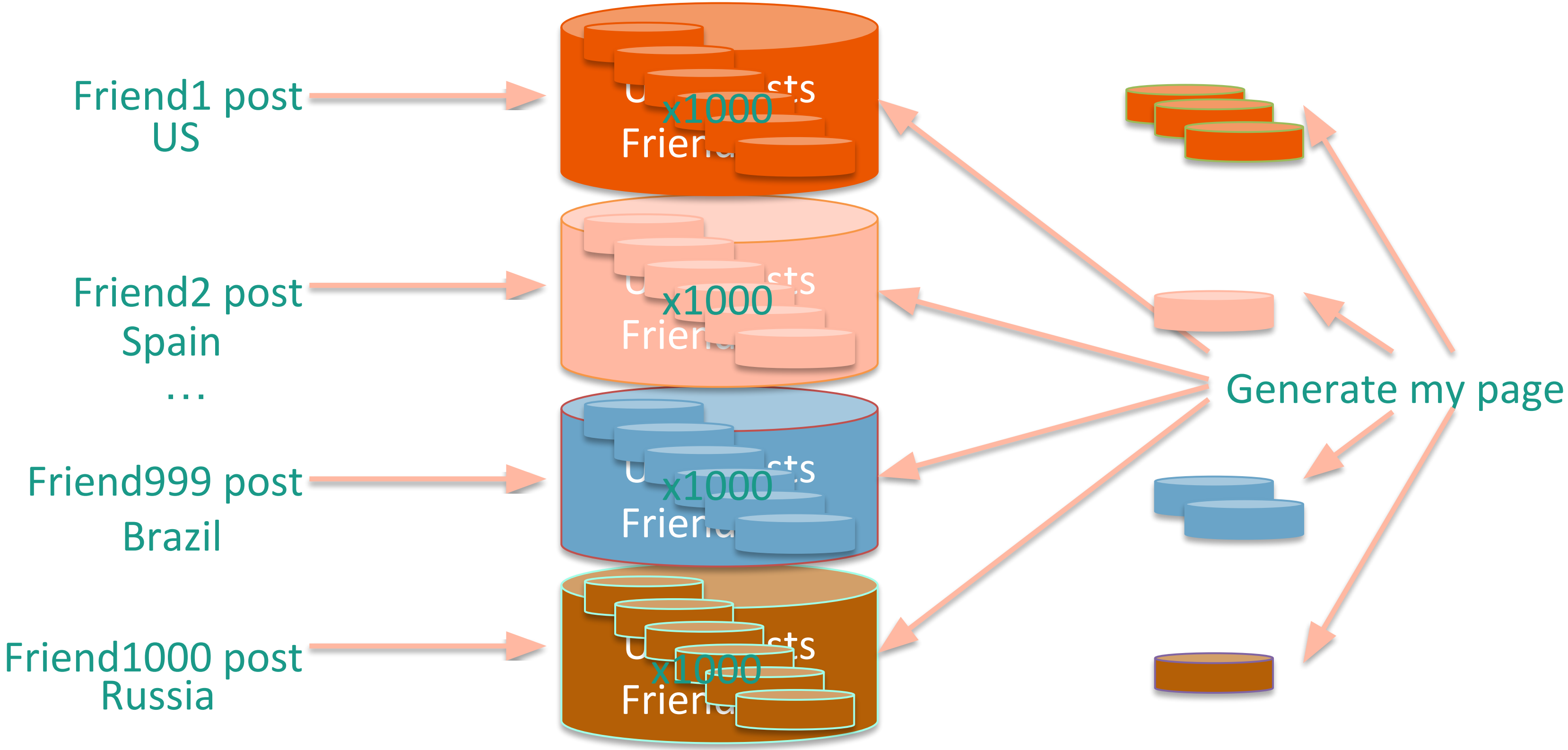
Sao Paulo
Santiago
Buenos Aires

San Francisco
Seattle
Arizona

London
Paris
Berlin
Madrid
Lisbon

Moscow
Berlin
Krakow

Multiple Datacenters



Friend1 post
US

Friend2 post
Spain

...

Friend999 post
Brazil

Friend1000 post
Russia

Generate my page

What is Spanner?

- Schematized tables
- Semi-relational data model
- General purpose transactions - ACID
- SQL based

Running in Production

- Storage for Google’s ad data
- Replaced a sharded MySQL database

More on Spanner

- Automatic load balancing
- Client application configurable
    - How far the data is from users
    - How far replicas are from each other
    - How many replicas to be maintained
- Consistent backups and atomic schema updates
- Globally meaningful commit timestamps to reflect serialization order

What makes Spanner a good Distributed Database

Scalable: Horizontally scalable across rows, regions, and continents, from 1 to hundreds or thousands of nodes

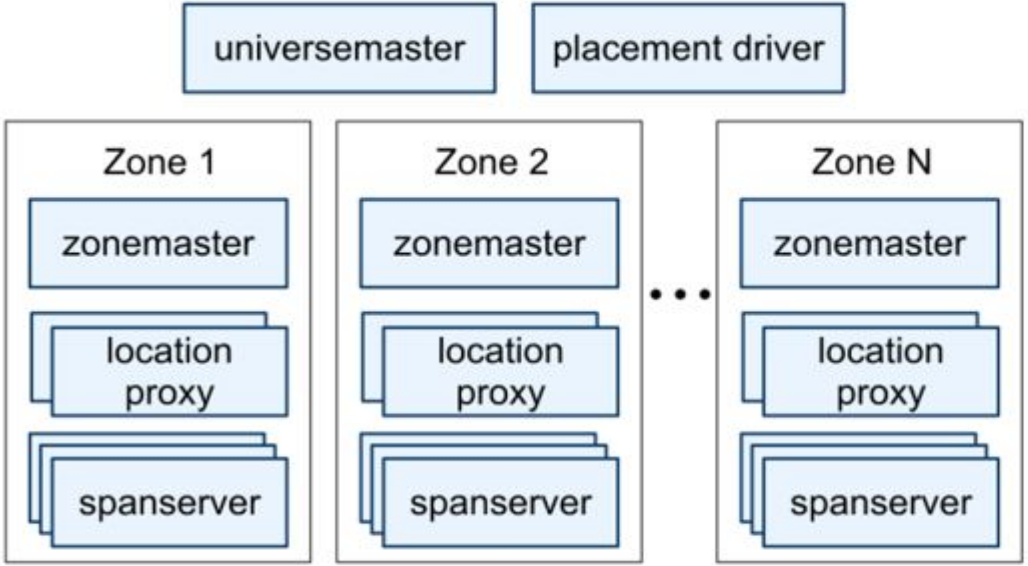
Multi-version: Data is versioned with timestamps

Globally distributed: Across continents, across hundreds of data centers, across thousands of machines

Synchronously-replicated: Copying data so there are multiple, up-to-data replicas of the data

Externally-consistent distributed transactions: The system behaves as if all transactions were executed sequentially

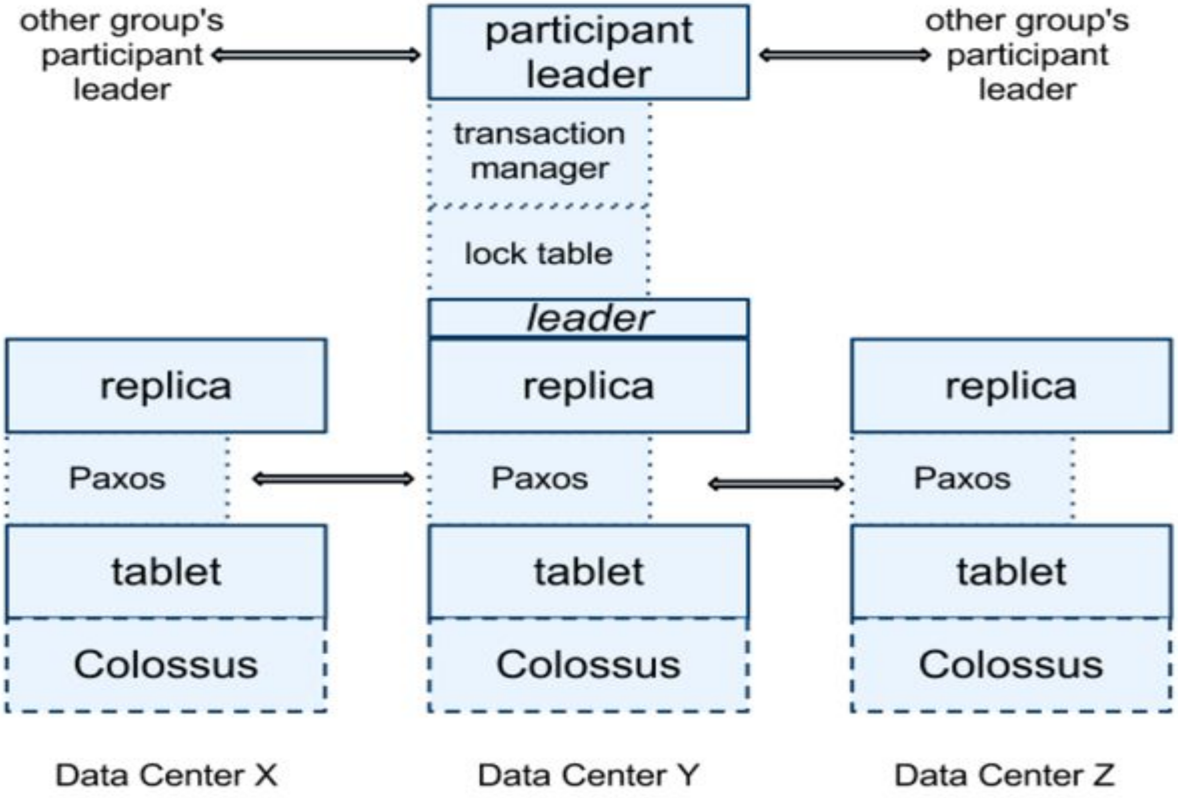
Implementation - Server Organization



Implementation

- A spanner deployment is called a *universe*
  - Auto shards and auto rebalances data across many sets of Paxos machines
- Consists of a set of *zones*
  - Locations across which data can be replicated
  - Physical isolation
- A zone has one *zonemaster* and possibly 1000s of *spanservers*
  - The former assigns data to spanservers; the latter serve data to clients
  - *Location proxies* are used by client apps to locate spanservers holding their data
- *Universe master*
  - Contains status information about all the zones
- *Placement driver*
  - Handles data movement across zones

Spanserver



(key:string, timestamp:int64) → string

Directories

- Unit of data movement across Paxos groups
- *Bucket* that stores a set of contiguous keys with common prefix
- Move a directory
    - Load balancing
    - Frequently accessed directories
    - Closer to accessors
- Done in the background

The research

Feature: Lock-free distributed read transactions

Property: External consistency of distributed transactions

- First system at global scale

Implementation: Integration of concurrency control, replication, and 2PC

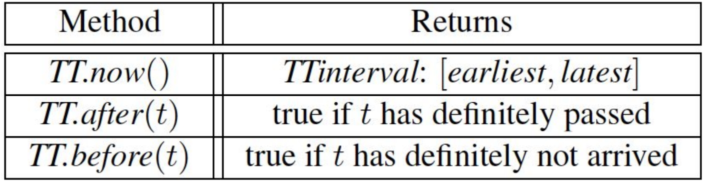
- Correctness and performance

Enabling technology: TrueTime

- Interval-based global time

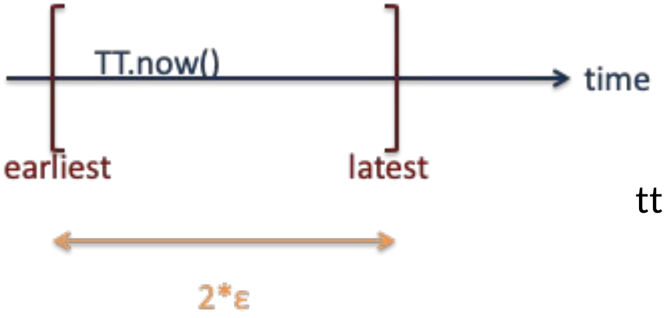
TrueTime

- *Global wall clock time* that reflects uncertainty
- Returns a time interval bounded with uncertainty
- Clocks - GPS clocks and atomic clocks
- *Daemon* polls a number of masters and uses Marzullo's alg



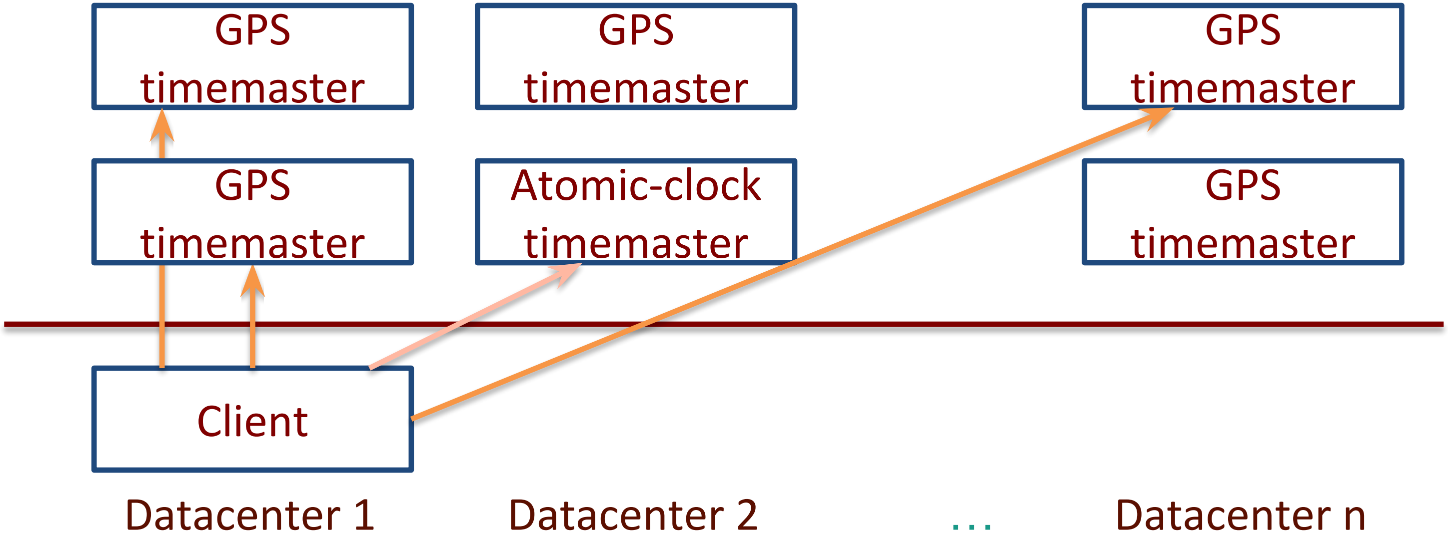
Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(t)	true if t has definitely passed
TT.before(t)	true if t has definitely not arrived

Table 1: TrueTime API. The argument t is of type *TTstamp*.



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

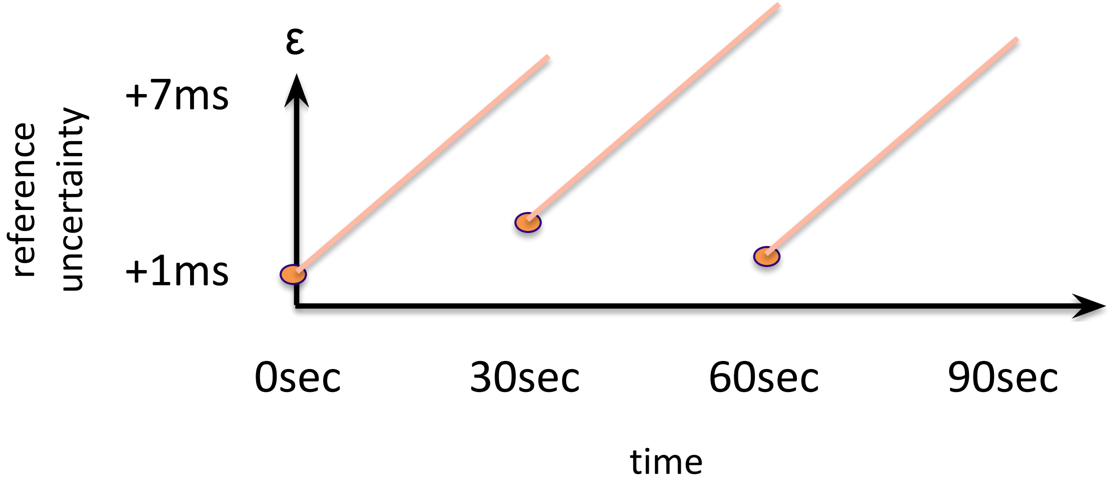
TrueTime Architecture



Compute reference [earliest, latest] = now ± ε

TrueTime Implementation

- ε is a sawtooth function of time, varying from 1 to 7 ms
- Average of 4 ms



200 µs/sec

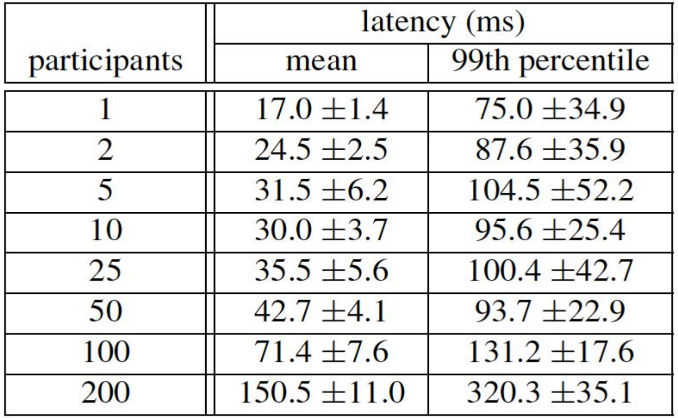
Concurrency Control - Read-Only Transactions

- Read operations execute at a system-chosen timestamp without locking
- Single paxos: Client issues the transaction to the group leader
    - S_{read} = LastTS()
- Multiple paxos: Client may wait for safe time to advance
    - S_{read} = TT.now().latest
- Reads can proceed on any replicas that is up-to-date

Concurrency Control- Read-Write Transactions

- Writes are buffered at the client until commit
    - Reads do not see the effects of the transaction’s writes
- Reads are issued to the leader replicas
    - Acquires read locks and reads the most recent data
- Two-phase commit at the end of the transaction
    - Client chooses a coordinator group and sends commit message
- Non-coordinator participant leaders
    - Write locks, prepare timestamps
- Coordinator leaders
    - Write locks,commit timestamp >= all prepare timestamps

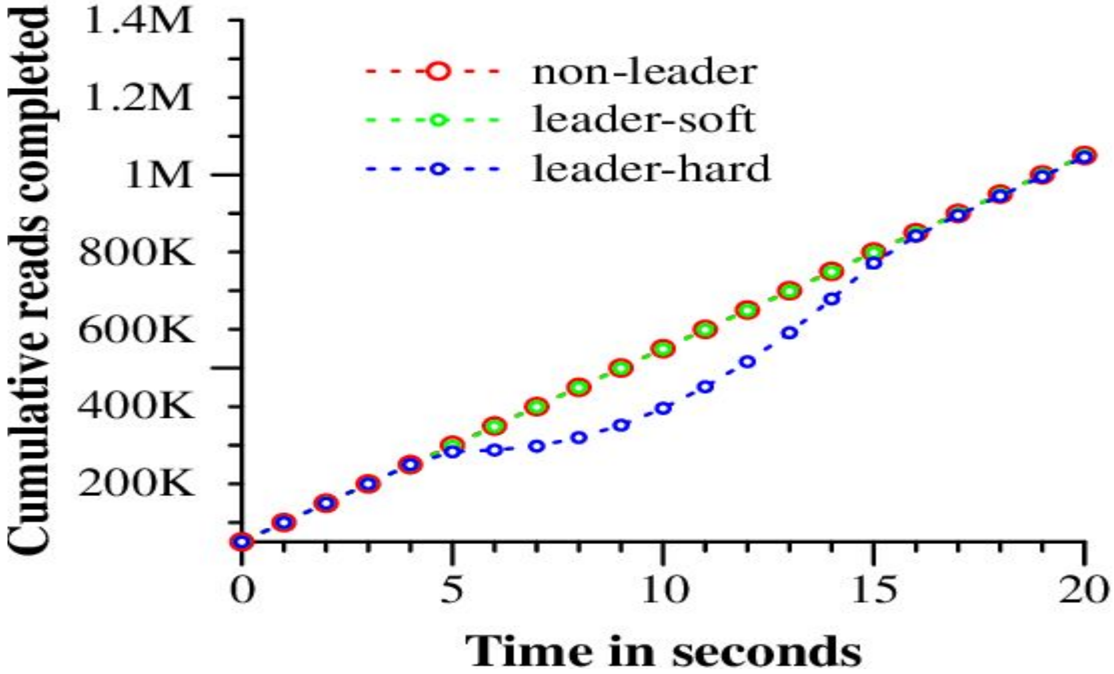
Scalability



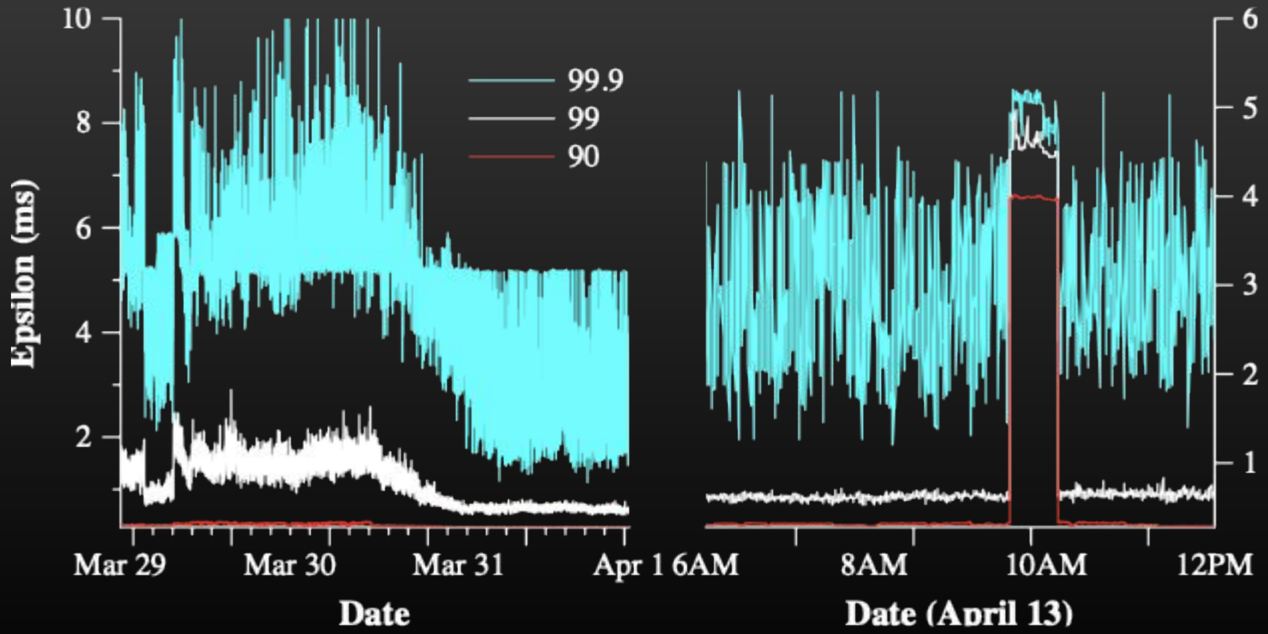
participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

Table 4: Two-phase commit scalability. Mean and standard deviations over 10 runs.

Availability



Network Induced Uncertainty



To Summarize

- Ensure consistency
- TrueTime API
- Strong distributed design with high synchronization
- Integrates database features with systems features

What Could Have Been Better

- Write API is not strictly SQL, read is SQL however
- Write latency is high
    - How it competes with less consistent data stores that provide less write latency
- Heavily based on Google systems
- Strongly based on the 200 microseconds/sec drift