

IEMS5709

Advanced Topics in Information Processing  
Big Data Systems and Information Processing  
Spring 2016

Big Data Stores (aka NoSQL Databases )

Prof. Wing C. Lau

Department of Information Engineering

wclau@ie.cuhk.edu.hk

# Acknowledgements

- The slides used in this chapter are adapted from the following sources:
  - CS5412 Cloud Computing, by Ken Birman, Cornell
  - CS498 Cloud Computing, by Roy Campbell and Reza Farivar, UIUC.
  - CS525 Advanced Distributed Systems, by Indranil Gupta, UIUC
  - Slides by Daniel J. Abadi, Yale University
  - Perry Hoekstra, Jiaheng Lu, Avinash Lakshman, Prashant Malik, and Jimmy Lin, “NoSQL and Big Data Processing, BigTable, Hbase, Cassandra, Hive and Pig”
- All copyrights belong to the original authors of the materials.

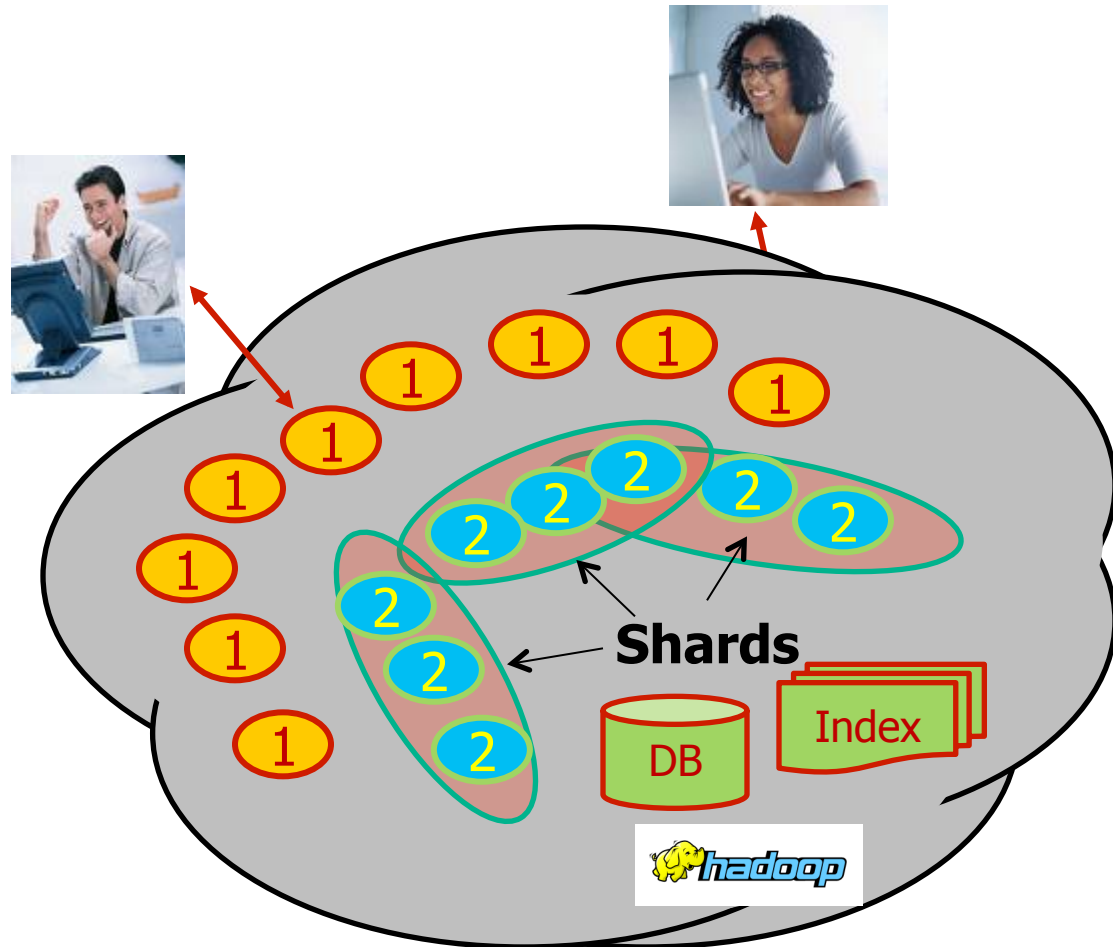
# The NoSQL Movement

# History of the World, Part 1

- Relational Databases – main stay of business
- Web-based applications caused spikes
  - Especially true for public-facing e-Commerce sites
- Developers begin to front RDBMS with memcache or integrate other caching mechanisms within the application (ie. Ehcache)

# Big picture overview

- Client requests are handled in the “first tier” by
  - PHP or ASP pages
  - Associated logic
- These lightweight services are fast and very nimble
- Much use of caching:  
the second tier



# Scaling Up

- Issues with scaling up when the dataset is just too big
- RDBMS were not designed to be distributed
- Began to look at multi-node database solutions
- Known as 'scaling out' or 'horizontal scaling'
- Different approaches include:
  - Master-slave
  - Sharding

# Scaling RDBMS – Master/Slave

- Master-Slave

- All writes are written to the master. All reads performed against the replicated slave databases
- Critical reads may be incorrect as writes may not have been propagated down
- Large data sets can pose problems as master needs to duplicate data to slaves

# Scaling RDBMS via various Sharding approaches

- Vertical Partitioning:
  - Have tables related to a specific feature sit on their own server
  - May have to rebalance or reshard if tables outgrow server
- Range-based Partitioning:
  - When a single table cannot sit on a server, split the rows (aka horizontal partitioning) of the table onto multiple servers based on some value range of a column/key
- Key/Hash-based Partitioning
  - Input a Key value to a Hash and use the resultant hash output as entry into multiple servers
- Directory-based Partitioning
  - Have a lookup service that has the knowledge of the partitioning scheme
  - This allows for the adding of servers or changing of partition scheme without changing the application



# Scaling RDBMS – Sharding (cont'd)

- Partition or Sharding
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
  - Can no longer have relationships/joins across partitions
  - Loss of referential integrity across shards

# Other ways to scale RDBMS

- Multi-Master replication
- INSERT only, not UPDATES/DELETES
- No JOINS, thereby reducing query time
  - This involves de-normalizing data
- In-memory databases

# What is NoSQL?

- Stands for **Not Only SQL**
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties

Note:

ACID = Atomicity, Consistency, Isolation, Durability

BASE = Basic Availability, Soft-state, Eventual consistency

# The Perfect Storm

- Large datasets, acceptance of alternatives, and dynamically-typed data has come together in a perfect storm
- Not a backlash/rebellion against RDBMS
- SQL is a rich query language that cannot be rivaled by the current list of NoSQL offerings

# Why NoSQL and Why Not ?

- For data storage, an RDBMS cannot be the be-all/end-all
- Just as there are different programming languages, need to have other data storage tools in the toolbox
- A NoSQL solution is more acceptable to a client now than even a year ago **BUT you should hear the arguments from the other side, e.g. Prof. M. Stonebraker:**
  - M. Stonebraker, “The NoSQL discussion has nothing to do with SQL,” [Blog@ACM,http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext](http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext)
  - M. Stonebraker, “New SQL: An alternative to NoSQL and Old SQL for new OLTP Apps,” Blog@CACM, Jun 2011, <http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext>
  - M. Stonebraker, “OldSQL vs. NoSQL vs. NewSQL on New OLTP,” Usenix LISA 2011, <https://www.usenix.org/legacy/events/lisa11/tech/>
- There are recent research success in building Scalable, Fully-ACID compliant transactional distributed database systems, e.g. the Calvin system from Yale and the “Coordination Avoidance” work by P. Bailis et al of Berkeley/Stanford in VLDB2015.

# How did NoSQL become popular ?

- Explosion of social media sites (Facebook, Twitter) with large data needs
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Just as moving to dynamically-typed languages (Ruby/Groovy), a shift to dynamically-typed data with frequent schema changes
- Open-source community

# Seeds of the NoSQL movement

- Three major papers were the seeds of the NoSQL movement:
  - BigTable (Google)
  - Dynamo (Amazon)
    - Gossip protocol (discovery and error detection)
    - Distributed key-value data store
    - Eventual consistency
  - CAP Theorem: which states that Strict Consistency can't be achieved at the same time as availability and partition-tolerance.

# Consistency Model

- A consistency model determines rules for visibility and apparent order of updates.
- For example:
  - Row X is replicated on nodes M and N
  - Client A writes row X to node N
  - Some period of time  $t$  elapses.
  - Client B reads row X from node M
  - Does client B see the write from client A?
  - Consistency is a continuum with tradeoffs
  - For NoSQL, the answer would be: maybe
    - e.g., Amazon statement of “*Usually* ships within 2 days” as it cannot prevent the last copy of an item to be sold multiple times !



# What kinds of NoSQL ?

- NoSQL solutions fall into two major areas:
  - Key/Value Store or 'the big hash table'.
    - Amazon S3 (Dynamo)
    - Voldemort
    - Scalaris
    - Memcached (in-memory key/value store)
    - Redis
  - Schema-less which comes in multiple flavors, column-based, document-based or graph-based.
    - Cassandra
    - CouchDB (document-based)
    - MongoDB (document-based)
    - Neo4J (graph-based)
    - HBase
- Different NoSQL DBs do support different Consistency Models, e.g.
  - BigTable/HBase: Strong-Consistency ;
  - Dynamo: Eventual Consistency
  - Cassandra: User-Tunable (e.g. setting the N,R,W parameters)
  - Basic version of Memcached: None

# Key-Value Store

## *Pros:*

- very fast
- very scalable
- simple model
- able to distribute horizontally

## *Cons:*

- many data structures (objects) can't be easily modeled as key value pairs

# Schema-Less

## *Pros:*

- Schema-less data model is richer than key/value pairs
- eventual consistency
- many are distributed
- still provide excellent performance and scalability

## *Cons:*

- typically no ACID transactions or joins

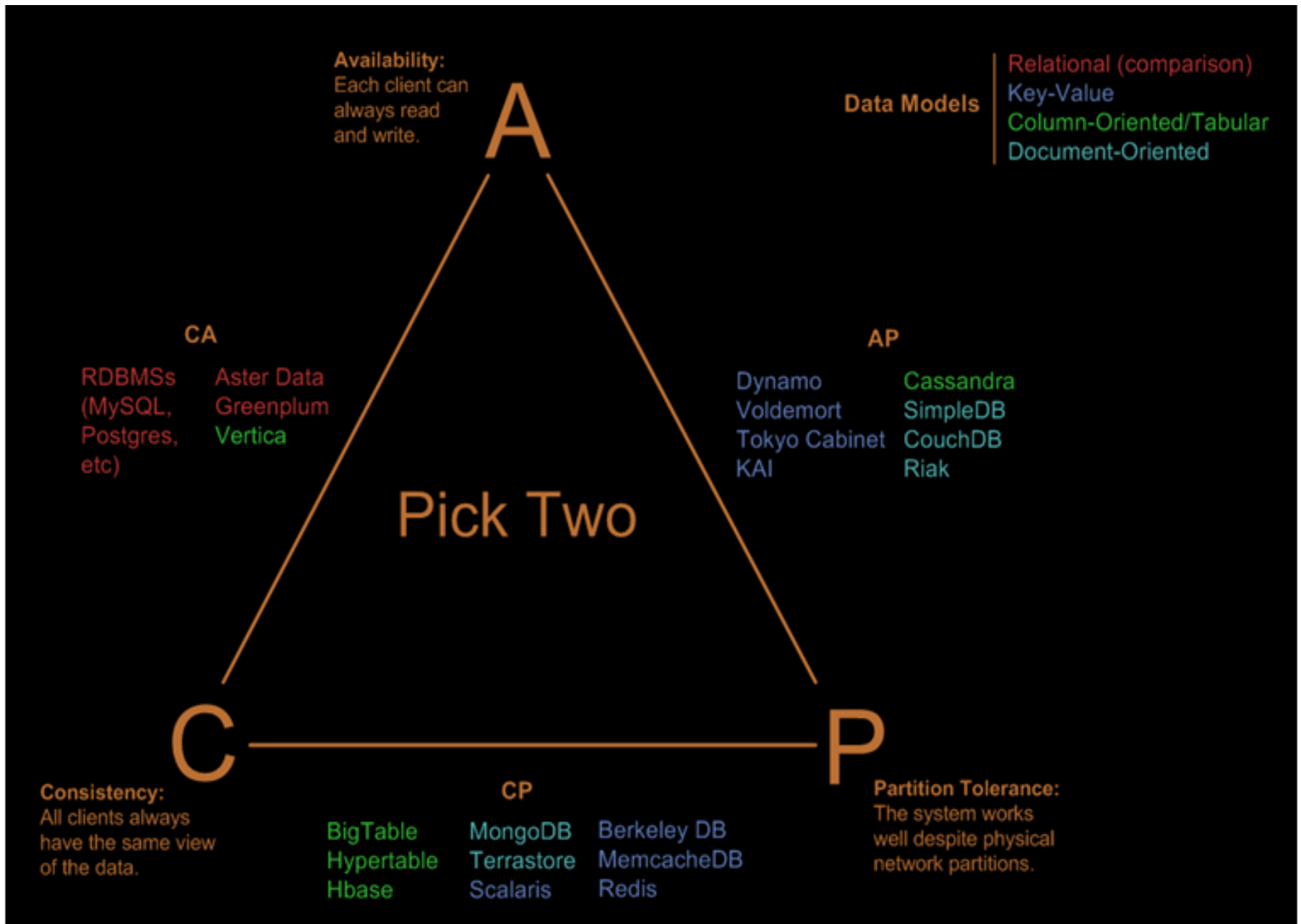
# Common Advantages

- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - Down nodes easily replaced
  - No single point of failure
- Easy to distribute
- Don't require a schema
- Can scale up and down
- Relax the data consistency requirement (CAP)

# What am I giving up?

- joins
- group by
- order by
- ACID transactions
- SQL as a sometimes frustrating but still powerful query language
  - But that's also changing with Integration support of High-level query language/systems like Hive/HiveQL
- easy integration with other applications that support SQL











# Visual Guide to NoSQL Systems



# Yet Another Comparison of Cloud Datastores









| <i>Year</i> | <i>System/<br/>Paper</i> | <i>Scale to<br/>1000s</i> | <i>Primary<br/>Index</i> | <i>Secondary<br/>Indexes</i> | <i>Transactions</i> | <i>Joins/<br/>Analytics</i> | <i>Integrity<br/>Constraints</i> | <i>Views</i> | <i>Language/<br/>Algebra</i> | <i>Data<br/>model</i> | <i>my label</i> |
|-------------|--------------------------|---------------------------|--------------------------|------------------------------|---------------------|-----------------------------|----------------------------------|--------------|------------------------------|-----------------------|-----------------|
| 1971        | RDBMS                    | 0                         | ✓                        | ✓                            | ✓                   | ✓                           | ✓                                | ✓            | ✓                            | tables                | sql-like        |
| 2003        | memcached                | ✓                         | ✓                        | 0                            | 0                   | 0                           | 0                                | 0            | 0                            | key-val               | nosql           |
| 2004        | MapReduce                | ✓                         | 0                        | 0                            | 0                   | ✓                           | 0                                | 0            | 0                            | key-val               | batch           |
| 2005        | CouchDB                  | ✓                         | ✓                        | ✓                            | record              | MR                          | 0                                | ✓            | 0                            | document              | nosql           |
| 2006        | BigTable (Hbase)         | ✓                         | ✓                        | ✓                            | record              | compat. w/MR                | /                                | 0            | 0                            | ext. record           | nosql           |
| 2007        | MongoDB                  | ✓                         | ✓                        | ✓                            | EC, record          | 0                           | 0                                | 0            | 0                            | document              | nosql           |
| 2007        | Dynamo                   | ✓                         | ✓                        | 0                            | 0                   | 0                           | 0                                | 0            | 0                            | key-val               | nosql           |
| 2008        | Pig                      | ✓                         | 0                        | 0                            | 0                   | ✓                           | /                                | 0            | ✓                            | tables                | sql-like        |
| 2008        | HIVE                     | ✓                         | 0                        | 0                            | 0                   | ✓                           | ✓                                | 0            | ✓                            | tables                | sql-like        |
| 2008        | Cassandra                | ✓                         | ✓                        | ✓                            | EC, record          | 0                           | ✓                                | ✓            | 0                            | key-val               | nosql           |
| 2009        | Voldemort                | ✓                         | ✓                        | 0                            | EC, record          | 0                           | 0                                | 0            | 0                            | key-val               | nosql           |
| 2009        | Riak                     | ✓                         | ✓                        | ✓                            | EC, record          | MR                          | 0                                |              |                              | key-val               | nosql           |
| 2010        | Dremel                   | ✓                         | 0                        | 0                            | 0                   | /                           | ✓                                | 0            | ✓                            | tables                | sql-like        |
| 2011        | Megastore                | ✓                         | ✓                        | ✓                            | entity groups       | 0                           | /                                | 0            | /                            | tables                | nosql           |
| 2011        | Tenzing                  | ✓                         | 0                        | 0                            | 0                   | 0                           | ✓                                | ✓            | ✓                            | tables                | sql-like        |
| 2011        | Spark/Shark              | ✓                         | 0                        | 0                            | 0                   | ✓                           | ✓                                | 0            | ✓                            | tables                | sql-like        |
| 2012        | Spanner                  | ✓                         | ✓                        | ✓                            | ✓                   | ?                           | ✓                                | ✓            | ✓                            | tables                | sql-like        |
| 2012        | Accumulo                 | ✓                         | ✓                        | ✓                            | record              | compat. w/MR                | /                                | 0            | 0                            | ext. record           | nosql           |
| 2013        | Impala                   | ✓                         | 0                        | 0                            | 0                   | ✓                           | ✓                                | 0            | ✓                            | tables                | sql-like        |

# Partial List of Managed NoSQL services

|                  | Model       | CAP | Scans   | Sec.<br>Indices  | Largest<br>Cluster | Lear-<br>ning | Lic.   | DBaaS  |
|------------------|-------------|-----|---|--|--------------------|---------------|--------|--|
| <b>HBase</b>     | Wide-Column | CP  | Over Row Key  |    | ~700               | 1/4           | Apache | <br>(EMR)       |
| <b>MongoDB</b>   | Doc-ument   | CP  | yes   |    | >100<br><500       | 4/4           | GPL    |                 |
| <b>Riak</b>      | Key-Value   | AP  |  |    | ~60                | 3/4           | Apache | <br>(Softlayer) |
| <b>Cassandra</b> | Wide-Column | AP  | With Comp. Index  |  | >300<br><1000      | 2/4           | Apache |               |
| <b>Redis</b>     | Key-Value   | CA  | Through Lists, etc.   | manual   | N/A                | 4/4           | BSD    |               |



# Partial List of Proprietary Database Service

|                           | Model        | CAP | Scans   | Sec.<br>Indices  | Queries   | API                | Scale-<br>out   | SLA   |
|---------------------------|--------------|-----|---|--|---|--------------------|---|---|
| <b>SimpleDB</b>           | Table-Store  | CP  | Yes (as queries)  | Auto-matic   | SQL-like (no joins, groups, ...)  | REST + SDKs        |  |    |
| <b>Dynamo-DB</b>          | Table-Store  | CP  | By range key / index  | Local Sec.<br>Global Sec.  | Key+Cond.<br>On Range Key(s)  | REST + SDKs        | Automatic over Prim. Key  |    |
| <b>Azure Tables</b>       | Table-Store  | CP  | By range key  |    | Key+Cond.<br>On Range Key   | REST + SDKs        | Automatic over Part. Key  | 99.9% uptime  |
| <b>AE/Cloud DataStore</b> | Entity-Group | CP  | Yes (as queries)  | Auto-matic   | Conjunct. of Eq. Predicates   | REST/ SDK, JDO,JPA | Automatic over Entity Groups  |  |
| <b>S3, Az. Blob, GCS</b>  | Blob-Store   | AP  |  |  |  | REST + SDKs        | Automatic over key  | 99.9% uptime (S3)   |

# Amazon's Dynamo

# Amazon's Dynamo System

<http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>

- Amazon was interested in improving the scalability of their shopping cart service
- A core component widely used within their system
  - Functions as a kind of **key-value storage** solution
  - Previous version was a transactional database and, just as the BASE folks predicted, wasn't scalable enough
  - Dynamo project created a new version from scratch

# Assumptions and Requirements

- Simple query model
  - Values/Objects are small ( $< 1\text{MB}$ ) binary objects
- Stringent latency requirements
  - Want guarantees on 99.9<sup>th</sup> percentile of latency  
e.g., 300ms response time for 99.9% of requests at peak load of 500 requests/s
- Non-hostile environment
- No ACID properties
  - Single key updates
  - No isolation guarantees
  - Weaker consistency

# Dynamo's System Interface

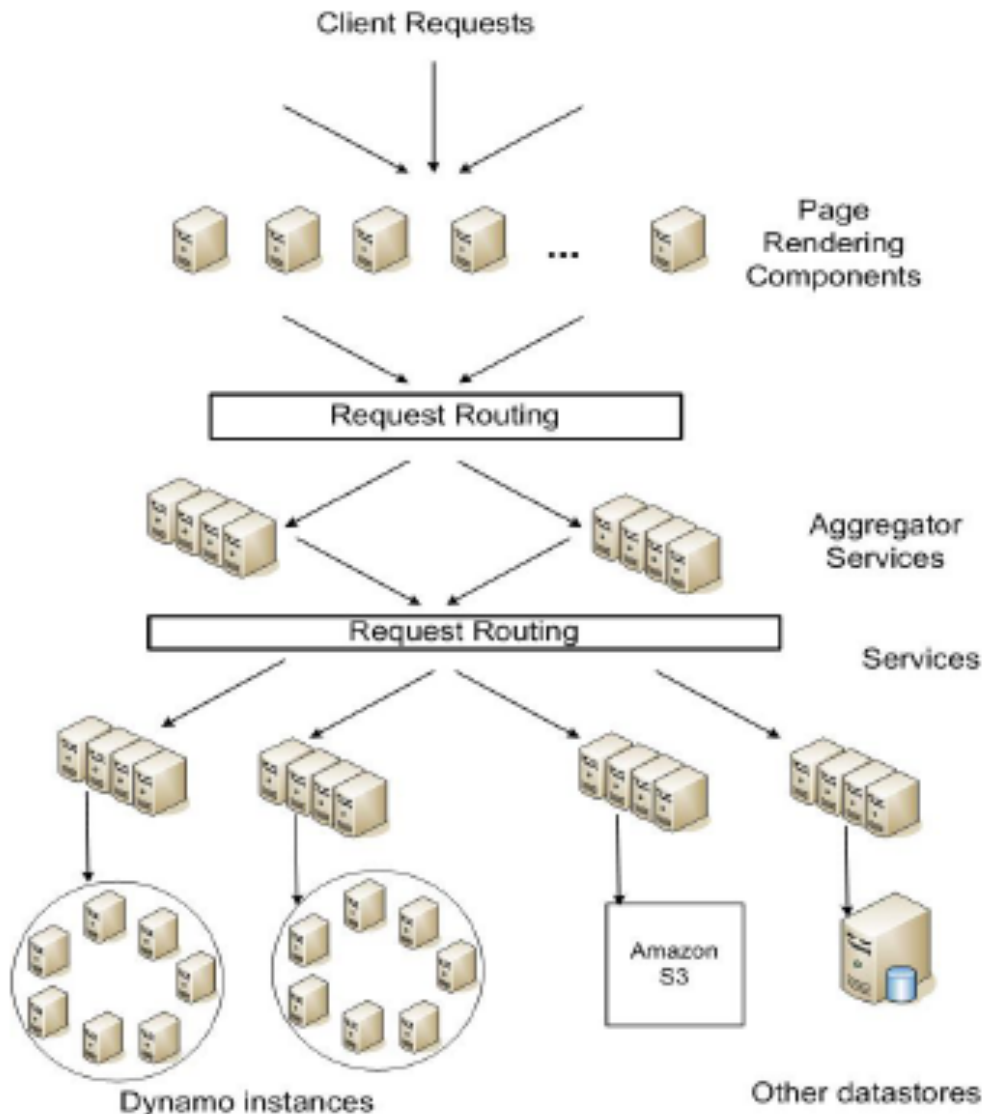
Only two operations

- put (key, context, object)
  - key: primary key associated with data object
  - context: vector clocks (some sort of time-stamp over a distributed system) and history (needed for merging)
  - object: data to store
- get (key)

# Dynamo Design Decisions

- Incremental Scalability
  - Must be able to add nodes on-demand with minimal impact
- Load Balancing & Exploiting Heterogeneity
- Symmetry
  - All nodes are Peers in responsibilities (i.e. a P2P system as opposed to a Master/Slave one)
- Avoid Single-Point-of-Failure (SPOF)

# Amazon Service Architecture and Service Level Agreement (SLA)



- SLAs are used widely at Amazon
- Sub-services must meet strict SLAs
  - Average-case SLAs are not good enough
  - Mentioned a cost-benefit analysis that said 99.9% is the right number
- Rendering a single page can make requests to 150 services

# Dynamo approach

- They made an initial decision to base Dynamo on a Chord-like Distributed Hash Table (DHT) structure
- Plan was to run this DHT in Tier 2 of the Amazon cloud system, with one instance of Dynamo in each Amazon data center and no “linkage” between them
- This works because each data center has “ownership” for some set of customers and handles all of that person’s purchases locally.



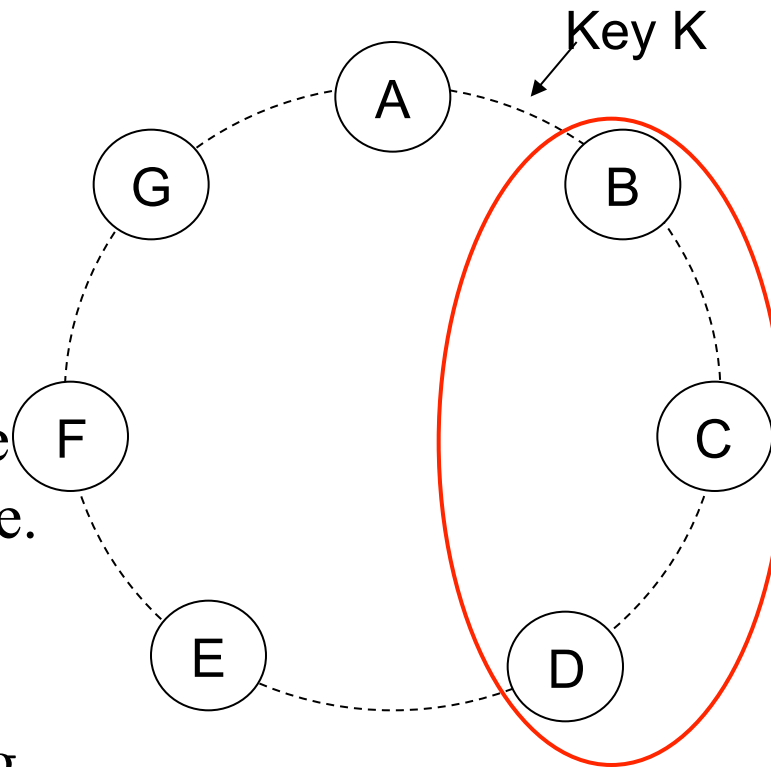
# Variant of Consistent Hashing

## Consistent hashing:

- The o/p range of a hash function is treated as a fixed circular space or “ring”.

## Virtual Nodes:

- Each physical node (machine) can be responsible for more than one virtual node.
- Flexible Load Balancing, Failure handling and dealing with server heterogeneity.

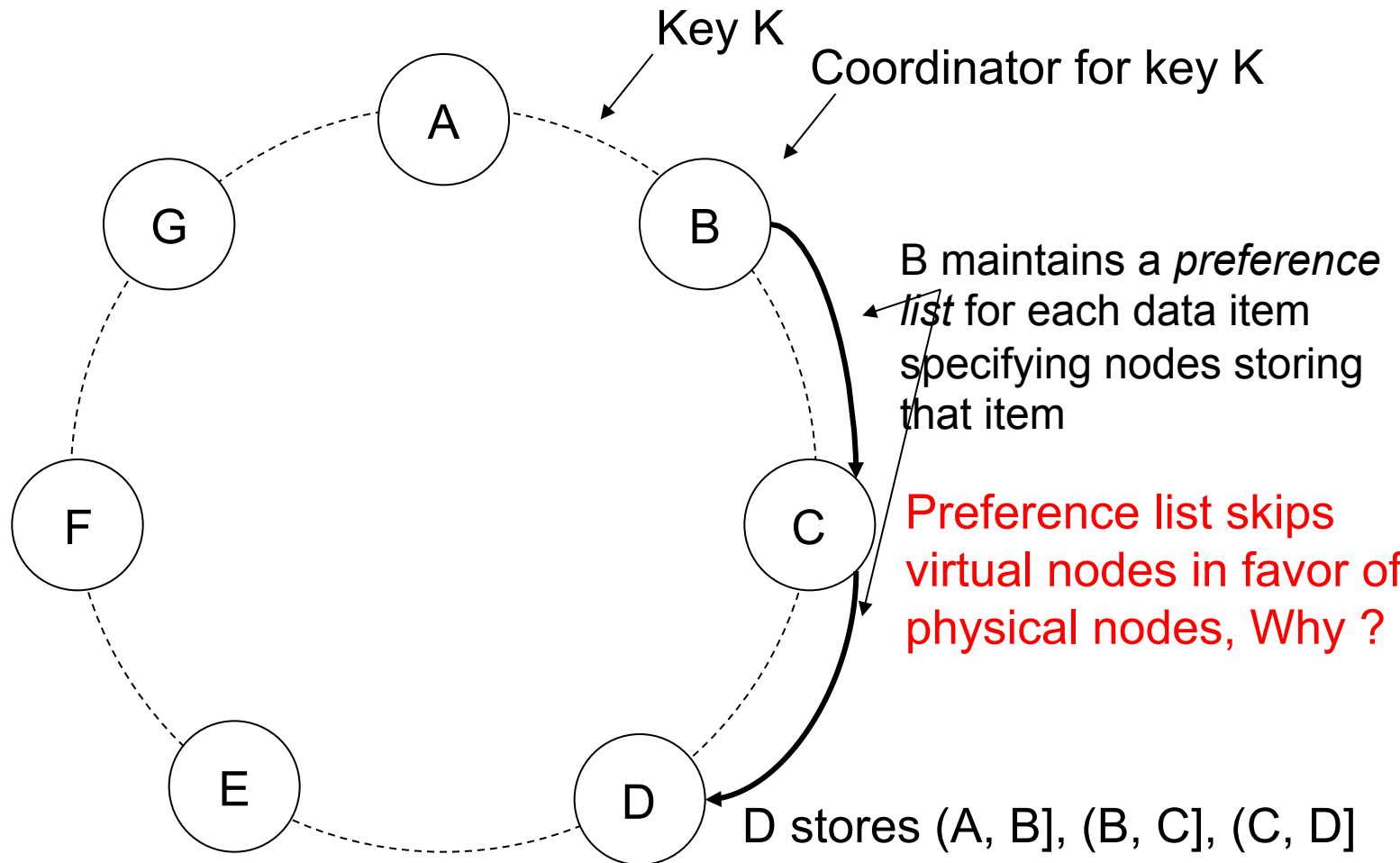


Each node is assigned to multiple points in the ring (e.g., B, C, D store keyrange (A, B))

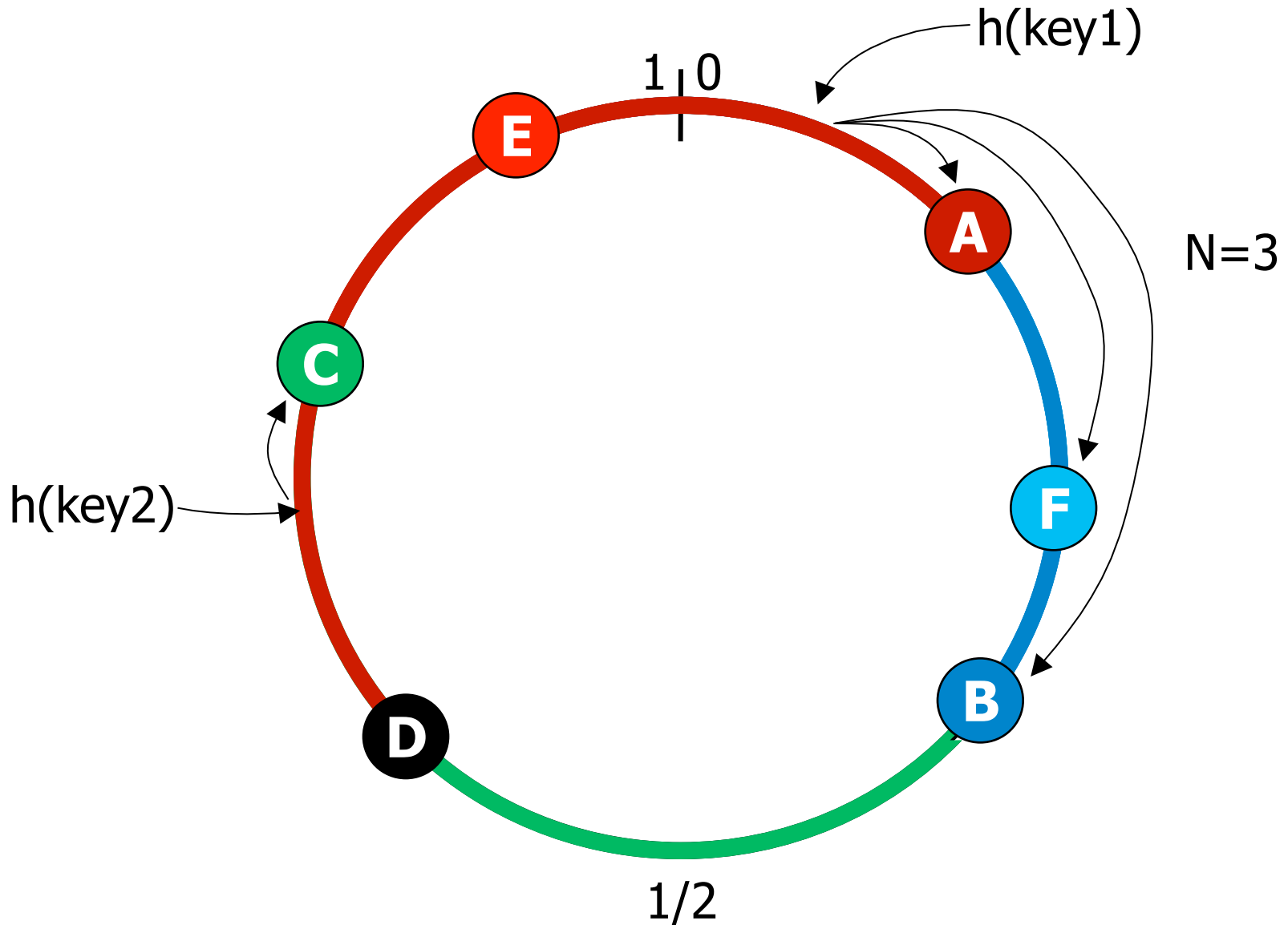
# of points can be assigned based on node's capacity

If node becomes unavailable, load is distributed to others

# Replication



# Partitioning & Replication



# Execution of get () and put () operations

## Two Implementation Choices:

1. Route its request through a generic load balancer that will select a node based on load information.
2. Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.

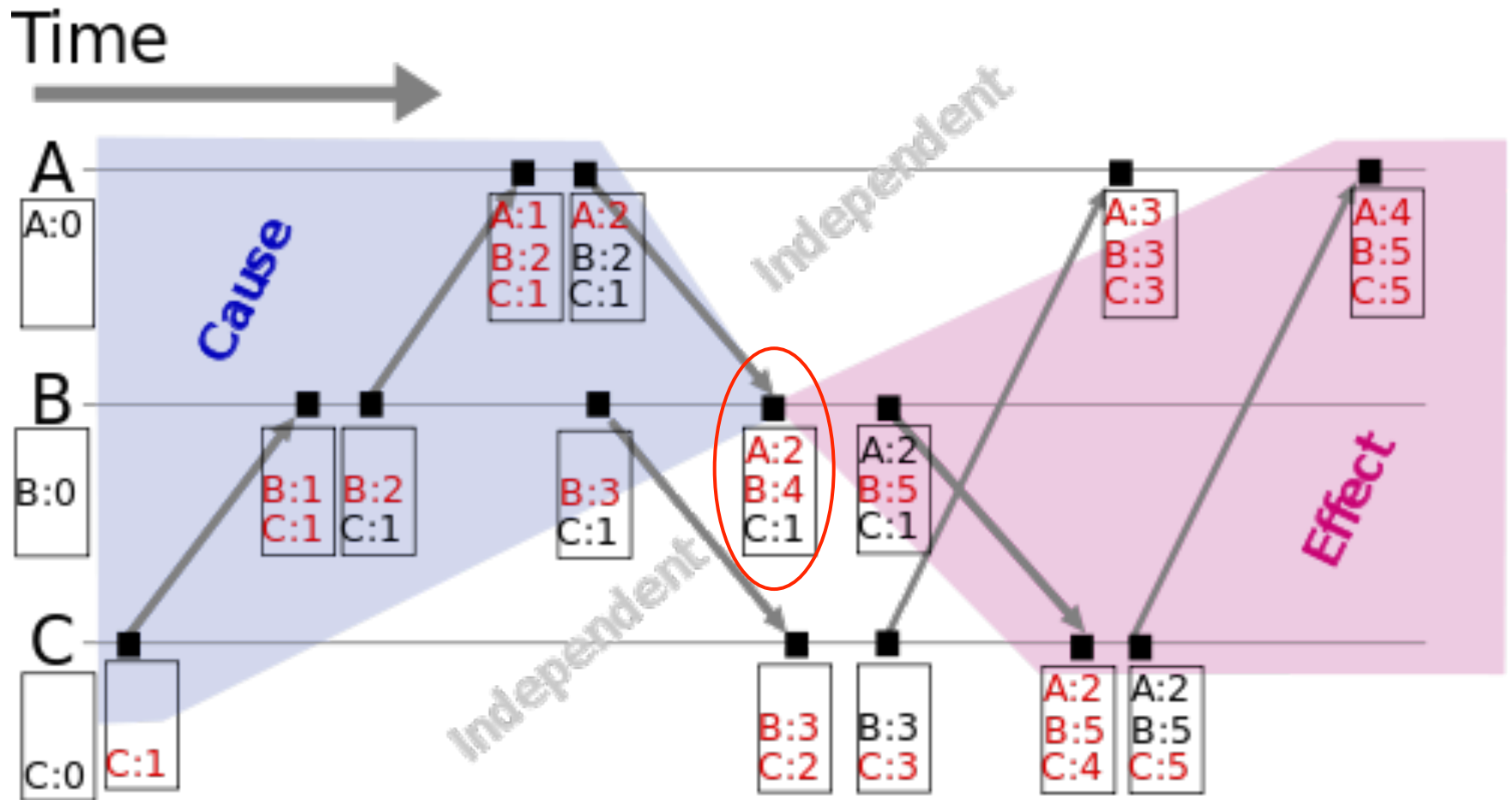
# Data Versioning

- 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 for tracking the *partial ordering* of events occurring in different nodes (processes) within a distributed system.
- Each time a node (process) experiences an internal event, it increments its own logical clock (counter) in the vector by one.
- Each time a node prepares to send out a message, it first increments its own logical counter in the vector by one before sending its entire vector along with the message being sent.
- Each time a node receives a message, it increments its own logical counter in the vector by one and then updates each element in its vector *by taking the maximum* of the value of its own vector clock and the value in the vector carried by the received message in an *element-by-element manner*.

# The Vector Clock

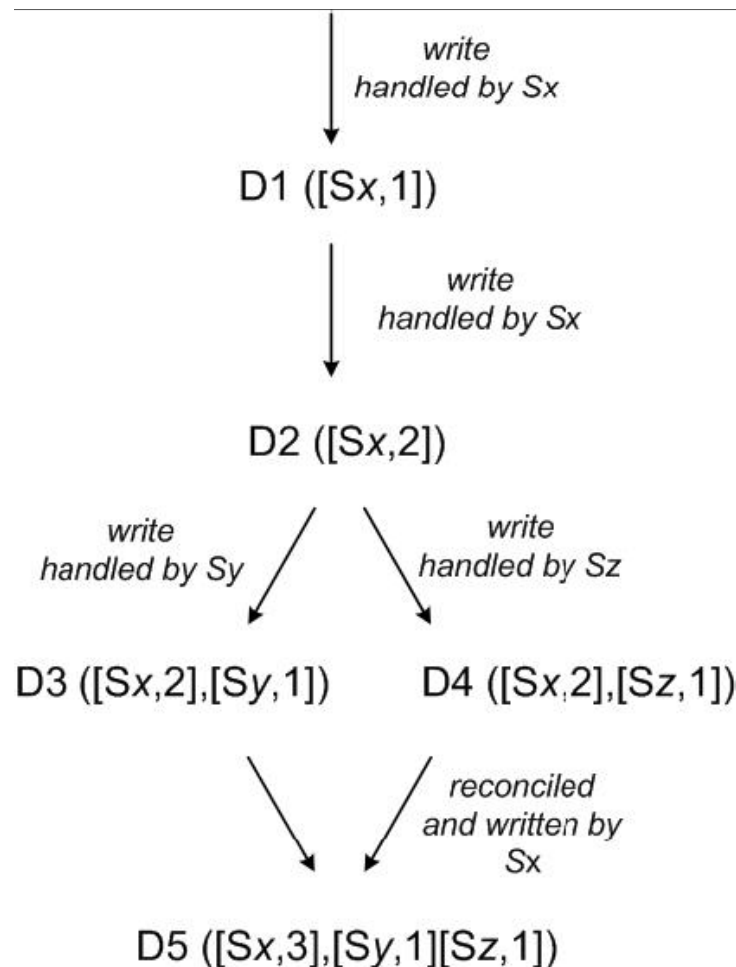


# Version tracking via Vector Clock

- When a node operates on an object, every version of the object will include a copy of the node's current vector clock values.
- *If the counters on the first object's clock are less-than-or-equal to (and not identical to) all of the nodes in a second object's vector clock counters, then the first object is an ancestor of the second one and can be forgotten.*



# An Example of Version tracking using Vector Clock



# Vector Clock Example

- A client writes D1 at server SX:
  - D1 ([SX,1])
- Another client reads D1, writes back D2; also handled by SX:
  - D2 ([SX,2]) (D1 garbage collected)
- Another client reads D2, writes back D3; handled by server SY:
  - D3 ([SX,2], [SY,1])
- Another client reads D2, writes back D4; handled by server SZ:
  - D4 ([SX,2], [SZ,1])
- Another client reads D3, D4: **CONFLICT !**

# Data Versioning

- Updates generate a new timestamp (Vector Clock)
- Eventual consistency
  - Multiple versions of the same object might co-exist
- Syntactic Reconciliation
  - System might be able to resolve conflicts automatically  
e.g. Dynamo enforces last-writer-wins
- Semantic Reconciliation
  - Conflict resolution pushed to application  
e.g., merge conflicting shopping carts

# Quantifying divergent versions (inconsistency)

- In a 24 hour trace
  - 99.94% of requests saw exactly one version
  - 0.00057% received 2 versions
  - 0.00047% received 3 versions
  - 0.00009% received 4 versions
- Experience showed that diversion came usually from concurrent writers due to automated client programs (robots), not humans

# “Quorum-likeness”

- `get()` & `put()` driven by two parameters:
  - R: the minimum number of replicas to read
  - W: the minimum number of replicas to write
- $R + W > N$  yields a “quorum-like” system
- Latency is dictated by the slowest R (or W) replicas
- **Sloppy quorum** to tolerate failures
  - Replicas can be stored on healthy nodes downstream in the ring, with metadata specifying that the replica should be sent to the intended recipient later

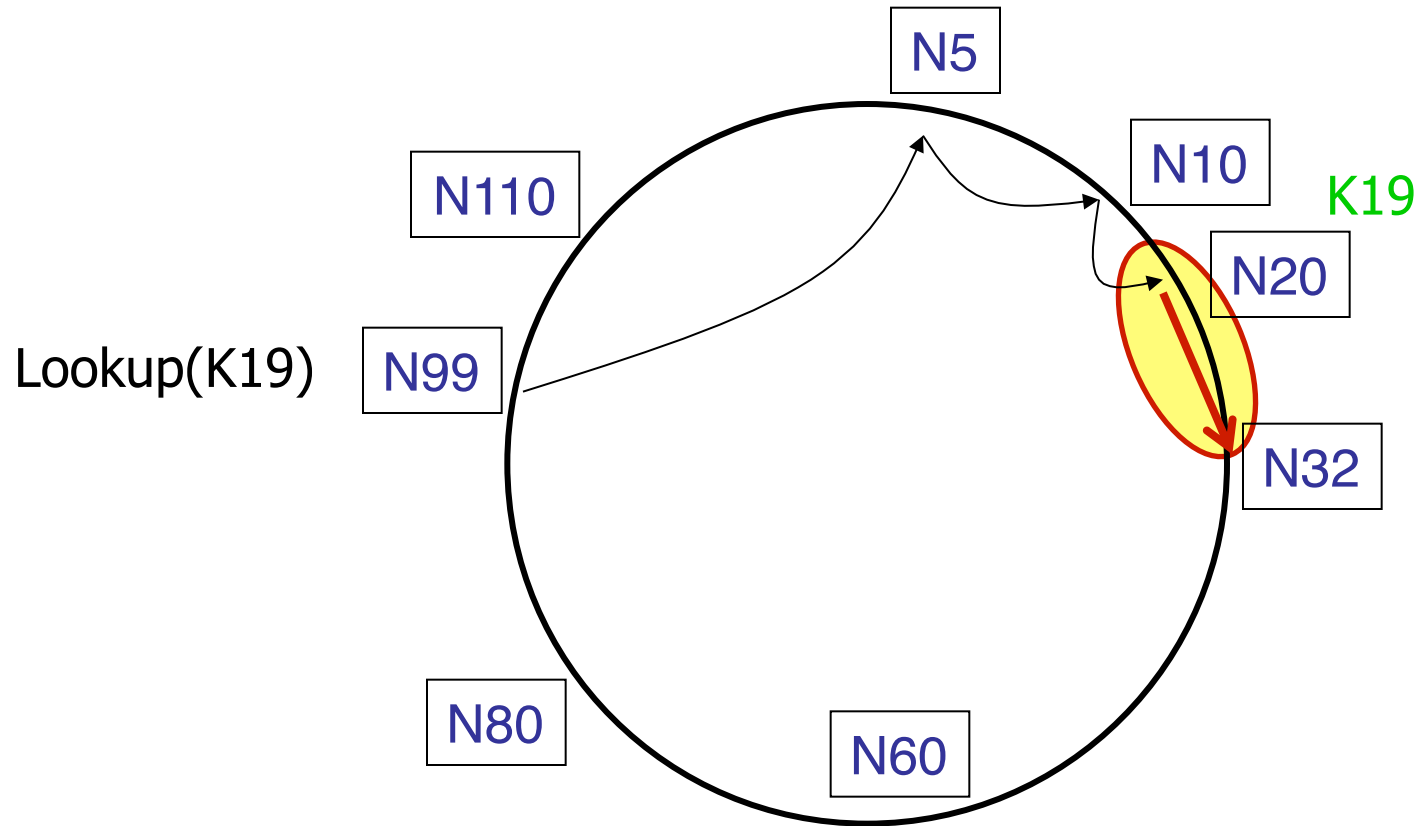
# The Challenge in Handling Temp. Failures

- Amazon quickly had their version of Chord up and running, but then encountered a problem
- Chord isn't very "delay tolerant"
  - So if a component gets slow or overloaded, Chord was very impacted
  - Yet delays are common in the cloud (not just due to failures, although failure is one reason for problems)
- Team asked: how can Dynamo tolerate delay?

## Idea they had

- Key issue is to find the node on which to store a key-value tuple, or one that has the value
- Routing can tolerate delay fairly easily
  - Suppose node N99 wants to use the finger to node N20 and gets no acknowledgement
  - Then Dynamo just tries again with node N32
  - This works at the “cost” of slight stretch in the routing path in the rare cases when it occurs

# Dynamo example: picture



- When N20 is temporarily down or unreachable during a write, send replica to N32.
- N32 is hinted that the replica is belong to N20 and it will deliver to N20 when N20 is recovered.

RESULTS: Dynamo is an **"always writeable"** data store ;  
Pushes conflict resolution to reads



## What if the actual “home” node fails?

- Suppose that we reach the point at which the next hop should take us to the owner for the hashed key
- But the target doesn't respond
  - It may have crashed, or have a scheduling problem (overloaded), or be suffering some kind of burst of network loss
  - All common issues in Amazon's data centers
- Then they do the Get/Put on the *next node that actually responds* even if this is the “wrong” one!

# Dynamo example in pictures

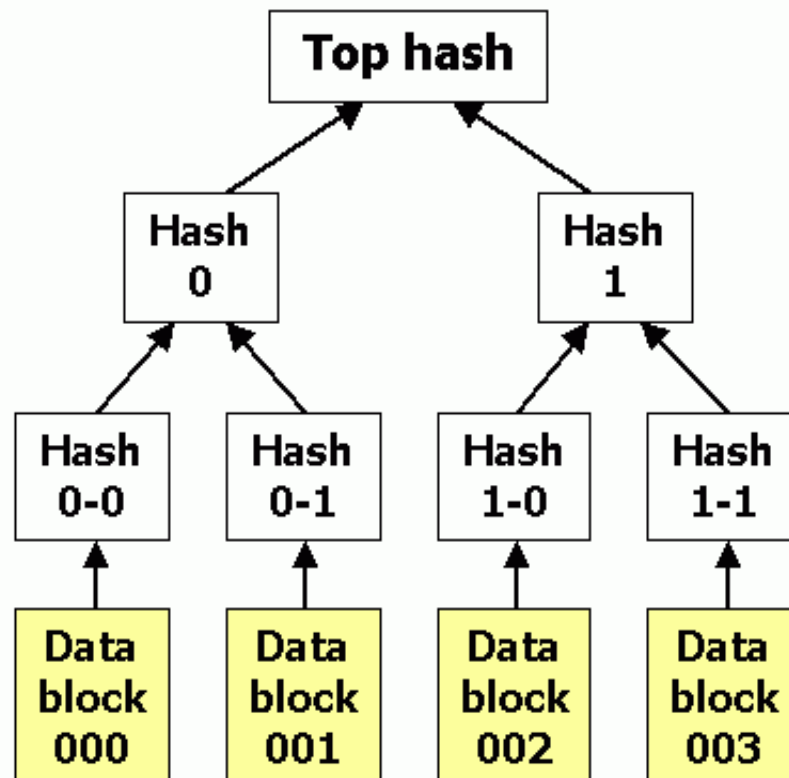
- Notice: Ideally, this strategy works perfectly
  - Recall that Chord normally replicates a key-value pair on a few nodes, so we would expect to see several nodes that “know” the current mapping: a shard
  - After the intended target recovers the repair code will bring it back up to date by copying key-value tuples
- But sometimes Dynamo jumps beyond the target “range” and ends up in the wrong shard

# Consequences?

- If this happens, Dynamo will eventually repair itself
  - ... But meanwhile, some slightly confusing things happen
- Put might succeed, yet a Get might fail on the key
- Could cause user to “buy” the same item twice
  - This is a risk they are willing to take because the event is rare and the problem can usually be corrected before products are shipped in duplicate

# Handling Non-Transient Failures

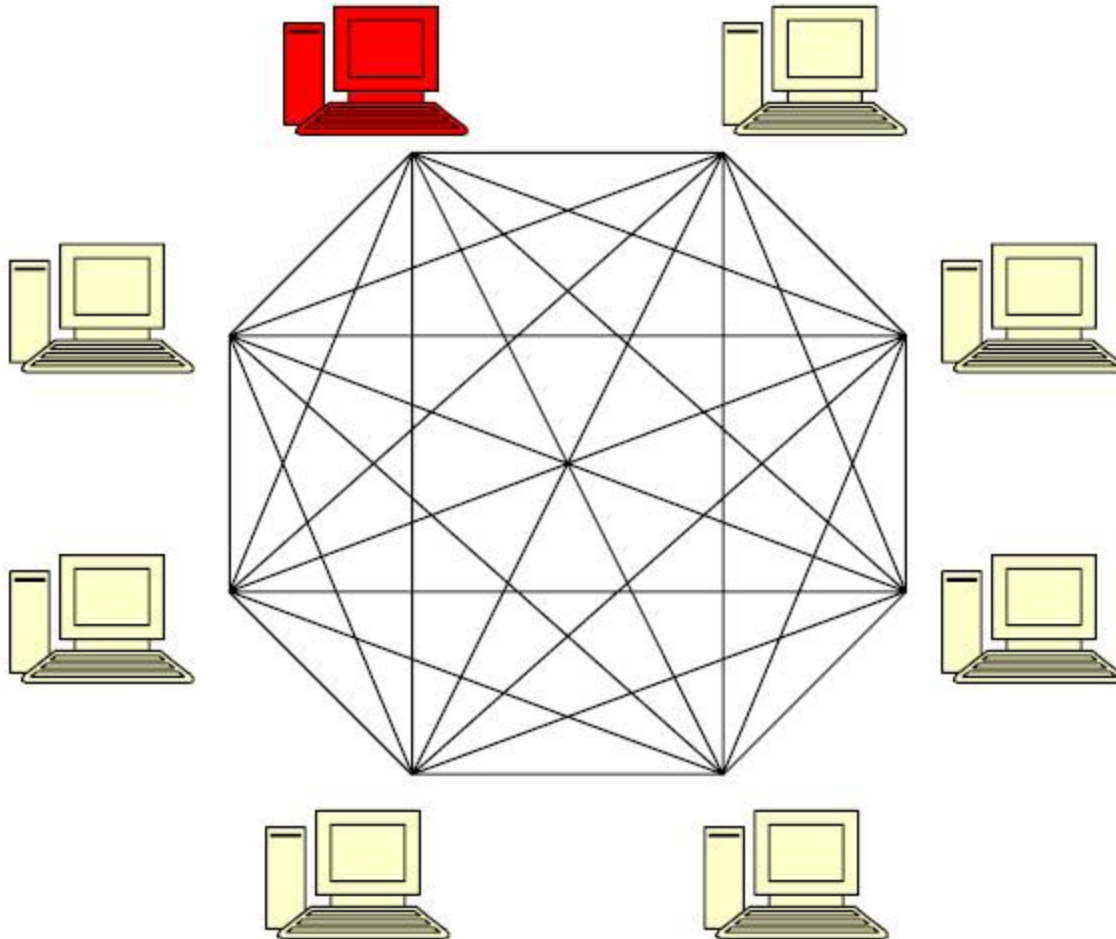
- Permanent failures: Replica Synchronization
  - Synchronize with another node
- Use **Merkle Trees** to speed-up detection of inconsistencies between data stored by replicas
  - Anti-Entropy operations: actively compared the content of different replicas and update all copies to the latest version



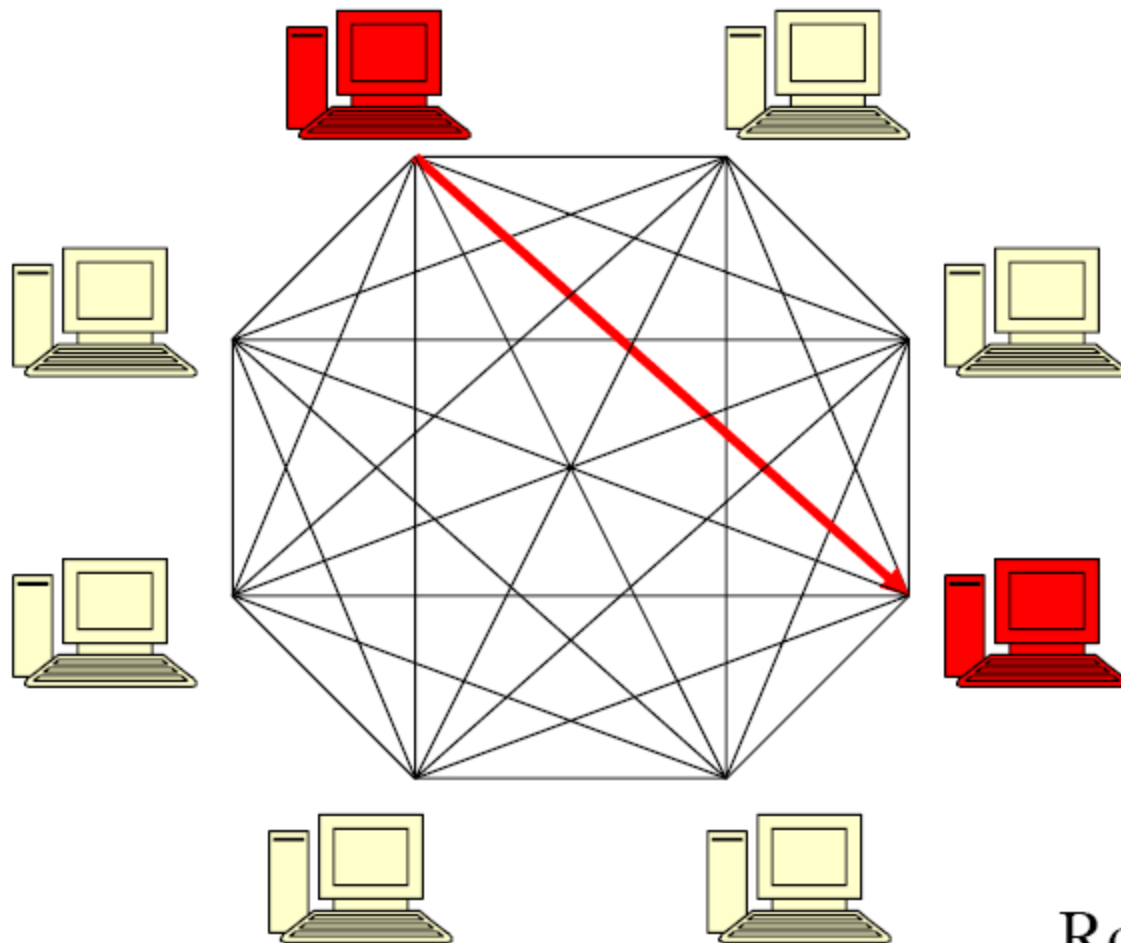
# Cluster Membership & Failure Detection

- Ring Membership
  - Use background gossip to build 1-hop DHT
  - Use external entity to bootstrap the system to avoid partitioned rings
- Failure Detection
  - Use standard gossip, heartbeats, and timeouts to implement failure detection
- System state disseminated via Gossiping in  $O(\log N)$  rounds where  $N = \#$  of nodes in the cluster.
  - Every  $T$  seconds each member increments its heartbeat counter and selects one other member to send its list to.
  - A member merges the list with its own list .

# Gossip

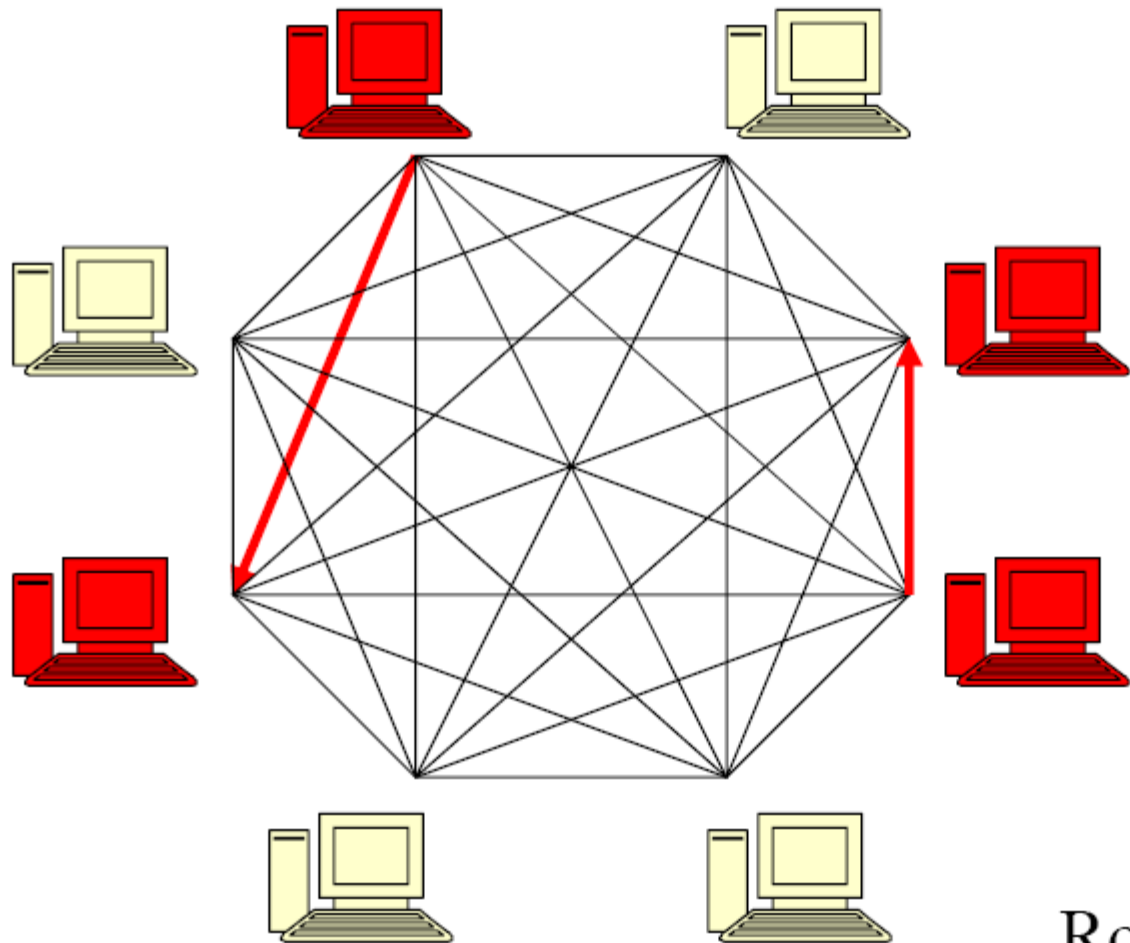


# Gossip



Round 1: 2

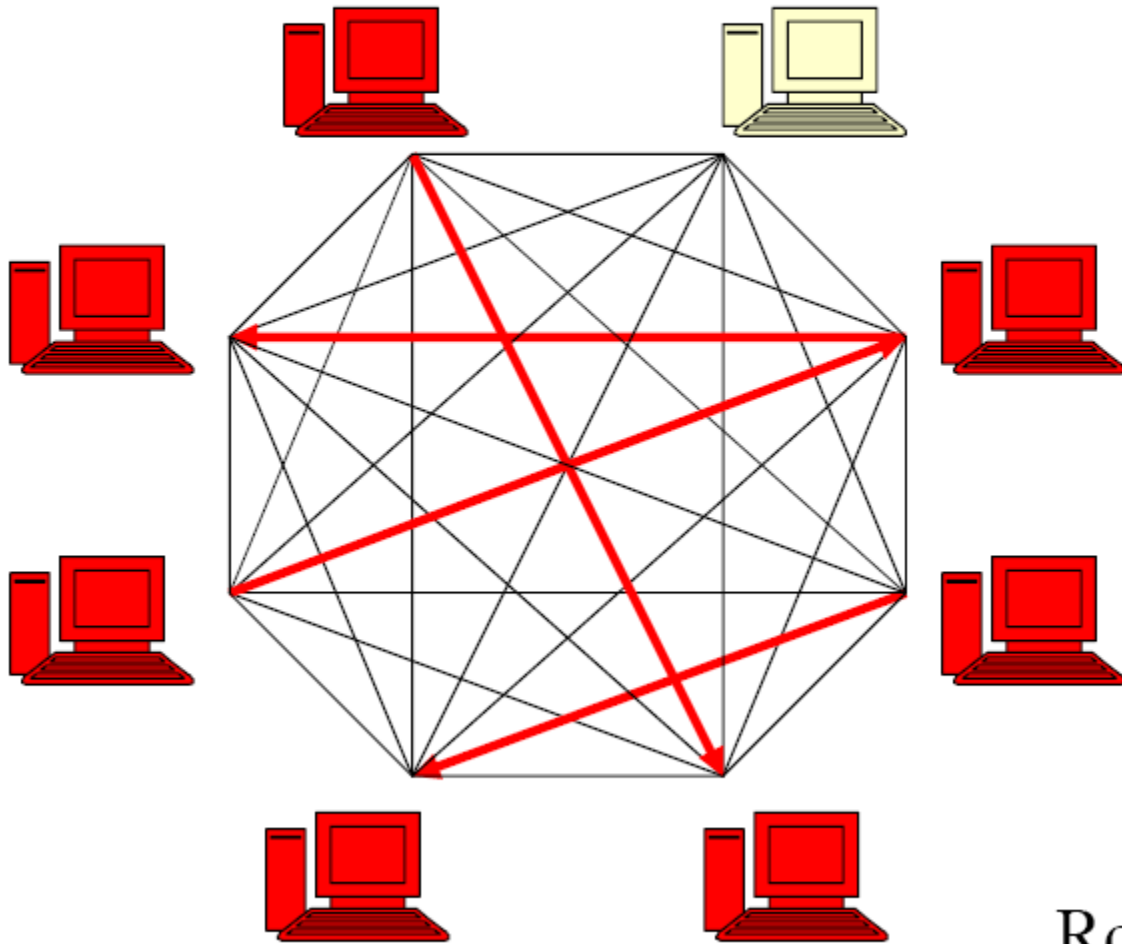
# Gossip



## Round 2: 4



# Gossip



Round 3: 7

# Implementation

- Persistent store either Berkeley DB Transactional Data Store, BDB Java Edition, MySQL, or in-memory buffer w/ persistent backend
- All in Java!
- Common N, R, W setting is (3, 2, 2)
- Results are from several hundred nodes configured as (3, 2, 2)
  - Not clear whether they run in a single datacenter...

# 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. |

# Werner Vogels on BASE

- He argues that delays as small as 100ms have a measurable impact on Amazon's income!
  - People wander off before making purchases
  - So snappy response is king
- True, Dynamo has weak consistency and may incur some delay to achieve consistency
  - There isn't any real delay "bound"
  - But they can hide most of the resulting errors by making sure that applications which use Dynamo don't make unreasonable assumptions about how Dynamo will behave

# BigTable and HBase (CP)

# Why Bigtable ?

- Performance of RDBMS system is good for transaction processing but for very large scale analytic processing, the solutions are commercial, expensive, and specialized.
- Very large scale analytic processing
  - Big queries – typically range or table scans.
  - Big databases (100s of TB)

## Why Bigtable ? (2)


- MapReduce on Bigtable/HBase with optionally Cascading on top to support some relational algebras may be a cost effective solution.
- Sharding is not a solution to scale open source RDBMS platforms
  - Application specific
  - Labor intensive (re)partitioning

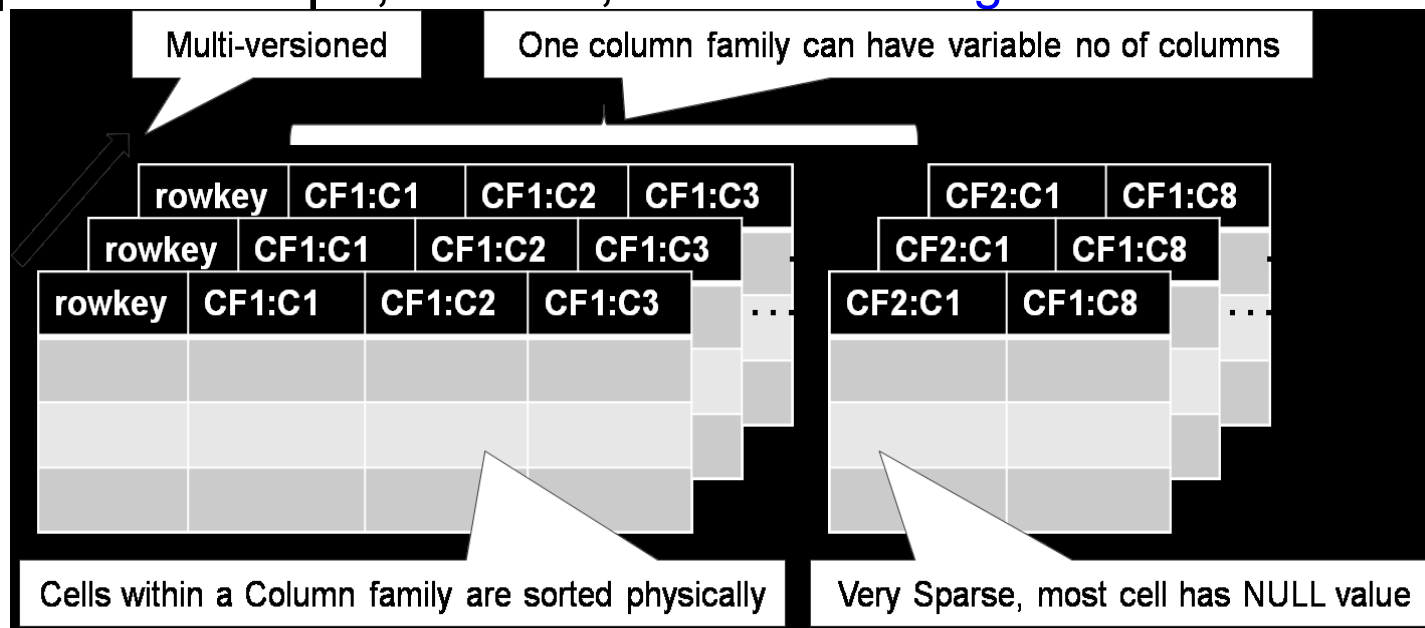
# Why BigTable/HBase ?

- If You need **Random Write/Read or both** on a large amount of data (remember GFS/HDFS not good with random access)
- Distributed storage
- Table-like in data structure
  - multi-dimensional map
- High scalability
- High availability
- High performance (1000's of operations/sec on Multiple TB of Data)



# Data Model

- Data is divided into various Tables
- Table is composed of Columns, Columns are grouped into Column Families (CF)
- A Table is a sparse, distributed, persistent multidimensional sorted Map
- Map indexed by a **Row key**, Column key, and a Timestamp
  - (row:string, column:string, time:int64)  uninterpreted byte array
- Supports lookups, inserts, deletes - **Single row transactions only**



# Rows and Columns

- Rows maintained in sorted lexicographic order
  - Applications can exploit this property for efficient row scans
  - Row ranges dynamically partitioned into tablets
- Columns grouped into column families
  - Column key = *family:qualifier*
  - Column families provide locality hints
  - Unbounded number of columns

# Bigtable Building Blocks

- GFS
  - Provide raw, fault-tolerant storage for log and data files
- Chubby
  - A distributed lock manager
  - Based on the Paxos algorithm to keep replicas to be consistent even in the presence of failures
- SSTable
  - A customized file format for storing Big Table data

# Chubby

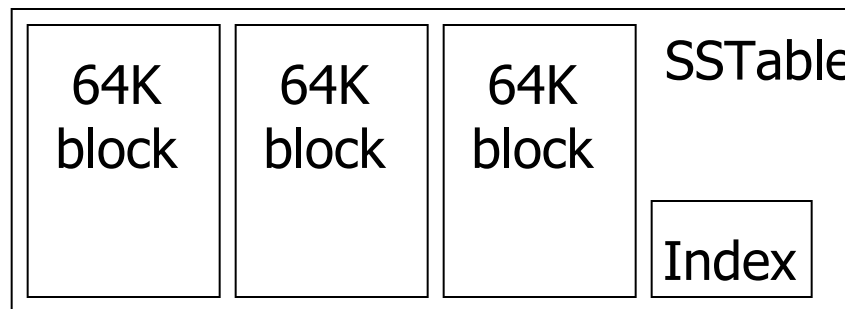
- Provide distributed lock service
- Five active replicas, among them, one is elected as the master and serve the service requests
- Run Paxos algorithm to provide consistency even under replica failures
- Each directory or file is used as a lock to support atomic Reads and Writes from/to a file
- Each Chubby client maintains a session with the Chubby service
  - When the session of the client expires, it loses any locks and open handles

## Use of Chubby:

- Store Bootstrap location
- Discover Tablet servers
- Store BigTable Schema information
- Store access control list

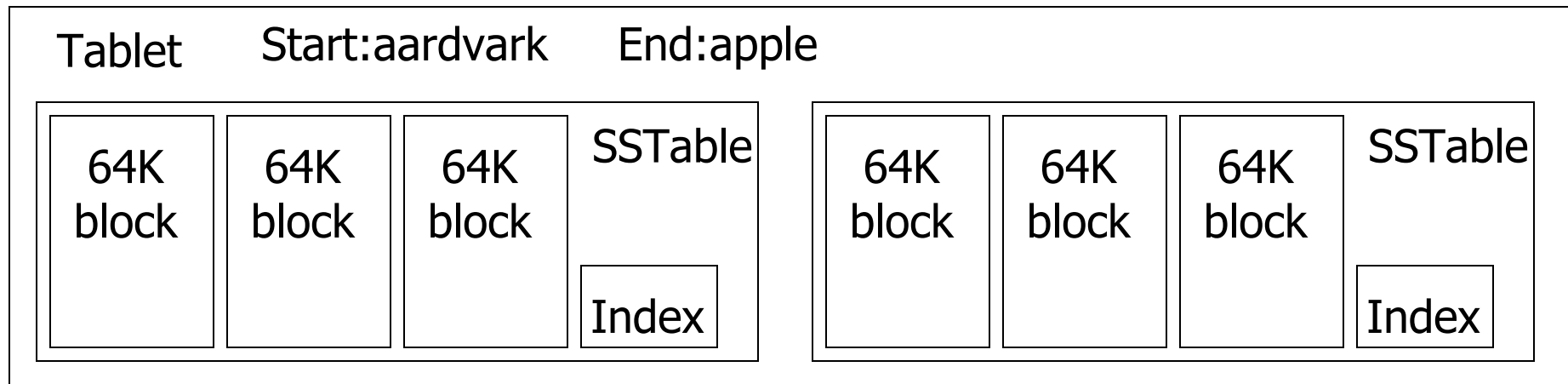
# SSTable

- Basic building block to hold data in Bigtable
- Persistent, ordered immutable map from keys to values
  - Stored in GFS
- Sequence of blocks on disk plus an index for block lookup
  - Can be completely mapped into memory
- Supported operations:
  - Look up value associated with key
  - Iterate key/value pairs within a key range



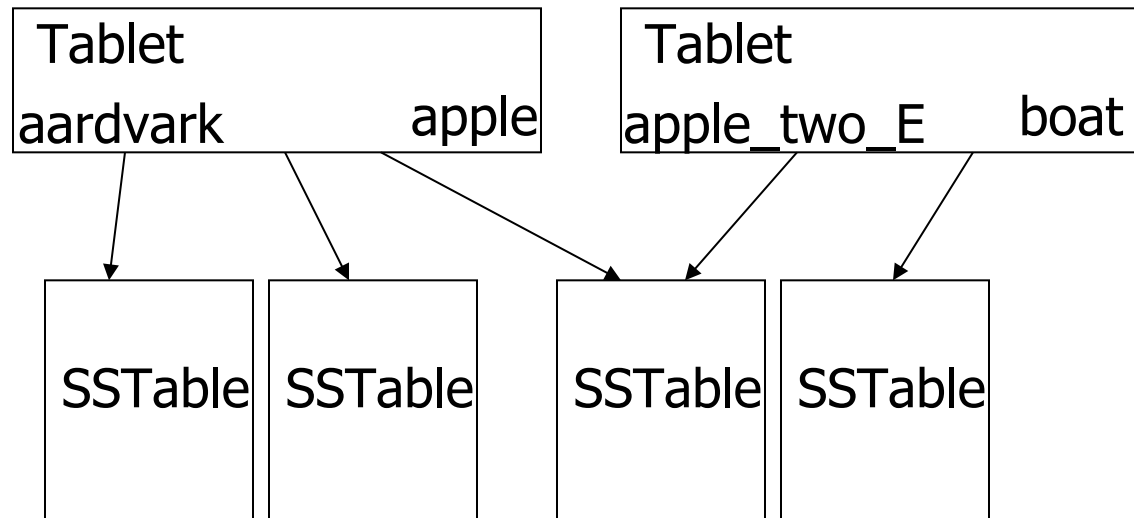
# Tablet

- Dynamically partitioned range of rows
- Built from multiple SSTables
- Data distribution and Load Balancing are performed at the granularity of Tablet



# Table

- Multiple tablets make up the table
- SSTables can be shared



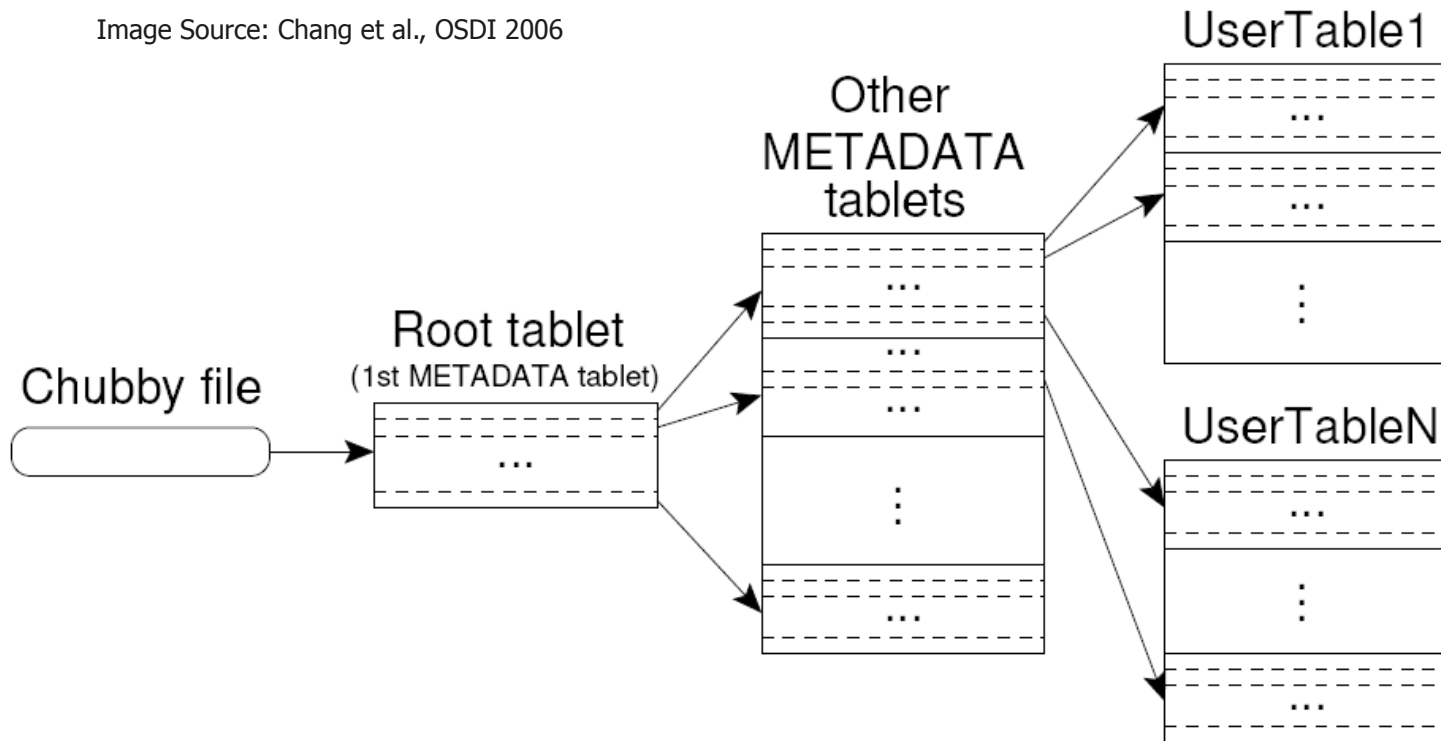
# System Architecture of Big Table

- Client library
- Single Master server
  - Assigns tablets to tablet servers
  - Detects addition and expiration of tablet servers
  - Balances tablet server load
  - Handles garbage collection
  - Handles schema changes
- Tablet servers
  - Each tablet server manages a set of tablets
    - Typically between ten to a thousand tablets
    - Each 100-200 MB by default
  - Handles read and write requests to the tablets
  - Splits tablets that have grown too large



# Tablet Location

Image Source: Chang et al., OSDI 2006

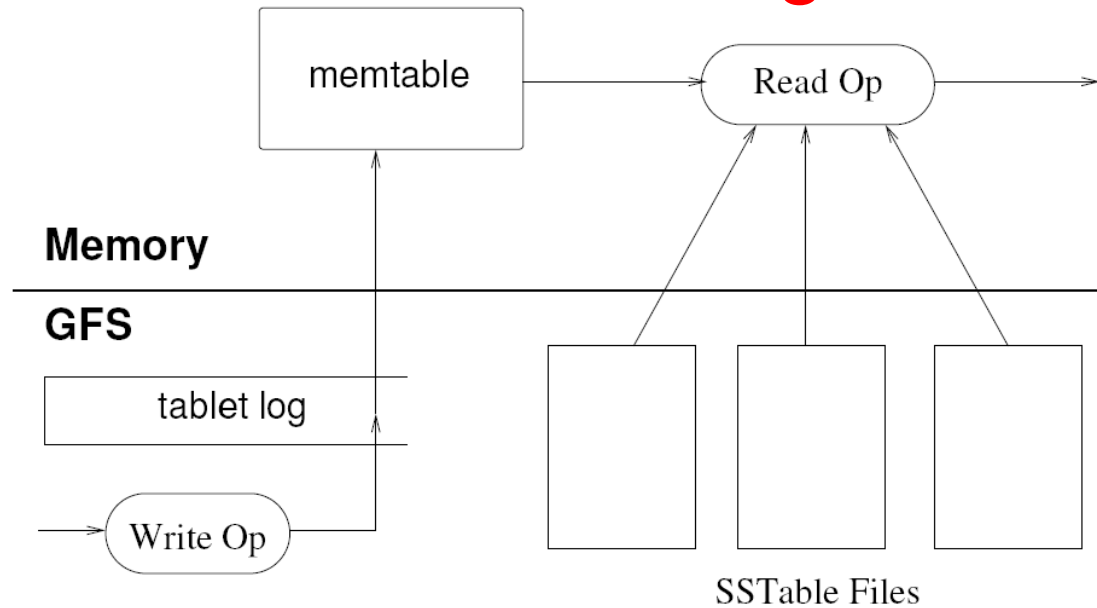


- 1<sup>st</sup> level: Root Tablet contains location of all tablets
- 2<sup>nd</sup> level: Metadata Tablet contain locations of user tablets
- 3<sup>rd</sup> level: User Tablets storing the actual user data

# Tablet Assignment

- Master keeps track of:
  - Set of live tablet servers
  - Assignment of tablets to tablet servers
  - Unassigned tablets
- Each tablet is assigned to one tablet server at a time
  - Tablet server maintains an exclusive lock on a file in Chubby
  - Master monitors tablet servers and handles assignment
- Changes to tablet structure
  - Table creation/deletion (master initiated)
  - Tablet merging (master initiated)
  - Tablet splitting (tablet server initiated)

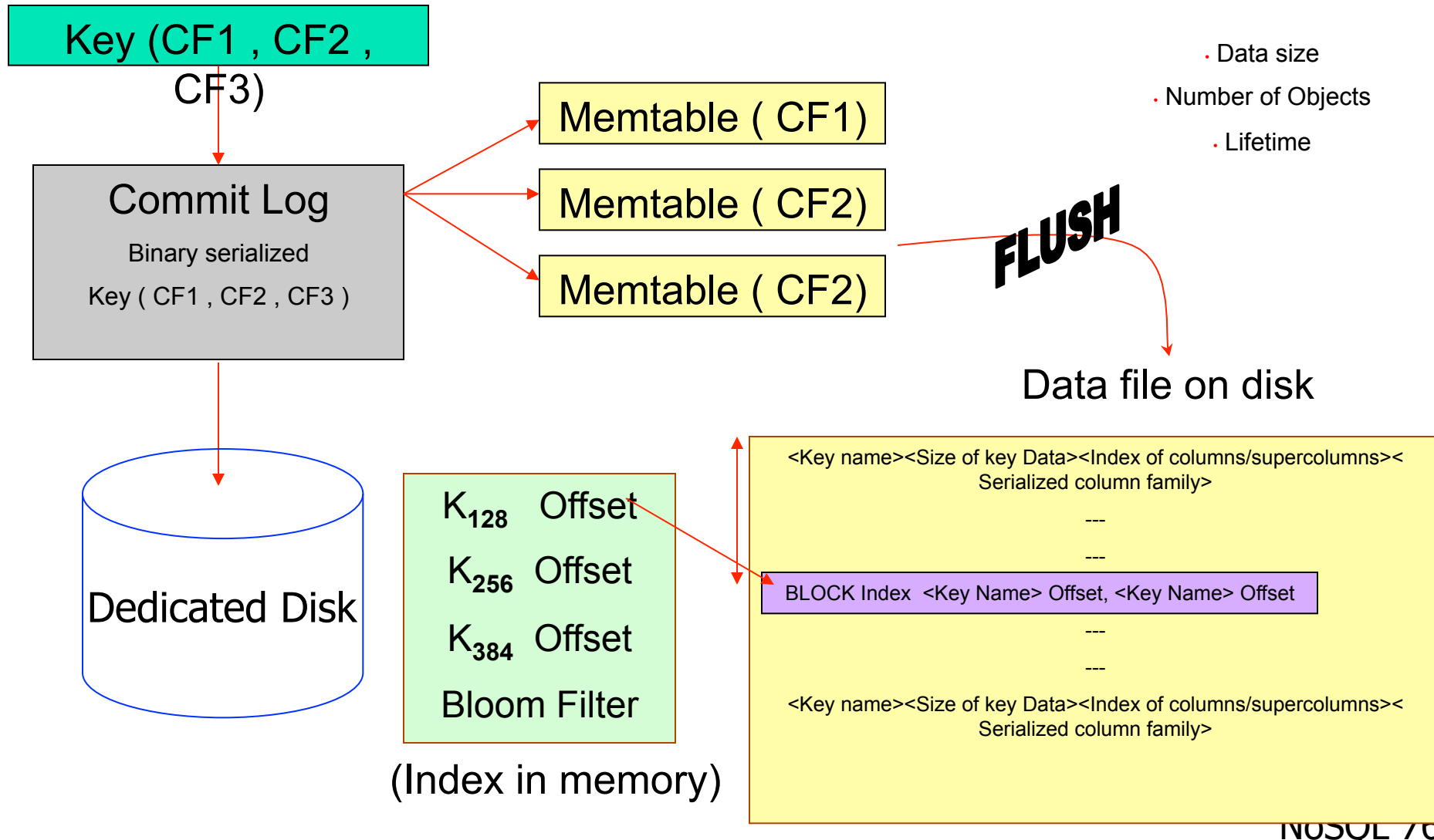
# Tablet Serving



Tablet representation

- Write Operation
  - Updates committed to a commit log
  - Recently committed updates are stored in memtable
  - Older updates are stored (flushed) to a sequence of SSTables
- Read Operation
  - Form a merged-view of SSTables and memtable
  - Read <key-value> pair

# Write cont'd



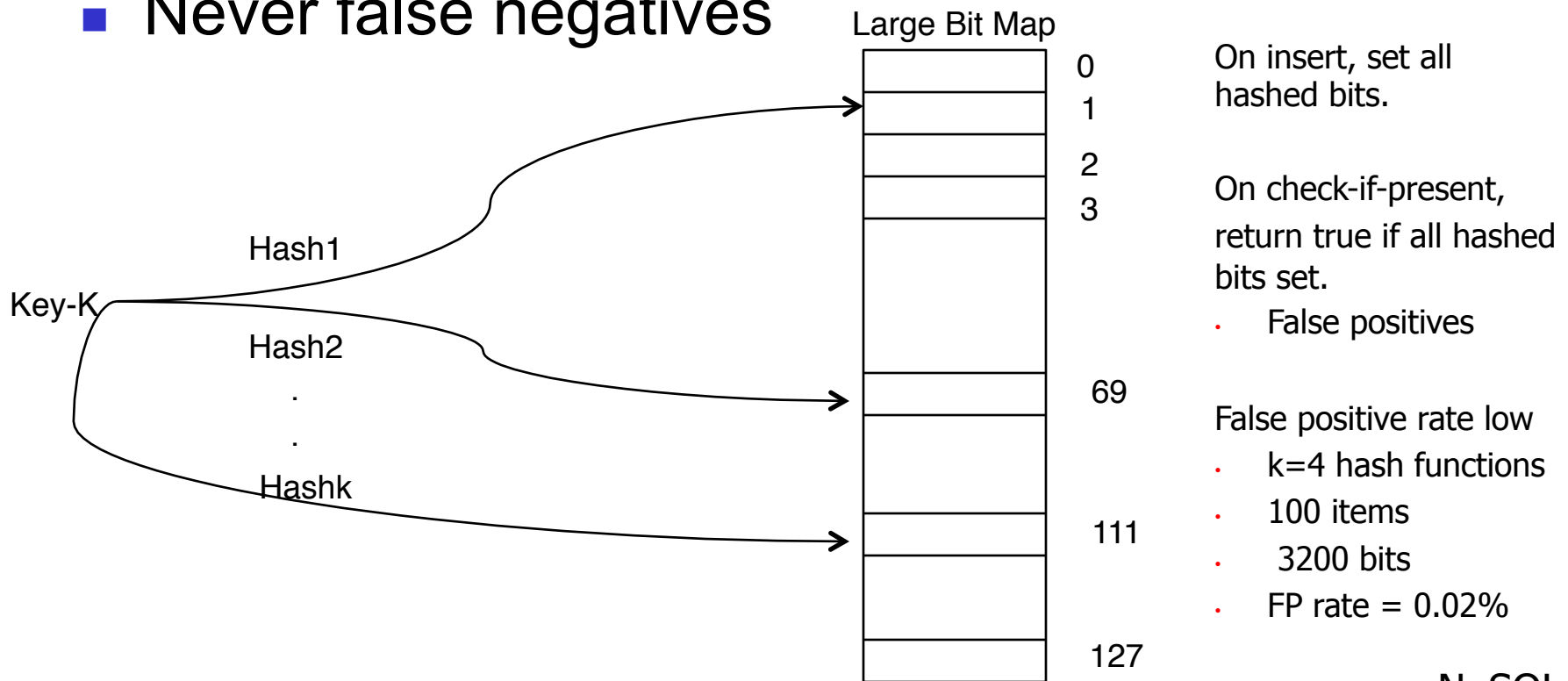
# Deletes and Reads

- Delete: don't delete item right away
  - Add a tombstone to the log
  - Compaction will eventually remove tombstone and delete item
- Read: Similar to writes, except
  - A row may be split across multiple SSTables

=> reads need to touch multiple SSTables => reads slower than writes (but still fast)

# 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

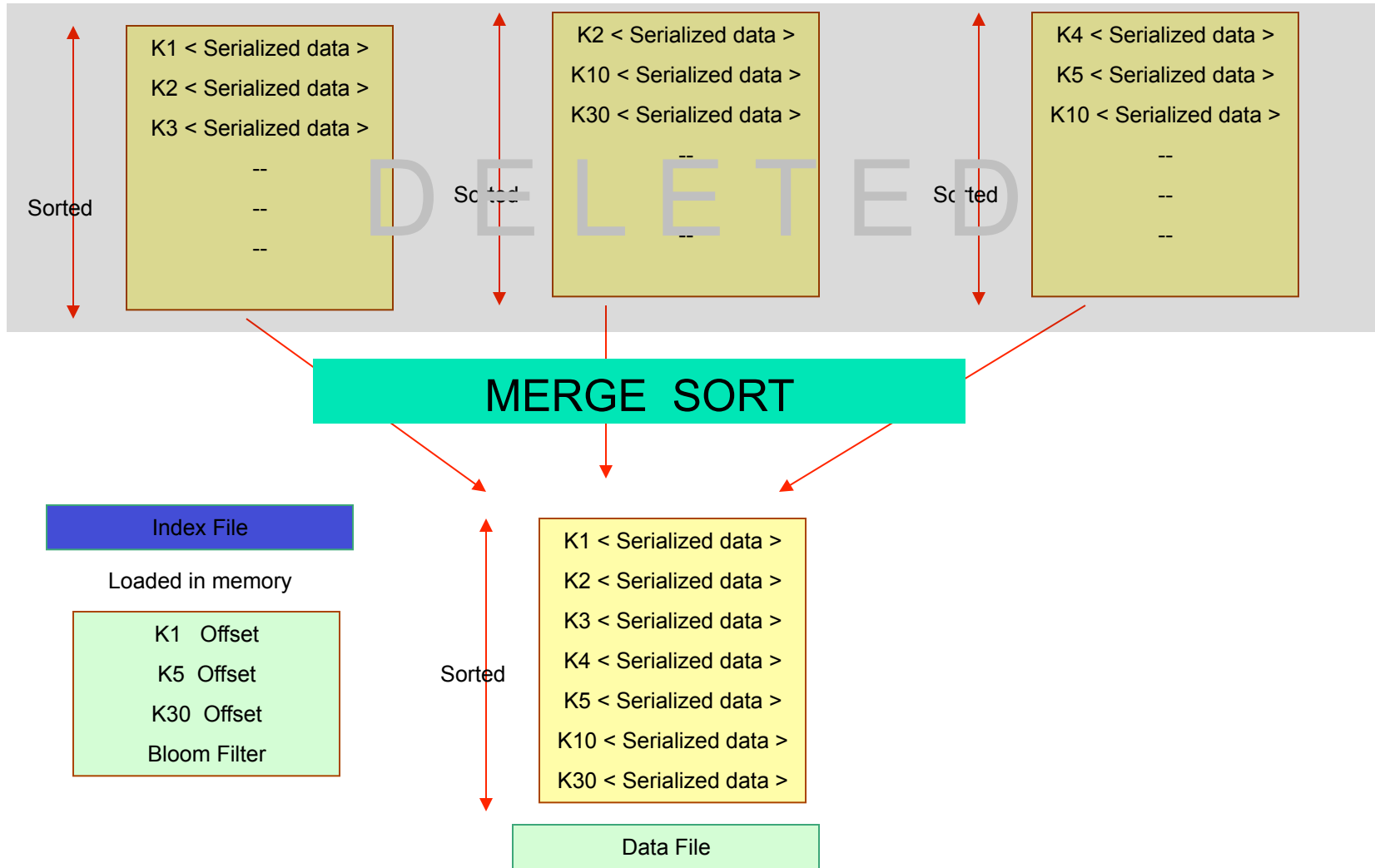


# Compactions

- Minor compaction
  - Converts the memtable into an SSTable
  - Reduces memory usage and log traffic on restart
- Merging compaction
  - Merging non-full SSTables created by Minor compaction
  - Not as thorough as Major compaction, e.g. does not clean-up deleted-records during the merge
- Major compaction
  - Can be triggered manually (via the HBase shell) or perform in the background periodically, e.g. every 24 hours
  - Reads the contents of a few SSTables and the memtable, and writes out a new SSTable
  - Reduces number of SSTables
  - Delete stale/excessive versions of a cell

Idea is to trade background writes to speedup subsequent Read operations

# Compactions





# Bigtable Applications

- Data source and data sink for MapReduce
- Google's web crawl
- Google Earth
- Google Analytics

# Lessons Learned

- Fault tolerance is hard
- Don't add functionality before understanding its use
  - Single-row transactions appear to be sufficient
- Keep it simple!

HBase is an **open-source, distributed,**  
database built on top of HDFS based on  
BigTable!

# HBase is ..

- A distributed data store that can scale horizontally to 1,000s of commodity servers and petabytes of indexed storage.
- Designed to operate on top of the Hadoop distributed file system (HDFS) or Kosmos File System (KFS, aka Cloudstore) for scalability, fault tolerance, and high availability.



# Backdrop

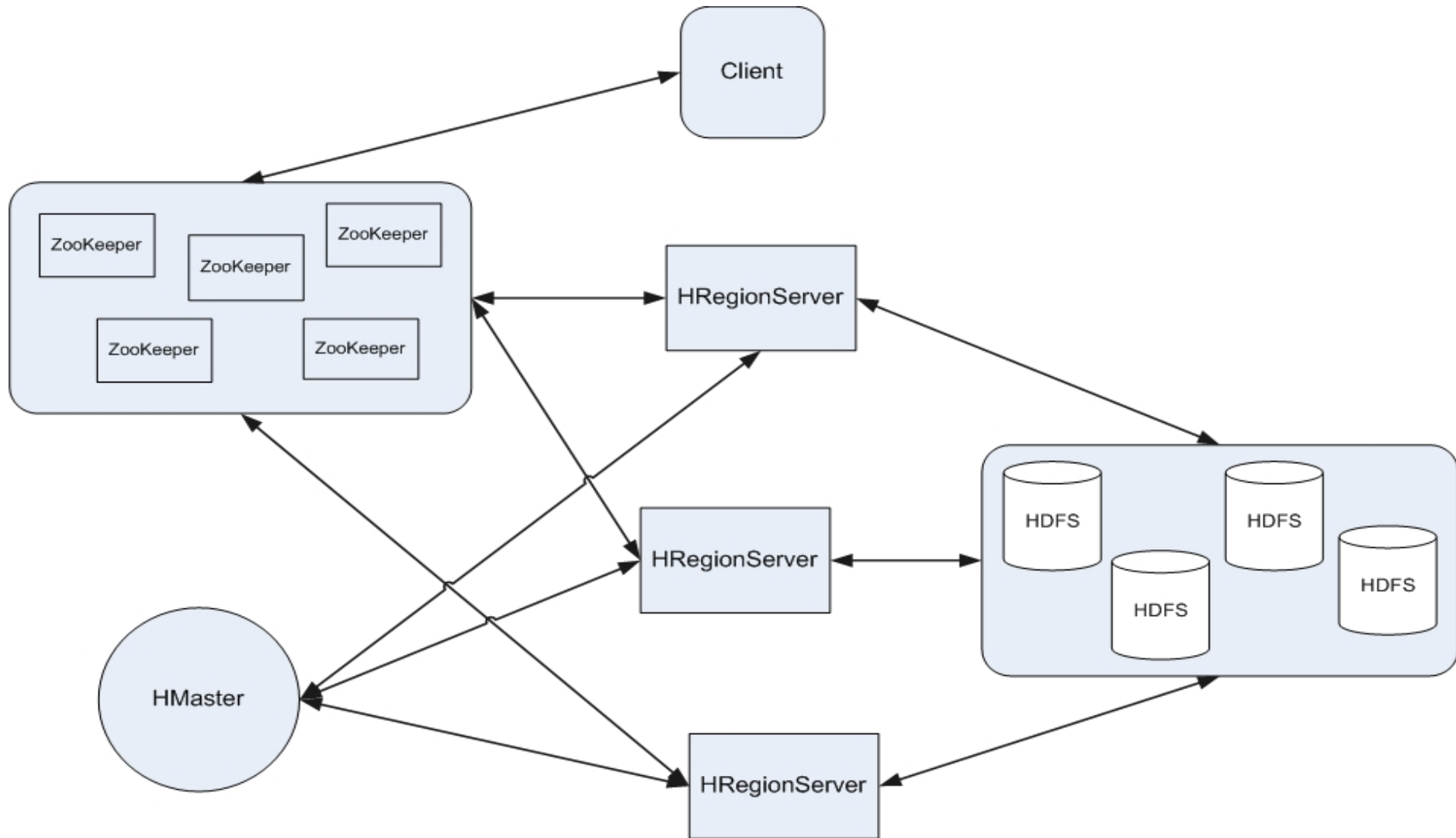
- Started toward by Chad Walters and Jim Kellerman
- 2006.11
  - Google releases paper on BigTable
- 2007.2
  - Initial HBase prototype created as Hadoop contrib.
- 2007.10
  - First useable HBase
- 2008.1
  - Hadoop become Apache top-level project and HBase becomes subproject
- 2008.10~
  - HBase 0.18, 0.19 released

# HBase Storage Model

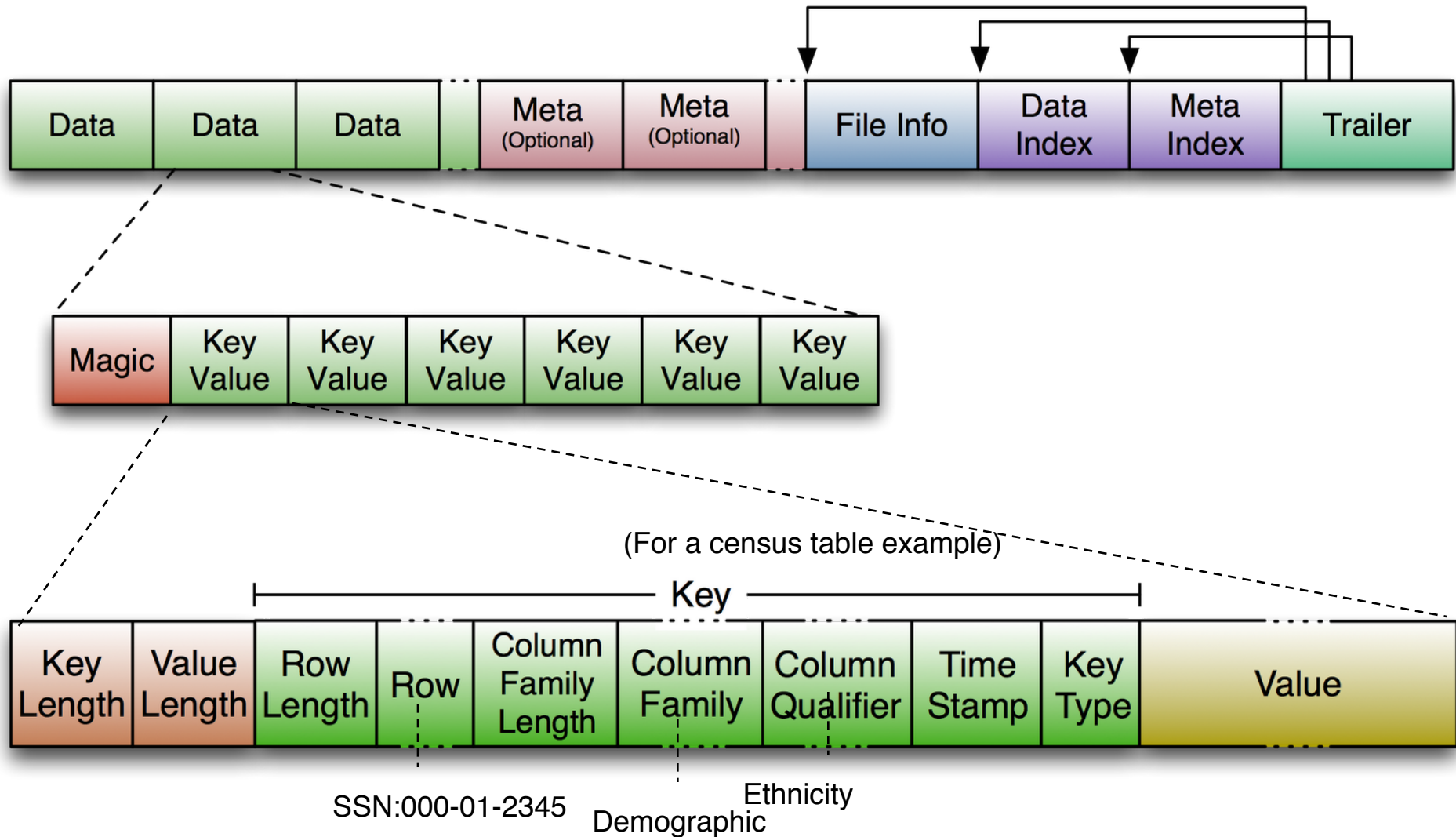
## (and different Terminologies vs. BigTable)

- Partitioning
  - A Table is horizontally partitioned into Regions, each region is composed of sequential range of keys
  - Each region is managed by a RegionServer
    - A RegionServer may hold multiple regions
- Persistence and Data availability
  - HBase stores its data in HDFS, it does NOT replicate RegionServers and relies on HDFS replication for Data Availability
  - Region data is cached in-memory
    - Updates and Reads are served from in-memory cache (Memstore)
    - MemStore is flushed periodically to HDFS
    - Write Ahead Log (stored in HDFS) is used for Durability of updates

# HBase Architecture



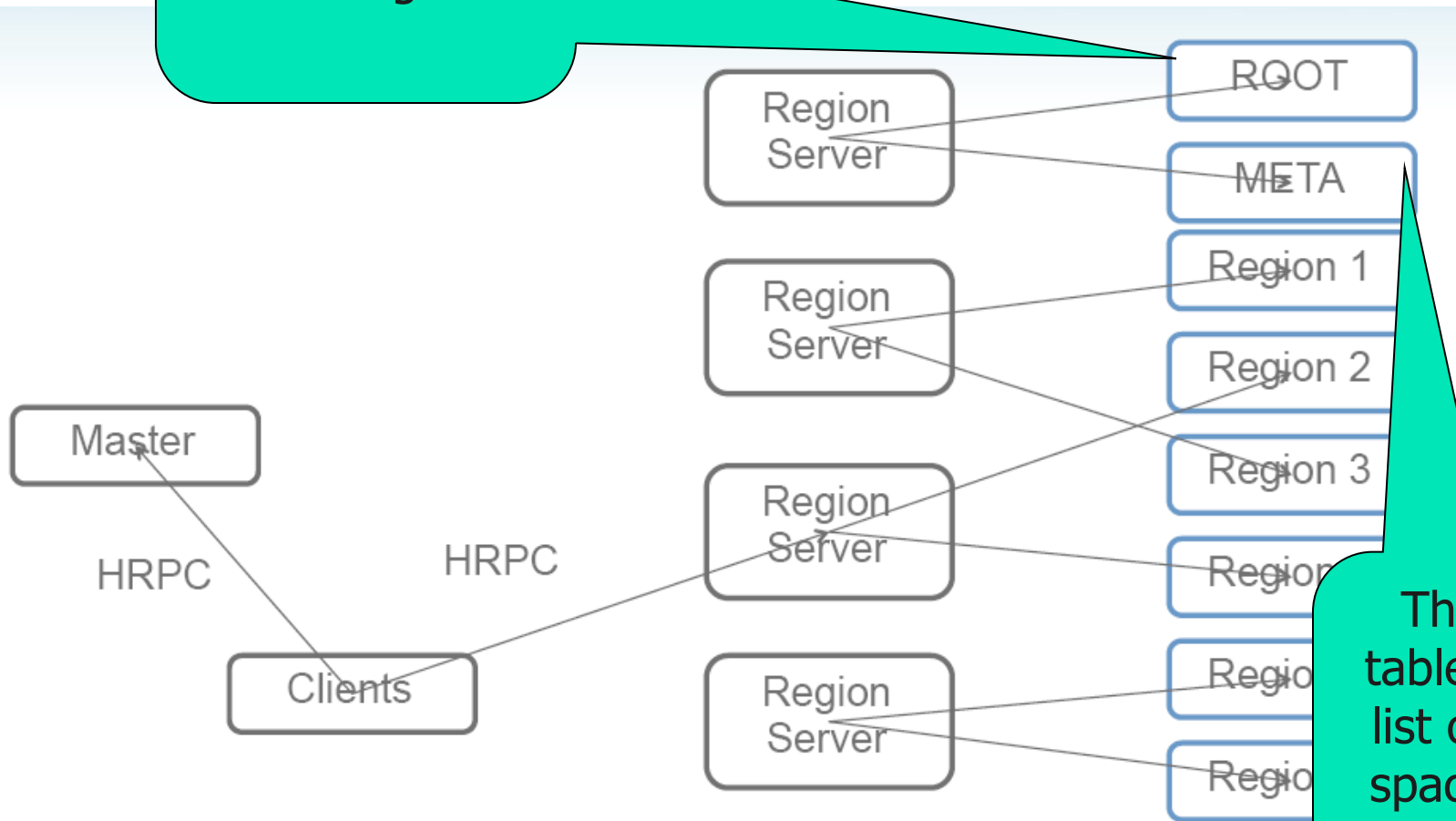
# Hfile of HBase = SSTable in BigTable





# Operation

The -ROOT-  
table holds the  
list of .META.  
table regions



The .META.  
table holds the  
list of all user-  
space regions.

# HBase API

## ■ API

- `get(row)`
- `put(row, Map<column,value>)`
- `scan(key range, filter)`
- `increment(row,columns)`
- Check and Put, Delete etc.

# HBase Shell

- HBase shell provides interactive commands for manipulating database
  - Create/Delete tables
  - Insert/Update/Read from tables
  - Manage Regions

# HBase Atomic Operations

- HBase provides single row atomic operations
  - CheckAndPut – Similar to test-and-set
  - CheckAndDelete
  - All row operations are atomic no matter how many columns are involved
- HBase also provides Row-level exclusive locks
  - One can use these locks to implement single row-level transactions

# HBase Characteristics/Features...

- Tables have one primary index, *the row key*.
- No join operators.
- Scans and queries can select a subset of available columns, perhaps by using a wildcard.
- 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 Characteristics/Features...(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, JPython etc.
  - **BUT this is also changing: as you can now use** the HiveQL to perform SQL-like queries on data stored in HBase tables.

## Why HBase (cont'd) ?

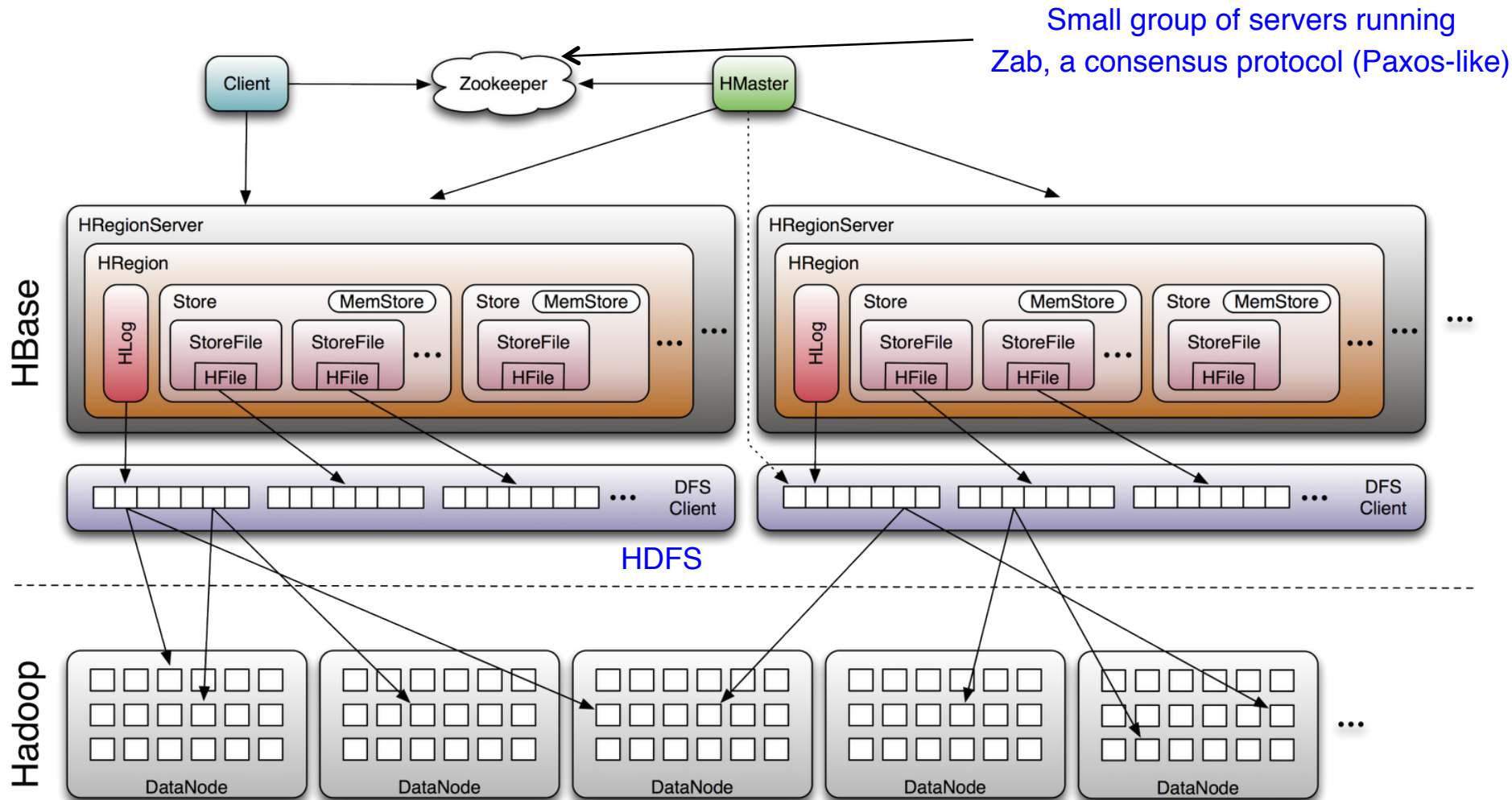
- HBase is a Bigtable clone.
- It is open source
- It has a good community and promise for the future
- It is developed on top of and has good integration for the Hadoop platform, if you are using Hadoop already.
- It has a Cascading connector.

# HBase benefits than RDBMS

- No real indexes
- Automatic partitioning
- Scale linearly and automatically with new nodes
- Commodity hardware
- Fault tolerance
- Batch processing



# HBase Architecture



Source: <http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html>

# Members

- *Master*

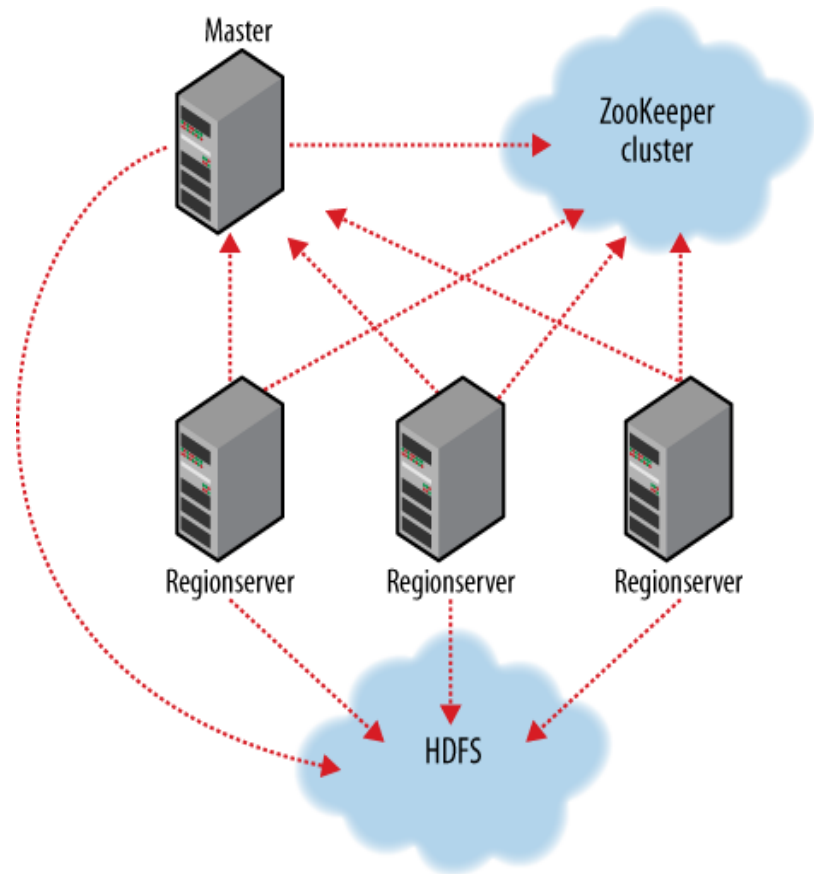
- Responsible for monitoring region servers
- Load balancing for regions
- Redirect client to correct region servers

- *Regionserver slaves*

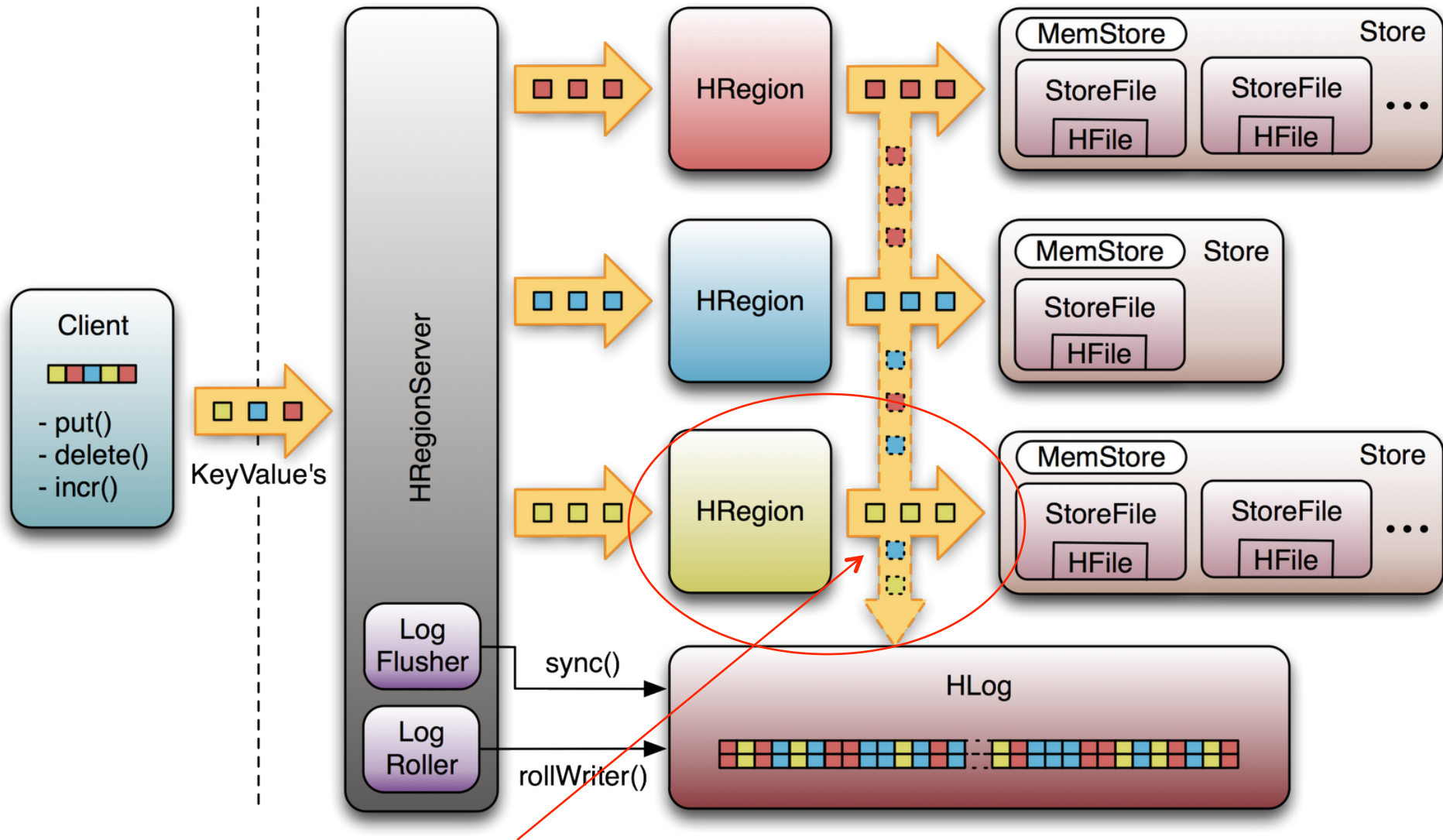
- Serving requests(Write/Read/Scan) of Client
- Send HeartBeat to Master
- Throughput and Region numbers are scalable by region servers

# ZooKeeper

- HBase depends on ZooKeeper and by default it manages a ZooKeeper instance as the authority on cluster state
- To manage master election and server availability



# Strong Consistency: HBase Write-Ahead Log

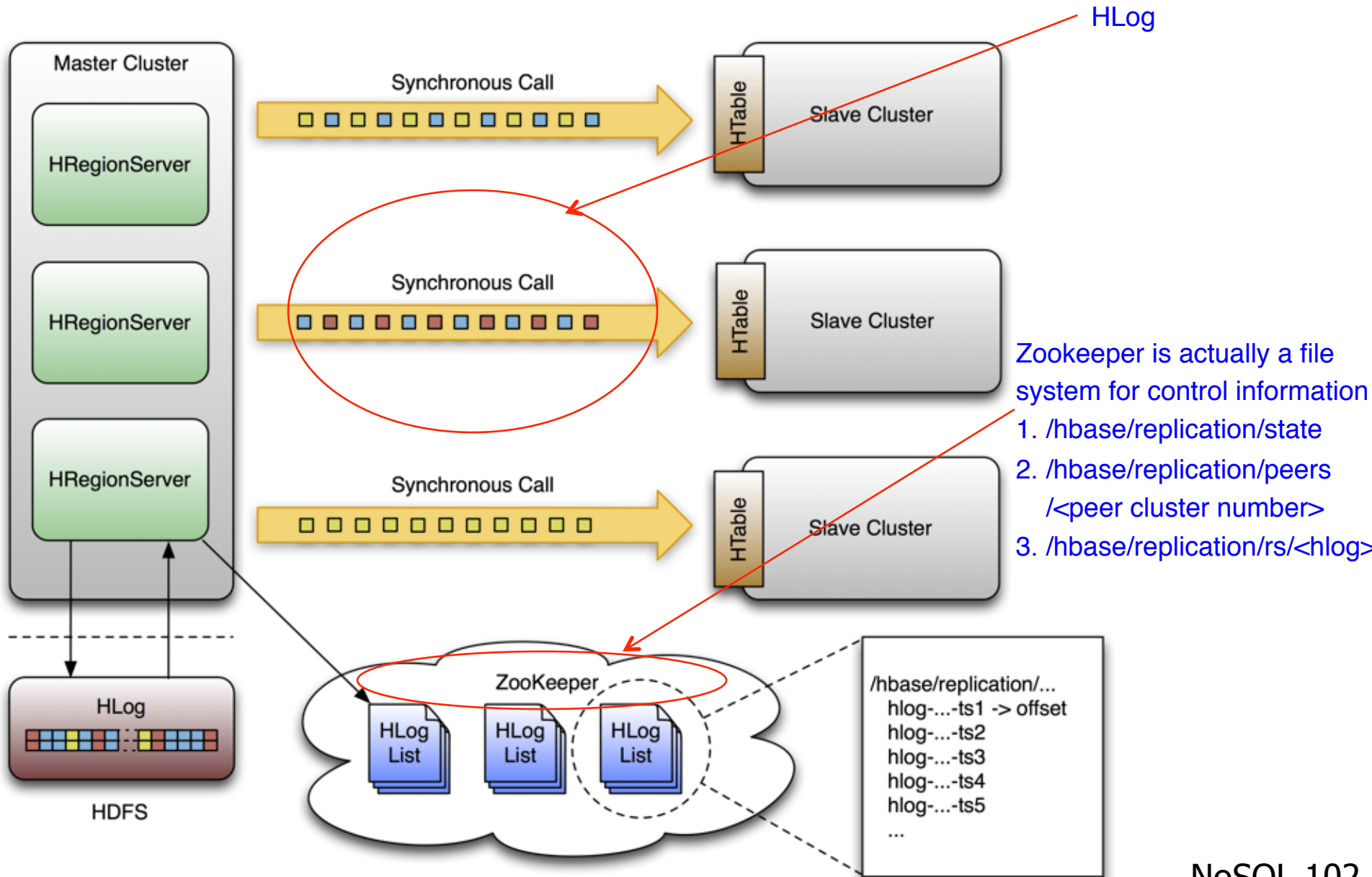


Write to HLog before writing to MemStore  
Thus can recover from failure

# Log Replay

- After recovery from failure, or upon bootup (HRegionServer/HMaster)
  - Replay any stale logs (use timestamps to find out where the database is w.r.t. the logs)
  - Replay: add edits to the MemStore
- Keeps one HLog per HRegionServer rather than per region
  - Avoids many concurrent writes, which on the local file system may involve many disk seeks

# Cross-data center replication



# Installation (1)

START Hadoop...

```
$ wget http://ftp.twaren.net/Unix/Web/apache/hadoop/  
hbase/hbase-0.20.2/hbase-0.20.2.tar.gz  
$ sudo tar -zxvf hbase-*.tar.gz -C /opt/  
$ sudo ln -sf /opt/hbase-0.20.2 /opt/hbase  
$ sudo chown -R $USER:$USER /opt/hbase  
$ sudo mkdir /var/hadoop/  
$ sudo chmod 777 /var/hadoop
```

# Setup (1)

```
$ vim /opt/hbase/conf/hbase-env.sh
    export JAVA_HOME=/usr/lib/jvm/java-6-sun
export HADOOP_CONF_DIR=/opt/hadoop/conf
export HBASE_HOME=/opt/hbase
export HBASE_LOG_DIR=/var/hadoop/hbase-logs
export HBASE_PID_DIR=/var/hadoop/hbase-pids
export HBASE_MANAGES_ZK=true
export HBASE_CLASSPATH=$HBASE_CLASSPATH:/opt/hadoop/
conf
```

```
$ cd /opt/hbase/conf
$ cp /opt/hadoop/conf/core-site.xml ./
$ cp /opt/hadoop/conf/hdfs-site.xml ./
$ cp /opt/hadoop/conf/mapred-site.xml ./
```



## Setup (2)

```
<configuration>
  <property>
    <name> name </name>
    <value> value </value>
  </property>
</configuration>
```

| Name                                    | value                                |
|---|--------------------------------------|
| hbase.rootdir                           | hdfs://secuse.nchc.org.tw:9000/hbase |
| hbase.tmp.dir                           | /var/hadoop/hbase-\${user.name}      |
| hbase.cluster.distributed               | true                                 |
| hbase.zookeeper.property<br>.clientPort | 2222                                 |
| hbase.zookeeper.quorum                  | Host1, Host2                         |
| hbase.zookeeper.property<br>.dataDir    | /var/hadoop/hbase-data               |

## Startup & Stop

```
$ start-hbase.sh
```

```
$ stop-hbase.sh
```

## Testing (4)

**\$ hbase shell**

**> create 'test', 'data'**

0 row(s) in 4.3066 seconds

**> list**

test

1 row(s) in 0.1485 seconds

**> put 'test', 'row1', 'data:1',  
'value1'**

0 row(s) in 0.0454 seconds

**> put 'test', 'row2', 'data:2',  
'value2'**

0 row(s) in 0.0035 seconds

**> put 'test', 'row3', 'data:3',  
'value3'**

0 row(s) in 0.0090 seconds

**> scan 'test'**

ROW COLUMN+CELL

row1 column=data:1, timestamp=1240148026198,  
value=value1

row2 column=data:2, timestamp=1240148040035,  
value=value2

row3 column=data:3, timestamp=1240148047497,  
value=value3

3 row(s) in 0.0825 seconds

**> disable 'test'**

09/04/19 06:40:13 INFO client.HBaseAdmin: Disabled  
test

0 row(s) in 6.0426 seconds

**> drop 'test'**

09/04/19 06:40:17 INFO client.HBaseAdmin: Deleted  
test

0 row(s) in 0.0210 seconds

**> list**

0 row(s) in 2.0645 seconds

# Connecting to HBase

- Java client
  - *get(byte [] row, byte [] column, long timestamp, int versions);*
- Non-Java clients
  - Thrift server hosting HBase client instance
- Sample ruby, c++, & java (via thrift) clients
  - REST server hosts HBase client
- TableInput/OutputFormat for MapReduce
  - HBase as MR source or sink
- HBase Shell
  - JRuby IRB with “DSL” to add get, scan, and admin
  - *./bin/hbase shell YOUR\_SCRIPT*

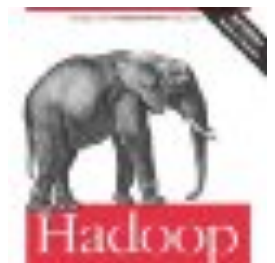
# Thrift

```
$ hbase-daemon.sh start thrift  
$ hbase-daemon.sh stop thrift
```

- a software framework for scalable cross-language services development.
- By Facebook
- seamlessly between C++, Java, Python, PHP, and Ruby.
- This will start the server instance, by default on port 9090
- The other similar project “rest”

# References

- Hbase the definitive guide by Lars George, 2011
  - <http://www.amazon.com/HBase-Definitive-Guide-Lars-George/dp/1449396100/>
- Hadoop: The Definitive Guide, 4th Edition by Tom White, 2015
  - <http://www.amazon.com/Hadoop-Definitive-Guide-Tom-White/dp/1449311520/>
- Introduction to HBase
  - [trac.nchc.org.tw/cloud/raw-attachment/wiki/.../hbase\\_intro.ppt](http://trac.nchc.org.tw/cloud/raw-attachment/wiki/.../hbase_intro.ppt)



# Cassandra

## Structured Storage System over a P2P Network

Lakshman, Avinash, and Prashant Malik. "Cassandra: a decentralized structured storage system." *ACM SIGOPS Operating Systems Review* 44.2 (2010): 35-40.

# Why Cassandra?

- Lots of data
  - Copies of messages, reverse indices of messages, per user data.
- Many incoming requests resulting in a lot of random reads and random writes.
- No existing production ready solutions in the market meet these requirements.



# Design Goals

- High availability
- Eventual consistency
  - trade-off strong consistency in favor of high availability
- Incremental scalability
- Optimistic Replication
- “Knobs” to tune tradeoffs between consistency, durability and latency
- Low total cost of ownership
- Minimal administration

# Innovation at scale

- Google Bigtable (2006)
  - Consistency model: strong
  - Data model: sparse map
  - Clones: Hbase, Hypertable
- Amazon Dynamo (2007)
  - O(1) Distributed Hash Table (DHT)
  - Consistency model: Client Tune-able
  - Clones: Riak, Voldemort

Cassandra  $\sim$  Data-Model of Bigtable/HBase  
+  
P2P DHT infrastructure of Dynamo

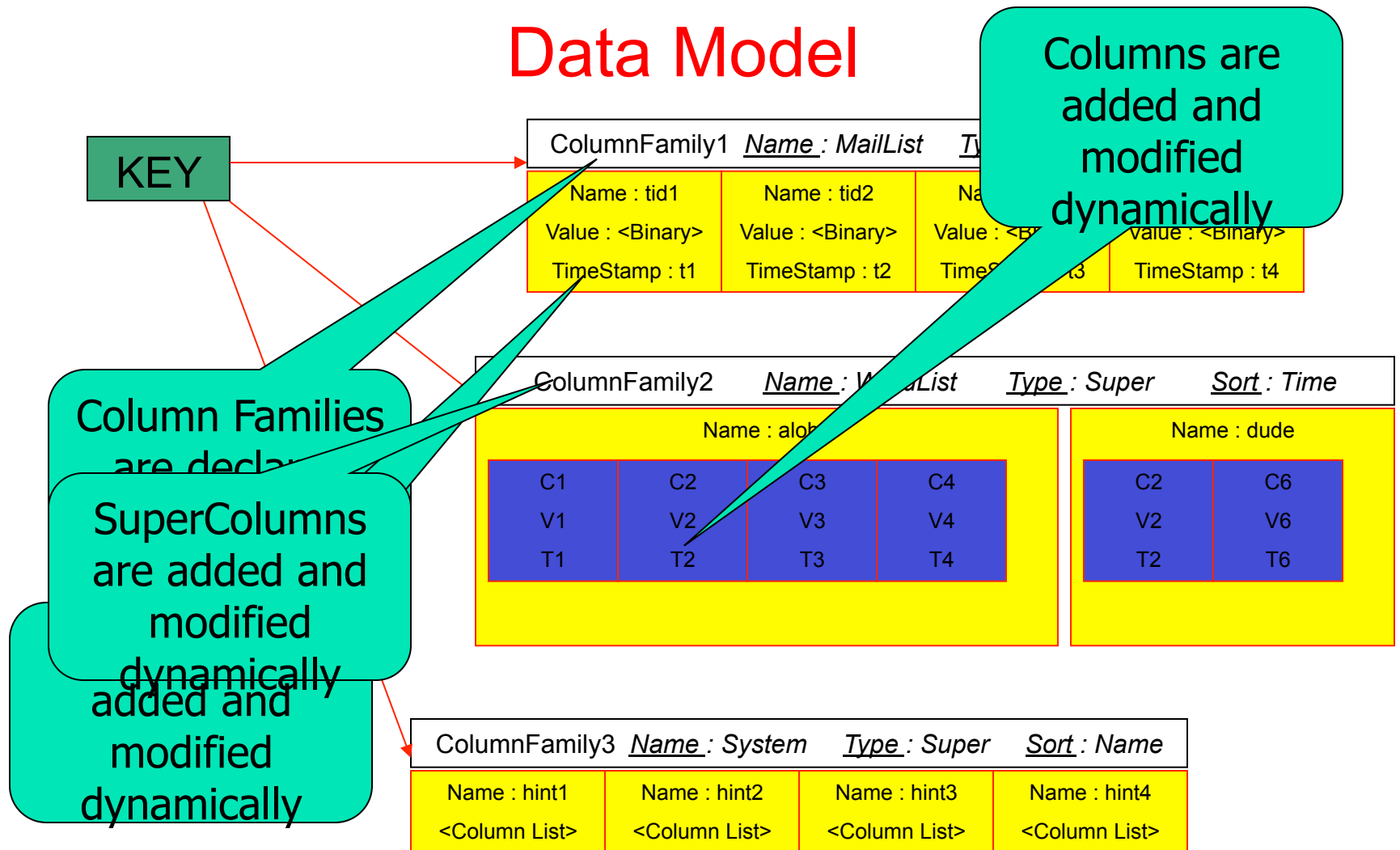
# Cassandra: A Proven Technology

- The Facebook stores 150TB of data on 150 nodes

web 2.0

- Used at Twitter, Rackspace, Mahalo, Reddit, Cloudkick, Cisco, Digg, SimpleGeo, Ooyala, OpenX, others

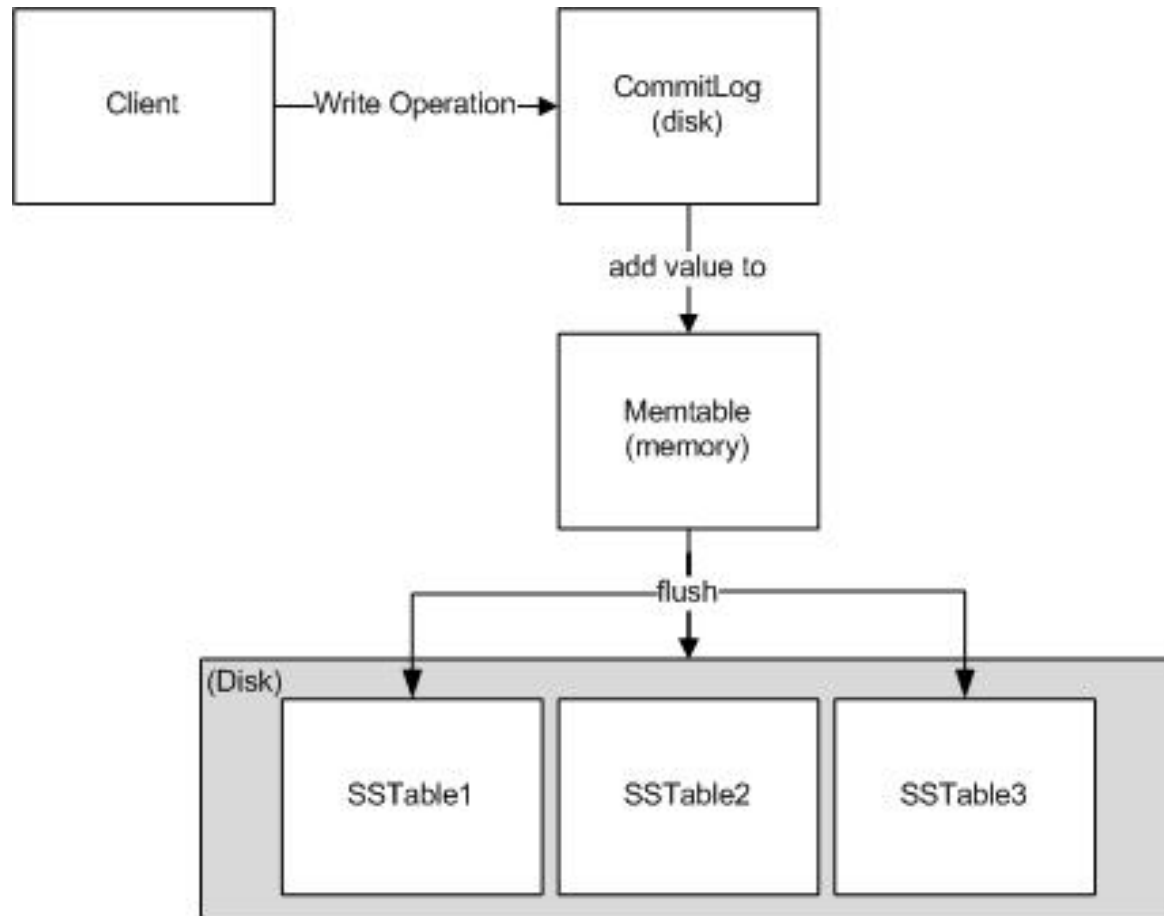
# Data Model



# Write Operations

- A client issues a write request to a random node in the Cassandra cluster.
- The “Partitioner” determines the nodes responsible for the data.
- Locally, write operations are logged and then applied to an in-memory version.
- Commit log is stored on a dedicated disk local to the machine.

# write op



# Write Properties

- No locks in the critical path
- Sequential disk access
- Behaves like a write back Cache
- Append support without read ahead
- Atomicity guarantee for a key
- “Always Writable”
  - accept writes during failure scenarios

# Deletes and Reads

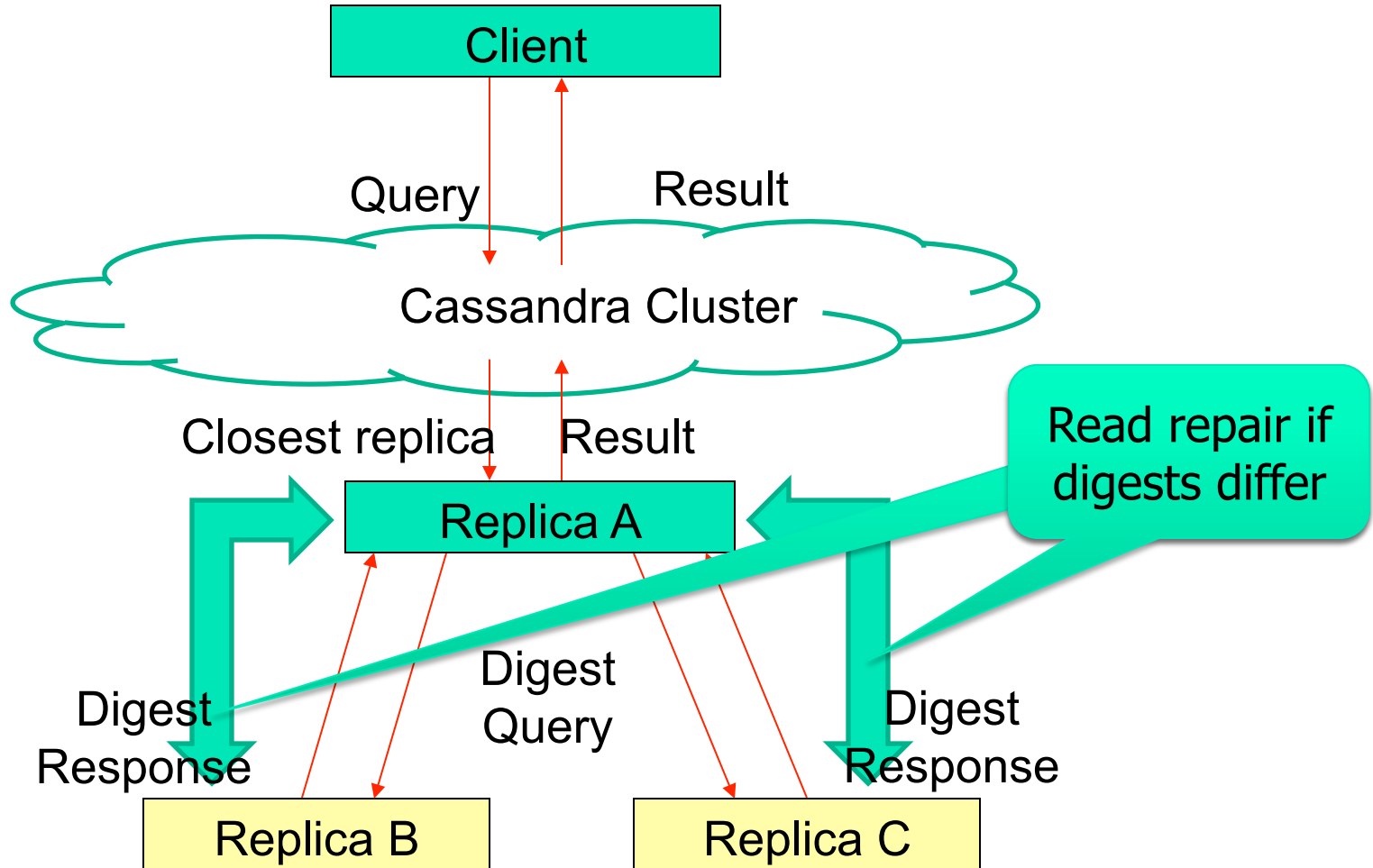
- Delete: don't delete item right away
  - Add a tombstone to the log
  - Compaction will eventually remove tombstone and delete item
- Read: Similar to writes, except
  - Coordinator can contact a number of replicas (e.g., in same rack) specified by consistency level
    - Forwards read to replicas that have responded quickest in past
    - Returns latest timestamp value
  - Coordinator also fetches value from multiple replicas
    - check consistency in the background, initiating a read-repair if any two values are different
    - Brings all replicas up to date
  - A row may be split across multiple SSTables => reads need to touch multiple SSTables => reads slower than writes (but still fast)



# Cassandra uses Quorums

- **Quorum** = way of selecting sets so that any pair of sets intersect
  - E.g., any arbitrary set with at least  $Q = N/2 + 1$  nodes
  - Where  $N$  = total number of replicas for this key
- **Reads**
  - Wait for  $R$  replicas ( $R$  specified by clients)
  - In the background, check for consistency of remaining  $N-R$  replicas, and initiate read repair if needed
- **Writes come in two default flavors**
  - Block until quorum is reached
  - Async: Write to any node
- $R$  = read replica count,  $W$  = write replica count
- If  $W+R > N$  and  $W > N/2$ , you have consistency, i.e., each read returns the latest written value
- Reasonable:  $(W=1, R=N)$  or  $(W=N, R=1)$  or  $(W=Q, R=Q)$

# Read



# Tunable Read Consistency Levels

| Consistency level | Implication   |
|-------------------|---|
| ZERO              | Unsupported. You cannot specify CL . ZERO for read operations because it doesn't make sense. This would amount to saying "give me the data from no nodes."  |
| ANY               | Unsupported. Use CL . ONE instead.  |
| ONE               | Immediately return the record held by the first node that responds to the query. A background thread is created to check that record against the same record on other replicas. If any are out of date, a <i>read repair</i> is then performed to sync them all to the most recent value. |
| QUORUM            | Query all nodes. Once a majority of replicas ( $(\text{replication factor} / 2) + 1$ ) respond, return to the client the value with the most recent timestamp. Then, if necessary, perform a read repair in the background on all remaining replicas.                                     |
| ALL               | Query all nodes. Wait for all nodes to respond, and return to the client the record with the most recent timestamp. Then, if necessary, perform a read repair in the background. If any nodes fail to respond, fail the read operation.   |

# Tunable Write Consistency Levels

| Consistency level | Implication   |
|-------------------|---|
| ZERO              | The write operation will return immediately to the client before the write is recorded; the write will happen asynchronously in a background thread, and there are no guarantees of success.                    |
| ANY               | Ensure that the value is written to a minimum of one node, allowing hints to count as a write.  |
| ONE               | Ensure that the value is written to the commit log and memtable of at least one node before returning to the client.  |
| QUORUM            | Ensure that the write was received by at least a majority of replicas ( $(\text{replication\_factor} / 2) + 1$ ).   |
| ALL               | Ensure that the number of nodes specified by <code>replication_factor</code> received the write before returning to the client. If even one replica is unresponsive to the write operation, fail the operation. |

# Eventual Consistency (User Tunable)

*BASE: If all writers stop (to a key), then all its values (replicas) will converge eventually.*

- If writes continue, then system always tries to keep converging.
  - Moving “wave” of updated values lagging behind the latest values sent by clients, but always trying to catch up
- Converges when  $R + W > N$ 
  - $R$  = # records to read,  $W$  = # records to write,  $N$  = replication factor
- Consistency Levels: (refer the tables in the previous pages)
  - ONE ->  $R$  or  $W$  is 1
  - QUORUM ->  $R$  or  $W$  is ceiling  $(N + 1) / 2$
  - ALL ->  $R$  or  $W$  is  $N$
- If you want to write with Consistency Level of ONE and get the same data when you read, you need to read with Consistency Level of ALL

# Cluster Membership and Failure Detection

- Like Dynamo, Gossip protocol is used for cluster membership and failure detection.

# Performance Benchmark

- Loading of data - limited by network bandwidth.
- Read performance for Facebook Inbox Search in production:

|         | Search Interactions | Term Search |
|---------|---------------------|-------------|
| Min     | 7.69 ms             | 7.78 ms     |
| Median  | 15.69 ms            | 18.27 ms    |
| Average | 26.13 ms            | 44.41 ms    |

# MySQL Comparison

- MySQL > 50 GB Data  
Writes Average : ~300 ms  
Reads Average : ~350 ms
- Cassandra > 50 GB Data  
Writes Average : 0.12 ms  
Reads Average : 15 ms



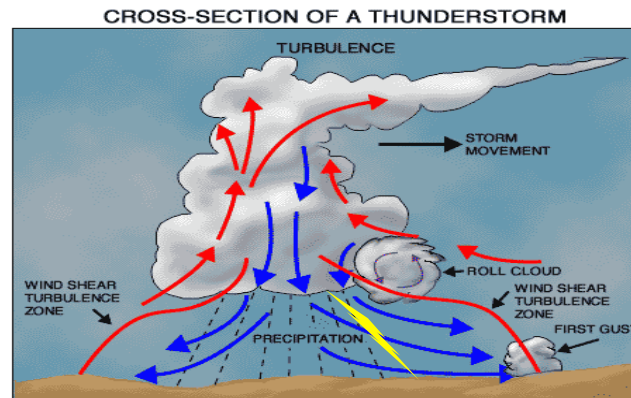
# Lessons Learnt

- Add fancy features only when absolutely required.
- Many types of failures are possible.
- Big systems need proper systems-level monitoring.
- Value simple designs

# Typical System Architecture for Multi-tier Cloud-based Services

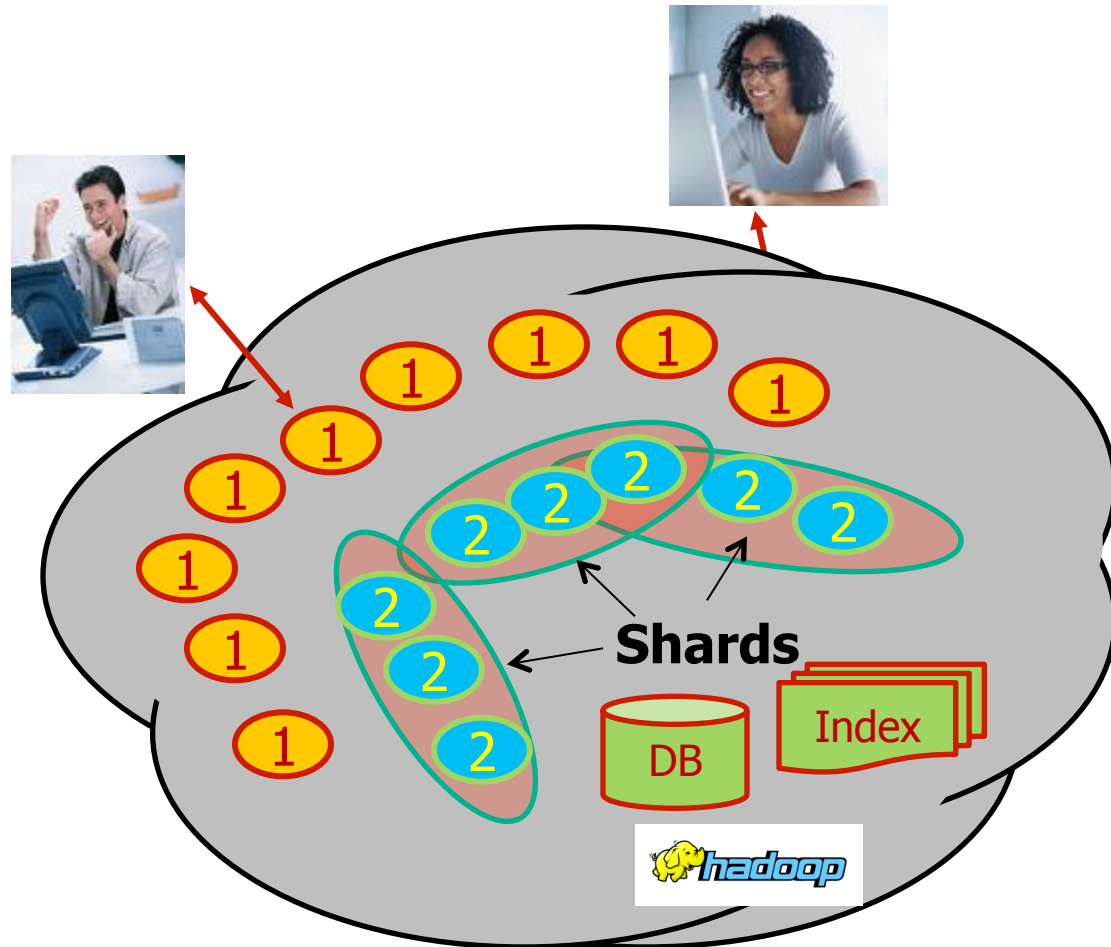
# How are cloud structured?

- Clients talk to clouds using web browsers or the web services standards
  - But this only gets us to the outer “skin” of the cloud data center, not the interior
  - Consider Amazon: it can host entire company web sites (like Target.com or Netflix.com), data (AC3), servers (EC2) and even user-provided virtual machines!



# Big picture overview

- Client requests are handled in the “first tier” by
  - PHP or ASP pages
  - Associated logic
- These lightweight services are fast and very nimble
- Much use of caching:  
the second tier



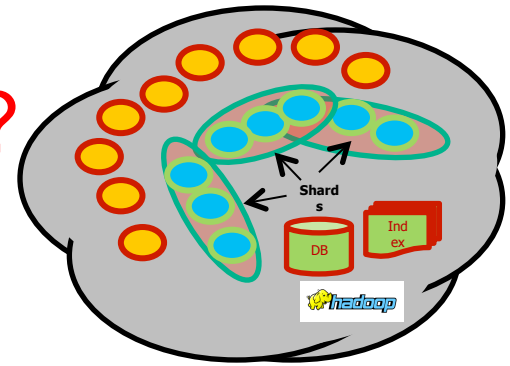
## Many styles of system

- Near the edge of the cloud focus is on vast numbers of clients and rapid response
- Inside we find high volume services that operate in a pipelined manner, asynchronously
- Deep inside the cloud we see a world of virtual computer clusters that are scheduled to share resources and on which applications like MapReduce (Hadoop) are very popular

# In the outer tiers replication is key

- We need to replicate
  - Processing: each client has what seems to be a private, dedicated server (for a little while)
  - Data: as much as possible, that server has copies of the data it needs to respond to client requests without any delay at all
  - Control information: the entire structure is managed in an agreed-upon way by a decentralized cloud management infrastructure

# What about the “shards”?



- The caching components running in Tier-two are central to the responsiveness of Tier-one services
  - Basic idea is to always use cached data if at all possible, so the inner services (here, a database and a search index stored in a set of files) are shielded from “online” load
  - We need to replicate data within our cache to spread loads and provide fault-tolerance
  - But not everything needs to be “fully” replicated. Hence we often use “shards” with just a few replicas

# Sharding used in many ways

- The second tier could be any of a number of caching services:
  - Memcached: a sharable in-memory key-value store
  - Other kinds of DHTs that use key-value APIs
  - Dynamo: A service created by Amazon as a scalable way to represent the shopping cart and similar data
  - BigTable: A very elaborate key-value store created by Google and used not just in tier-two but throughout their “GooglePlex” for sharing information
- Notion of sharding is cross-cutting
  - Most of these systems replicate data to some degree



# Do we *always* need to shard data?

- Imagine a tier-one service running on 100k nodes
  - Can it ever make sense to replicate data on the entire set?
- Yes, if some kinds of information might be so valuable that almost every external request touches it.
  - Must think hard about patterns of data access and use
  - Some information needs to be heavily replicated to offer blindingly fast access on vast numbers of nodes
  - The principle is similar to the way Beehive operates.
    - Even if we don't make a dynamic decision about the level of replication required, the principle is similar
    - We want the level of replication to match level of load and the degree to which the data is needed on the critical path

## And it isn't just about updates

- Should also be thinking about patterns that arise when doing reads (“queries”)
  - Some can just be performed by a single representative of a service
  - But others might need the parallelism of having several (or even a huge number) of machines do parts of the work concurrently
- The term sharding is used for data, but here we might talk about “parallel computation on a shard”