CSE 487/587
Data Intensive Computing

Lecture 15: BigTable/Spanner
Part 3 of noSQL

Vipin Chaudhary

vipin@buffalo.edu

716.645.4740
305 Davis Hall

Overview of Lecture Series

- Chubby
- BigTable
- Spanner
- Zookeeper
- Dynamo

But before we start

- Pop quiz!

BigTable: A System for Distributed Structured Storage

What is BigTable?

- BigTable is a distributed Database
- A more application-friendly storage service
- Data in Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank,…
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …

Question: Why not just use commercial DB ?

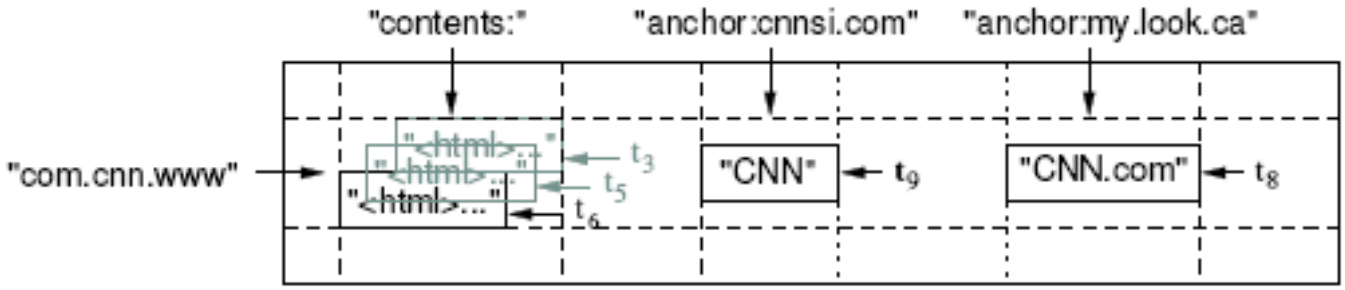
- TeraData, Oracle, MySql with sharding
- Data is large
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands of q/sec
  - PB+ of satellite image data
- Many incoming requests
- Scale to thousands of machines

  - 450,000 machines (NYTimes estimate, June 14th 2006)
- No commercial system big enough
  - Cost is too high
  - Might not have made appropriate design choices
  - Low-level storage optimizations help improving performance

Goals of BigTable

- Need to support:
  - Data is highly available at any time
  - Very high read/write rates
  - Efficient scans over all or interesting subsets of data
  - Asynchronous and continuously updates
  - High Scalability

Data model: a big map



- BigTable is a distributed multi-dimensional sparse map
  (row, column, timestamp) → cell contents
- Provides lookup, insert, and delete API
  - *Row keys and Column keys are strings*
- Arbitrary "columns"
  - Column family:qualifier
    - Column-oriented physical store
- Does not support a relational model
  - No table-wide integrity constraints
  - No multirow transaction

Timestamps

- Timestamps allow version index
  - 64-bit integers
  - Can be assigned by BigTable or Client
  - Versions stored in decreasing timestamp order

- Two per-column-family settings
  - client can specify either that only the last n versions of a cell be kept,
  - or that only new-enough versions be kept

APIs

- Creating and deleting tables and column families.
- Changing cluster, table, and column family metadata, such as access control rights
- Writing to big table

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");
// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

- RowMutation abstraction to perform a series of updates
- Call to Apply performs an atomic mutation to the Webtable: it adds one anchor to www.cnn.com and deletes a different anchor.

APIs

- Reading from big table

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
printf("%s %s %lld %s\n",
scanner.RowName(),
stream->ColumnName(),
stream->MicroTimestamp(),
stream->Value());
}
```

- Scanner abstraction to iterate over all anchors in a particular row.
- Clients can iterate over multiple column families, and there are several mechanisms for limiting the rows, columns, and timestamps produced by a scan.
- For example, we could restrict the scan above to only produce anchors whose columns match the regular expression anchor:*.cnn.com, or to only produce anchors whose timestamps fall within ten days of the current time.

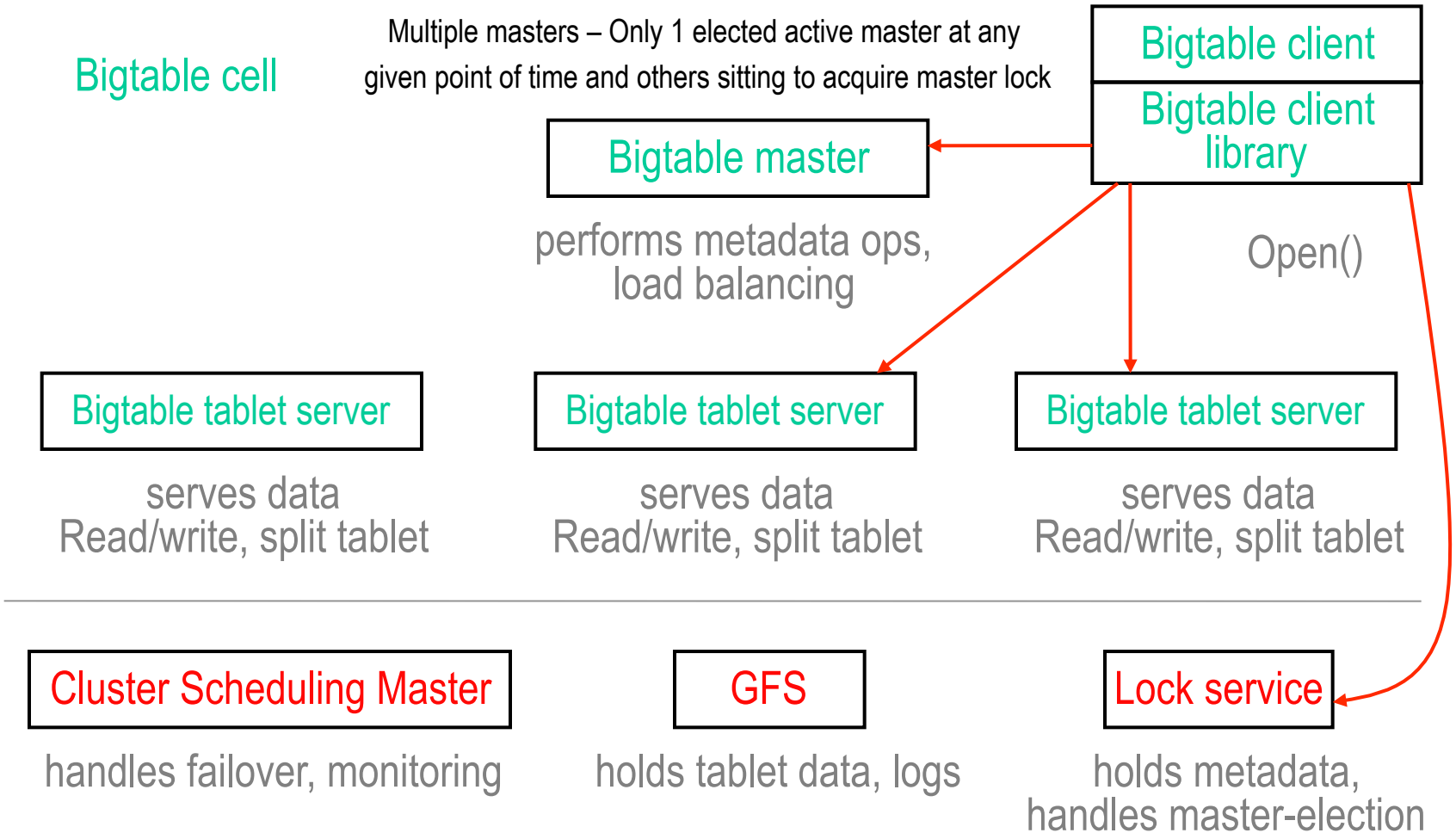
APIs

- Supports single-row transactions
  - can be used to perform atomic read-modify-write sequences on data stored under a single row key
  - does not currently support general transactions across row keys,
  - But provides an interface for batching writes across row keys at the clients
- Supports execution of client-supplied scripts
  - The scripts are written in Sawzall, a language developed at Google
- Set of wrappers that allow a Bigtable to be used both as an input source and as an output target for MapReduce jobs.

Building Blocks

- Scheduler (Google WorkQueue)
- Google Filesystem
  - SSTable file format
- Chubby
  - {lock/file/name} service
  - Coarse-grained locks
  - discover tablet server
  - store meta data of tablets
- MapReduce

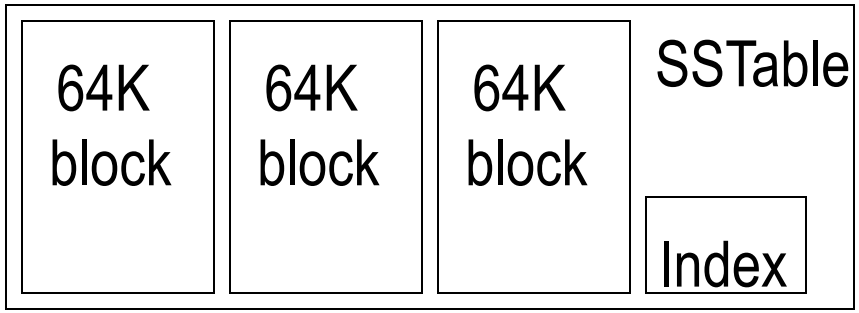
Tablet Serving Structure



Where is data stored

SSTable (**"Static and Sorted Table"**)

- Sorted file of key-value string pairs
- Chunks of data plus an index
  - Index is of block ranges, not values
  - triplicated across three machines in GFS



- block index (stored at the end of the SSTable) is used to locate blocks
  - index is loaded into memory when the SSTable is opened
- lookup can be performed with a single disk seek:
  - First find the appropriate block by performing a binary search in the in-memory index, and
  - then reading the appropriate block from disk.
- Optionally, an SSTable can be completely mapped into memory

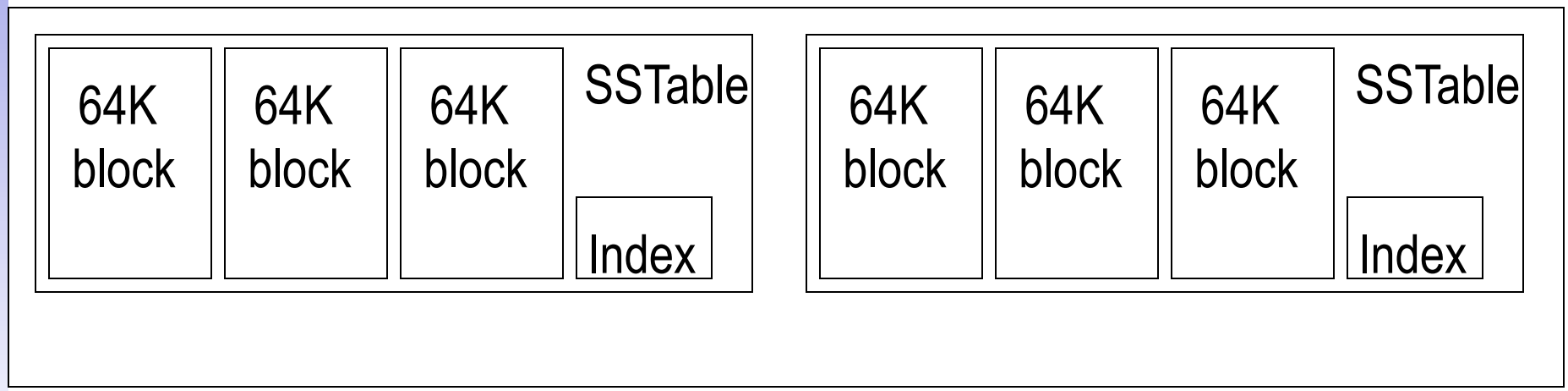
How is Chubby used?

- Ensure at most one active master at any time
- Store the bootstrap location of Bigtable data
- Discover tablet servers and finalize tablet server deaths
- Store Bigtable schema information (the column family information for each table);
- Store access control lists.
- If Chubby unavailable for extended period of time, Bigtable becomes unavailable.
- Measured this effect in 14 Bigtable clusters spanning 11 Chubby instances.
  - average percentage of Bigtable server hours during which some data stored in Bigtable was not available due to Chubby unavailability (caused by either Chubby outages or network issues) was 0.0047%.
  - The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

Tablet

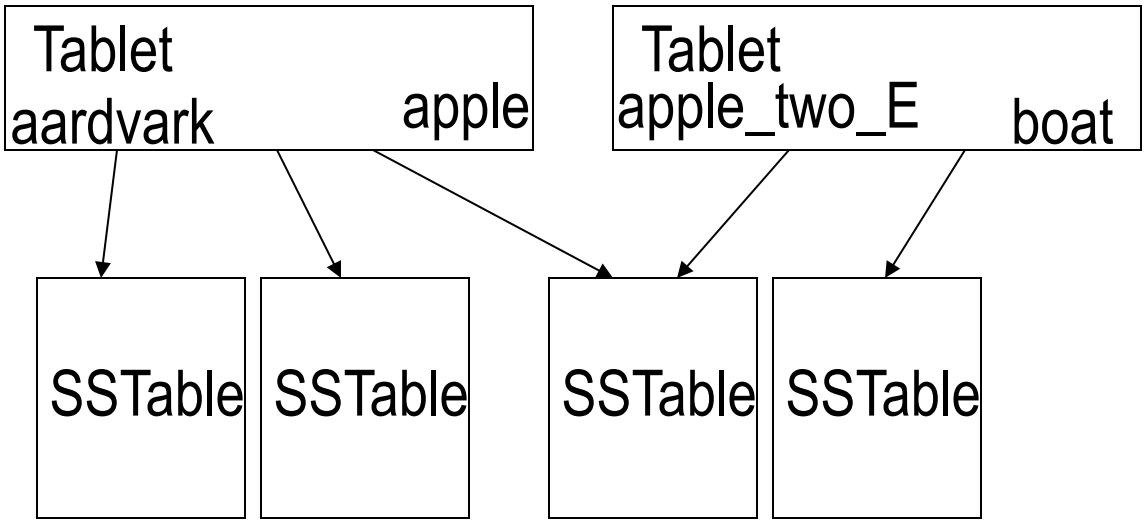
- Contains some range of rows of the table
- Built out of multiple SSTables
- Typical size: 100-200 MB
- Tablets are stored in Tablet servers (~100 per server)

Tablet Start:aardvark End:apple



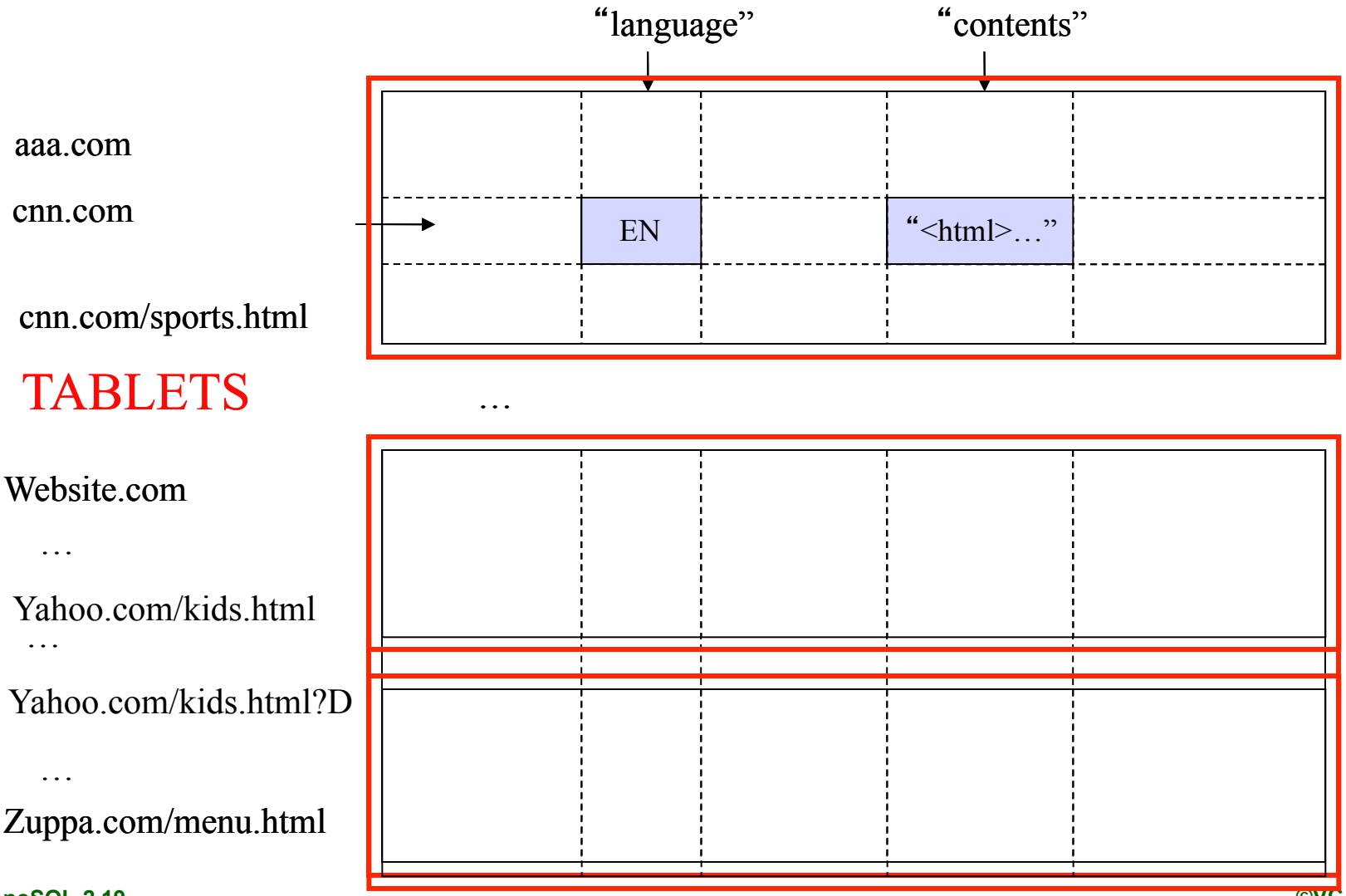
Table

- Multiple tablets make up the table
- SSTables can be shared (pointer)
- Tablets do not overlap, SSTables can overlap

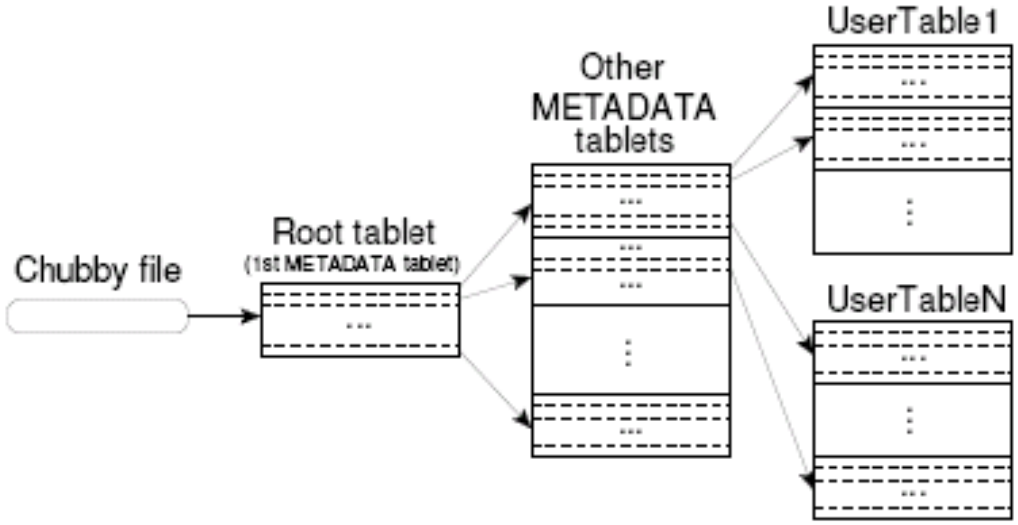


Tablets & Splitting

Large tables broken into *tablets* at row boundaries



Locating Tablets



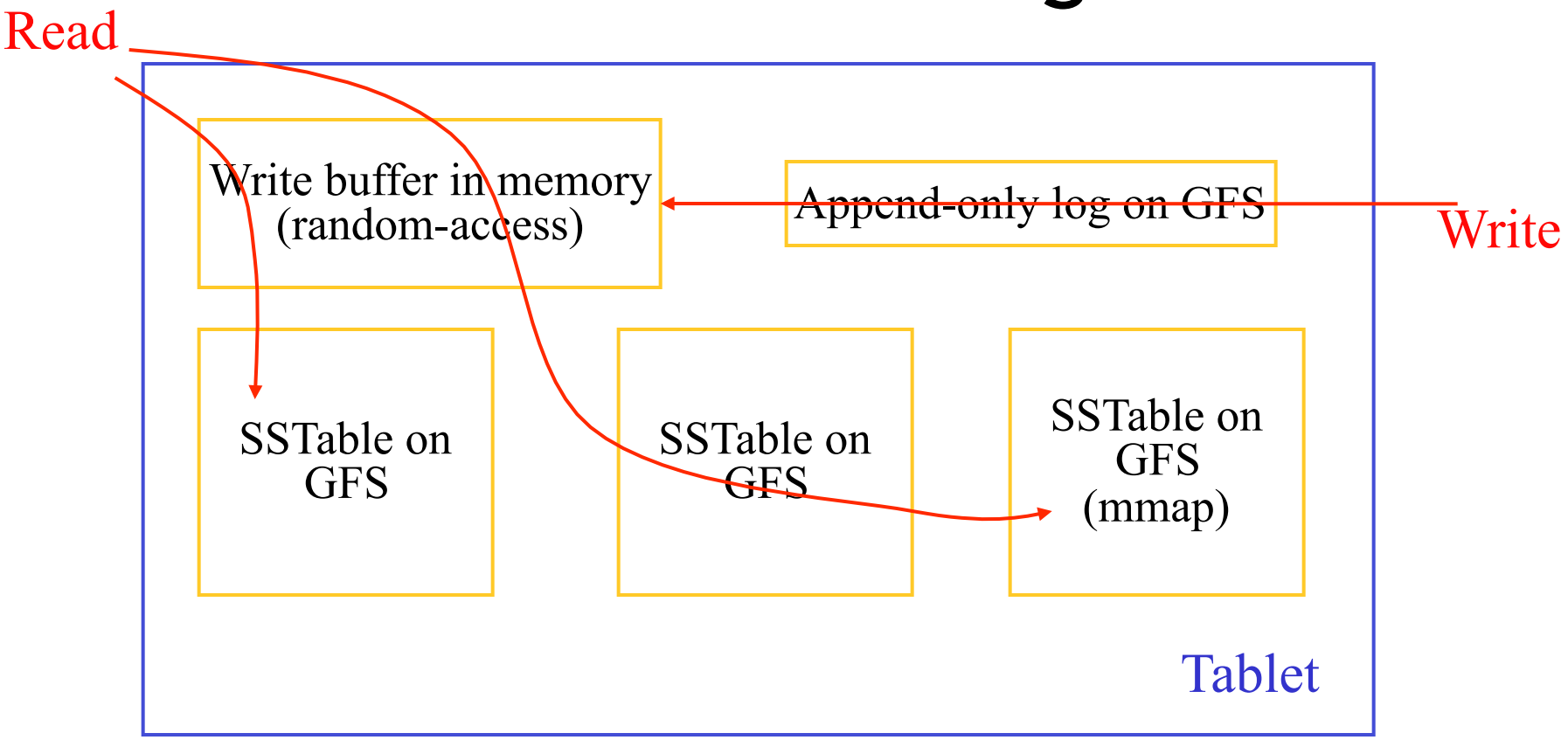
Approach: 3-level hierarchical lookup scheme for tablets

- Location is *ip:port* of relevant server, all stored in META tablets
- 1st level: bootstrapped from lock server, points to owner of META0
- 2nd level: Uses META0 data to find owner of appropriate META1 tablet
- 3rd level: META1 table holds locations of tablets of all other tables
  - META1 table itself can be split into multiple tablets

Client caches tablet locations

- Each METADATA row stores ~1KB of data in memory.
- With a modest limit of 128 MB METADATA tablets, the three-level location scheme is sufficient to address 2^{34} tablets (or 2^{61} bytes in 128 MB tablets).

Tablet Serving



Writes

- Updates committed to a commit log
- Recently committed updates are stored in memory (memtable)
- Older updates are stored in a sequence of SSTables.

Reads

- form a merged view of SSTables in memory
- read <key-value> pair

Fault tolerance and load balancing

- Master responsible for load balancing and fault tolerance
  - GFS replicates data
  - Use Chubby to keep locks of tablet servers, restart failed servers
  - Master checks the status of tablet servers
  - Keep track of available tablet servers and unassigned tablets
  - if a server fails, start tablet recovering

- Recovering tablet
  - New tablet server reads data from METADATA table.
  - Metadata contains list of SSTables and pointers into any commit log that may contain data for the tablet.
  - Server reads the indices of the SSTables in memory
  - Reconstructs the memtable by applying all of the updates since redo points.

Implementation Refinements - Locality Groups

- Group column families together into an SSTable
  - Avoid mingling data
  - Can keep some groups all in memory
- Can compress locality groups
- Bloom Filters on locality groups – avoid searching SSTable

  - space-efficient

  - can test membership of a set

  - False positives are possible but false negatives are not

Refinements - Shared Logs for editing a table

- Mutations are logged, then applied to an in-memory version
- Logfile stored in GFS

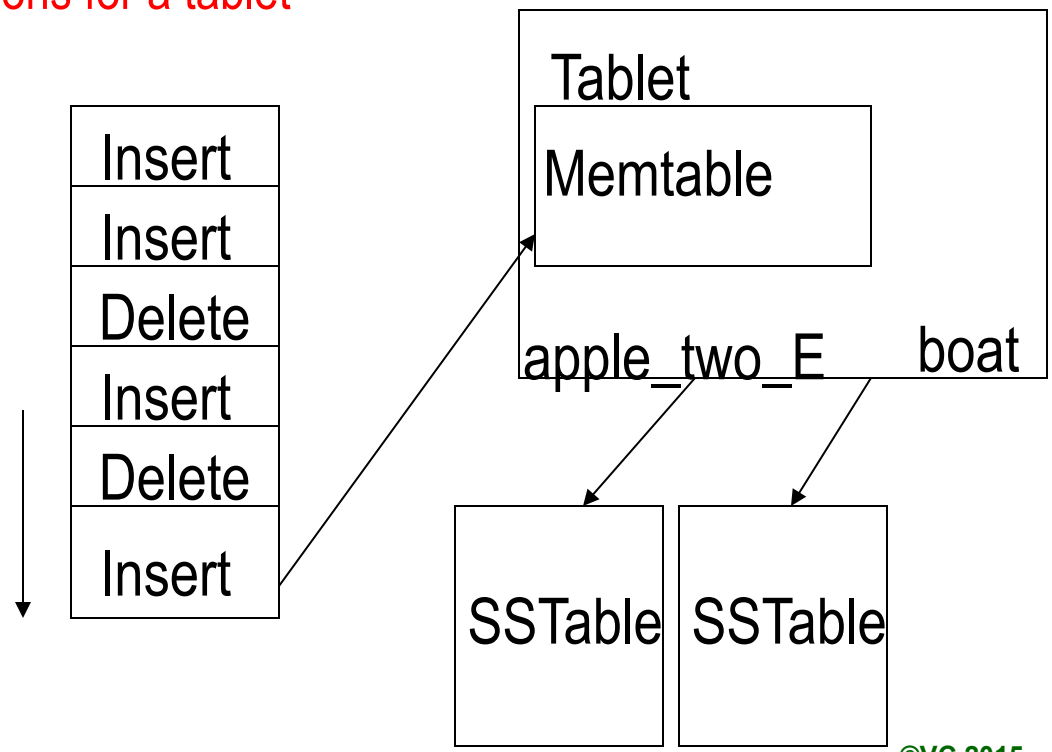
  -Write log file per tablet server instead of per tablet

Any Problem for doing this?

Problem: during recovery, server needs to read log data to apply mutations for a tablet

Solution:

- Master aggregates and sort the needed chunks by tablet in the log
- Other tablet servers ask master which servers have sorted chunks they need
- Tablet servers issue direct RPCs to peer tablet servers to read sorted data efficiently



Refinements - Compression

- Many opportunities for compression
  - Similar values in the same row/column at different timestamps
  - Similar values in different columns
  - Similar values across adjacent rows

- Within each SSTable for a locality group, encode compressed blocks
  - Keep blocks small for random access (~64KB compressed data)
  - Exploit the fact that many values very similar
  - Needs to be low CPU cost for encoding/decoding

Microbenchmarks- setup

- Tablet servers were configured to use 1 GB of memory and to write to a GFS cell
  - 1786 machines with two 400 GB IDE hard drives each.
- N client machines generated the Bigtable load used for these tests.
  - used the same number of clients as tablet servers to ensure that clients were never a bottleneck.
  - each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single 1 GbE
- Machines arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root.
  - All machines in same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.
- The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server.
- R is the distinct number of Bigtable row keys involved in the test.
  - R was chosen so that each benchmark read or wrote approximately 1 GB of data per tablet server.

Microbenchmarks- speed per server

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Number of 1k-byte values read/written per second per server

- Sequential write benchmark used row keys with names 0 to R-1 .
- Row keys partitioned into 10N equal-sized ranges.
- These ranges were assigned to the N clients by a central scheduler that assigned the next available range to a client as soon as client finished processing the previous range assigned to it.
- Dynamic assignment helped mitigate the effects of performance variations caused by other processes running on the client machines.
- Wrote a single string under each row key.
- Each string was generated randomly and was therefore uncompressible.
- In addition, strings under different row key were distinct, so no cross-row compression was possible.

Microbenchmarks- speed per server

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Number of 1k-byte values read/written per second per server

- Random write benchmark similar to sequential write except
  - row key was hashed modulo R immediately before writing
    - so that the write load was spread roughly uniformly across the entire row space for the entire duration of the benchmark.

Microbenchmarks- speed per server

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Number of 1k-byte values read/written per second per server

- sequential read benchmark generated row keys in exactly the same way as the sequential write benchmark, but
  - instead of writing under the row key, it read the string stored under the row key (which was written by an earlier invocation of the sequential write benchmark)

- Similarly, random read benchmark shadowed the operation of the random write benchmark.

Microbenchmarks- speed per server

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Number of 1k-byte values read/written per second per server

- The scan benchmark is similar to the sequential read benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range.
- Using a scan reduces the number of RPCs executed by the benchmark since a single RPC fetches a large sequence of values from a tablet server.

Microbenchmarks- speed per server

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Number of 1k-byte values read/written per second per server

- random reads (mem) benchmark is similar to the random read benchmark, but
  - the locality group that contains the benchmark data is marked as in-memory
    - the reads are satisfied from the tablet server's memory instead of requiring a GFS read.
  - For just this benchmark, we reduced the amount of data per tablet server from 1GB to 100 MB so that it would fit comfortably in the memory available to the tablet server.

Microbenchmarks- speed per server

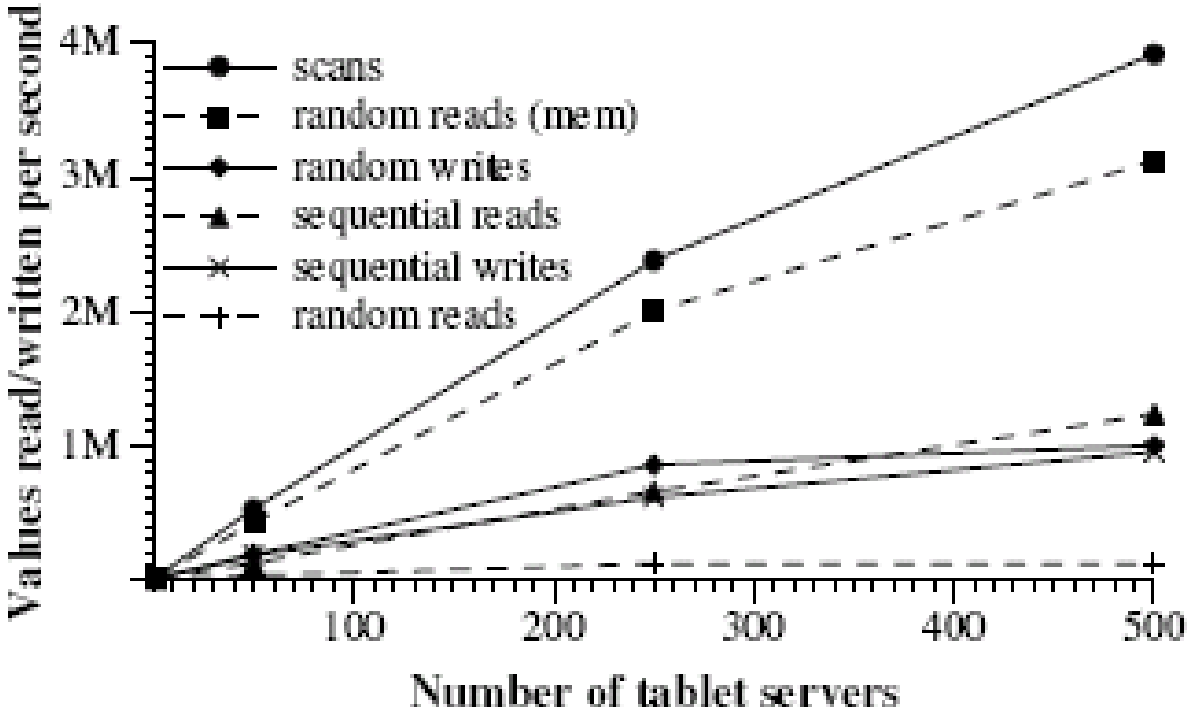
Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Number of 1k-byte values read/written per second per server

Remarks:

1. Random reads are 10-times slower than other ops
2. Writes are faster than reads (why?)
   - each tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes efficiently to GFS
3. Random reads in memory is the fastest

Microbenchmarks- scaling



Number of 1k-byte values read/written per second in total

Remarks:

1. Sequential reads and writes behave like linearly, others are sub-linear
2. Performance order doesn't change compared to per-server
    1. caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network.

Application at Google

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Discussion

- Will it work over Internet? (clusters spread across geographically distant

Spanner: Google's Globally-Distributed Database

Terrific presentation by

Wilson Hsieh

representing a host of authors

OSDI 2012

https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

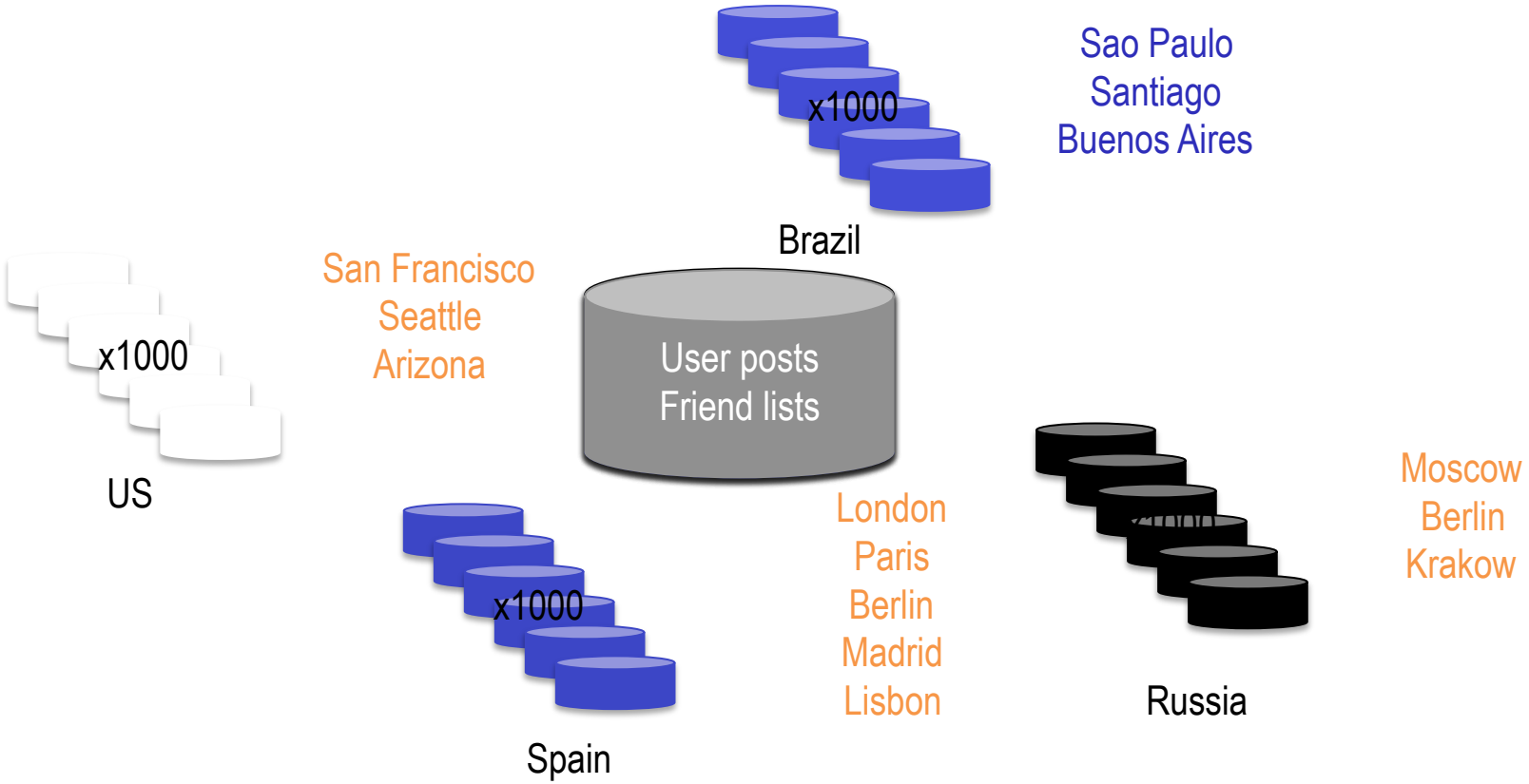
What is Spanner?

- BigTable was difficult to use
  - For complex evolving schema
  - For strong consistency with wide area replication
- Many applications at Google used Megastore because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput
  - Thus the evolution of Spanner from BigTable
- Distributed multiversion database
  - dB that shards data across DCs globally
  - General-purpose transactions (ACID)
  - SQL query language
  - Schematized tables
  - Semi-relational data model
- Running in production
  - Storage for Google's ad data, F1 (5 replicas spread across datacenters in US)
  - Replaced a sharded MySQL database

What is Spanner?

- Auto-sharding, auto-rebalancing, automatic failure response.
- Exposes control of data replication and placement to user/ application.
  - how far data is from its users (control read latency)?
  - how far replicas are from each other (control write latency)?
  - how many replicas are maintained (control durability, availability, and read performance).
  - Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters.
- Enables transaction serialization via global timestamps
  - Acknowledges clock uncertainty and guarantees a bound on it
    - Generally less than 10ms
  - Uses novel TrueTime API to accomplish concurrency control
  - Uses GPS devices and Atomic clocks to get accurate time
  - Enables consistent backups, atomic schema updates during ongoing transactions

Example: Social Network



Overview

- Feature: Lock-free distributed read transactions
- Property: External consistency of distributed transactions
  - Commit order is same as user sees transactions
  - First system at global scale
- Implementation: Integration of concurrency control, replication, and 2PC (2 Phase Commit)
  - Necessary for Correctness and performance
- Enabling technology: TrueTime
  - Interval-based global time
  - Exposes uncertainty in clock

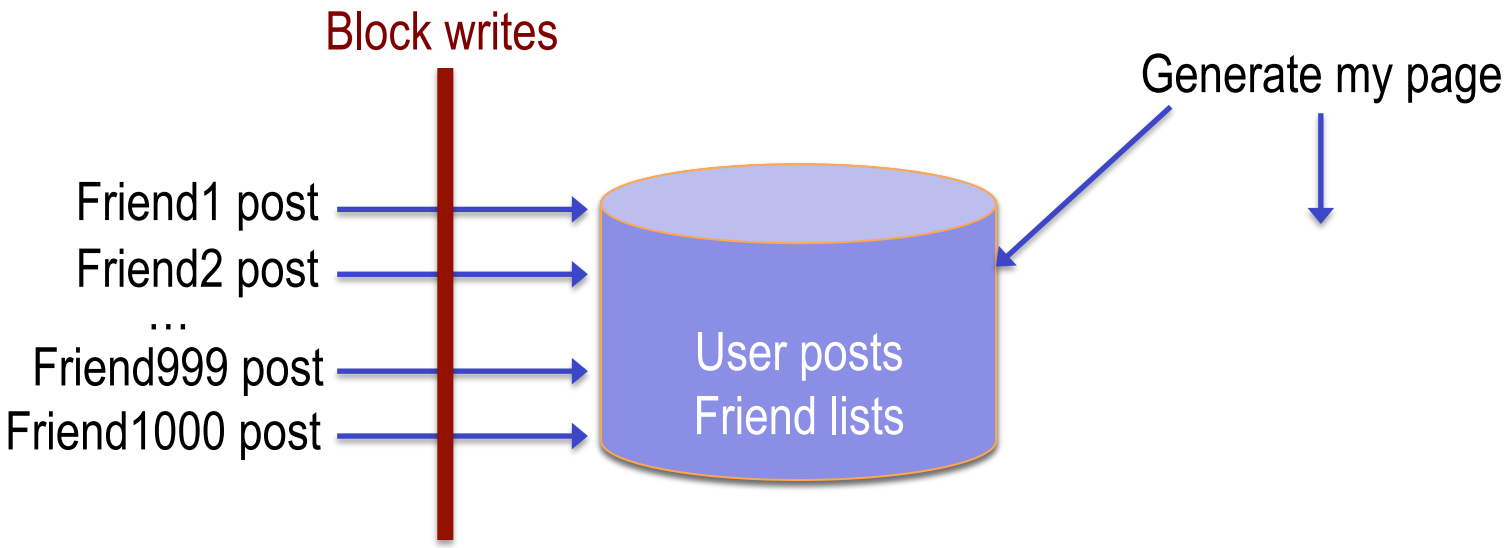
Read Transactions

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

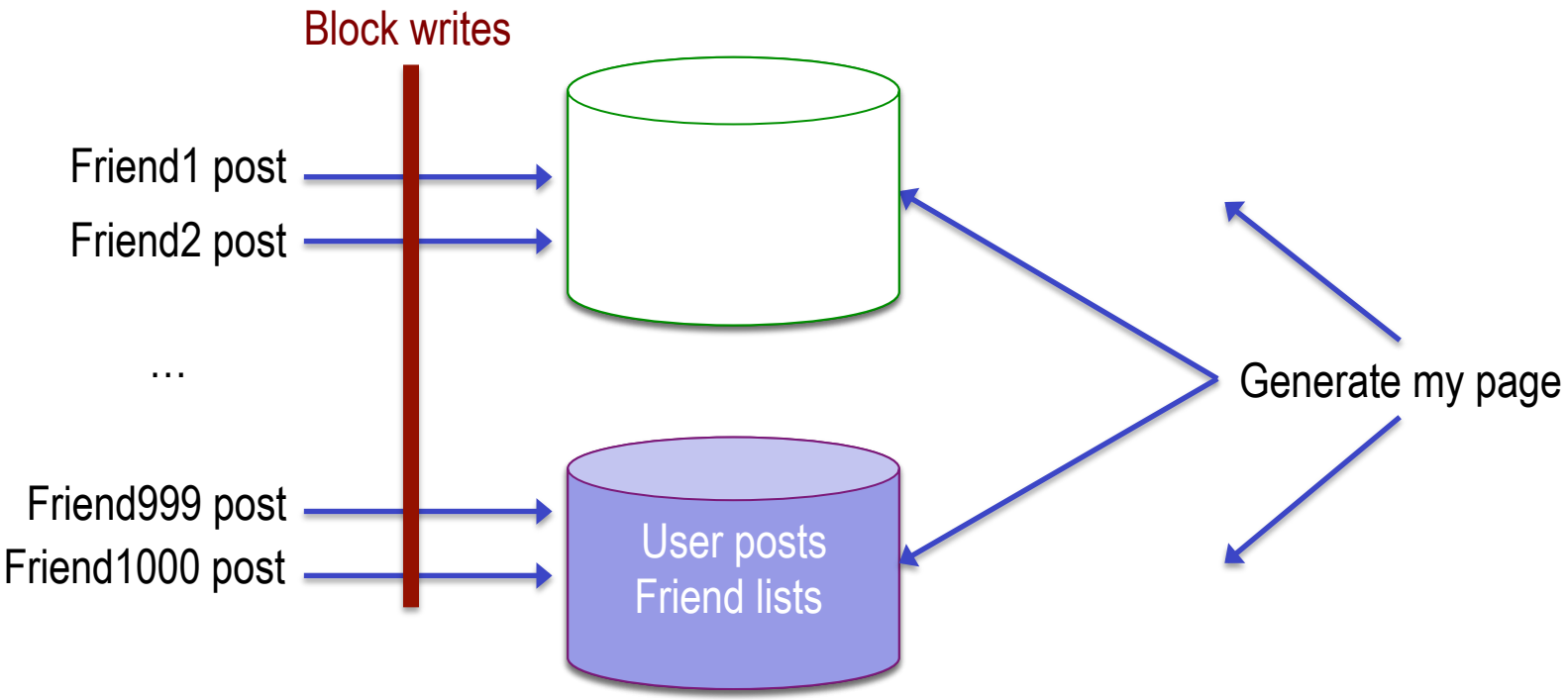
Why consistency matters

1. Remove untrustworthy person X as friend
2. Post P: "My government is repressive…"

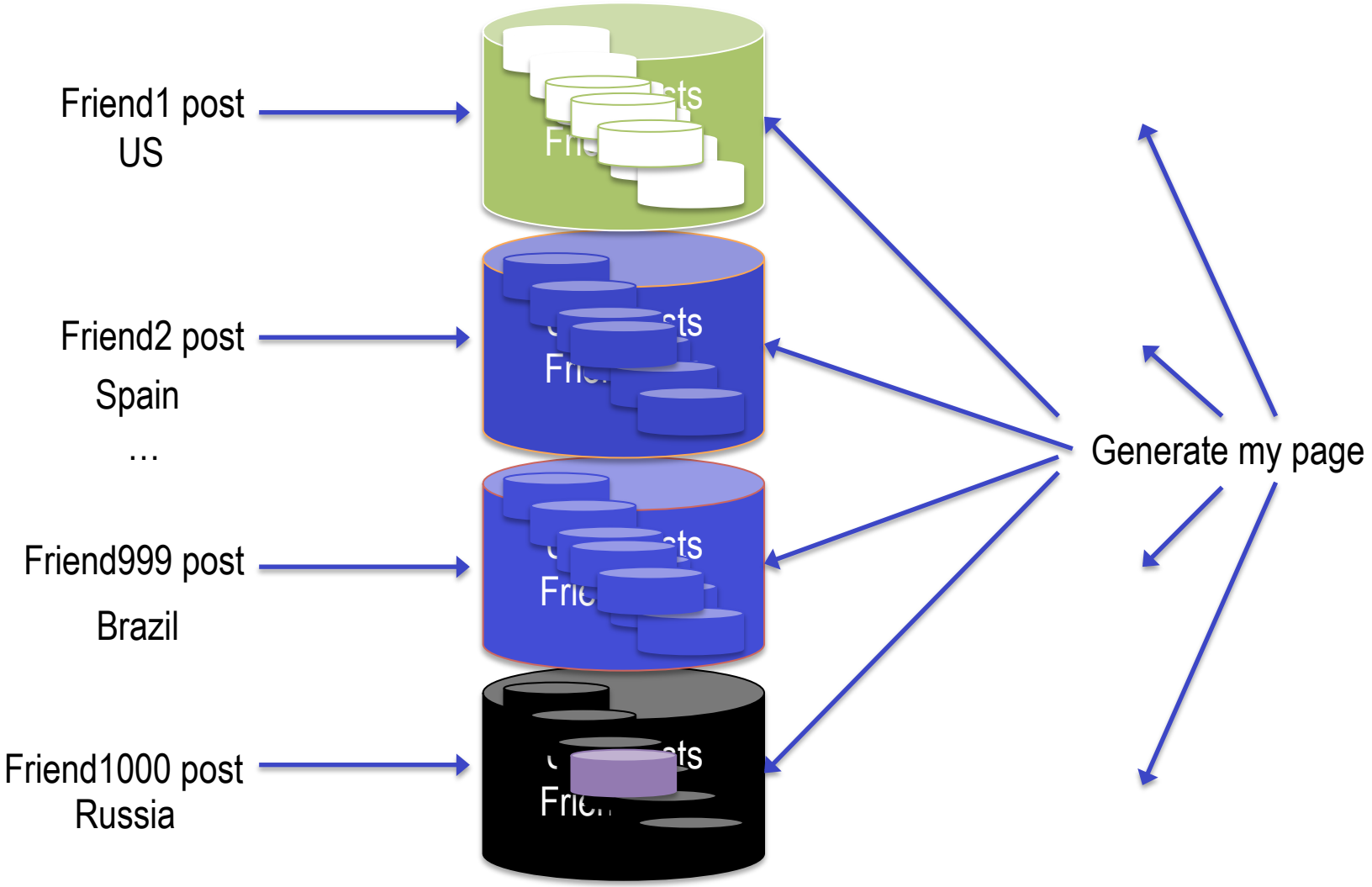
Single Machine



Multiple Machines



Multiple Datacenters



Version Management

- Transactions that write use strict 2PL
  - Each transaction *T* is assigned a timestamp *s*
  - Data written by *T* is timestamped with *s*

Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

Synchronizing Snapshots

Global wall-clock time

==

External Consistency:

Commit order respects global wall-time order

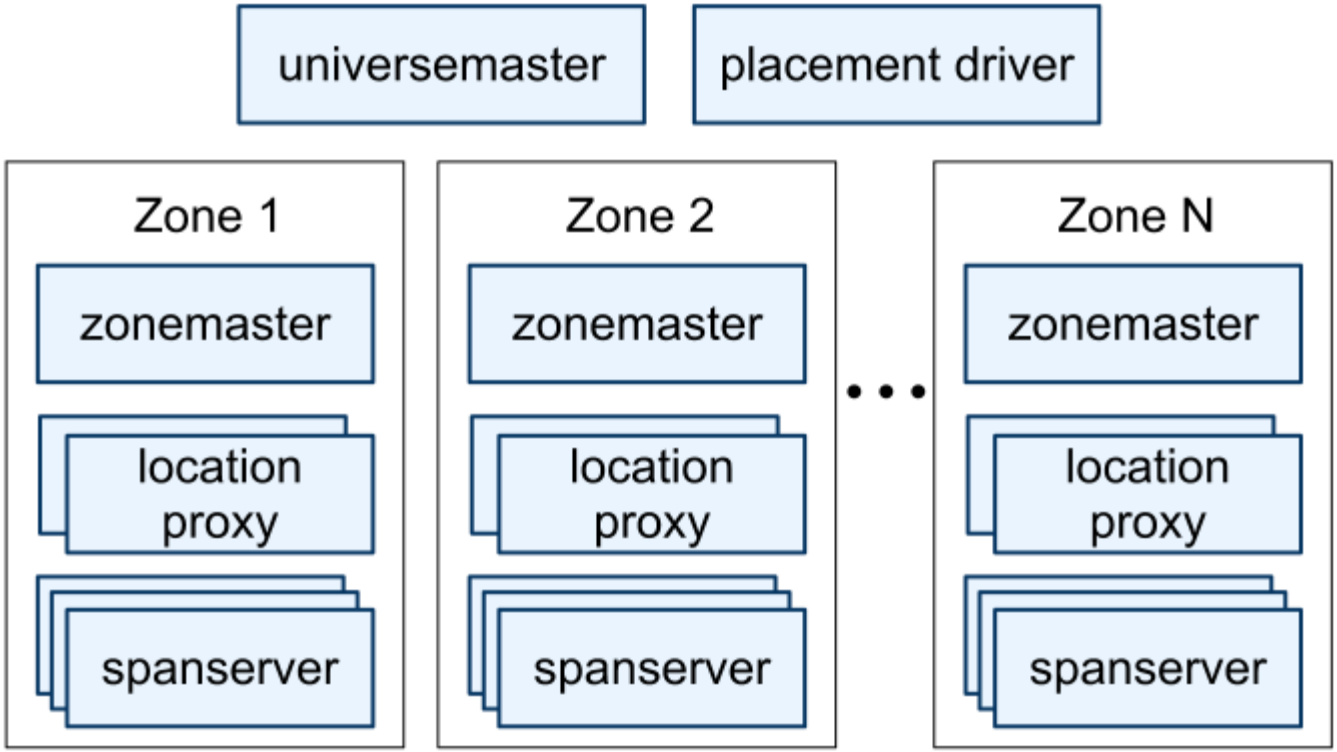
==

Timestamp order respects global wall-time order

given

timestamp order == commit order

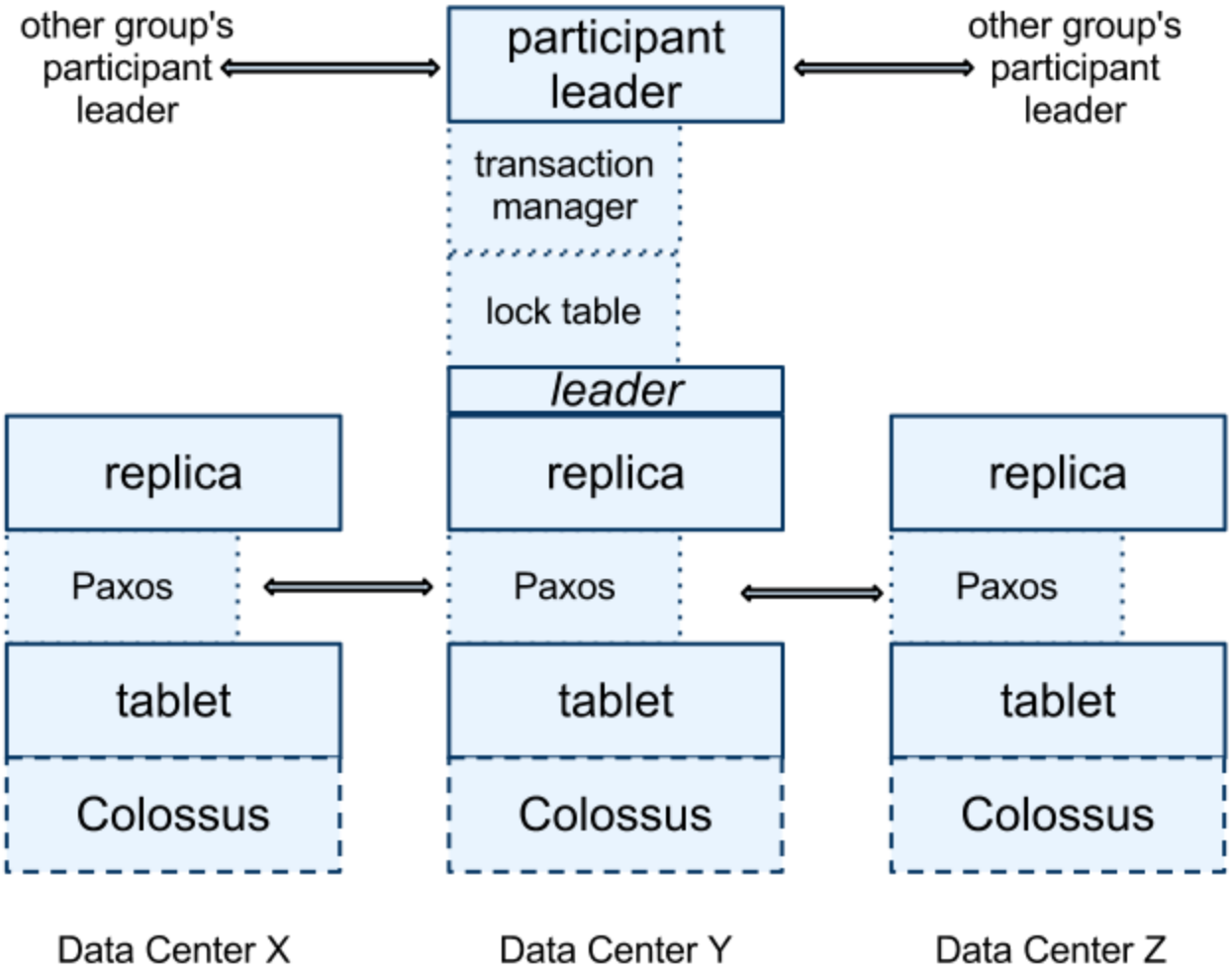
System Architecture



- Spanner deployment is called a universe
- Spanner organized as set of zones
  - zone is analog of Bigtable deployment
  - Unit of administrative deployment
  - Set of zones is set of locations across which data can be replicated
  - Unit of physical isolation, ie., more than one zone in DC

- can be added to/removed from a running system as new datacenters are brought into service and old ones are turned off
- One zonemaster with 100/1000s spanservers
- per-zone location proxies used by clients to locate spanservers assigned
- placement driver handles automated movement of data across zones on the timescale of minutes

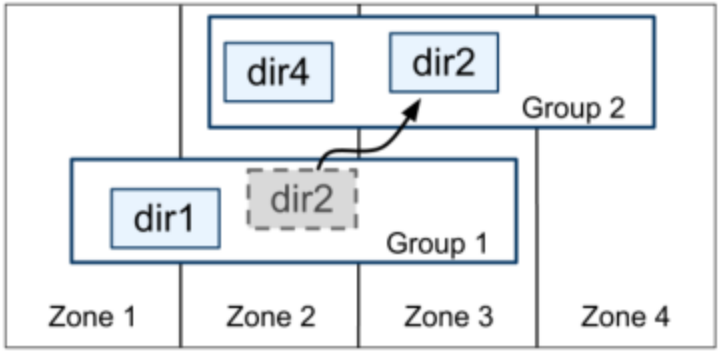
Spanserver Software Stack



- each spanserver is responsible for between 100 and 1000 instances of tablets
- Tablet similar to BigTable tablet except it assigns a timestamp to data
  (key:string, timestamp:int64) → string

Spanserver Software Stack

- Back End: Colossus (successor to GFS)
- Paxos State Machine on top of each tablet stores meta data and logs of the tablet.
- Leader among replicas in a Paxos group is chosen and all write requests for replicas in that group initiate at leader.
  - For replica (leader), spanserver implements a lock table to implement concurrency control
  - Lock table contains the state for two-phase locking: it maps ranges of keys to lock states
- Directory – analogous to bucket in BigTable
  - Smallest unit of data placement; Smallest unit to define replication properties
  - Move directory to shed load from a Paxos group
  - Put directories that are frequently accessed together into the same group
  - move a directory into a group that is closer to its accessors.

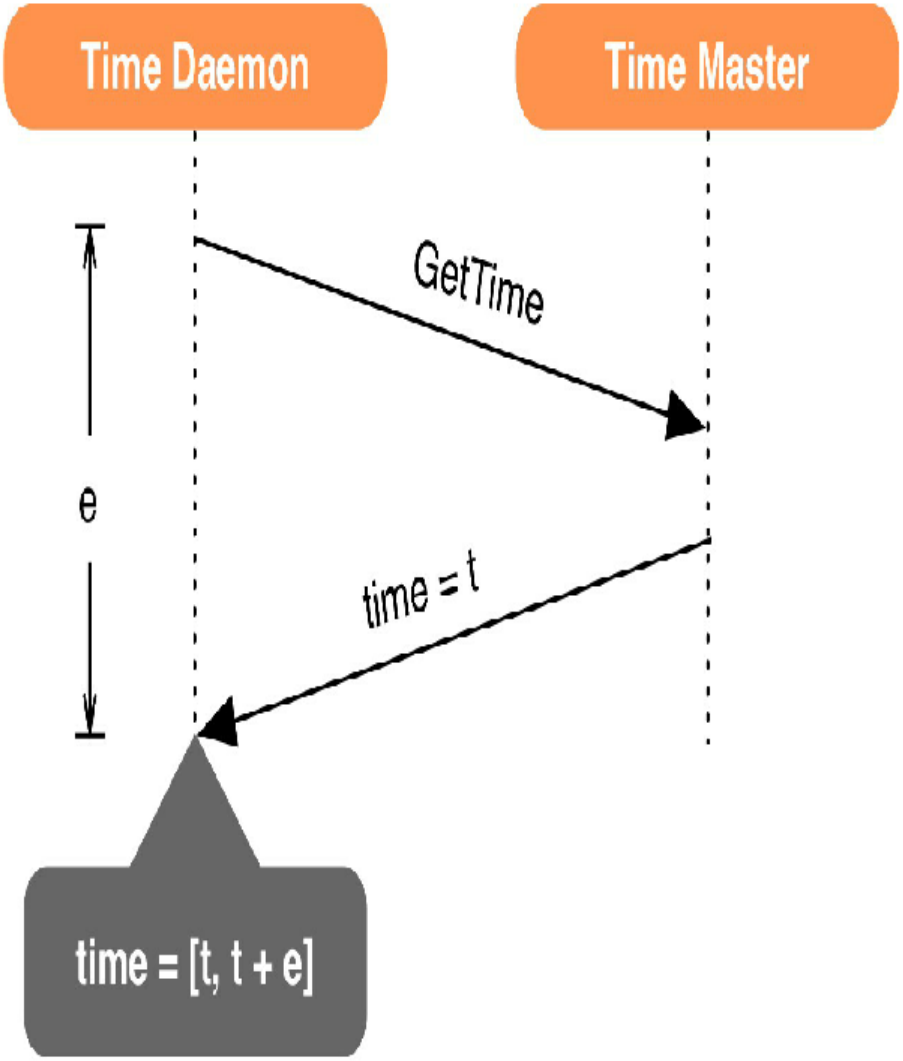


- Directory might in turn be sharded into Fragments if it grows too large.

TrueTime

- Novel API behind Spanner's core innovation
- Leverages hardware features like GPS and Atomic Clocks
- Implemented via TrueTime API.
  - Key method being now() which not only returns current system time but also another value (ε) which tells the maximum uncertainty in the time returned
- Set of *time master* server per datacenters and *time slave daemon* per machine.
- Majority of time masters are GPS fitted and few others are atomic clock fitted (Armageddon masters).
- Daemon polls variety of masters and reaches a consensus about correct timestamp.

TrueTime



- TrueTime uses both GPS and Atomic clocks since they are different failure rates and scenarios.
- Methods in API are
  - TT.now() – TTinterval:[earliest, latest]
  - TT.After(t) – returns TRUE if t is definitely passed
  - TT.Before(t) – returns TRUE if t is definitely not arrived
- TrueTime uses these methods in concurrency control and t serializes transactions.

TrueTime

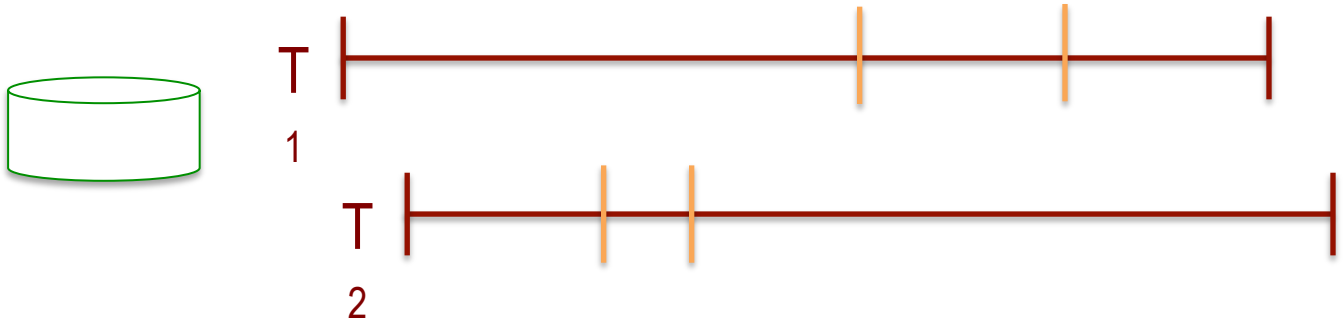
- After() is used for Paxos Leader Lease
  - Uses after(S_{max}) to check if S_{max} is passed so that Paxos Leader can abdicate its slaves.
- Paxos Leaders cannot assign timestamps(S_i) greater than S_{max} for transactions(T_i) and clients cannot see the data committed by transaction T_i till after(S_i) is true.
  - After(t) – returns TRUE if t is definitely passed
  - Before(t) – returns TRUE if t is definitely not arrived
- Replicas maintain a timestamp t_{safe} which is the maximum timestamp at which that replica is up to date.
- Read-Write – requires lock.
- Read-Only – lock free.
  - Requires declaration before start of transaction.
  - Reads information that is up to date
- Snapshot Read – Read information from past by specifying a timestamp or bound
  - User specifies specific timestamp from past or timestamp bound so data till that point will be read.

Timestamps, Global Clock

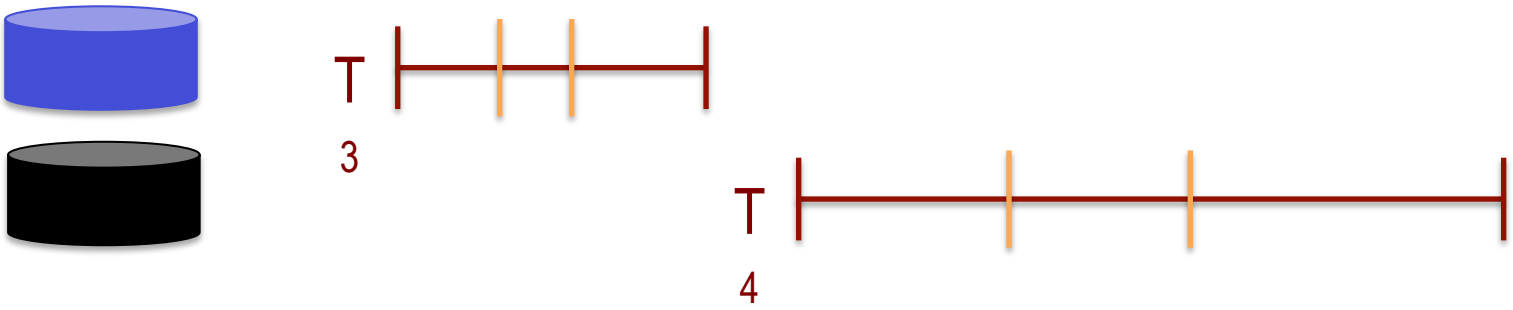
- Strict two-phase locking for write transactions
- Assign timestamp while locks are held

Timestamp Invariants

- Timestamp order == commit order

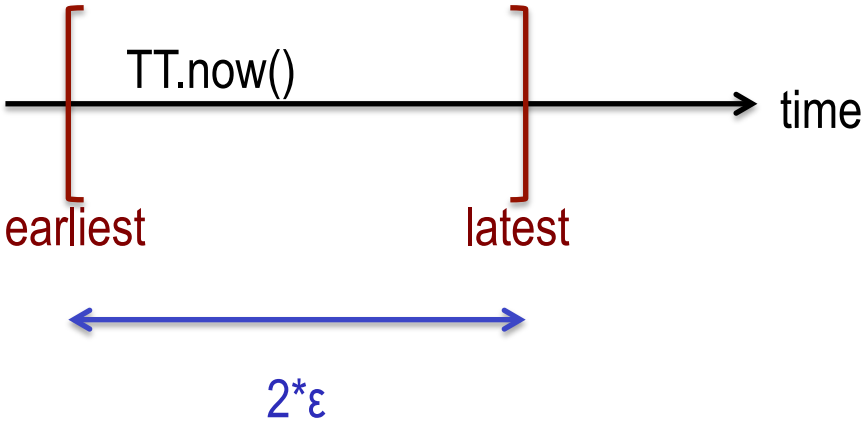


- Timestamp order respects global wall-time order

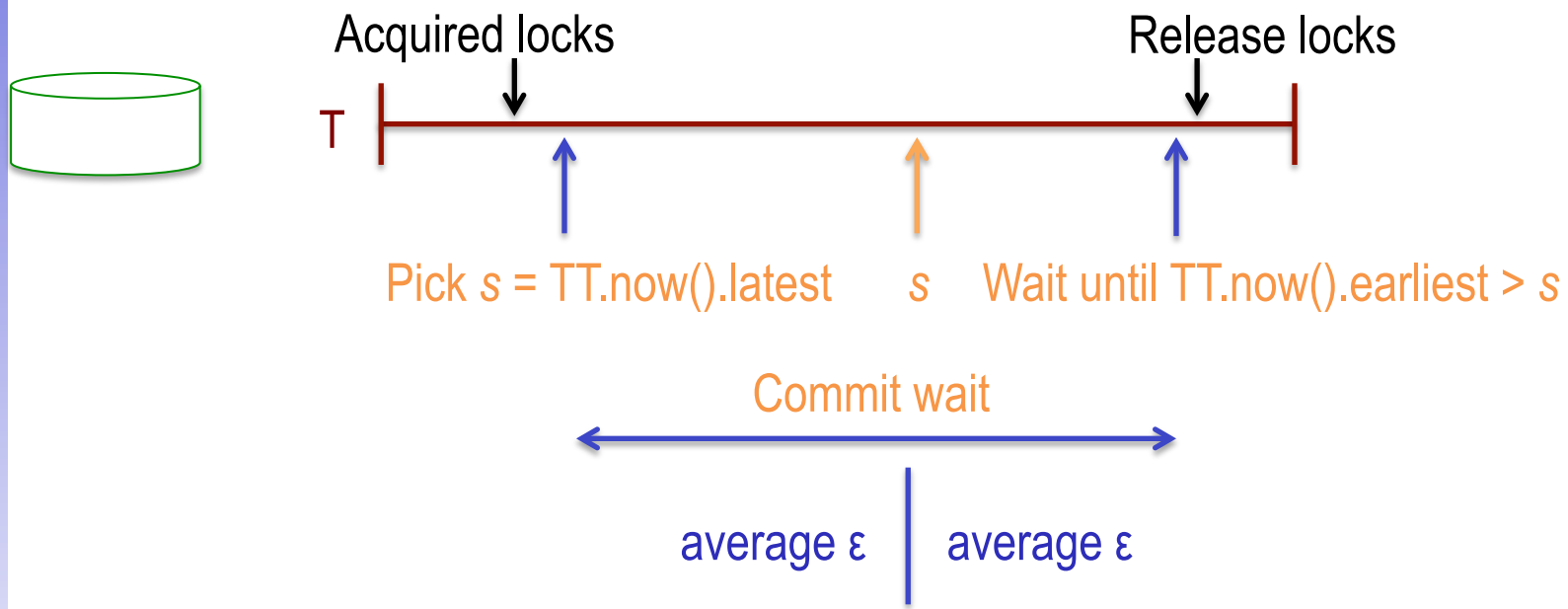


TrueTime

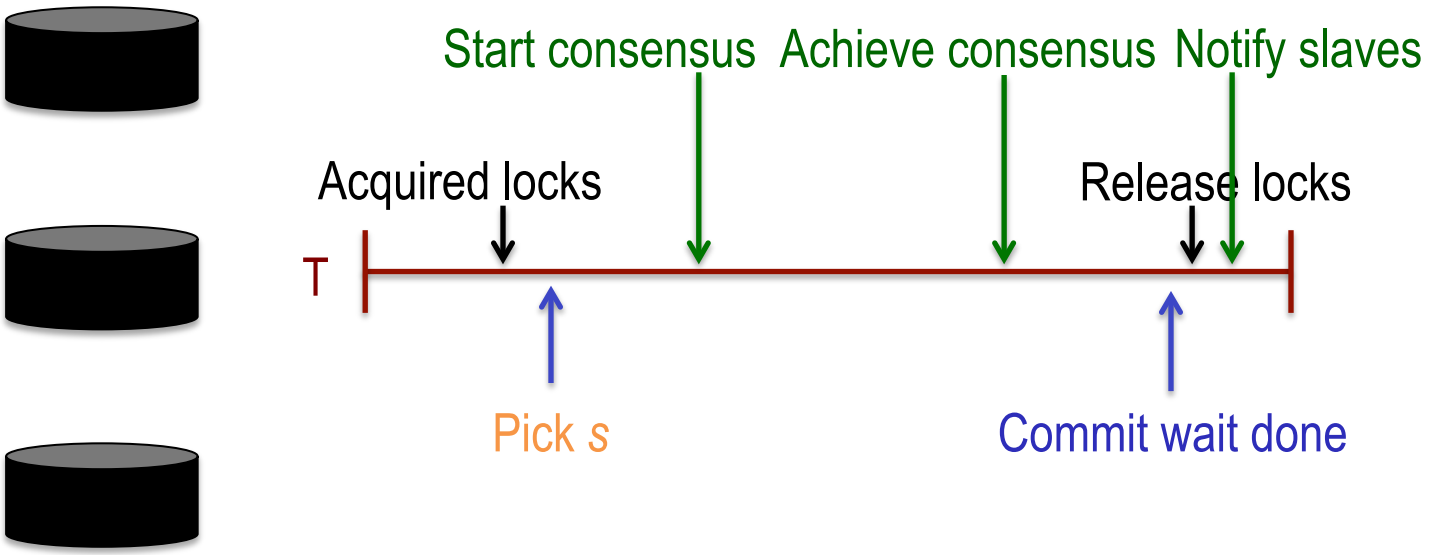
- "Global wall-clock time" with bounded uncertainty



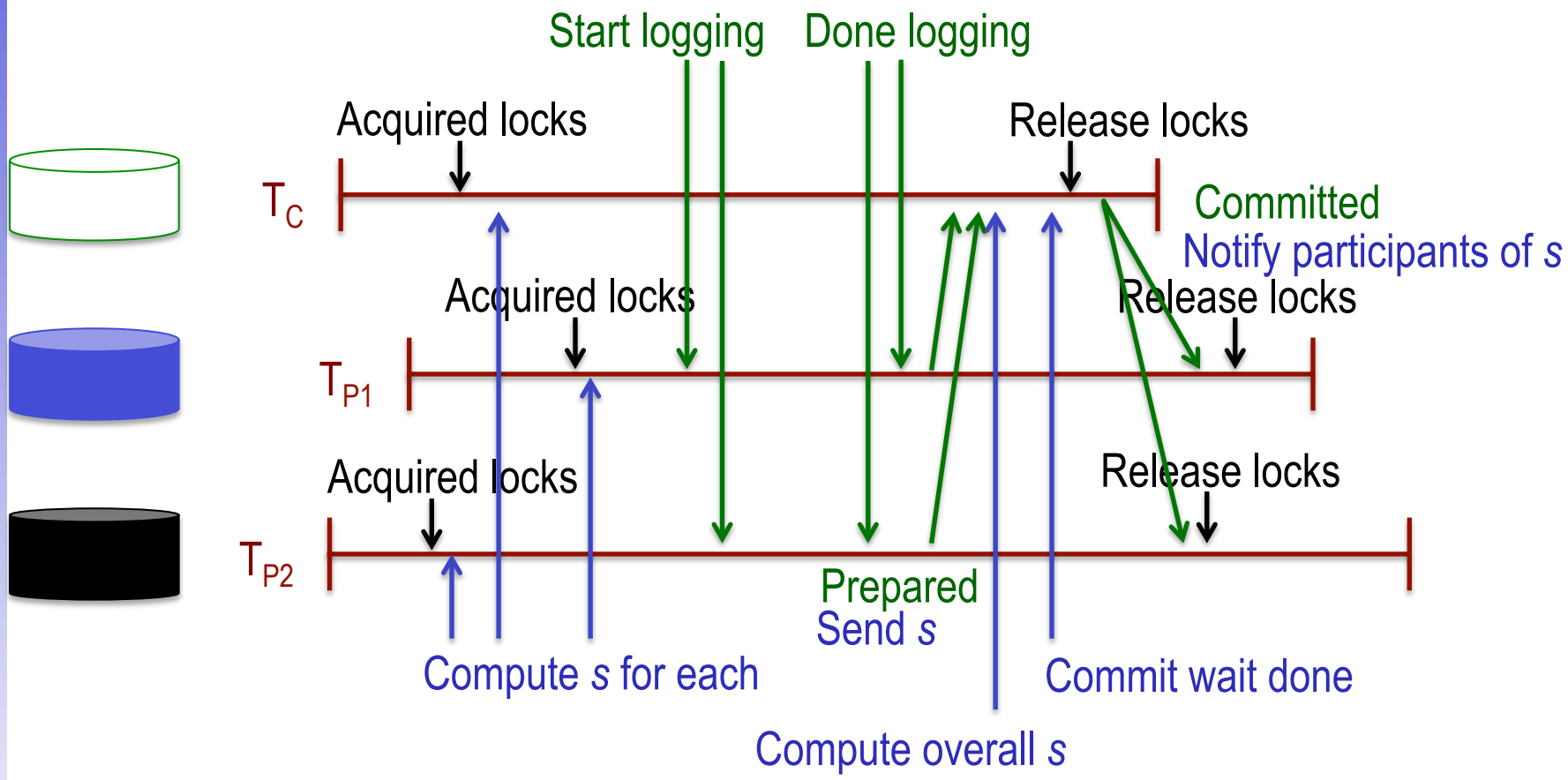
Timestamps and TrueTime



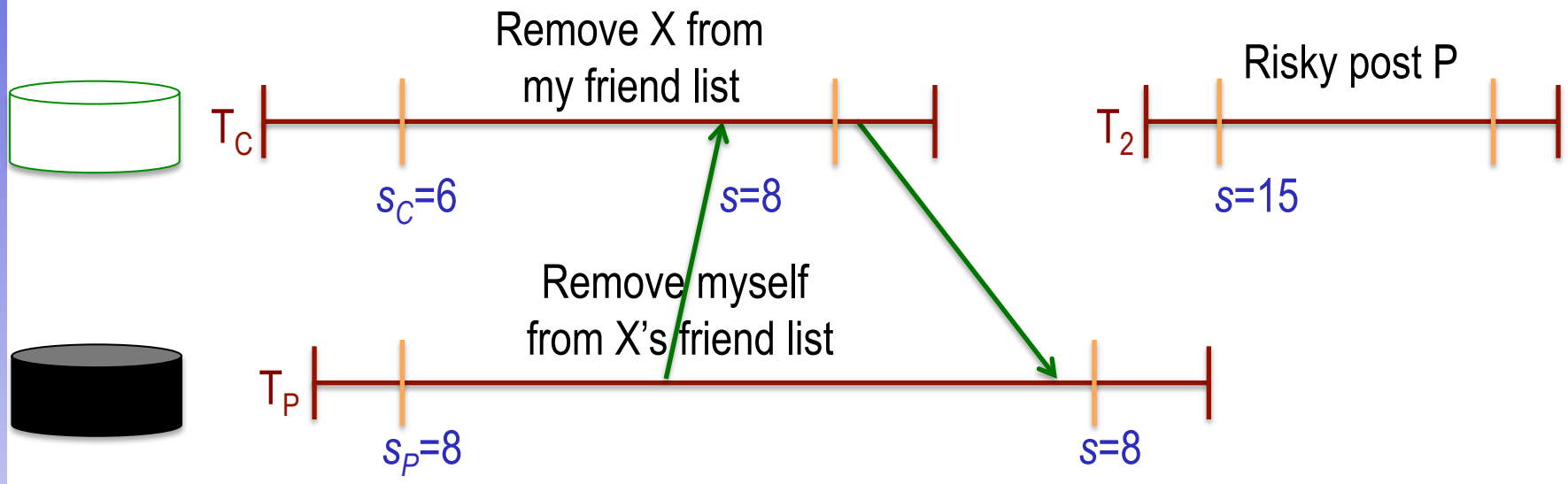
Commit Wait and Replication



Commit Wait and 2-Phase Commit



Example

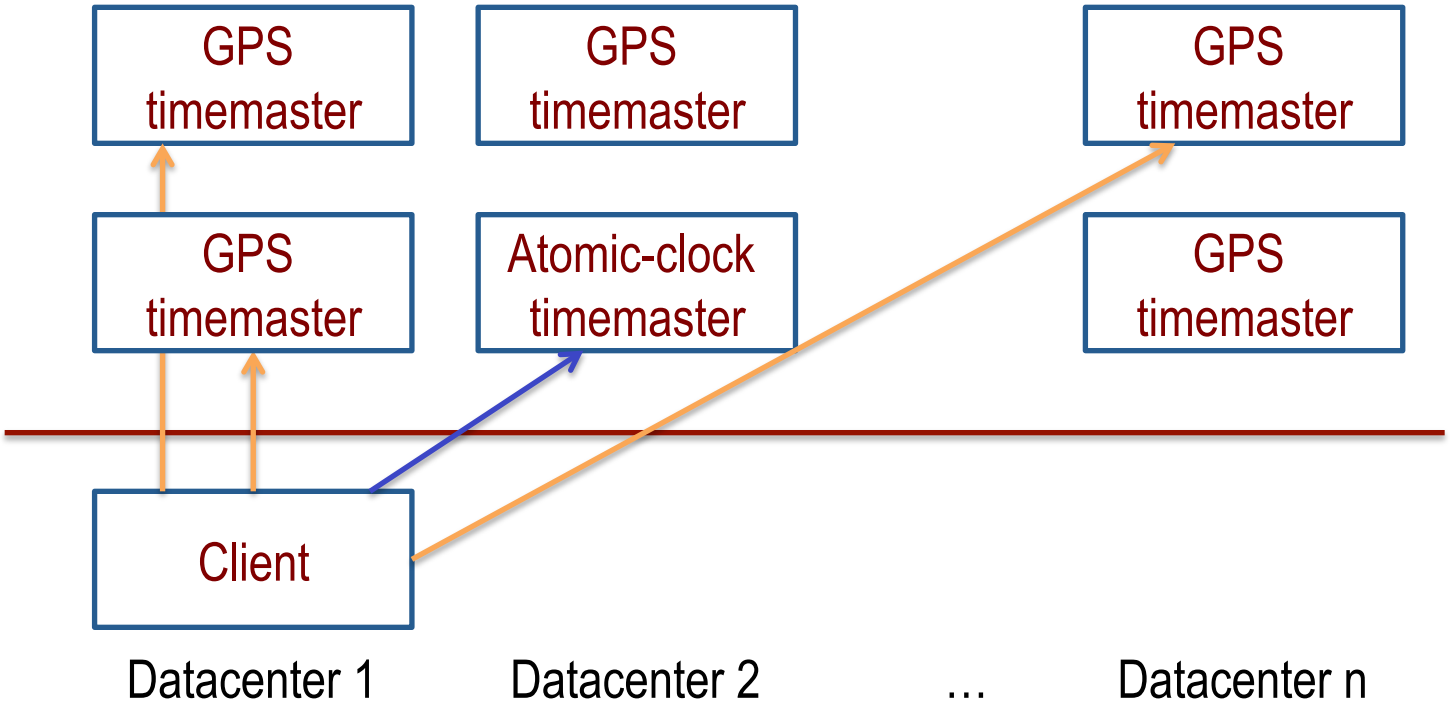


Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

We are not covering entire paper!

- Lock-free read transactions across datacenters
- External consistency
- Timestamp assignment
- TrueTime
  - Uncertainty in time can be waited out

TrueTime Architecture

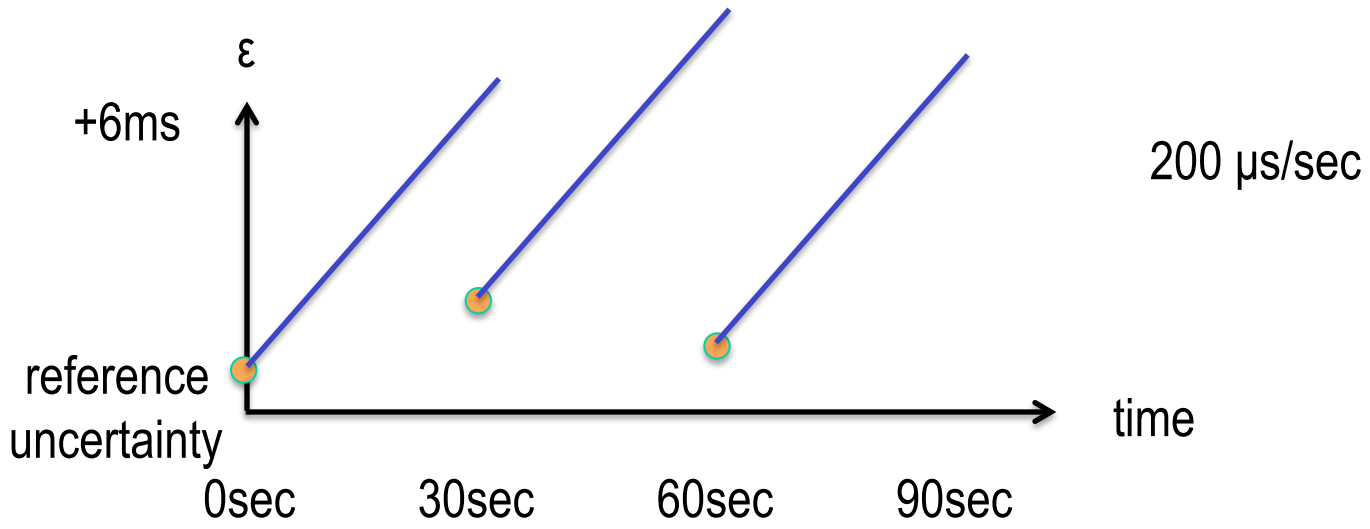


Compute reference [earliest, latest] = now ± ε

TrueTime implementation

now = reference now + local-clock offset

ε = reference ε + worst-case local-clock drift

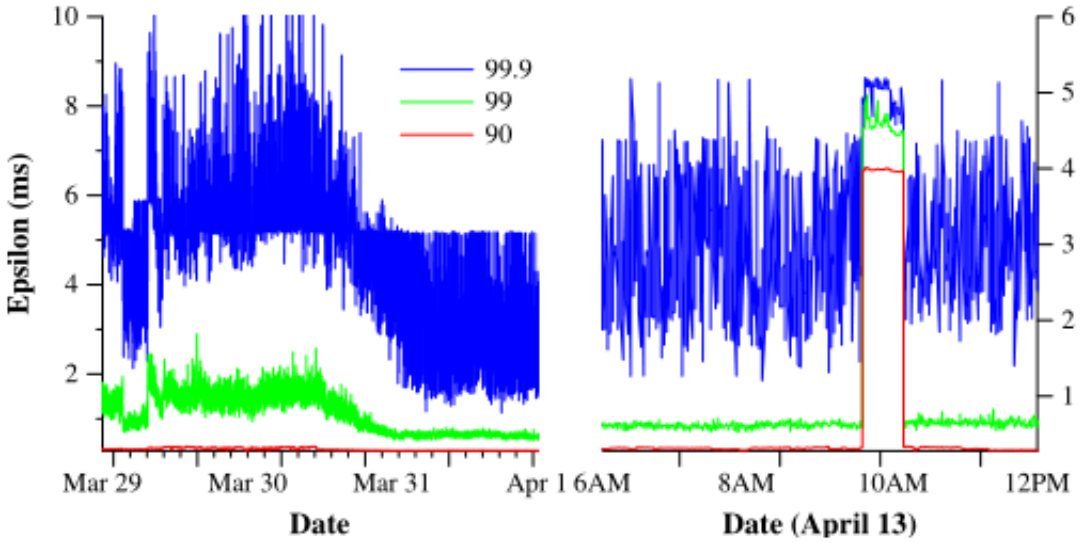


What If a Clock Goes Rogue?

- Timestamp assignment would violate external consistency
- Empirically unlikely based on 1 year of data
  - Bad CPUs 6 times more likely than bad clocks

Evaluation

- Evaluated for replication, transactions and availability.
- Results on epsilon of TrueTime
- Network induced Uncertainty across data centers 2200 kms apart



- Reduction in tail latencies beginning on March 30 were due to networking improvements that reduced transient network-link congestion
- Increase in epsilon on April 13, approximately one hour in duration, resulted from the shutdown of 2 time masters at a datacenter for routine maintenance.

F1 Case Study

- Spanner in production used by Google's advertising backend F1.
- F1 previously used MySQL for strong transactional semantics
- Spanner provides synchronous replication and automatic failover for F1.
- Enabled F1 to specify data placement via directories of spanner based on their needs.
- F1 operation latencies measured over 24 hours

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

Future Work

- Improving TrueTime
  - Lower ε < 1 ms
- Building out database features
  - Finish implementing basic features
  - Efficiently support rich query patterns

Conclusions

- Realize clock uncertainty in time APIs
- Stronger semantics are achievable
  - Greater scale != weaker semantics

Dynamo: Amazon's Highly Available Key-value Store

- Build a distributed storage system:
  - Scale
  - Simple: key-value
  - Highly available (sacrifice consistency)
  - Guarantee Service Level Agreements

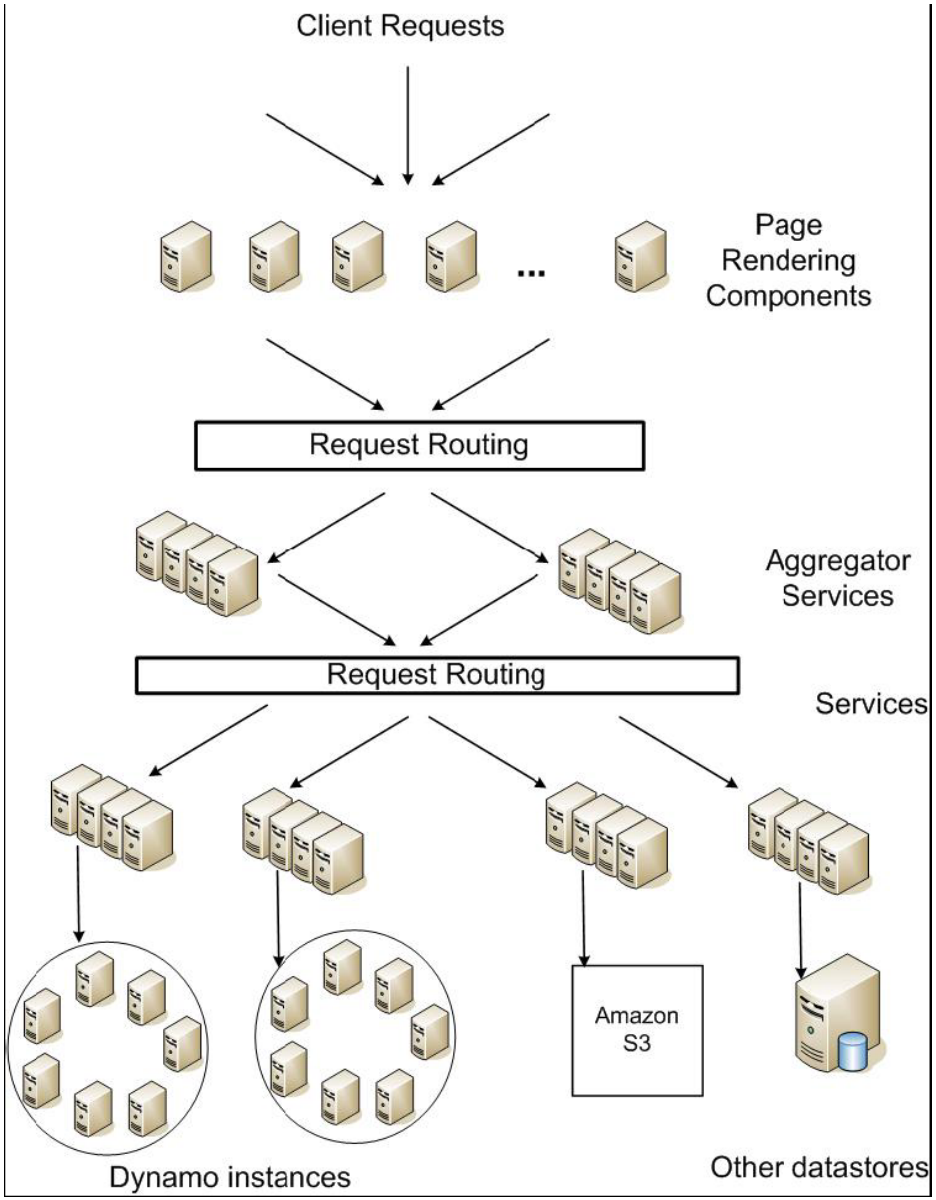
System Assumptions and Requirements

- Simple read and write operations to a data item that is uniquely identified by a key.
  - Most of Amazon's services can work with this simple query model and do not need any relational schema.
  - Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability.
- targeted applications - store objects that are relatively small (usually less than 1 MB)
- Operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization.
- Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability.

Service Level Agreements (SLA)

- Application can deliver its functionality in bounded time
  - Every dependency in the platform needs to deliver its functionality with even tighter bounds.
- Example
  - service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

Service-oriented architecture



Design Consideration

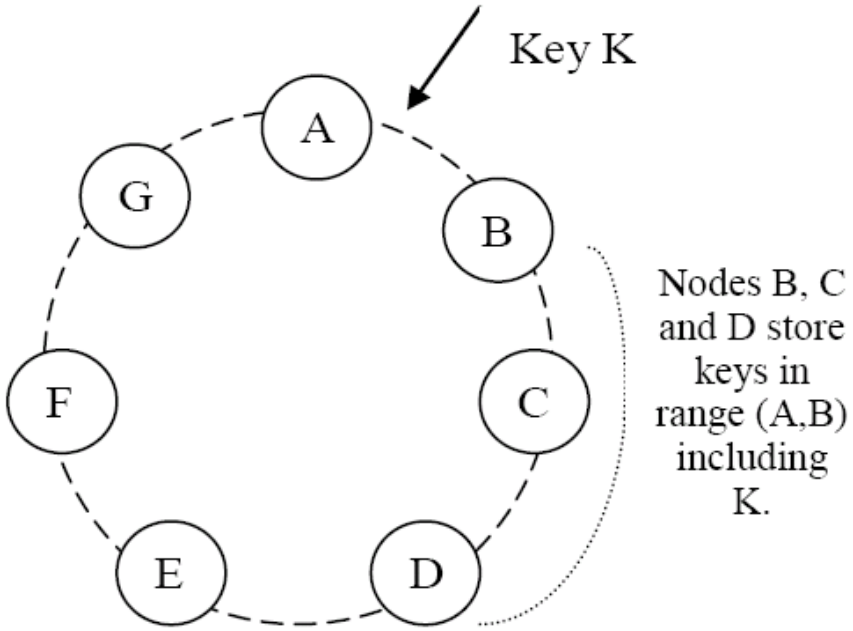
- Incremental scalability.
- Symmetry
  - Every node in Dynamo should have the same set of responsibilities as its peers.
- Decentralization.
  - In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible.
- Heterogeneity.
  - This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.
- "always writeable" data store where no updates are rejected due to failures or concurrent writes.
  - No updates rejected due to failures or concurrent writes
  - Conflict resolution is executed during ***read*** instead of ***write***, i.e. "always writeable".

System architecture

- Partitioning data over set of nodes to scale incrementally
- High Availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection

Partition Algorithm

- Consistent hashing: the output range of a hash function is treated as a fixed circular space or "ring".
- "Virtual Nodes": Each node can be responsible for more than one virtual node.

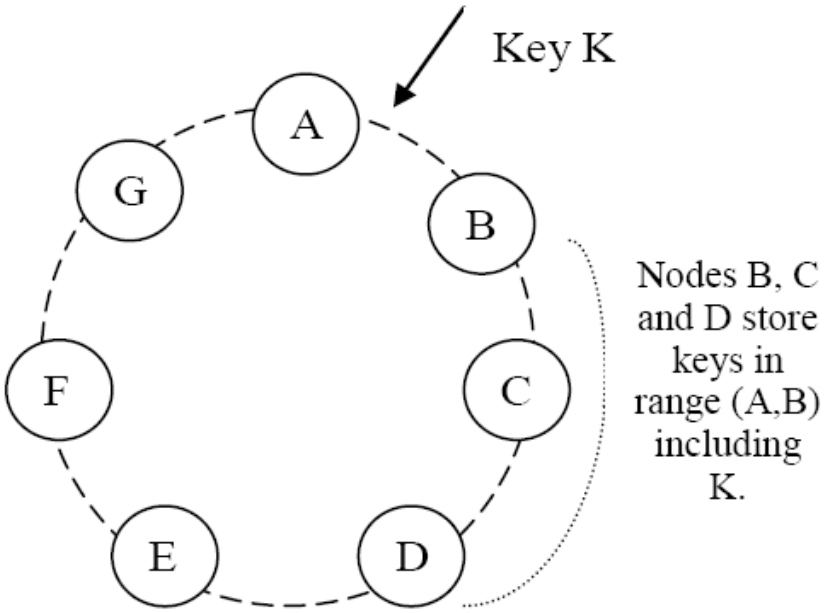


Partition (Cont'd)

- Advantages of using virtual nodes:
- If node becomes unavailable
  - load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

Replication

- Each data item is replicated at N hosts.
- "*preference list*": The list of nodes that is responsible for storing a particular key.



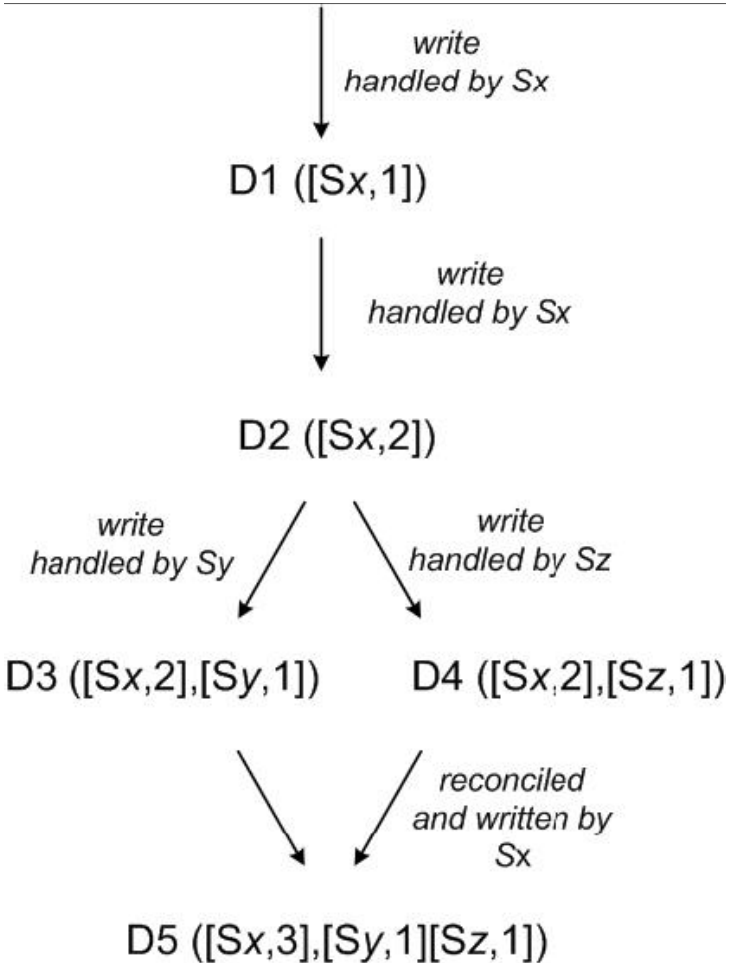
Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- Key Challenge:
  - distinct version sub-histories - need to be reconciled.
- Solution:
  - uses vector clocks in order to capture causality between different versions of the same object.

Vector Clock

- A vector clock is a list of (node, counter) pairs.
  - Every version of every object is associated with one vector clock.
- If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.
- In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow.
- Dynamo stores a *timestamp* that indicates the last time the node updated the data item.
- When the number of (node, counter) pairs in the vector clock reaches a *threshold* (say 10), the oldest pair is removed from the clock.

Vector clock example



get () and put () operations

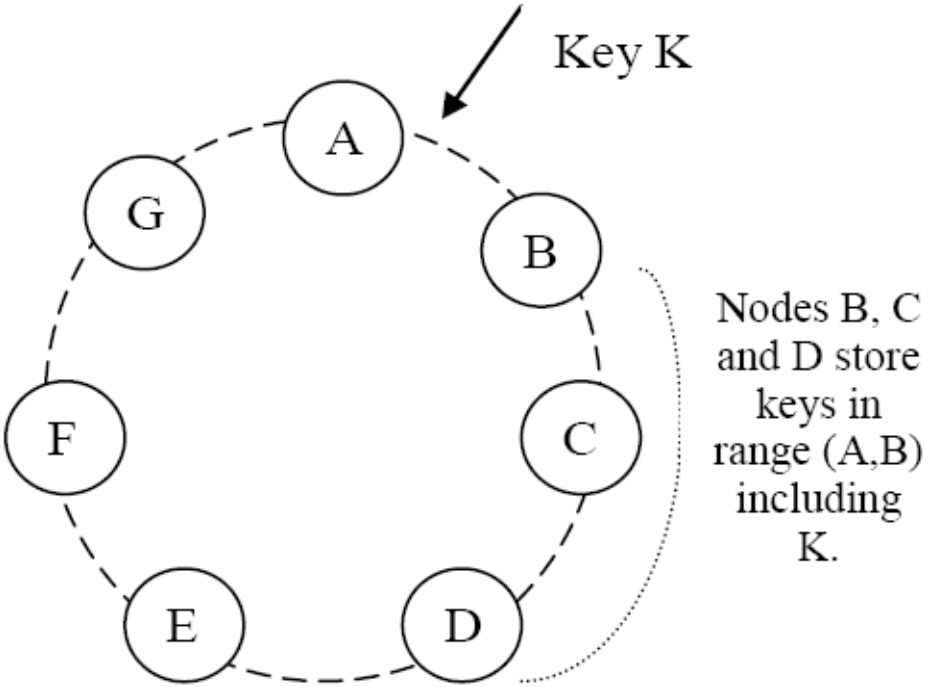
- Any storage node in Dynamo is eligible to receive client get and put operations for any key
- Two strategies to select a node:
  1. Route its request through a generic load balancer that will select a node based on load information.
     - The advantage of this approach is that the client does not have to link any code specific to Dynamo in its application
  2. Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.
     - This strategy can achieve lower latency because it skips a potential forwarding step.

Temporary Failures: Sloppy Quorum

- Coordinator node handles a read or write operation
  - Typically, this is the first among the top N nodes in the preference list.
  - If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring.
  - node that receives the request will not coordinate it if the node is not in the top N of the requested key's preference list.
  - Instead, that node will forward the request to the first among the top N nodes in the preference list.
- When all nodes are healthy, the top N nodes in a key's preference list are accessed
- When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed. So, consistency protocol is used.
- R/W is the minimum number of nodes that must participate in a successful read/write operation.
- Setting R + W > N yields a quorum-like system.
- In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas.
  - R and W are usually configured to be less than N, to provide better latency.

Hinted handoff

- Assume N = 3. When A is temporarily down or unreachable during a write, send replica to D.
- D is hinted that the replica belongs to A and it will deliver to A when A is recovered.
- Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically.
- Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, D may delete the object from its local store without decreasing the total number of replicas in the system.



Again: "always writeable"

Replica synchronization

- There are scenarios under which hinted replicas become unavailable before they can be returned to the original replica node. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized
- Merkle tree:
  - a hash tree where leaves are hashes of the values of individual keys.
  - Parent nodes higher in the tree are hashes of their respective children.
- Advantage of Merkle tree:
  - Each branch of the tree can be checked independently without requiring nodes to download the entire tree.
  - Help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas.

Membership Detection

- Explicit mechanism to initiate addition/removal of nodes from dynamo ring
  - The node that serves the request writes the membership change and its time of issue to persistent store.
  - The membership changes form a history because nodes can be removed and added back multiple times.
  - A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership.
  - Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

Summary of techniques used in *Dynamo* and their advantages

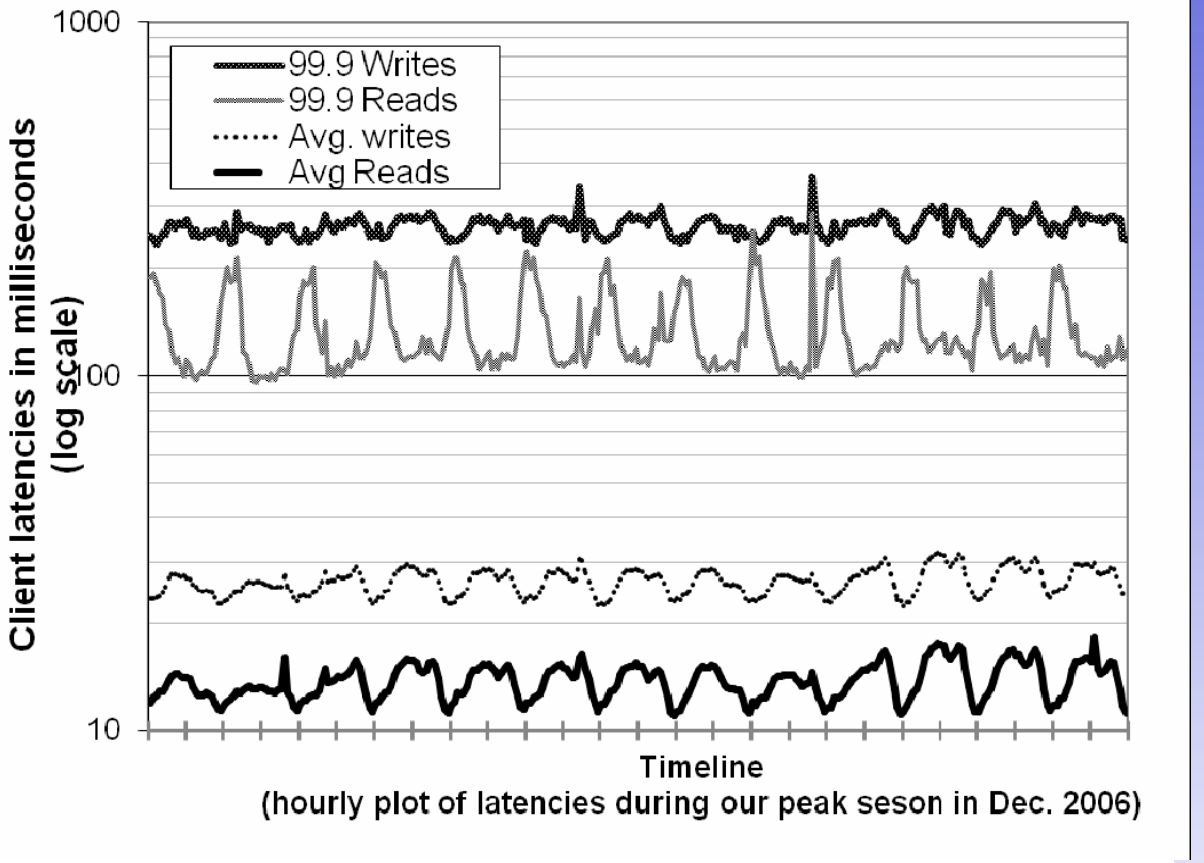
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Implementation

- Java
  - three main software components: request coordination, membership and failure detection, and a local persistence engine
- Local persistence component allows for different storage engines to be plugged in:
  - Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
  - MySQL: object of > tens of kilobytes
  - BDB Java Edition, etc.

Performance

- **Guarantee Service Level Agreements (SLA)**
- the latencies exhibit a clear diurnal pattern (incoming request rate of day/night)
- write operations always results in disk access.
- affected by several factors such as variability in request load, object sizes, and locality patterns

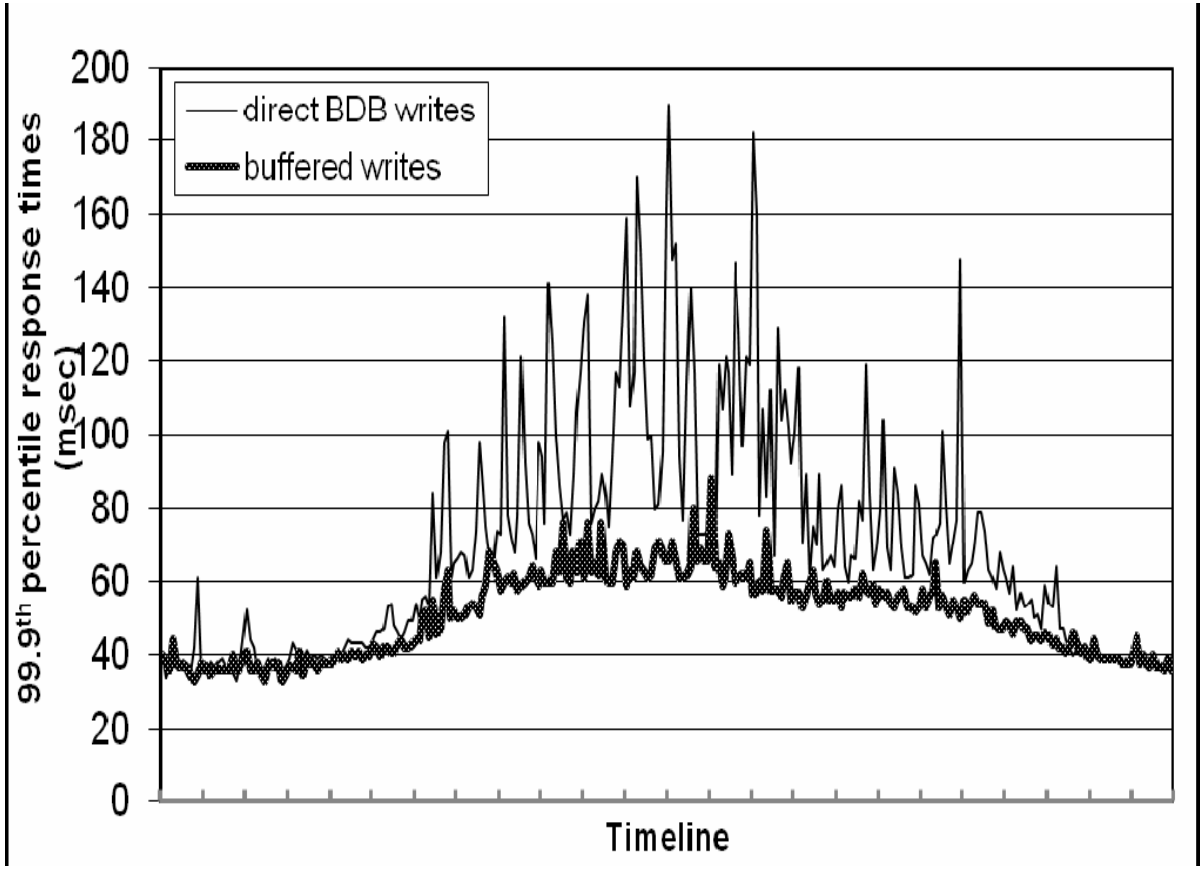


Performance Improvements

- A few customer-facing services required higher levels of performance.
- Each storage node maintains an object buffer in its main memory.
- Each write operation is stored in the buffer and gets periodically written to storage by a *writer thread*.
- Read operations first check if the requested key is present in the buffer

Object Buffer Improvements

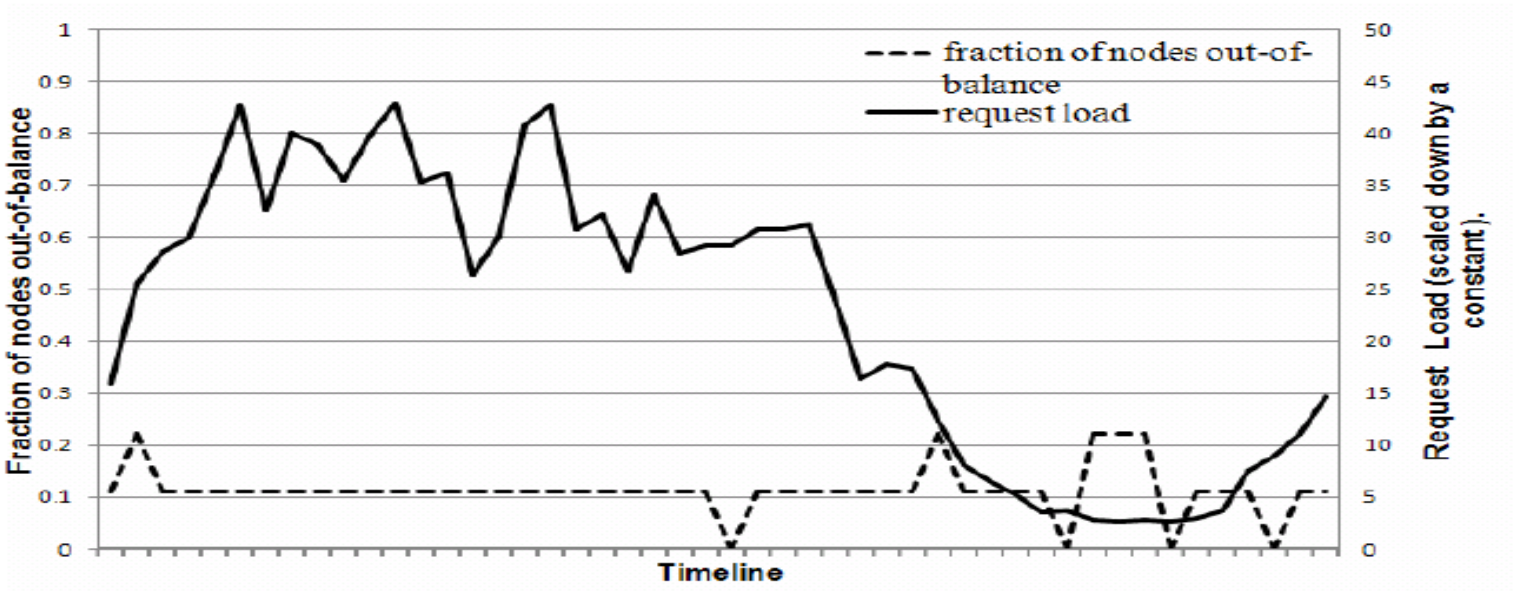
- lowering the 99.9th percentile latency by a factor of 5 during peak traffic
- write buffering smoothens out higher percentile latencies



Improvements

- Server crash can result in missing writes that were queued up in the buffer.
- To reduce the durability risk, the write operation is refined to have the coordinator choose one out of the N replicas to perform a “durable write”
- Since the coordinator waits only for W responses, the performance of the write operation is not affected by the performance of the durable write operation

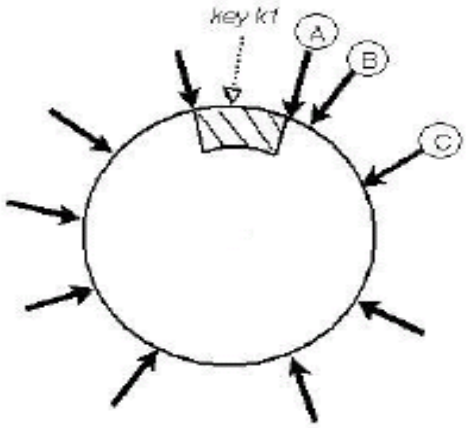
Load Balance



- Out-of-balance
  - If the node's request load deviates from the average load by a value more than a certain threshold (here 15%)
- Imbalance ratio decreases with increasing load
  - under high loads, a large number of popular keys are accessed and the load is evenly distributed

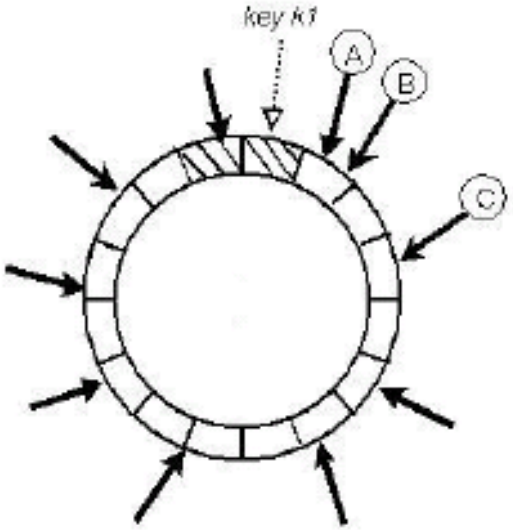
Partitioning and placement of key

- Space needed to maintain membership at each node increases linearly with number of nodes in system



Strategy 1

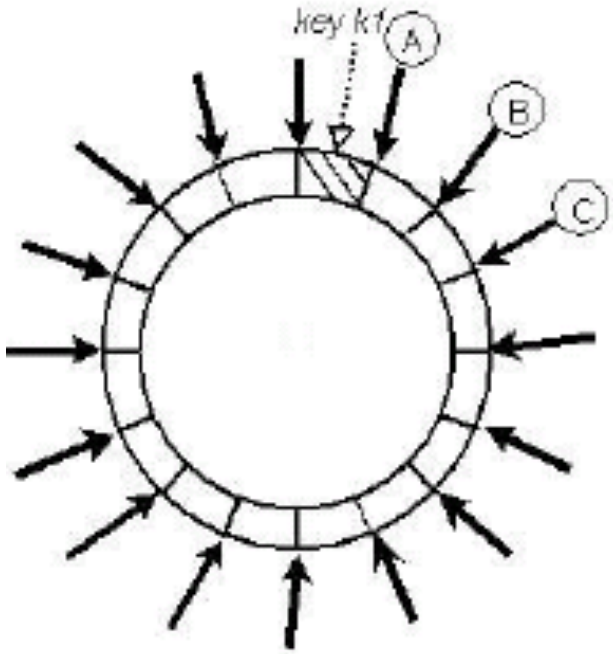
- divides the hash space into Q equally sized partitions
- The primary advantages of this strategy are:
  1. decoupling of partitioning and partition placement,
  2. enabling the possibility of changing the placement scheme at runtime.



Strategy 2

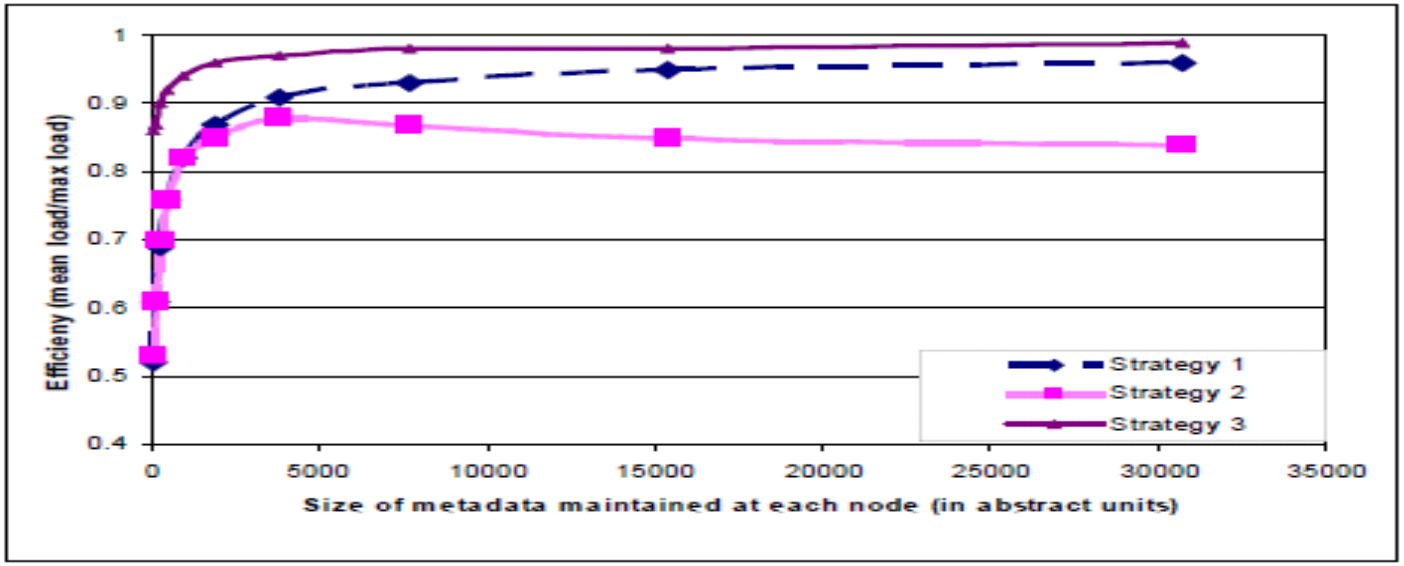
Partitioning and placement of key

- divides the hash space into Q equally sized partitions
- each node is assigned Q/S tokens where S is the number of nodes in the system.
- When a node leaves the system, its tokens are randomly distributed to the remaining nodes
- when a node joins the system it "steals" tokens from nodes in the system



Strategy 3

Load Balancing Efficiency



- Strategy 3 achieves better efficiency
- *Faster bootstrapping/recovery:*
  - Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items).
- *Ease of archival*
  - Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services.
  - Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately.

Coordination

- Dynamo has a request coordination component that uses a state machine to handle incoming requests. Client requests are uniformly assigned to nodes in the ring by a load balancer.
- An alternative approach to request coordination is to move the state machine to the client nodes. In this scheme client applications use a library to perform request coordination locally.
- The latency improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node.

Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

Conclusion

- Dynamo is a highly available and scalable data store for Amazon.com's e-commerce platform.
- Dynamo has been successful in handling server failures, data center failures and network partitions.
- Dynamo is incrementally scalable and allows service owners to scale up and down based on their current request load.
- Dynamo allows service owners to customize their storage system by allowing them to tune the parameters N, R,and W.