Cristina Nita-Rotaru


# 7610: Distributed Systems

## BigTable. Hbase.Spanner. Dynamo. Casandra

# 1: BigTable

# Acknowledgement


- Slides based on material from course at UMichigan, U Washington, and the authors of BigTable and Spanner.

# REQUIRED READING

- Bigtable: A Distributed Storage System for Structured Data. 2008. ACM Trans. Comput. Syst. 26, 2 (Jun. 2008), 1-26
- Spanner, Google's globally distributed database. OSDI 2012.

# BigTable

- Distributed storage system for managing structured data such as:
  - 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, …
- Designed to scale to a very large size: petabytes of data distributed across thousands of servers
- Used for many Google applications
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, … and more

# Why BigTable?

- Scalability requirements not met by existent commercial systems:
  - Millions of machines
  - Hundreds of millions of users
  - Billions of URLs, many versions/page
  - Thousands or queries/sec
  - 100TB+ of satellite image data
- Low-level storage optimization helps performance significantly

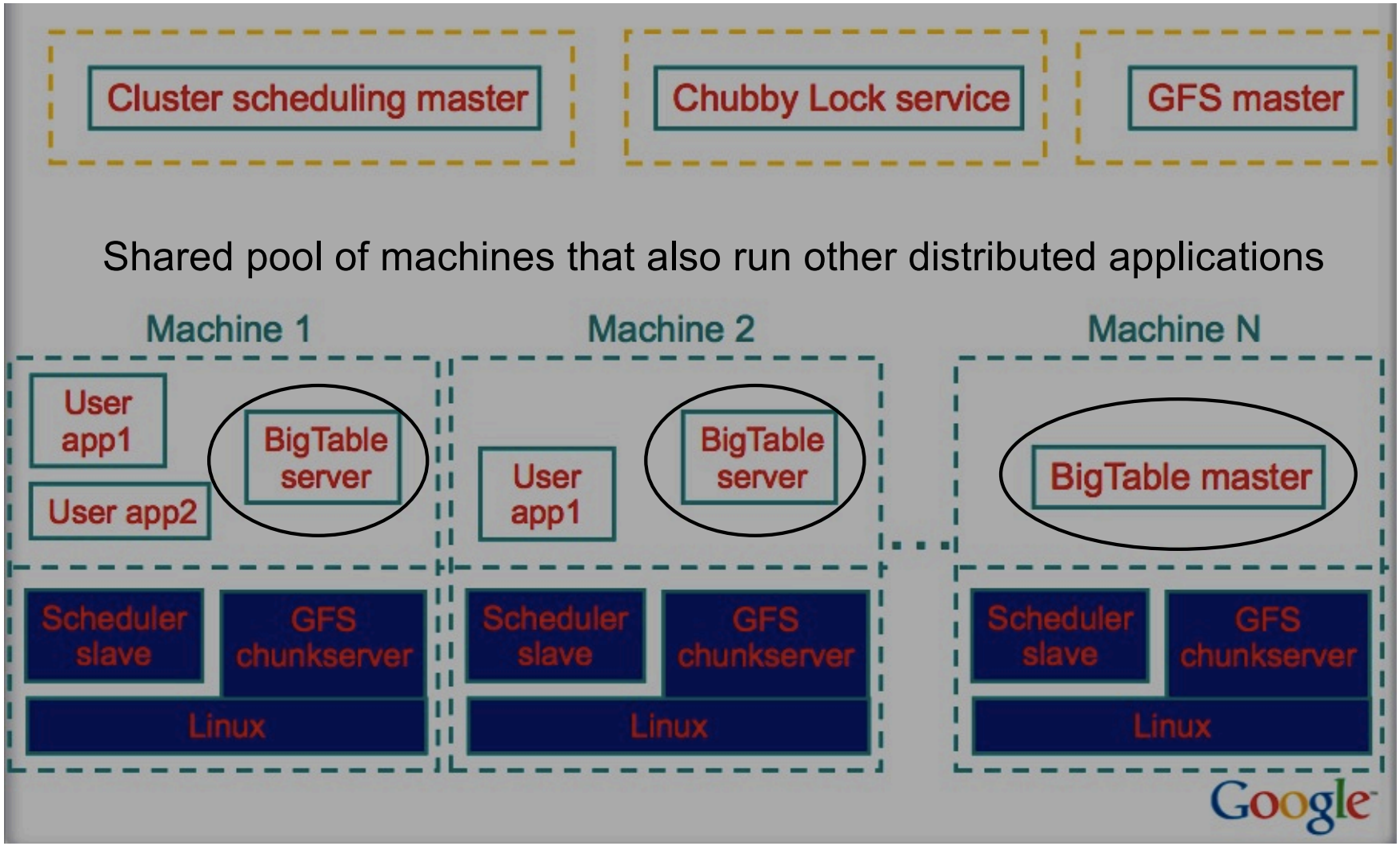
# Goals

- Simpler model that supports dynamic control over data and layout format
- Want asynchronous processes to be continuously updating different pieces of data: access to most current data at any time
- Examine data changes over time: e.g. contents of a web page over multiple crawls
- Support for:
  - Very high read/write rates (millions ops per second)
  - Efficient scans over all or subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets

# Design Overview

- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

# Typical Google Cluster



# Building Blocks

- Google File System (GFS)
  - Stores persistent data (SSTable file format)
- Scheduler
  - Schedules jobs onto machines
- Chubby
  - Lock service: distributed lock manager, master election, location bootstrapping
- MapReduce (optional)
  - Data processing
  - Read/write BigTable data

# Chubby

- {lock/file/name} service
- Coarse-grained locks
  - Provides a namespace that consists of directories and small files.
  - Each of the directories or files can be used as a lock.
- Each client has a session with Chubby
  - The session expires if it is unable to renew its session lease within the lease expiration time.
- 5 replicas Paxos, need a majority vote to be active

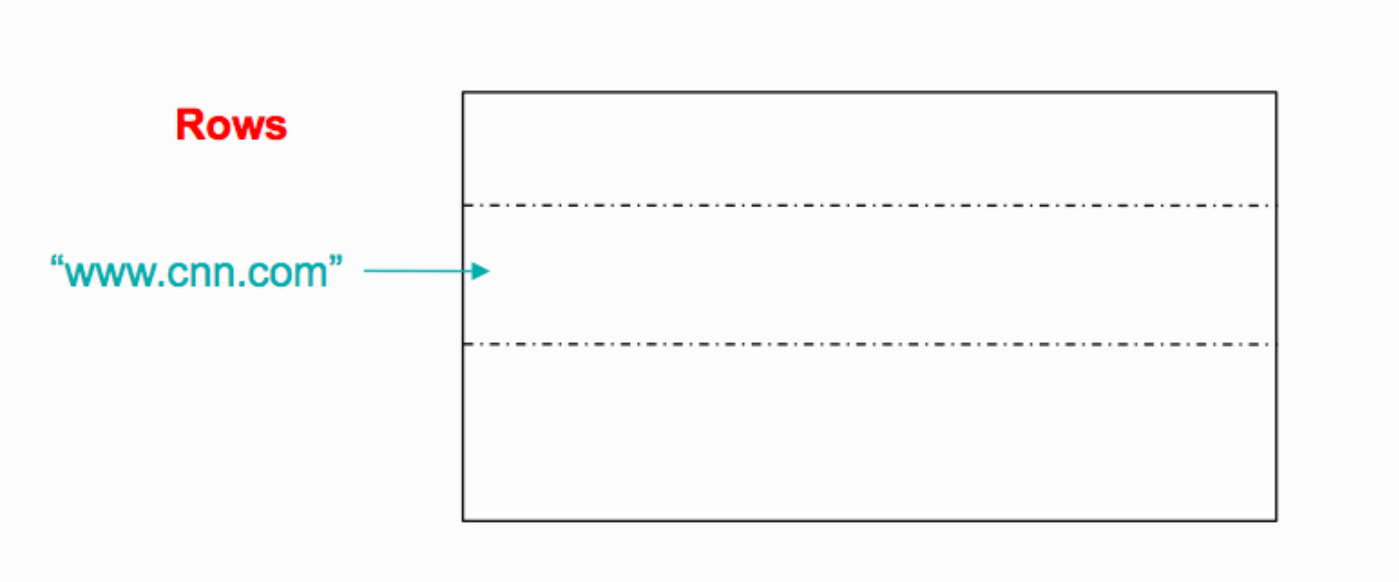
# Data Model

- A sparse, distributed persistent multi-dimensional sorted map
- Rows, column are arbitrary strings

- (row, column, timestamp) -> cell contents

# Data Model: Rows

- Arbitrary string
- Access to data in a row is atomic
  - Row creation is implicit upon storing data
  - Ordered lexicographically

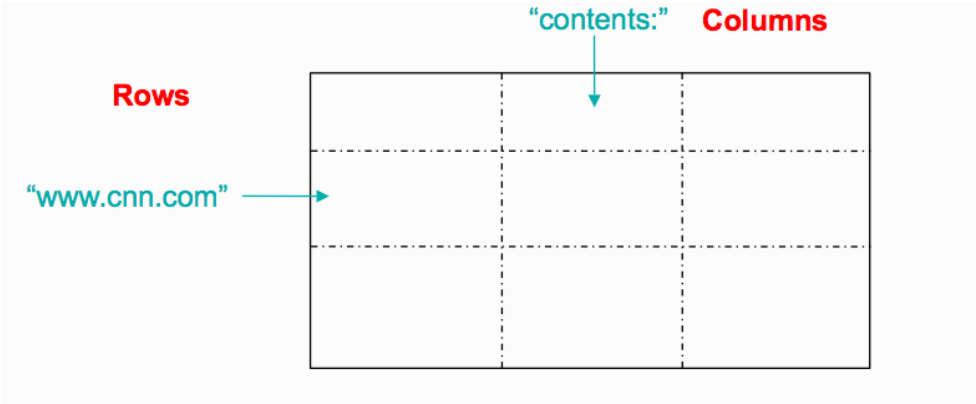


# Rows (cont.)

- Rows close together lexicographically usually on one or a small number of machines
- Reads of short row ranges are efficient and typically require communication with a small number of machines
- Can exploit lexicographic order by selecting row keys so they get good locality for data access
- Example:
  - math.gatech.edu, math.uga.edu, phys.gatech.edu, phys.uga.edu
  - VS
  - edu.gatech.math, edu.gatech.phys, edu.uga.math, edu.uga.phys

# Data Model: Columns

- Two-level name structure: family: qualifier
- Column family:
  - Is the unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional levels of indexing, if desired

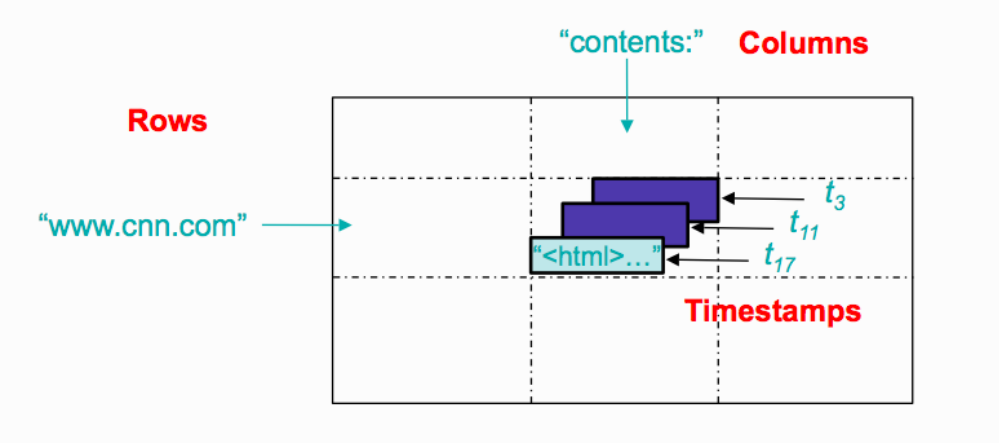


# Data Model: Timestamps (64bit integers)

Store different versions of data in a cell:

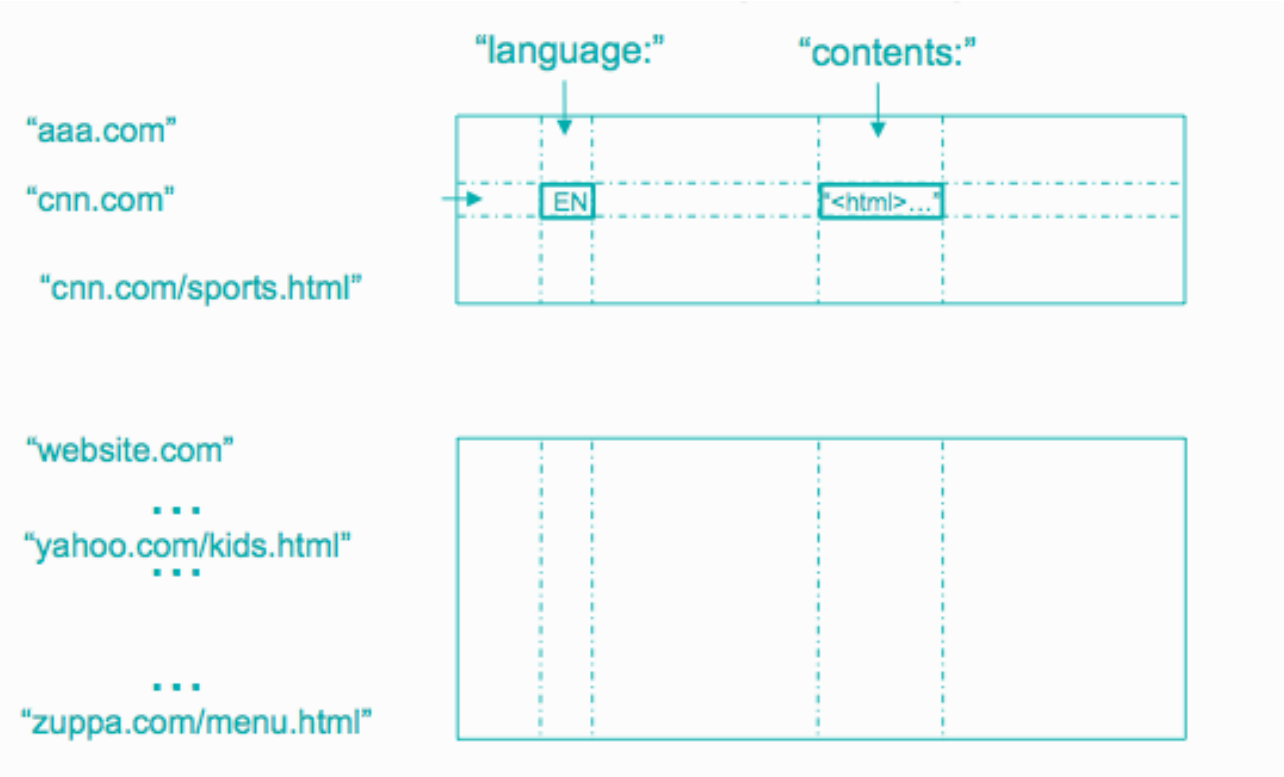
- New writes default to current time, but timestamps for writes can also be set explicitly by clients

- Lookup options
  - Return most recent K values
  - Return all values
- Column families can be marked w/ attributes:
  - Retain most recent K values in a cell
  - Keep values until they are older than K seconds



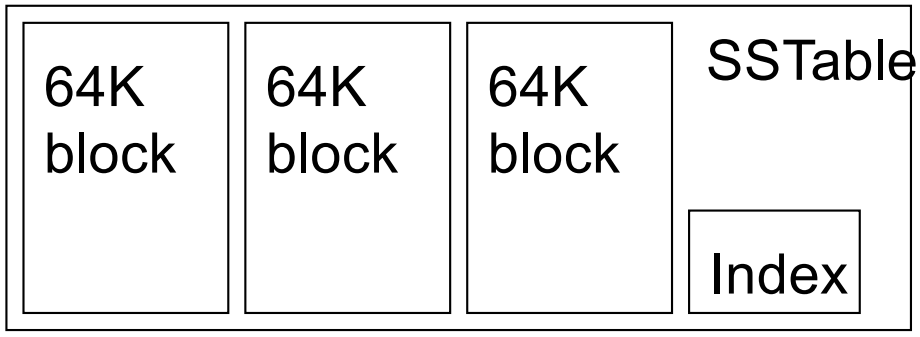
# Data Model: Tablet

- The row range for a table is dynamically partitioned
- Each row range is called a tablet (typically 10-100 bytes)
- Tablet is the unit for distribution and load balancing



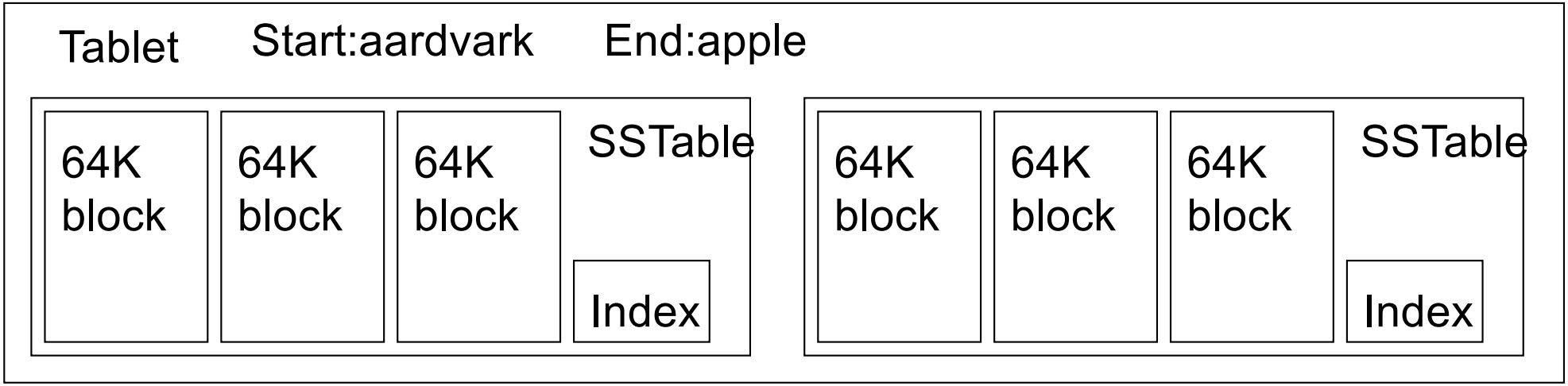
# Storage: SSTable

- Immutable, sorted file of key-value pairs
- Optionally, SSTable can be completely mapped into memory
- Chunks of data plus an index
  - Index is of block ranges, not values
  - Index is loaded into memory  when SSTable is open



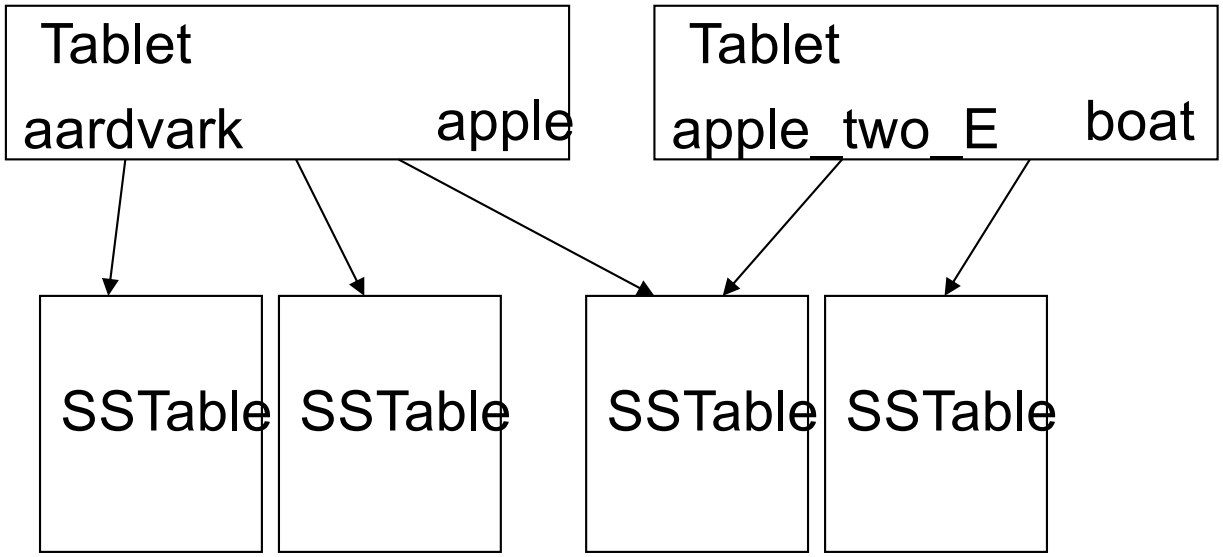
# Tablet vs. SSTable

- Tablet is built out of multiple SSTables

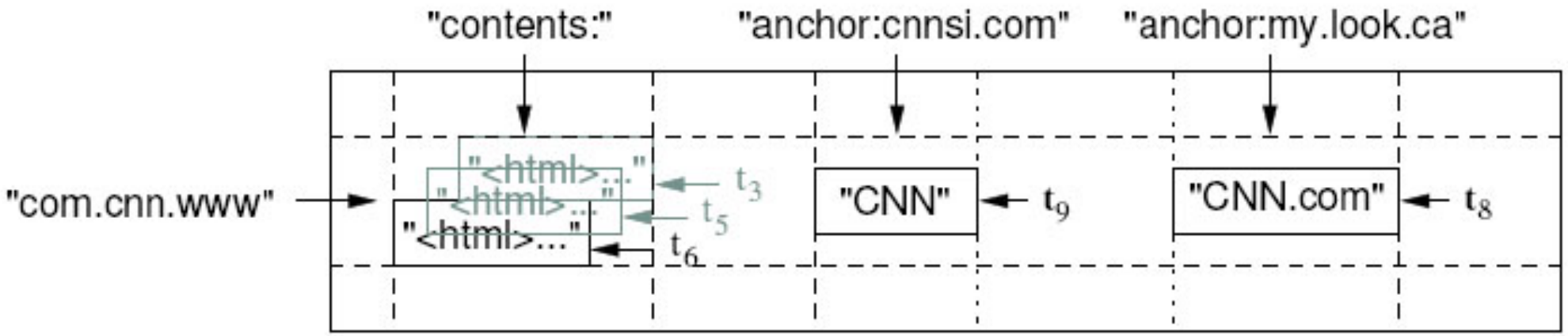


# Table vs. Tablet vs. SSTable

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



# Example: WebTable

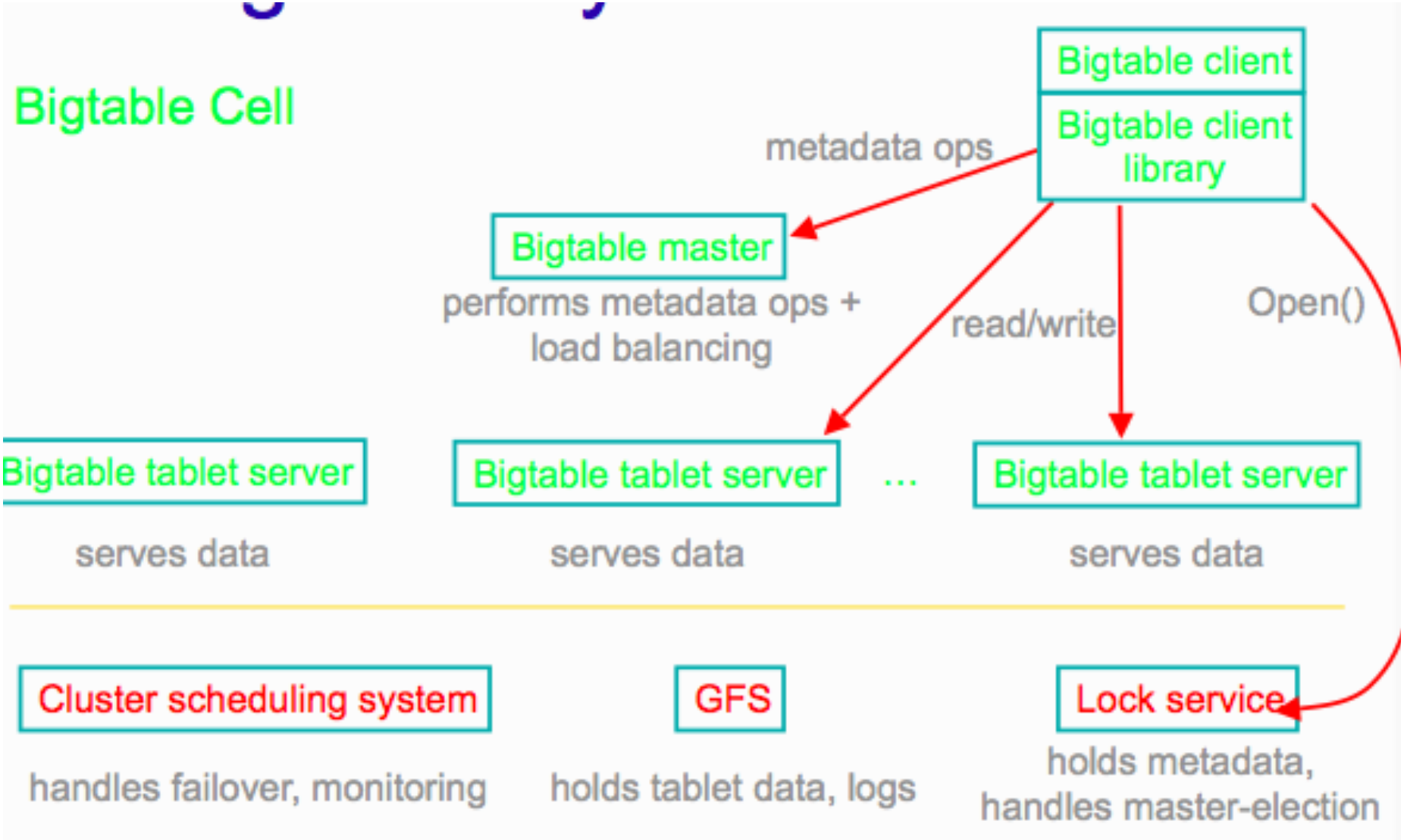



- Want to keep copy of a large collection of web pages and related information
- Use URLs as row keys
- Various aspects of web page as column names
- Store contents of web pages in the contents: column under the timestamps when they were fetched.

# Implementation

- Library linked into every client
- One master server responsible for:
  - Assigning tablets to tablet servers
  - Detecting addition and expiration of tablet servers
  - Balancing tablet-server load
  - Garbage collection
  - Handling schema changes such as table and column family creation
- Many tablet servers, each of them:
  - Handles read and write requests to its table
  - Splits tablets that have grown too large
- Clients communicate directly with tablet servers for reads and writes.

# Deployment

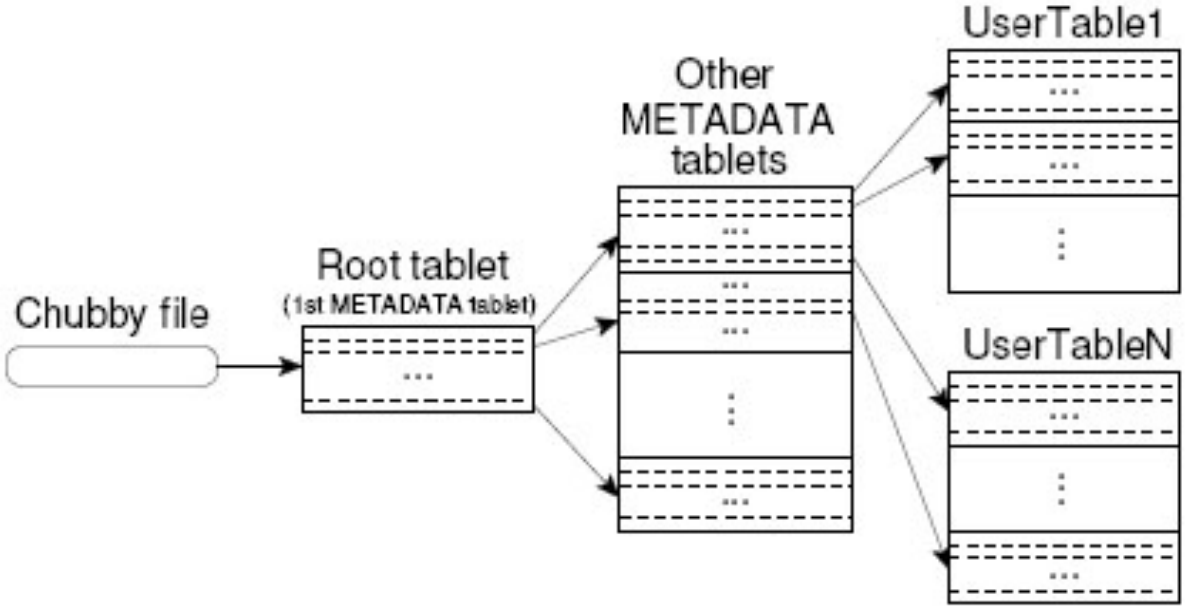

Bigtable Cell
Bigtable client
Bigtable client library
metadata ops
Bigtable master
performs metadata ops + load balancing
read/write
Open()
Bigtable tablet server
Bigtable tablet server
...
Bigtable tablet server
serves data
serves data
serves data
Cluster scheduling system
GFS
Lock service
handles failover, monitoring
holds tablet data, logs
holds metadata, handles master-election

# More about Tablets

- Serving machine responsible for 10 - 1000
  - Usually about 100 tablets
- Fast recovery:
  - 100 machines each pick up 1 tablet for failed machine
- Fine-grained load balancing:
  - Migrate tablets away from overloaded machine
  - Master makes load-balancing decisions

# Tablet Location

- Since tablets move around from server to server, given a row, how do clients find the right machine
  - Find tablet whose row range covers the target row
- METADATA: Key: table id + end row,  Data: location
- Aggressive caching and prefetching at client side



# Tablet Assignment

- Each tablet is assigned to one tablet server at a time.
- Master server
  - Keeps track of the set of live tablet servers and current assignments of tablets to servers.
  - Keeps track of unassigned tablets.
- When a tablet is unassigned, master assigns the tablet to a tablet server with sufficient room.
- It uses Chubby to monitor health of tablet servers, and restart/replace failed servers.

# Tablet Assignment: Chubby

- Tablet server registers itself with Chubby by getting a lock in a specific directory of Chubby
- Chubby gives "lease" on lock, must be renewed periodically
- Server loses lock if it gets disconnected
- Master monitors this directory to find which servers exist/are alive
  - If server not contactable/has lost lock, master grabs lock and reassigns tablets
  - GFS replicates data. Prefer to start tablet server on same machine that the data is already at

# API

- Metadata operations
  - Create/delete tables, column families, change metadata
- Writes (atomic)
  - Set(): write cells in a row
  - DeleteCells(): delete cells in a row
  - DeleteRow(): delete all cells in a row
- Reads
  - Scanner: read arbitrary cells in a bigtable
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row, all rows, etc.
    - Can ask for all columns, just certain column families, or specific columns

# Refinements: Locality Groups

- Can group multiple column families into a locality group
  - Separate SSTable is created for each locality group in each tablet.
- Segregating columns families that are not typically accessed together enables more efficient reads.
  - In WebTable, page metadata can be in one group and contents of the page in another group.

# 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
- Two-pass custom compressions scheme
  - First pass: compress long common strings across a large window
  - Second pass: look for repetitions in small window
- Speed emphasized, but good space reduction (10-to-1)

# Refinements: Bloom Filters

- Read operation has to read from disk when desired SSTable is not in memory
- Reduce number of accesses by specifying a Bloom filter:
  - Allows to ask if a SSTable might contain data for a specified row/column pair.
  - Small amount of memory for Bloom filters drastically reduces the number of disk seeks for read operations
  - Results in most lookups for non-existent rows or columns not needing to touch disk

# Real Applications

| 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 |

# Limitations

- No transactions supported
- Does not support full relational data model
- Achieved throughput is limited by GFS

# Lessons Learnt

- Large distributed systems vulnerable to many type of failures
  - Memory and network corruption
  - Large clock skew
  - Hung machines
  - Extended and asymmetric network partitions
  - Bugs in other systems
- Proper system-level monitoring critical
- Simple design better
- Do not add new features before they are needed

# 2: HBase

# HBase

- Open-source, distributed, versioned, column-oriented data store, modeled after Google's Bigtable
- Random, real time read/write access to large data:
  - Billions of rows,  millions of columns
  - Distributed across clusters of commodity hardware

# History

- 2006.11
  - Google releases paper on BigTable
- 2007.2
  - Initial HBase prototype created as Hadoop contrib.
- 2007.10
  - First useable HBase
- 2008.1
  - Hadoop become Apache top-level project and HBase becomes subproject
- Current stable release **Sept** 2019

# HBase Is Not …

- Tables have one primary index, the row key.
- No join operators.
- Scans and queries can select a subset of available columns.
- There are three types of lookups:
  - Fast lookup using row key and optional timestamp.
  - Full table scan
  - Range scan from region start to end.

# HBase Is Not …(2)

- Limited atomicity and transaction support.
  - HBase supports multiple batched mutations of single rows only.
  - Data is unstructured and untyped.
- No accessed or manipulated via SQL.
  - Programmatic access via Java, REST, or Thrift APIs.
  - Scripting via JRuby.

# 3: Spanner

# Limitations of BigTable

- Difficult to use for applications that
  - have complex, evolving schemas
  - want strong consistency in the presence of wide-area replication

# What is Spanner

- Scalable, multi-version, globally- distributed, and synchronously-replicated database
- Distribute data at global scale and support externally-consistent distributed transactions
- Features:
  - non- blocking reads in the past
  - lock-free read-only transactions
  - atomic schema changes
- Scale up to
  - millions of machines
  - hundreds of datacenters
  - trillions of database rows

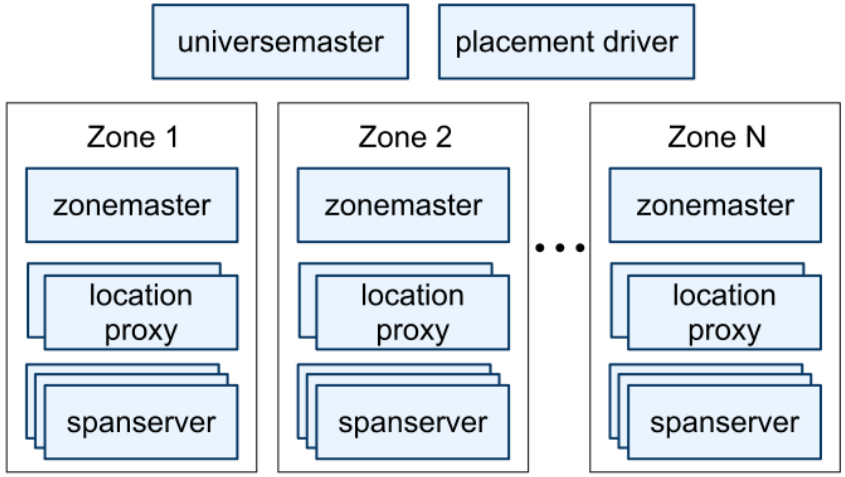
# What is Spanner

- Applications can control replication configurations for data
- Applications can specify constraints
  - to control which datacenters contain which data, how far data is from its users (to control read latency)
  - how far replicas are from each other (to control write latency)
  - how many replicas are maintained (to 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

# Spanner – key idea

- Consistent reads and writes
- How:
  - use global commit timestamps to transactions, even though transactions may be distributed.
  - timestamps represent serialization order
    - **if a transaction T1 commits before another transaction T2 starts, then T1's commit timestamp is smaller than T2's.**
  - provide such guarantees at global scale
- How to get the global timestamps: TrueTime
- Use existing algorithms such as Paxos and 2PC

# Architecture

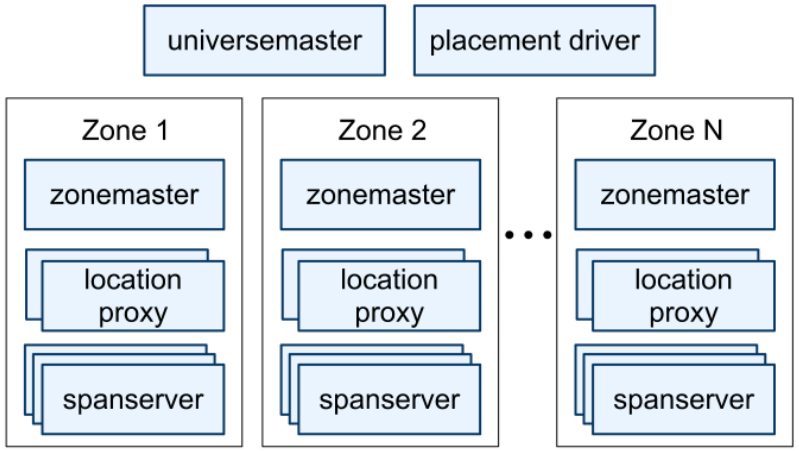



- Instance – it's called universe;

test, deployment, production

  - Universe master
  - Placement master
    - handles automated movement of data across zones on the timescale of minutes
    - periodically communicates with the spanservers to find data that needs to be moved, either to meet updated replication constraints or to balance load.
  - Universe consists of zones
    - Denotes physical isolation
    - Several zones can be in a datacenter

# Zones



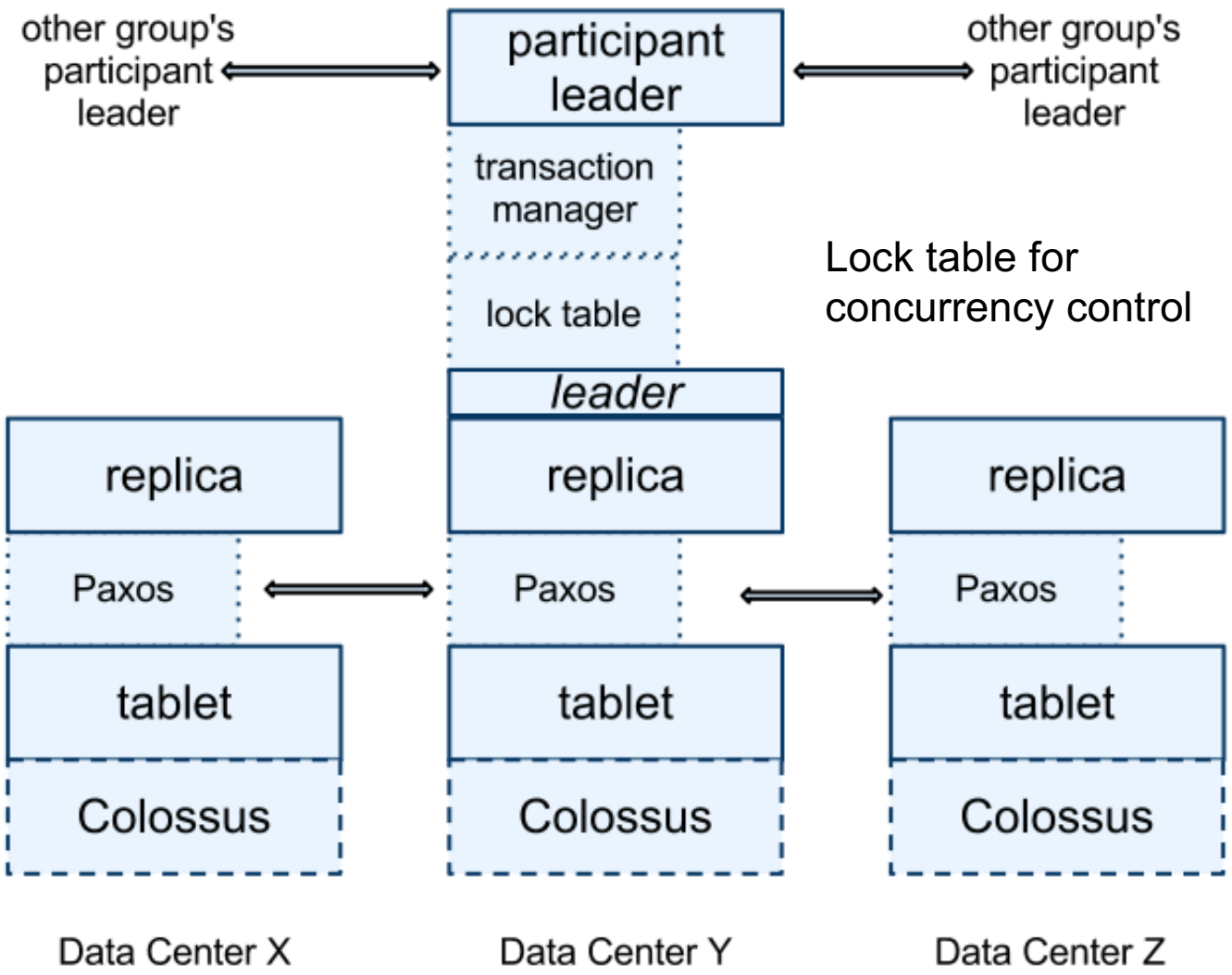

- Zonemaster
  - assigns the data to span servers
- Spanservers
  - hundreds to thousands
  - store data
  - responsible for between 100 and 1000 instances of a data structure called a *tablet* (different from the BigTable tablet)
- Location proxies
  - used by clients to locate the spanservers assigned to serve their data

# Tablet and Directory

- Tablet: Implements a bag of the following mappings:

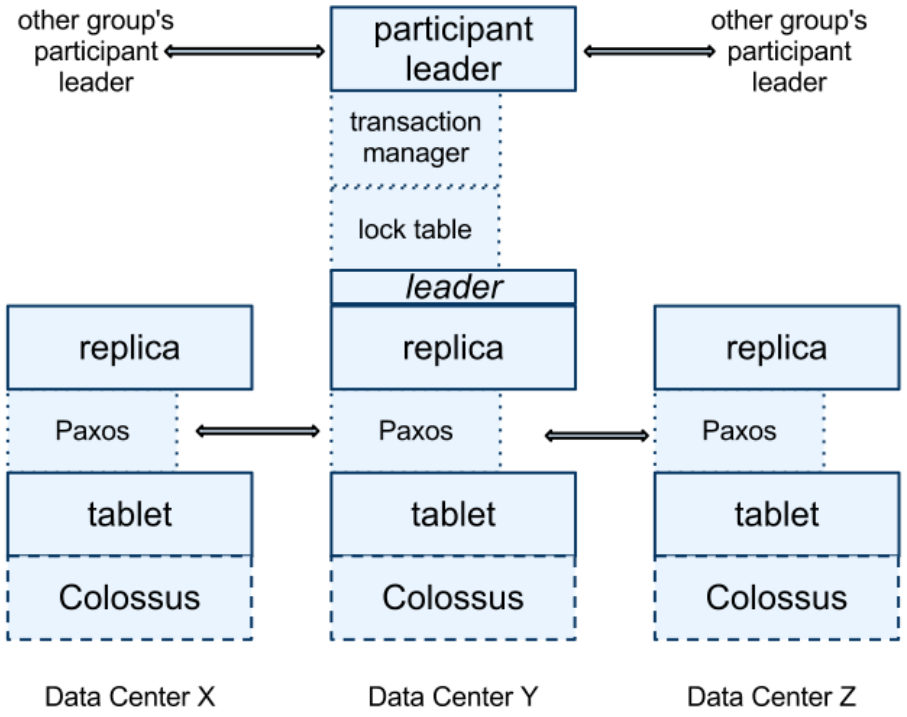
  (key:string, timestamp:int64) → string
- Unlike Bigtable, Spanner **assigns timestamps** to data
  - Spanner is more like a multi-version database than a key-value store
- Directory:
  - Set of contiguous keys that share a common prefix
  - Smallest unit of data placement
  - Smallest unit to define replication properties
  -

# Replication



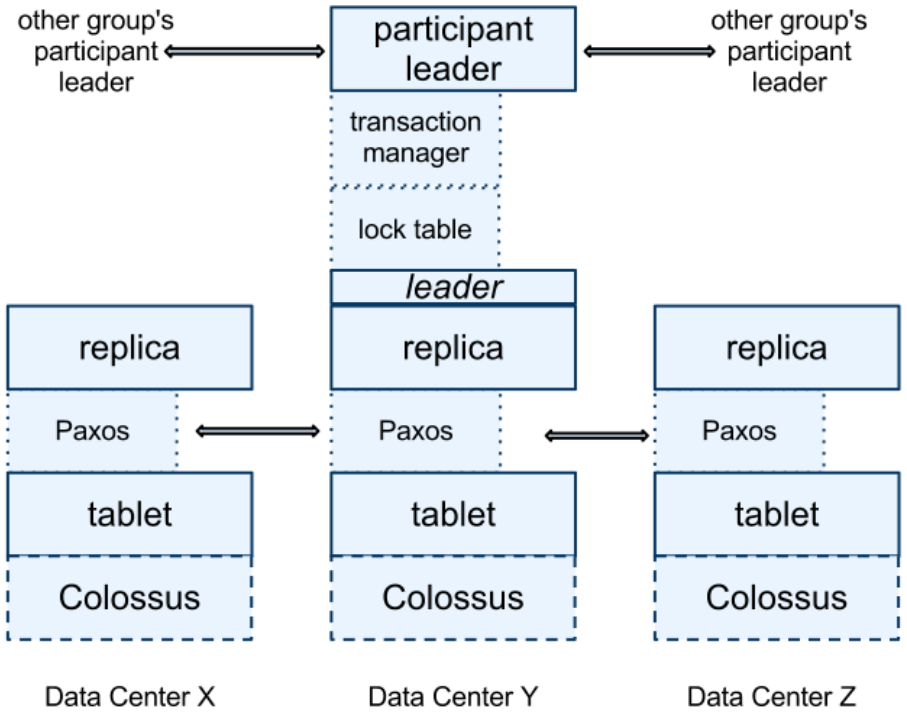
# Replication (cont.)

- Replication performed per tablet using Paxos
  - 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
  - long-lived leaders with time-based leader leases, (default length is 10 seconds)



# Transaction manager

- If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform two- phase commit.
- One of the participant groups is chosen as the coordinator
- The state of each transaction manager is stored in the underlying Paxos group (and therefore is replicated).



# TrueTime

- Leverages hardware features like GPS and Atomic Clocks
- Implemented via TrueTime API
  - Key method is **now()** which not only returns current system time but also value (ε) which tells the maximum uncertainty (less than 10ms) in the time returned
- Set of time master server per datacenters and time slave daemon per machines
- 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 have different failure rates and scenarios
- Two other boolean methods in API are
  - After(t) – returns TRUE if t is definitely passed
  - Before(t) – returns TRUE if t is definitely not arrived
- TrueTime uses these methods in concurrency control and to serialize transactions

# TrueTime and operations

- 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
  - Use specifies specific **timestamp** from past or **timestamp bound** so that data till that point will be read

# Paxos leaders leases

- Paxos uses timed leases to make leadership long-lived (10 seconds by default)
  - potential leader sends requests for timed *lease votes*
  - upon receiving a quorum of lease votes the leader knows it has a lease
  - a replica extends its lease vote implicitly on a successful write, and the leader requests lease-vote extensions if they are near expiration
- Spanner depends on (and enforces) the following disjointness invariant: for each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's.

# Paxos leaders and TrueTime

- A Paxos leader can abdicate by releasing its slaves from their lease votes. To preserve the disjointness invariant, Spanner constrains when abdication is permissible
  - ***Smax*** is the maximum timestamp used by a leader.
- A Paxos leader uses after(**Smax**) to check if Smax is passed so it can abdicate its slaves
- Paxos Leaders can not assign timestamps(Si) greater than Smax for transactions(Ti) and clients can not see the data committed by transaction Ti till after(Si) is true
- Replicas maintain a timestamp **tsafe** which is the maximum timestamp at which that replica is up to date.

# TBF Commit Wait Invariant

- If the start of a transaction T 2 occurs after the commit of a transaction T1, then the commit timestamp of T2 must be greater than the commit timestamp of T1.
- Define the start and commit events for a transaction T by e*start* and e*commit*; and iii the commit timestamp of a transaction Ti by si. The invariant becomes t (e*commit*) < t (e*start*) ⇒ s < s .
- 4.2 Details
- This section explains some of the practical details of read-write transactions and read-only transactions elided earlier, as well as the implementation of a special transaction type used to implement atomic schema changes.
- 4.1.3

# Read-write transactions (1)

- Writes that occur in a transaction are buffered at the client until commit
- When a client has completed all reads and buffered all writes, it begins a two-phase commit
  - The client chooses a coordinator group and sends a commit message to each participant's leader with the identity of the coordinator and any buffered writes
  - A non-coordinator-participant leader first acquires write locks. It then chooses a prepare timestamp that must be larger than any timestamps it has assigned to previous transactions and logs a prepare record through Paxos
  - Each participant then notifies the coordinator of its prepare timestamp

# Read-write transactions (2)

- The coordinator leader
  - first acquires write locks, but skips the prepare phase.
  - it chooses a timestamp for the entire transaction after hearing from all other participant leaders. The commit timestamp *s* must be greater or equal to all prepare timestamps greater than *TT.now*().*latest* at the time the coordinator received its commit message, and greater than any timestamps the leader has assigned to previous transactions
  - The coordinator leader then logs a commit record through Paxos (or an abort if it timed out while waiting on the other participants).

# Read-write transactions (3)

- Before allowing any coordinator replica to apply the commit record, the coordinator leader waits until *TT.after*(*s*), so as to obey the commit-wait rule
- After commit wait, the coordinator sends the commit timestamp to the client and all other participant leaders.
- Each participant leader logs the transaction's outcome through Paxos.
- All participants apply at the same timestamp and then release locks.

# Read-only transactions (1)

- Assigning a timestamp requires a negotiation phase between all of the Paxos groups that are involved in the reads
  - Spanner requires a *scope* expression for every read-only transaction, which is an expression that summarizes the keys that will be read by the entire transaction.
- If the scope's values are served by a single Paxos group, then the client issues the read-only transaction to that group's leader.
- That leader assigns *sread* and executes the read.

# Read-only transactions (2)

- If the scope's values are served by multiple Paxos groups"
- Option 1: Do a round of communication with all of the groups's leaders to negotiate *sread* based on *LastTS*().
- Option 2: The client avoids a negotiation round, and just has its reads execute at *sread* = *TT.now*().*latest* (which may wait for safe time to advance). All reads in the transaction can be sent to replicas that are sufficiently up-to-date.

# Schema-change transactions

- TrueTime enables Spanner to support atomic schema changes.
- A Spanner schema-change transaction is a generally non-blocking variant of a standard transaction.
  - It is explicitly assigned a timestamp in the future, which is registered in the prepare phase; schema changes across thousands of servers can complete with minimal disruption to other concurrent activity.
  - Reads and writes, which implicitly depend on the schema, synchronize with any registered schema-change timestamp at time *t*: they may proceed if their timestamps precede *t*, but they must block behind the schema- change transaction if their timestamps are after *t*.

# Applications

- Google advertising backend application – F1
- Replicated across 5 datacenters spread across US

# 4: Dynamo

*Dynamo: Amazon's Highly Available Key-value Store* Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels. SOSP 2007.

Slides taken from www.slideworld.com created by paper authors

# CAP

- States that any networked shared-data system can have at most two of three desirable properties:
  - consistency (C) –
  - high availability (A) of that data (for updates);
  - tolerance to network partitions (P).
- During a network partition and recovery from partition one can not have perfect availability and consistency
- Modern CAP goal should be to maximize combinations of consistency and availability that make sense for a specific application

# PACELC

- States that:
  - In case of network partitioning (P) one has to choose between availability (A) and consistency (C)
  - but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).
- Addresses the fact that CAP does not capture the consistency/latency tradeoff of replicated systems present at all times during system operation
- Example: Dynamo, Cassandra are PA/EL systems if a partition occurs, they give up consistency for availability, and under normal operation they give up consistency for lower latency.

# Motivation

- Even the slightest outage has significant financial consequences and impacts customer trust.
- The platform is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world.
- Persistent state is managed in the face of these failures - drives the reliability and scalability of the software systems

# Dynamo: Motivation

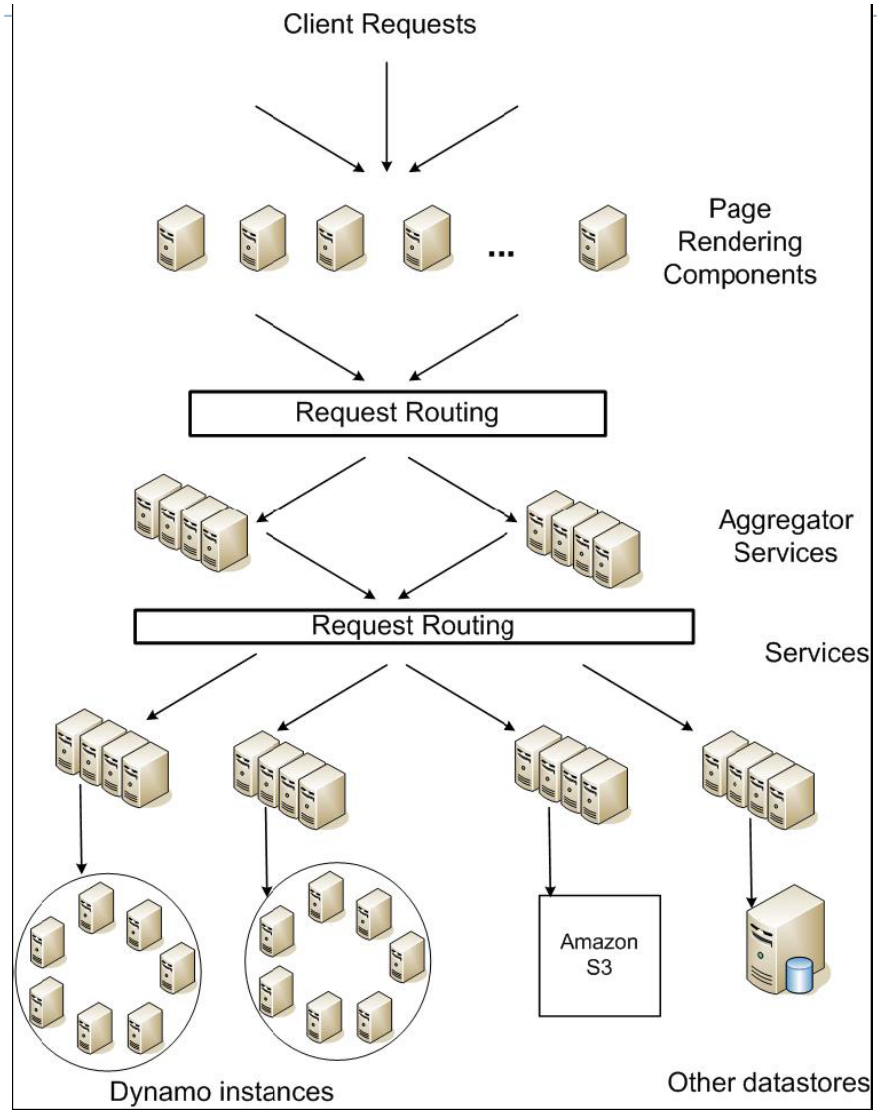
- Build a distributed storage system:
  - Scale
  - Simple: key-value
  - **Highly available**
  - **Guarantee Service Level Agreements (SLA)**

# System Assumptions and Requirements

- *Query Model*: simple read and write operations to a data item that is uniquely identified by a key.
- *ACID Properties:* *Atomicity, Consistency, Isolation, Durability.*
- *Efficiency:* latency requirements which are in general measured at the 99.9th percentile of the distribution.
- Other Assumptions: operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization.

# Service Level Agreements (SLA)

- Application can deliver its functionality in a 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 of Amazon's platform

# Design Consideration

- Sacrifice strong consistency for availability
- Conflict resolution is executed during ***read*** instead of ***write***, i.e. "always writeable".
- Other principles:
  - 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.

# Design Consideration (Cont'd)

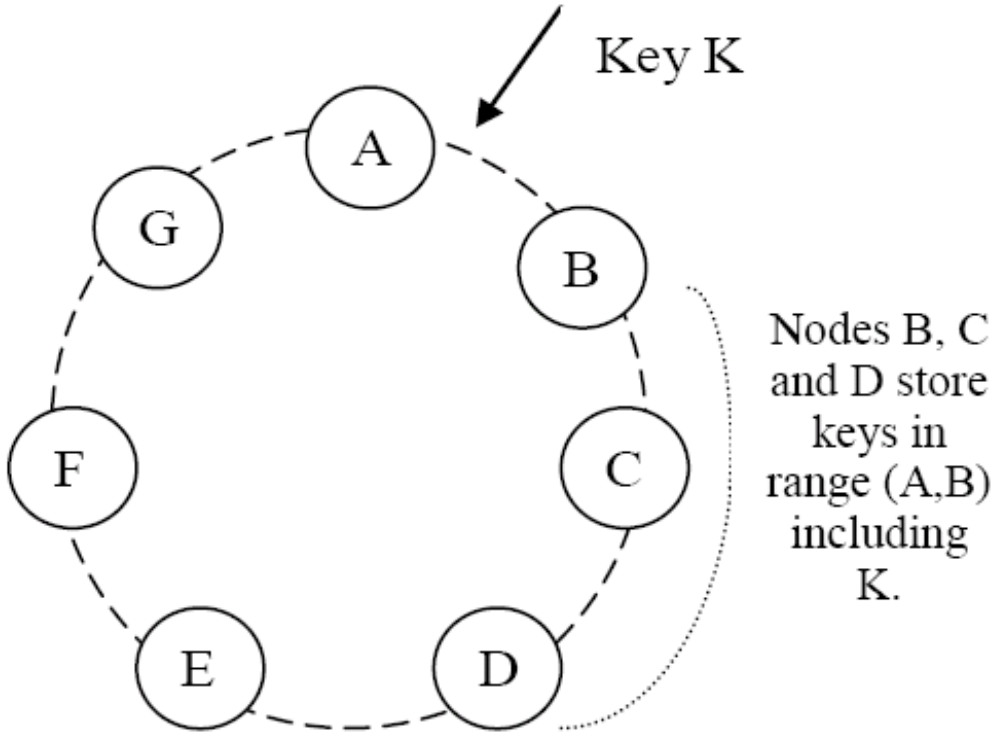
- "always writeable" data store where no updates are rejected due to failures or concurrent writes.
- an infrastructure within a single administrative domain where all nodes are assumed to be trusted.

# 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. |

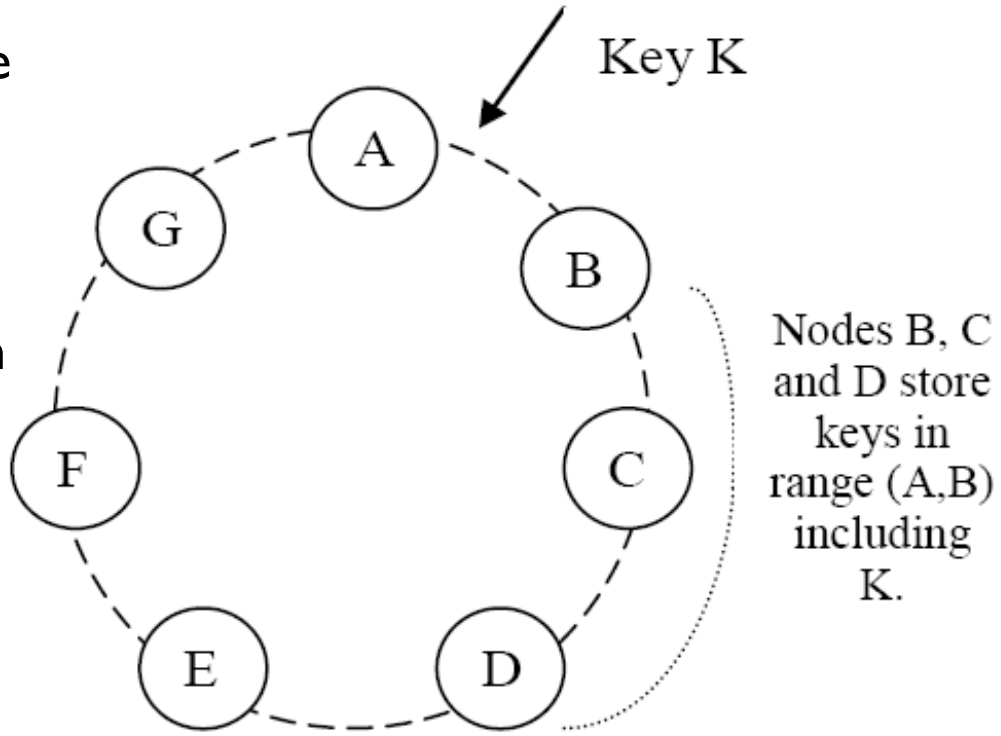
# Partition Algorithm

- **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.



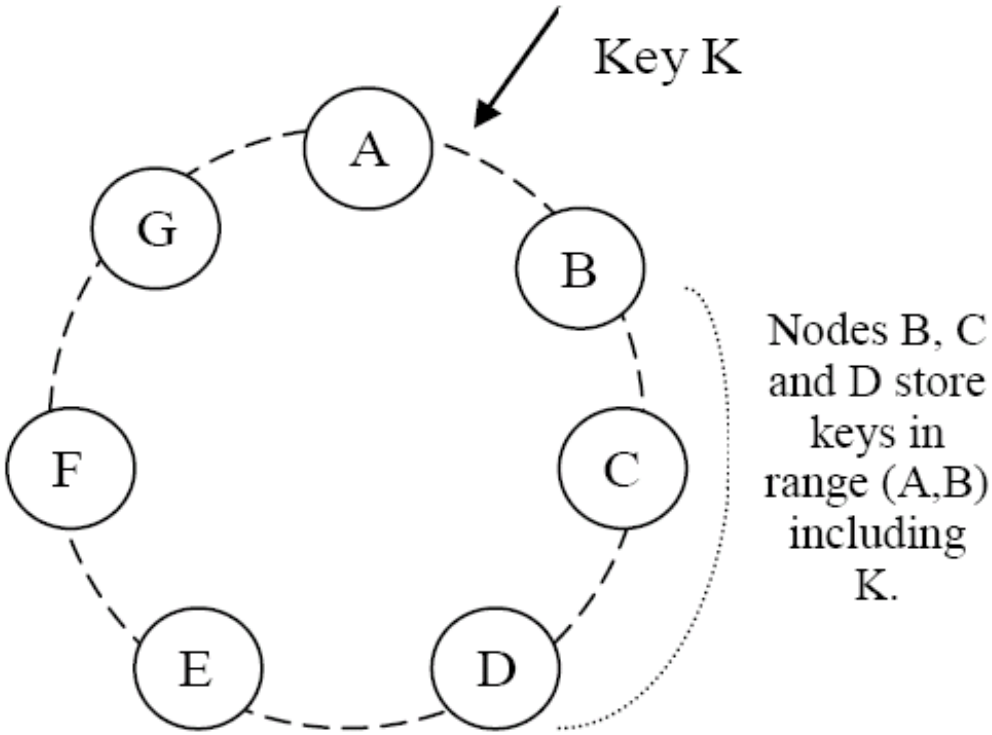
# Advantages of using virtual nodes

- If a node becomes unavailable the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, 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.



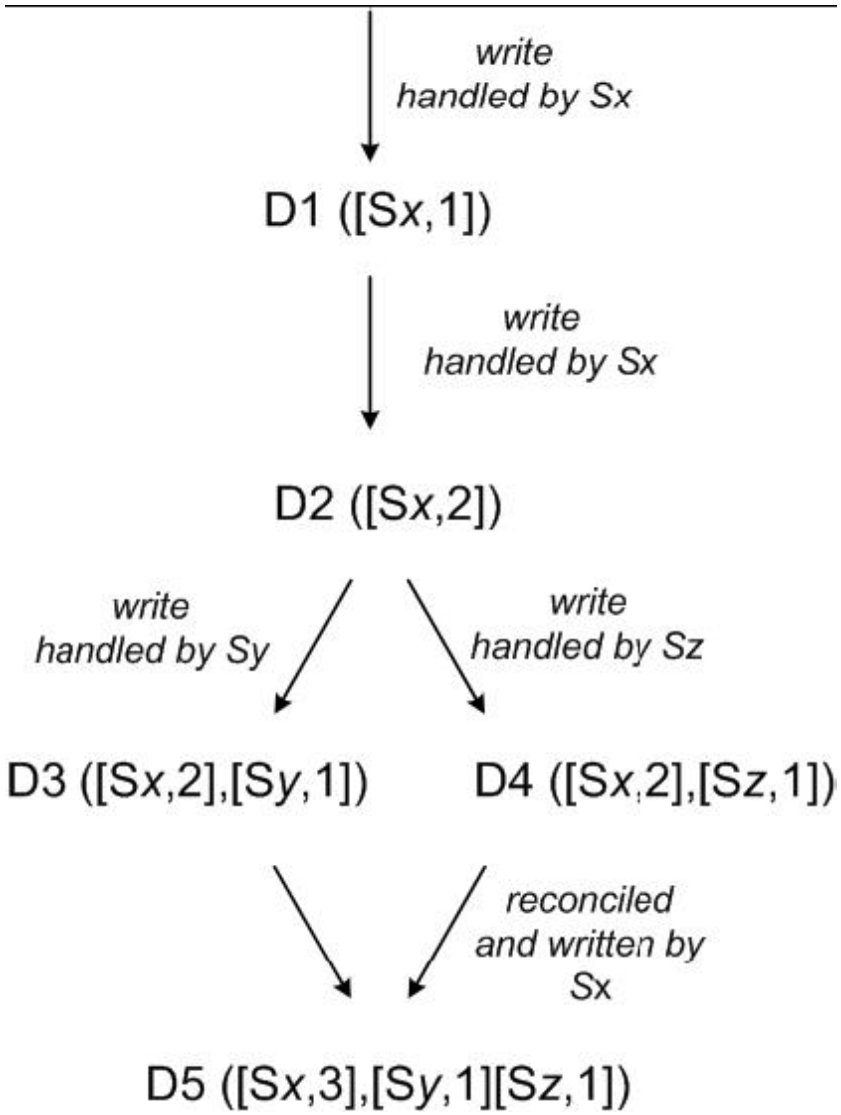
# Data Versioning

- 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.
- Challenge: an object having distinct version sub-histories, which the system will need to reconcile in the future.
- 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.*

# Vector clock example



# Vector clock

- 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.

# Execution of get () and put () operations

1. Route its request through a generic load balancer that will select a node based on load information.
    1. Advantage: client does not have to link any code specific to Dynamo in its application

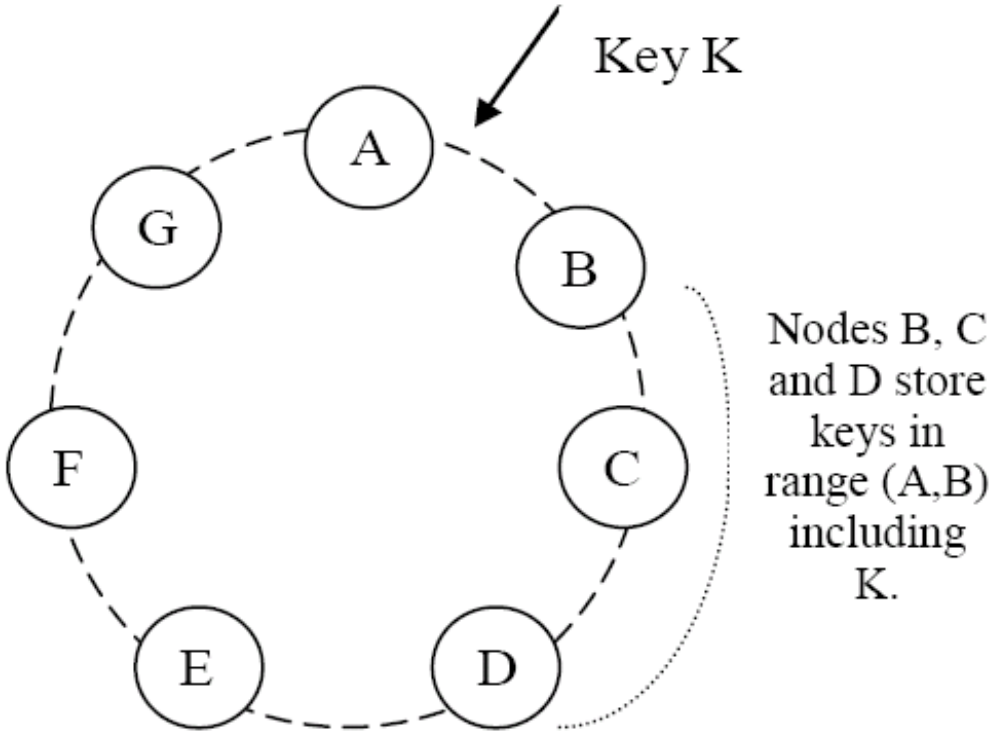
2. Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.
    1. Advantage: can achieve lower latency because it skips a potential forwarding step.

# Temporary failures: Sloppy Quorum

- 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. For this reason, 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 is belong to A and it will deliver to A when A is recovered.
- Again: "always writeable"



# Other techniques

- Replica synchronization:
  - Merkle hash tree.
- Membership and Failure Detection:
  - Gossip

# Replica synchronization

- 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.

# Implementation

- Java
- 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.

# Improvement

- 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

# 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.

# 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.

# 5: Cassandra

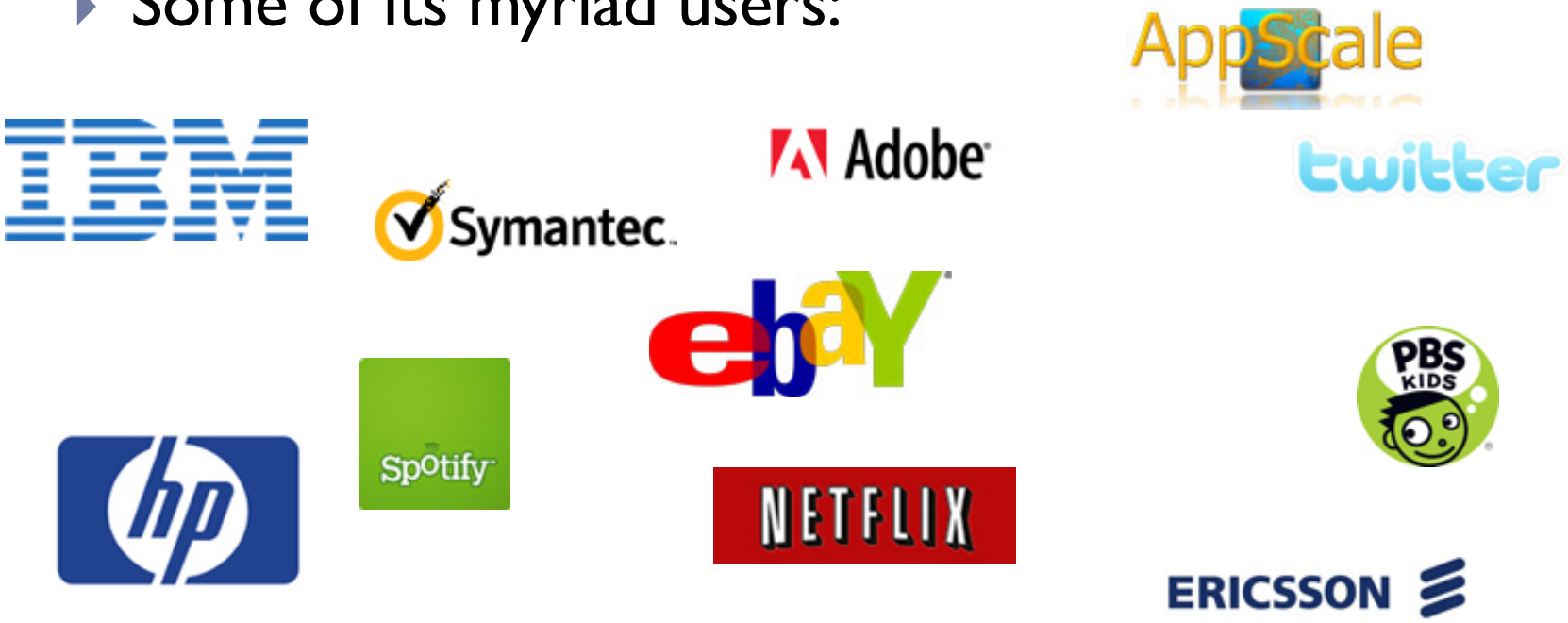
## Slides by Indranil Gupta, UIUC

Based mostly on

- Cassandra NoSQL presentation
- Cassandra 1.0 documentation at datastax.com
- Cassandra Apache project wiki
- HBase

# Cassandra

- Originally designed at Facebook
- Open-sourced
- Some of its myriad users:

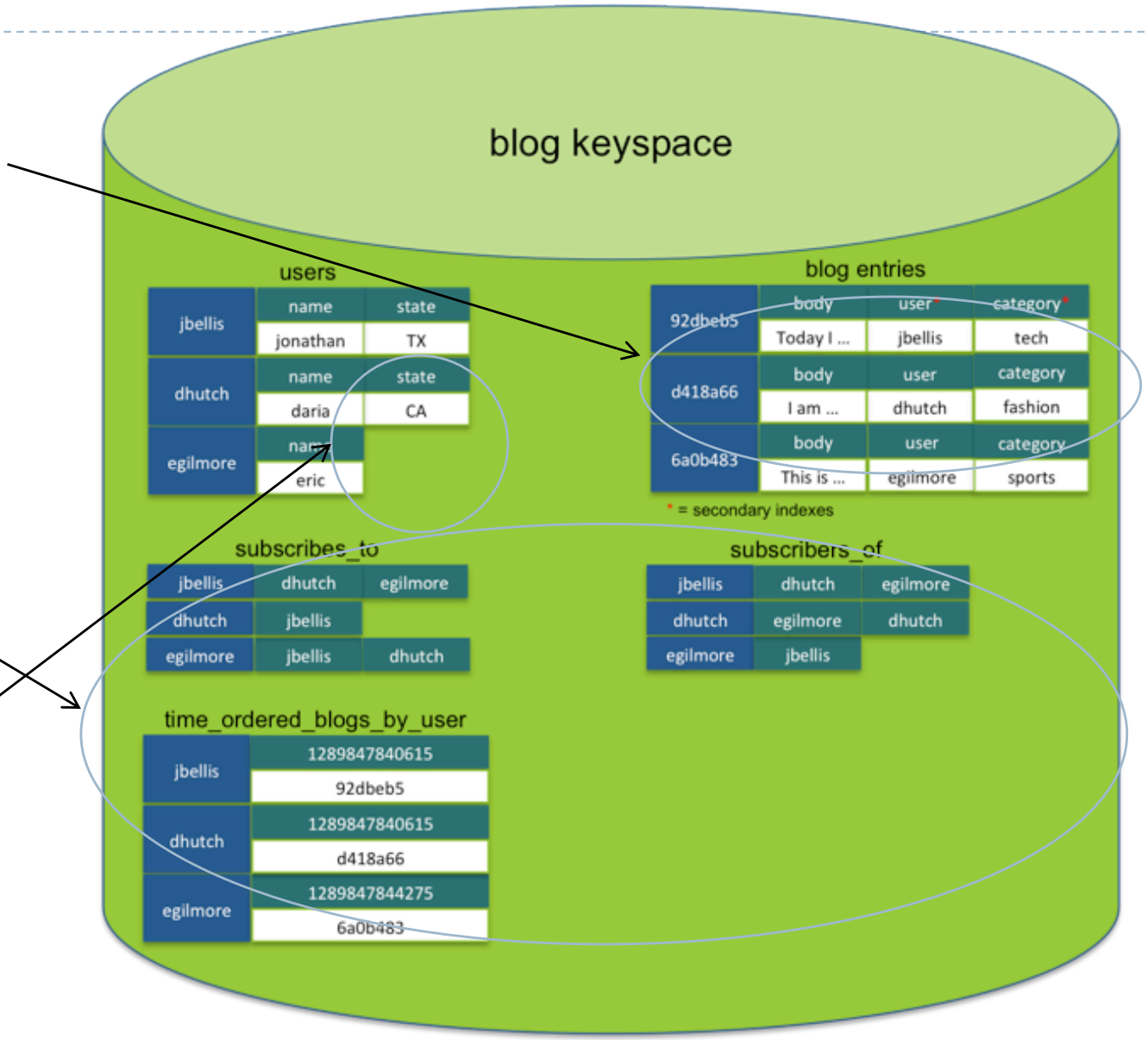


# Issues with today's workloads

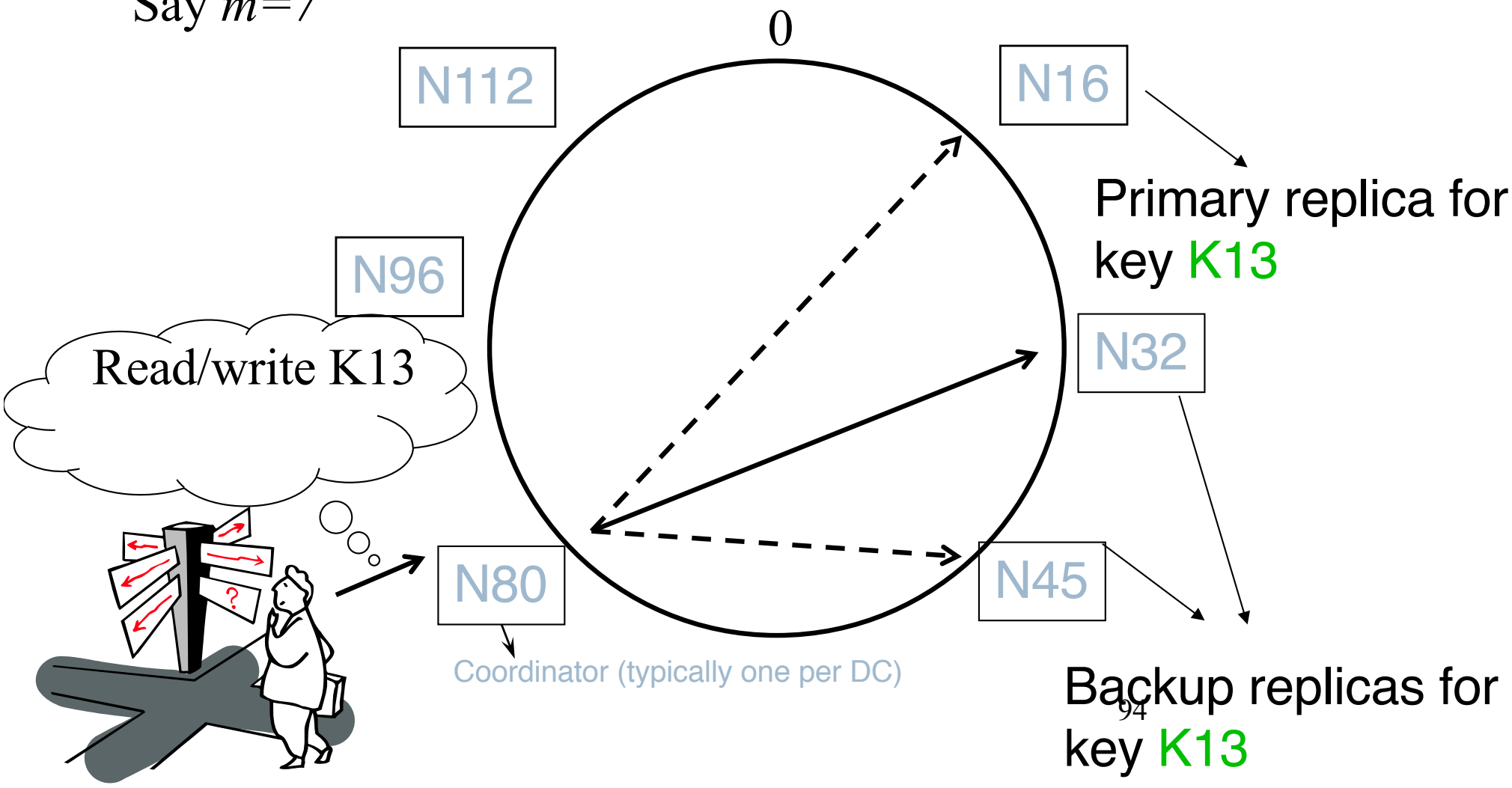
- Data: Large and unstructured
- Lots of random reads and writes
- Foreign keys rarely needed
- Need
  - Incremental Scalability
  - Speed
  - No Single point of failure
  - Low TCO and admin
  - Scale out, not up

# Cassandra data model

- Column Families:
  - Like SQL tables
  - but may be unstructured (client-specified)
  - Can have index tables
- Hence "column-oriented databases"/ "NoSQL"
  - No schemas
  - Some columns missing from some entries
  - "Not Only SQL"
  - Supports get(key) and put(key, value) operations
  - Often write-heavy workloads



Say $m=7$




Cassandra uses a Ring-based DHT but without routing

# Writes

- Need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one front-end node in Cassandra cluster (Coordinator)
- Which (via Partitioning function) sends it to all replica nodes responsible for key
  - Always writable: Hinted Handoff
    - If any replica is down, the coordinator writes to all other replicas, and keeps the write until down replica comes back up.
    - When all replicas are down, the Coordinator (front end) buffers writes (for up to an hour).
  - Provides Atomicity for a given key (i.e., within ColumnFamily)
- One ring per datacenter
  - Coordinator can also send write to one replica per remote datacenter

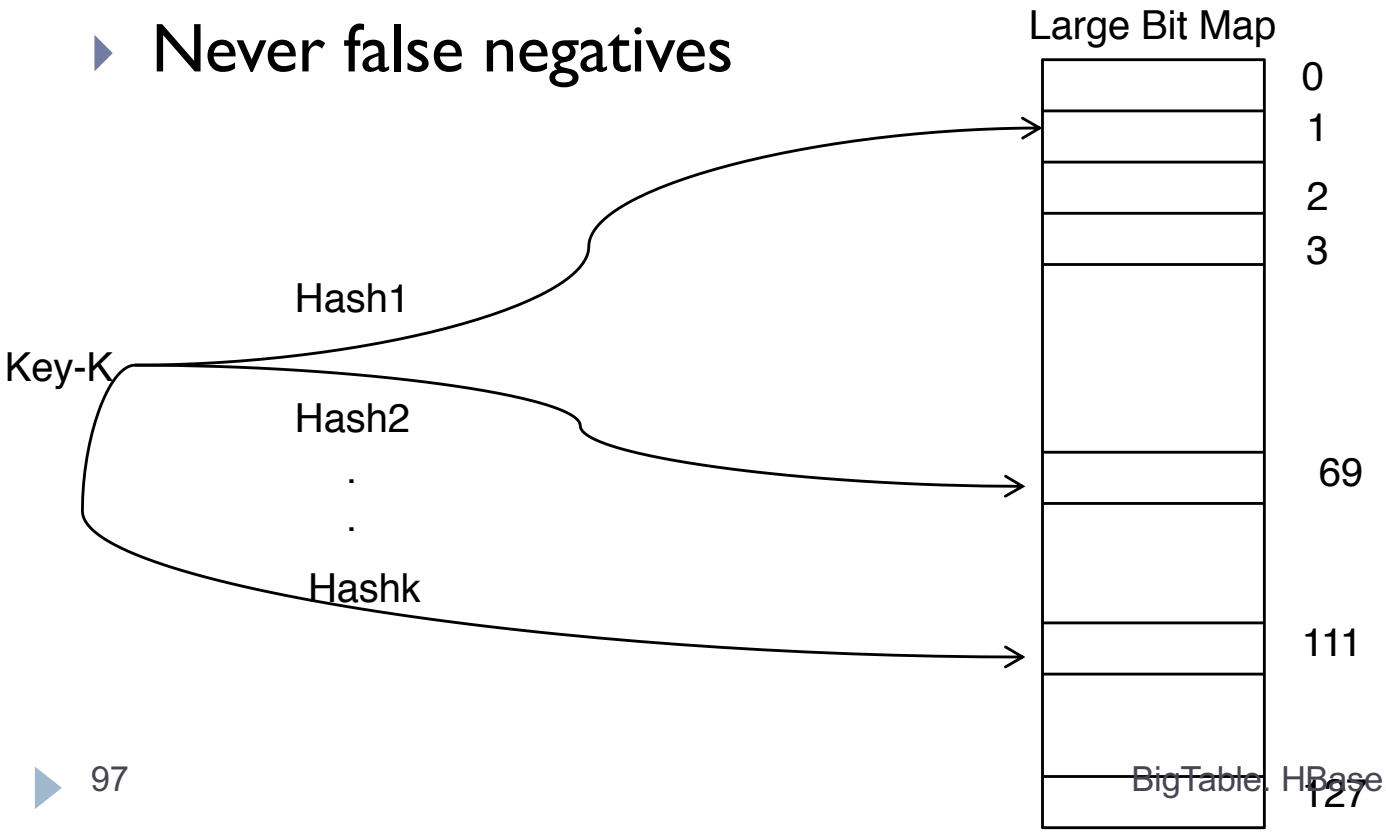
# Writes at a replica node

On receiving a write

- 1. log it in disk commit log
- 2. Make changes to appropriate memtables
  - In-memory representation of multiple key-value pairs
- Later, when memtable is full or old, flush to disk
  - Data File: An SSTable (Sorted String Table) – list of key value pairs, sorted by key
  - Index file: An SSTable – (key, position in data sstable) pairs
    - And a Bloom filter
- Compaction: Data udpates accumulate over time and sstables and logs need to be compacted
  - Merge key updates, etc.
- Reads need to touch log and multiple SSTables
  - May be slower than writes

# Bloom Filter

- Compact way of representing a set of items
- Checking for existence in set is cheap
- Some probability of false positives: an item not in set may check true as being in set
- Never false negatives




On insert, set all hashed bits.

On check-if-present, return true if all hashed bits set.

- False positives

False positive rate low

- k=4 hash functions
- 100 items
- 3200 bits
- FP rate = 0.02%

# Deletes and Reads

- Delete: don't delete item right away
  - Add a tombstone to the log
  - Compaction will remove tombstone and delete item

- Read: Similar to writes, except
  - Coordinator can contact closest replica (e.g., in same rack)
  - Coordinator also fetches from multiple replicas
    - check consistency in the background, initiating a read-repair if any two values are different
    - Makes read slower than writes (but still fast)
    - Read repair: uses gossip

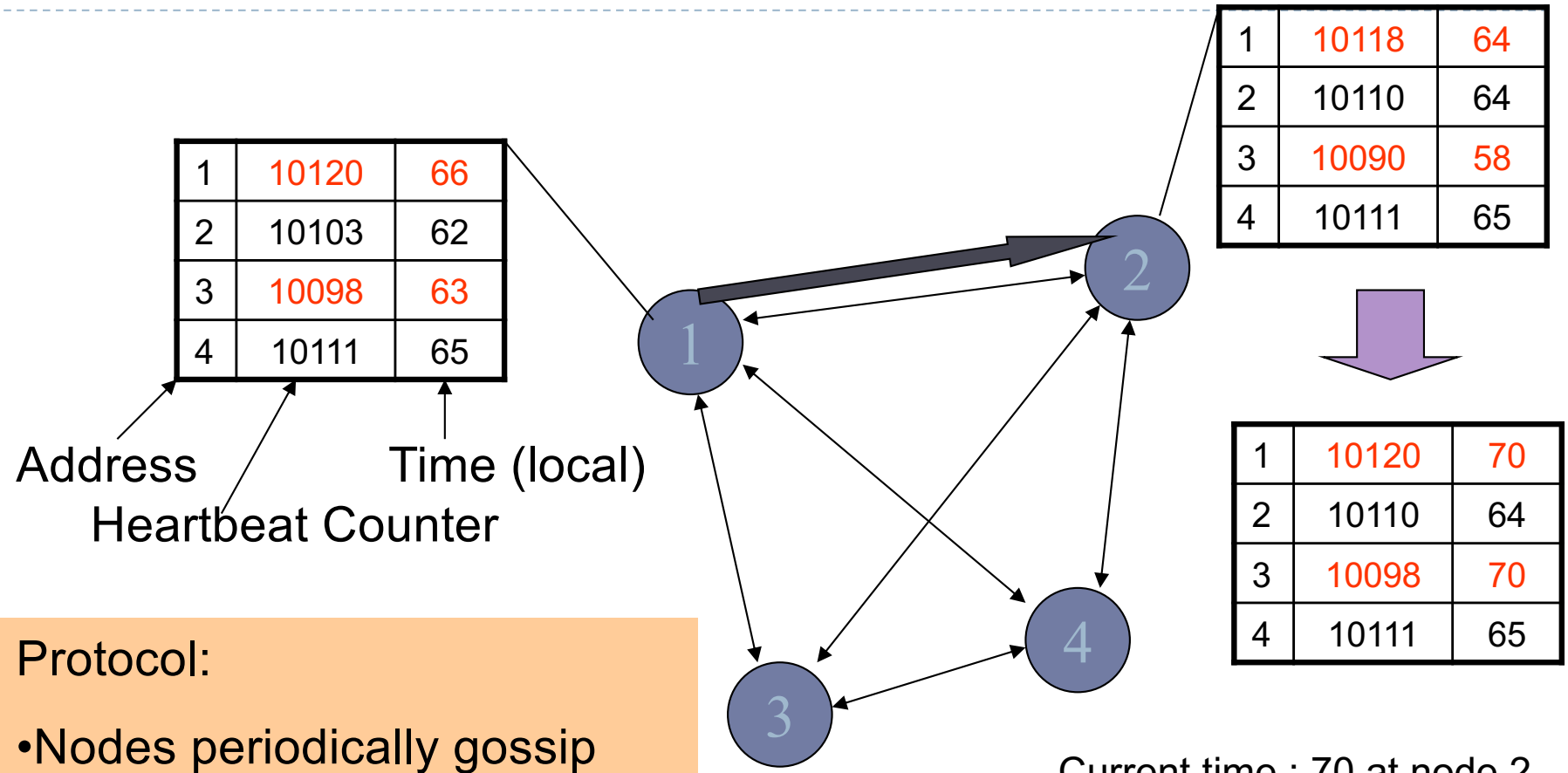
# Cassandra uses Quorums

- Reads
  - Wait for R replicas (R specified by clients)
  - In background check for consistency of remaining N-R replicas, and initiate read repair if needed (N = total number of replicas for this key)
- Writes come in two flavors
  - Block until quorum is reached
  - Async: Write to any node
- Quorum Q = N/2 + 1
- R = read replica count, W = write replica count
- If W+R > N and W > N/2, you have consistency
- Allowed (W=1, R=N) or (W=N, R=1) or (W=Q, R=Q)

# Cassandra uses Quorums

- In reality, a client can choose one of these levels for a read/write operation:
  - ANY: any node (may not be replica)
  - ONE: at least one replica
  - QUORUM: quorum across all replicas in all datacenters
  - LOCAL_QUORUM: in coordinator's DC
  - EACH_QUORUM: quorum in every DC
  - ALL: all replicas all DCs

# Cluster Membership




Protocol:

- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated

Current time : 70 at node 2

(asynchronous clocks)

Cassandra uses gossip-based cluster membership



Fig and animation by: Dongyun Jin and Thuy Nguyen

# Cluster Membership, contd.

- Suspicion mechanisms
- Accrual detector: FD outputs a value (PHI) representing suspicion
- Apps set an appropriate threshold
- PHI = 5 => 10-15 sec detection time
- PHI calculation for a member
  - Inter-arrival times for gossip messages
  - PHI(t) = - log(CDF or Probability(t_now – t_last))/log 10
  - PHI basically determines the detection timeout, but is sensitive to actual inter-arrival time variations for gossiped heartbeats

Cassandra uses gossip-based cluster membership

Spanner. Dynamo. Cassandra

Fig and animation by: Dongyun Jin and Thuy Nguyen

# Vs. SQL

- MySQL is the most popular (and has been for a while)
- On > 50 GB data
- MySQL
  - Writes 300 ms avg
  - Reads 350 ms avg
- Cassandra
  - Writes 0.12 ms avg
  - Reads 15 ms avg

# Cassandra Summary

- While RDBMS provide ACID (Atomicity Consistency Isolation Durability)
- Cassandra provides **BASE**
  - Basically Available Soft-state Eventual Consistency
  - Prefers Availability over consistency
- Other NoSQL products
  - MongoDB, Riak