Key-value/NoSQL Store

Yao Liu

The Key-value abstraction

- (Business) Key → Value
- (twitter.com) tweet id → information about tweet
- (amazon.com) item number → information about it
- (kayak.com) Flight number → information about flight, e.g., availability
- (yourbank.com) Account number → information about it

The Key-value abstraction

- It's a dictionary datastructure.
    - Insert, lookup, and delete by key
    - e.g., hash table, binary tree
- But distributed.
- Sound familiar? Remember Distributed Hash tables (DHT) in P2P systems?
- It's not surprising that key-value stores reuse many techniques from DHTs.

Database?

- Relational Database Management Systems (RDBMSs) have been around for ages
- MySQL is the most popular among them
- Data stored in tables
- Schema-based, i.e., structured tables
- Each row (data item) in a table has a primary key that is unique within that table
- Queried using SQL (Structured Query Language)
- Supports joins

Today's workloads

- Data: Large and unstructured
- Lots of random reads and writes
- Sometimes write-heavy
- Foreign keys rarely needed
- Joins infrequent

Needs of Today’s Workloads

- Speed
- Avoid single point of failure
- Low TCO (total cost of operation)
- Fewer system administrators
- Incremental scalability
- Scale out, not scale up

Scale out, not Scale up

- Scale up = grow your cluster capacity by replacing with more powerful machines
  - Traditional approach
  - Not cost-effective, as you're buying above the sweet spot on the price curve
  - And you need to replace machines often
- Scale out = incrementally grow your cluster capacity by adding more COTS machines (Components Off the Shelf)
  - Cheaper
  - Over a long duration, phase in a few newer (faster) machines as you phase out a few older machines
  - Used by most companies who run datacenters and clouds today

Key-value/NoSQL data model

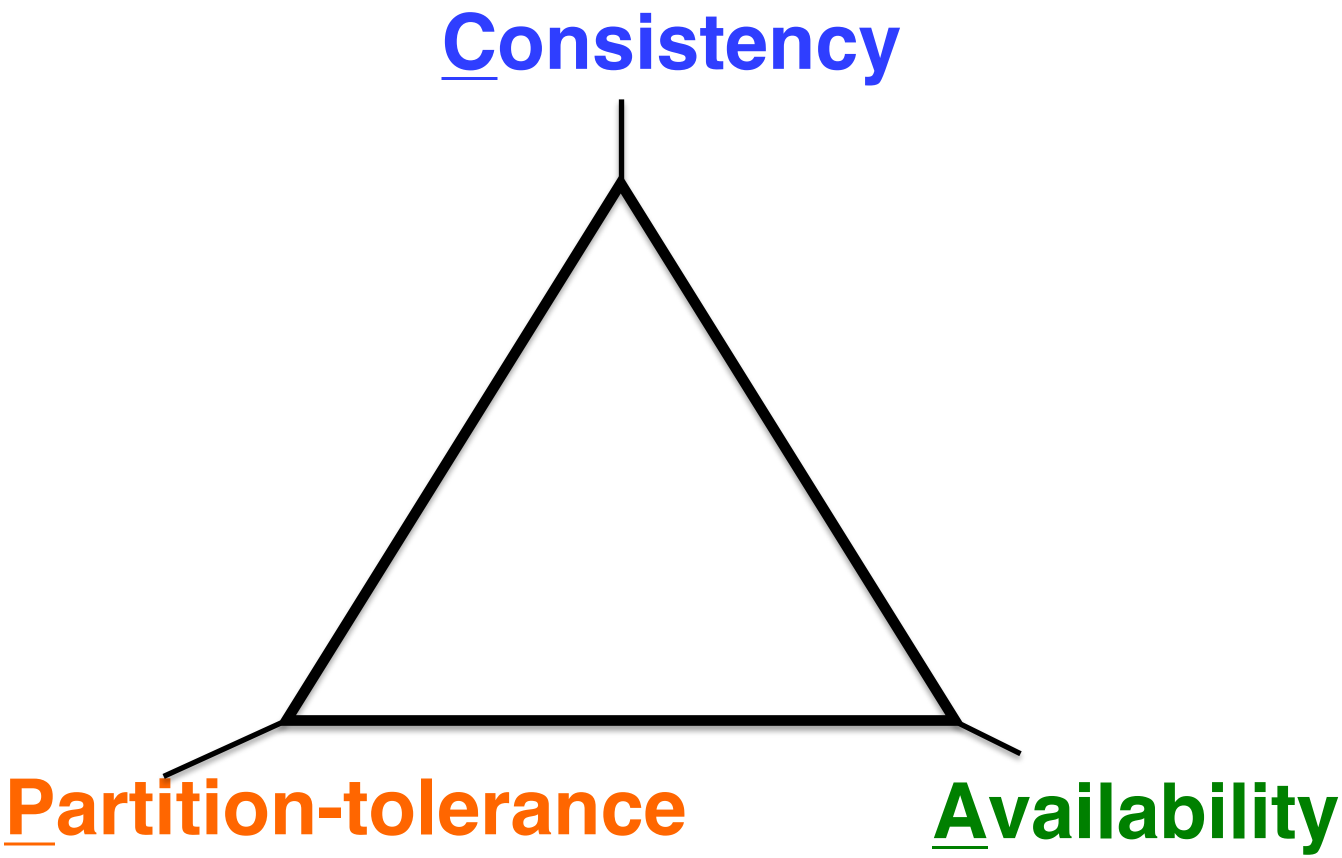
- NoSQL = “Not Only SQL”

- Necessary API operations:
  - get(key)
  - put(key, value)

CAP Theorem

- Proposed by Eric Brewer (Berkeley)
- Subsequently proved by Gilbert and Lynch (NUS and MIT)
- In a distributed system you can satisfy at most 2 out of the 3 guarantees:
  1. **Consistency**: all nodes see same data at any time, or reads return latest written value by any client
  2. **Availability**: the system allows operations all the time, and operations return quickly
  3. **Partition-tolerance**: the system continues to work in spite of network partitions

CAP Theorem



Why is Availability Important?

- Availability = Reads/writes complete reliably and quickly.
- Measurements have shown that a 500 ms increase in latency for operations at Amazon.com or at Google.com can cause a 20% drop in revenue.
- User cognitive drift: if more than a second elapses between clicking and material appearing, the user’s mind is already somewhere else

Why is Consistency Important?

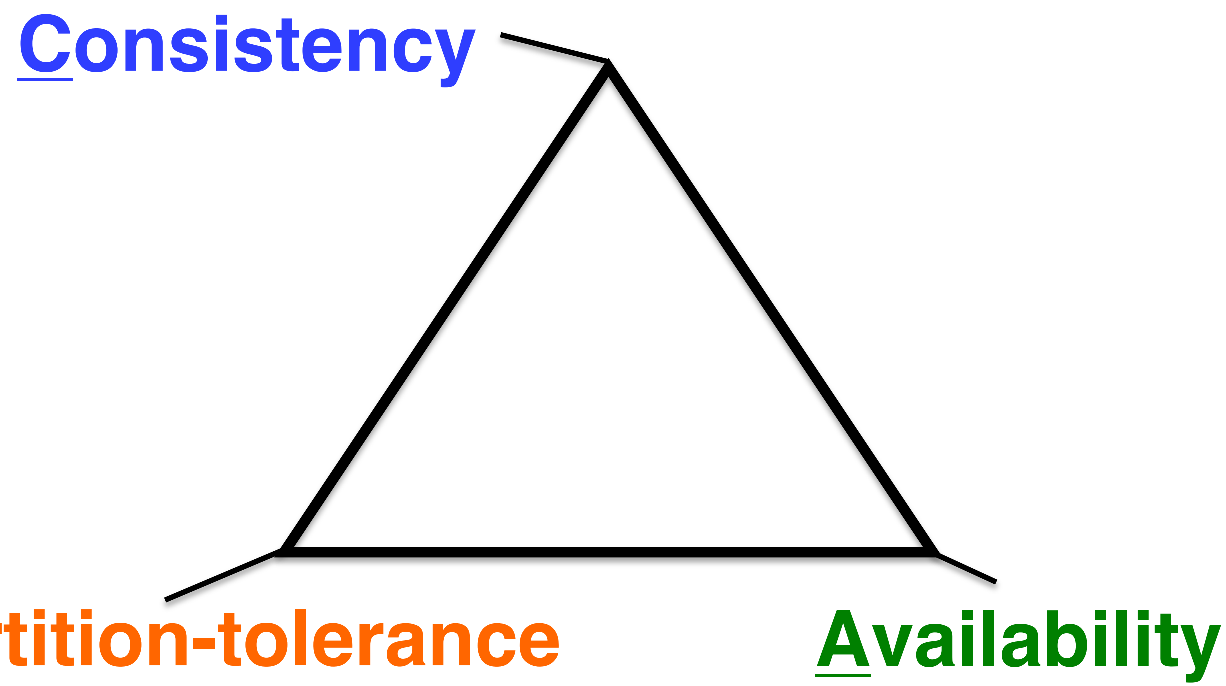
- Consistency = all nodes see same data at any time, or reads return latest written value by any client.
- When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.
- When thousands of customers are looking to book a flight, all updates from any client (e.g., book a flight) should be accessible by other clients.

Why is Partition-Tolerance Important?

- Partitions can happen across datacenters when the Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working
- Partitions can also occur within a datacenter, e.g., a rack switch outage
- Still desire system to continue functioning normally under this scenario

CAP Theorem Fallout

- Since partition-tolerance is essential in today's cloud computing systems,
- CAP theorem implies that a system has to choose between **consistency** and **availability**



Dynamo: Amazon’s Highly Available Key-value Store

Amazon's Dynamo

- Distributed key-value storage
  - Only accessible with the primary key
  - put(key, value) & get(key)
- Used for many Amazon services
  - Shopping cart, best seller lists, customer preferences, etc.
- Inspired many NoSQL implementations
  - Cassandra
  - Riak
  - Project Voldemont

Design consideration

- Sacrifice strong consistency for availability
- “Always writeable” data store where no updates are rejected due to failures or concurrent writes.
- Conflict resolution is executed during ***read*** instead of ***write***, i.e., “always writeable”.

Design consideration (cont’d)

- Incremental scalability
  - be able to scale out one storage host at a time, with minimal impact on both operators of the system and the system itself.
- 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.

System architecture

- Partitioning
- High Availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection

Partition algorithm

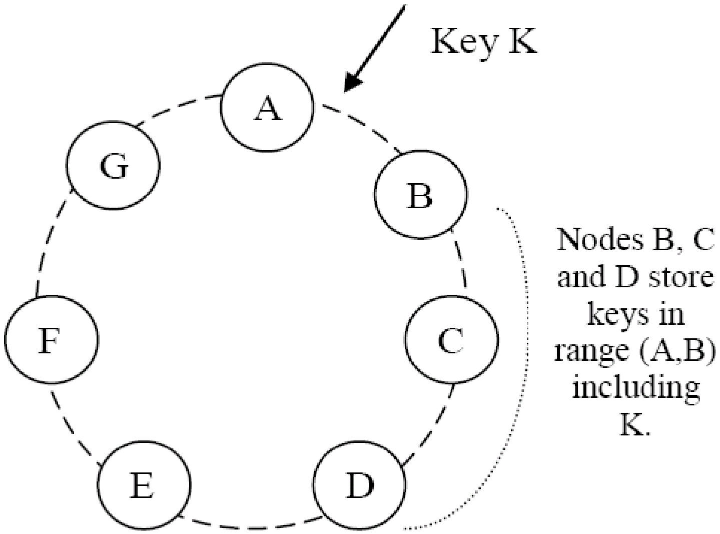
- Consistent hashing
  - the output range of a hash function is treated as a fixed circular space or “ring”.
  - With original consistent hashing, e.g., used by Chord DHT, load may become uneven
- “Virtual Nodes” for better load balancing
  - Each node can be responsible for more than one virtual nodes.

Partition algorithm (cont'd)

- 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, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
  - The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

Replication

- Each data item is replicated at N hosts; configurable.
- The first is stored regularly with consistent hashing
- N-1 replicas are stored in the N-1 successor nodes
- "*preference list*": The list of nodes that is responsible for storing a particular key.



Quorum

- Parameters:
  - N replicas
  - R readers
  - W writers
- Static quorum approach:
  - $R+W>N$
- Typical Dynamo configuration:
  - $N=3$, $R=2$, $W=2$

Sloppy quorum

- But we know that quorum-like approaches sacrifice availability for consistency!
- Dynamo uses a “sloppy” quorum.
  - all read and write operations are performed on the first N healthy nodes from the preference list, which may not always be the first N nodes encountered while walking the ring

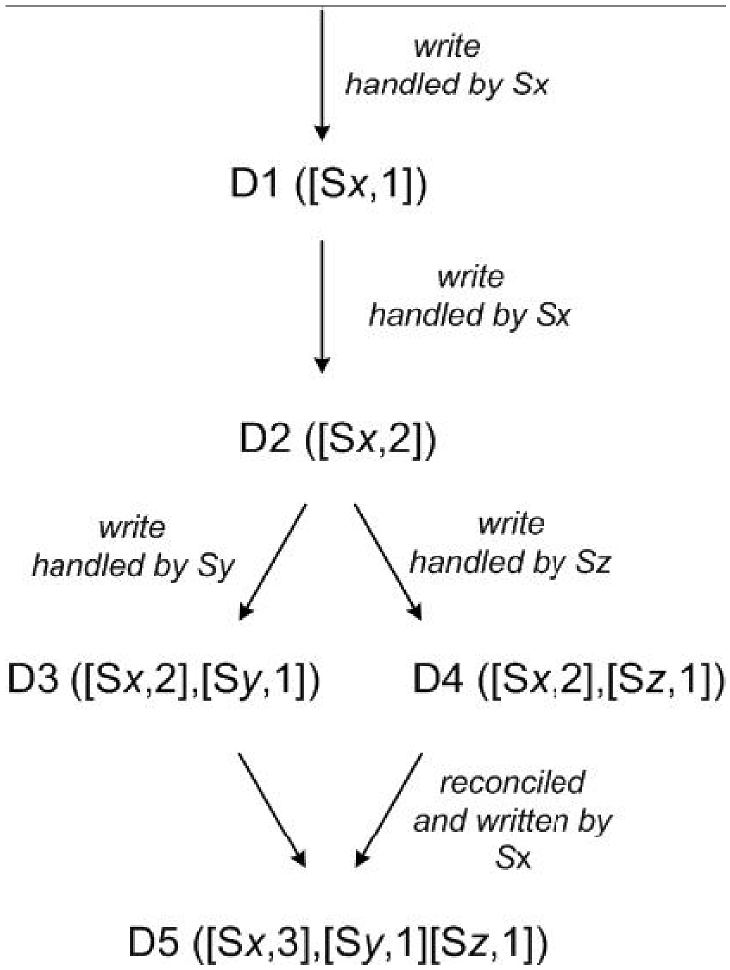
Data versioning

- Writes should succeed all the time
  - e.g., “Add to Cart”
- Need to reconcile inconsistent data due to network partitioning/failures
- Each object has a vector clock
  - e.g., D1 ([Sx, 1], [Sy, 1]): Object D1 has written once by server Sx and Sy, respectively.
  - Each node keeps all versions until the data becomes consistent
- If inconsistent, reconcile later.
  - e.g., deleted items might reappear in the shopping cart.

Vector clock

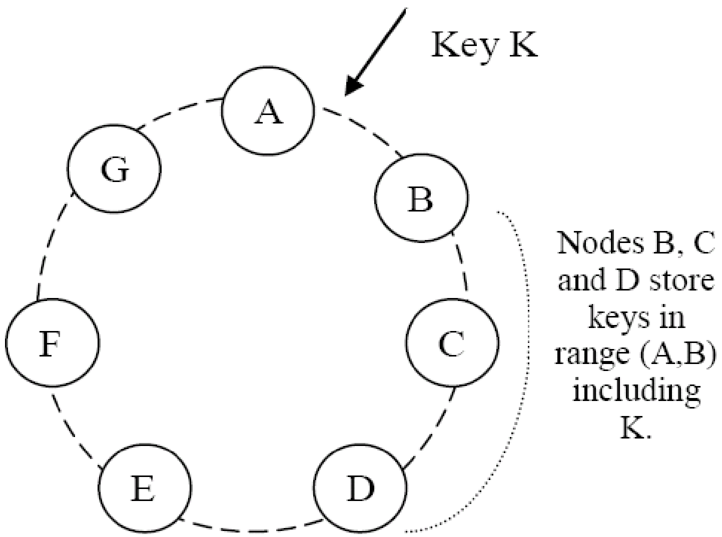
- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- *If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.*

Vector clock example



Handling failures: hinted handoff

- Assume N = 3. When C is temporarily down or unreachable during a write, send replica to E.
- E is hinted that the replica belongs to C and it will deliver to C when C is recovered.
- Again: “always writeable”

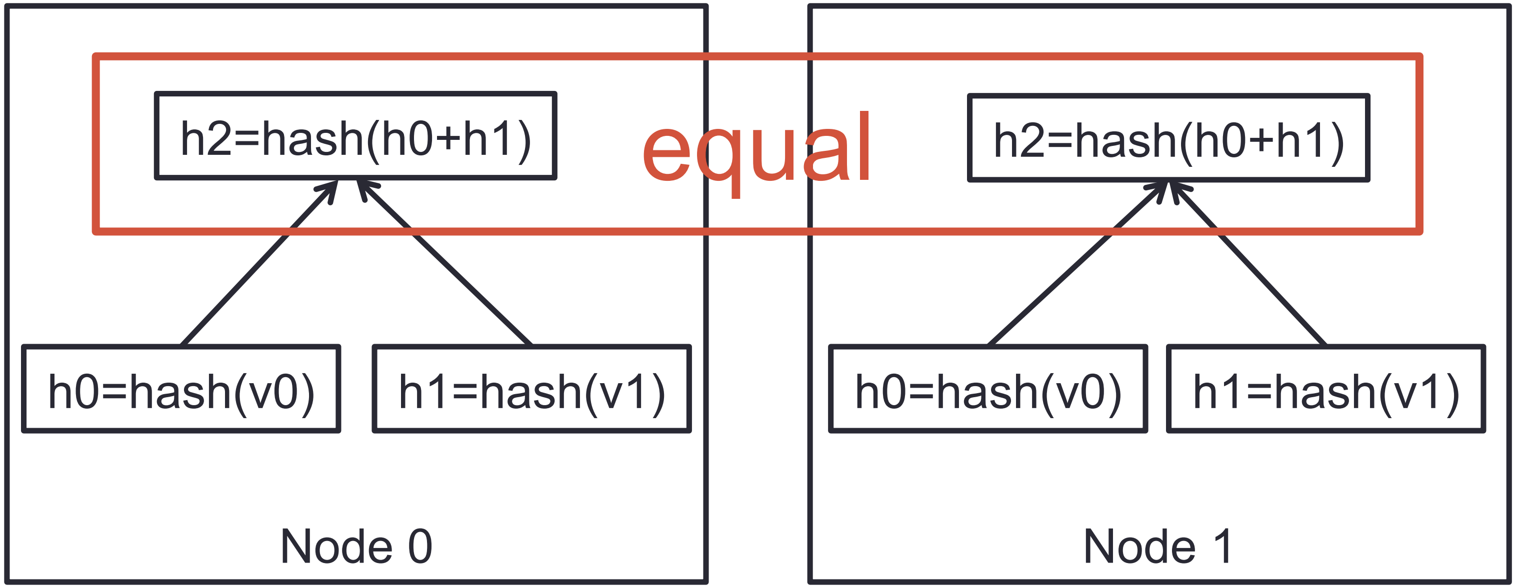


Replica synchronization

- Key ranges are replicated.
- If a node fails and recovers, it needs to quickly determine whether it needs to resynchronize or not.
  - Transferring entire (key, value) pairs for comparison is not an option.
- 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.

Replica synchronization

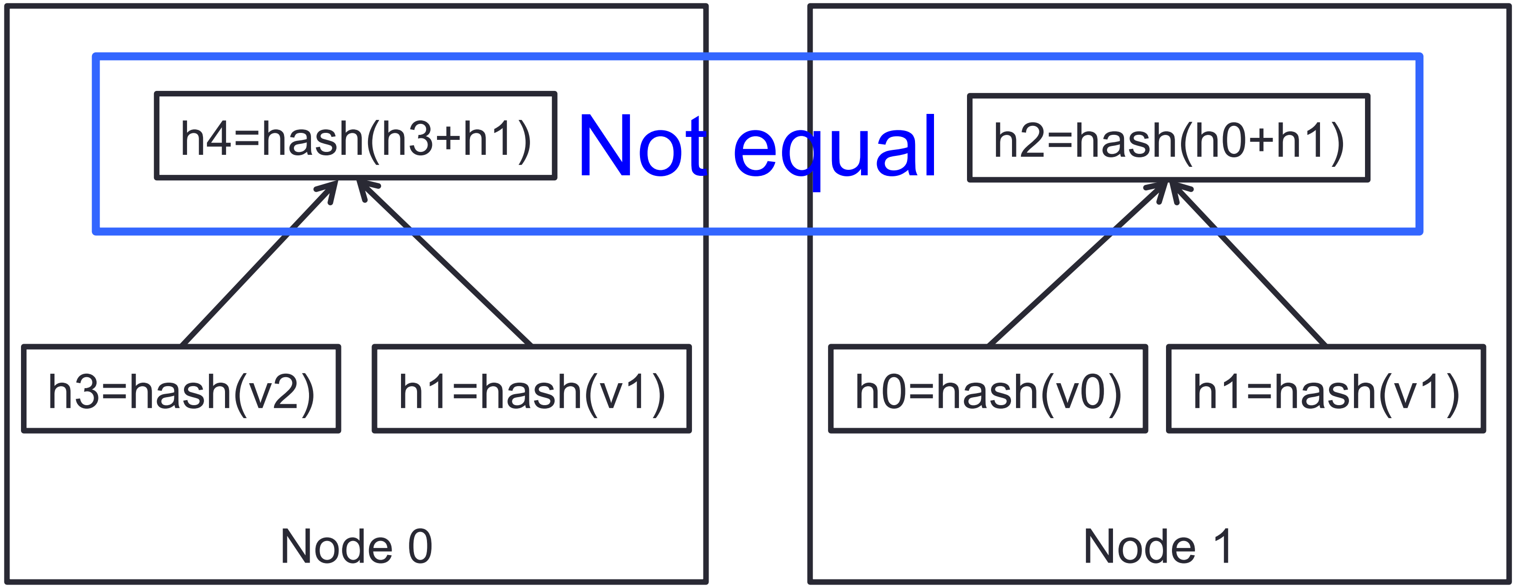
- Comparing two nodes that are synchronized
  - Two (key, value) pairs: (k0, v0) & (k1, v1)



+ indicates concatenate

Replica synchronization

- Comparing two nodes that are *not synchronized*
  - One: (k0, v2) & (k1, v1)
  - The other: (k0, v0) & (k1, v1)



+ indicates concatenate

Membership

- Nodes are organized as a ring just like Chord using consistent hashing
- But everyone knows everyone else.
- Node join/leave
  - Manually done
  - An operator uses a console to add/delete a node
  - Reason: it’s a well-maintained system; nodes come back pretty quickly and don’t depart permanently most of the time
- Membership change propagation
  - Each node maintains its own view of the membership & the history of the membership changes
  - Gossip-based protocol for propagation (each node contacts a randomly selected node every second)
- Eventually-consistent membership protocol

Failure detection

- Avoid communicating with unreachable peers
  - during put() and get()
  - transferring partitions and hinted replicas
- Does not use a separate protocol; each request serves as a ping
  - Dynamo has enough requests at any moment anyway
- If a node doesn’t respond to a request, it is considered to be failed.

Conclusion

- Amazon Dynamo is a highly available and scalable data store.
- Techniques:
  - Gossiping for propagation of membership changes
  - Consistent hashing for node and key distribution
  - Data versioning for eventually-consistent data objects
  - Sloppy quorum for partition/failure tolerance
  - Merkle tree for replica synchronization

Reading

- Dynamo: Amazon's Highly Available Key-value Store
- http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

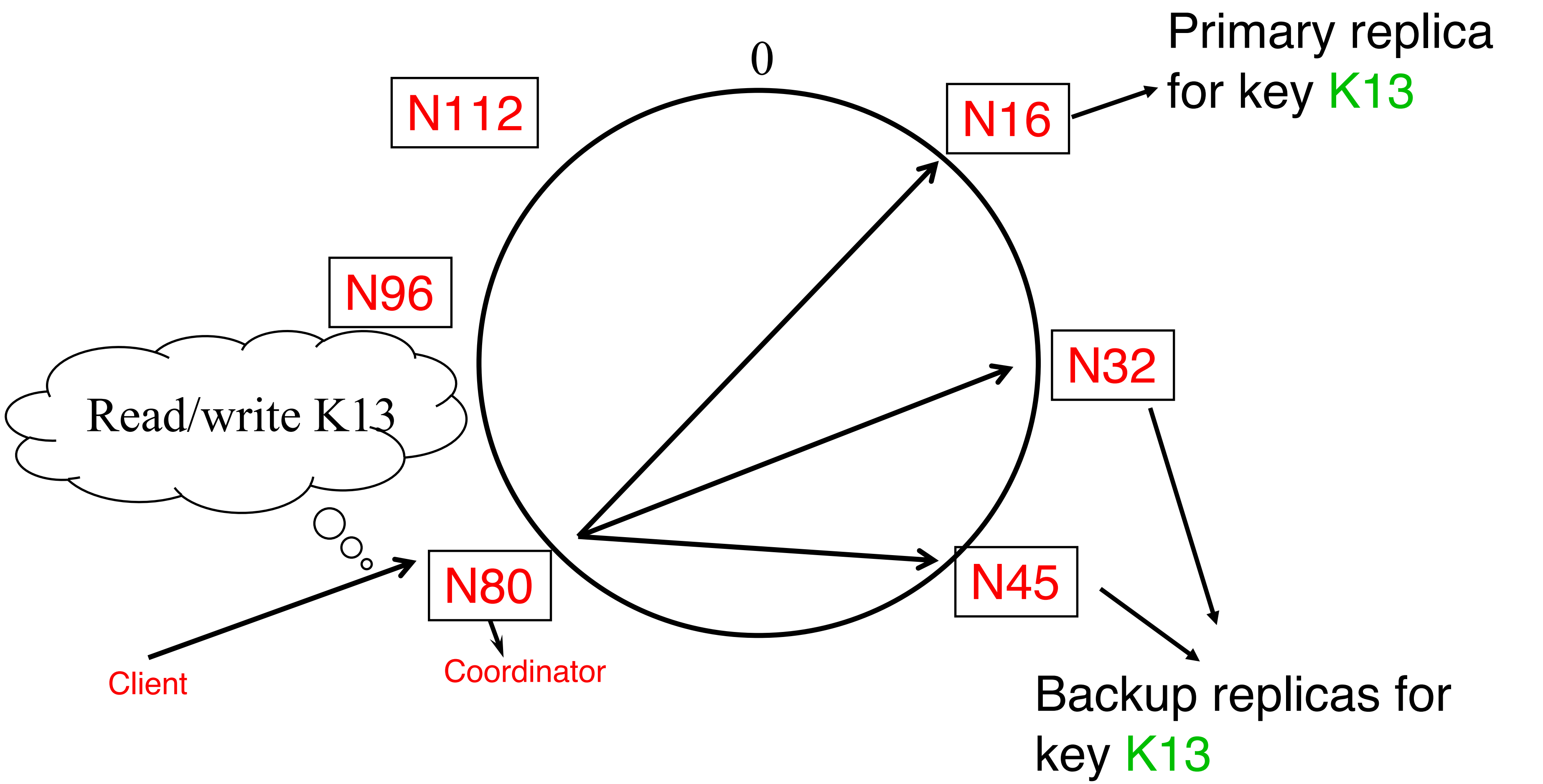
Cassandra

Cassandra

- A distributed key-value store
- Intended to run in a datacenter (and also across DCs)
- Originally designed at Facebook
- Open-sourced later, today an Apache project
- Some of the companies that use Cassandra in their production clusters
  - IBM, Adobe, HP, eBay, Ericsson, Symantec
  - Twitter, Spotify
  - Netflix: uses Cassandra to keep track of your current position in the video you're watching

Partitioning

- Which server(s) a key-value resides on?



Partitioner

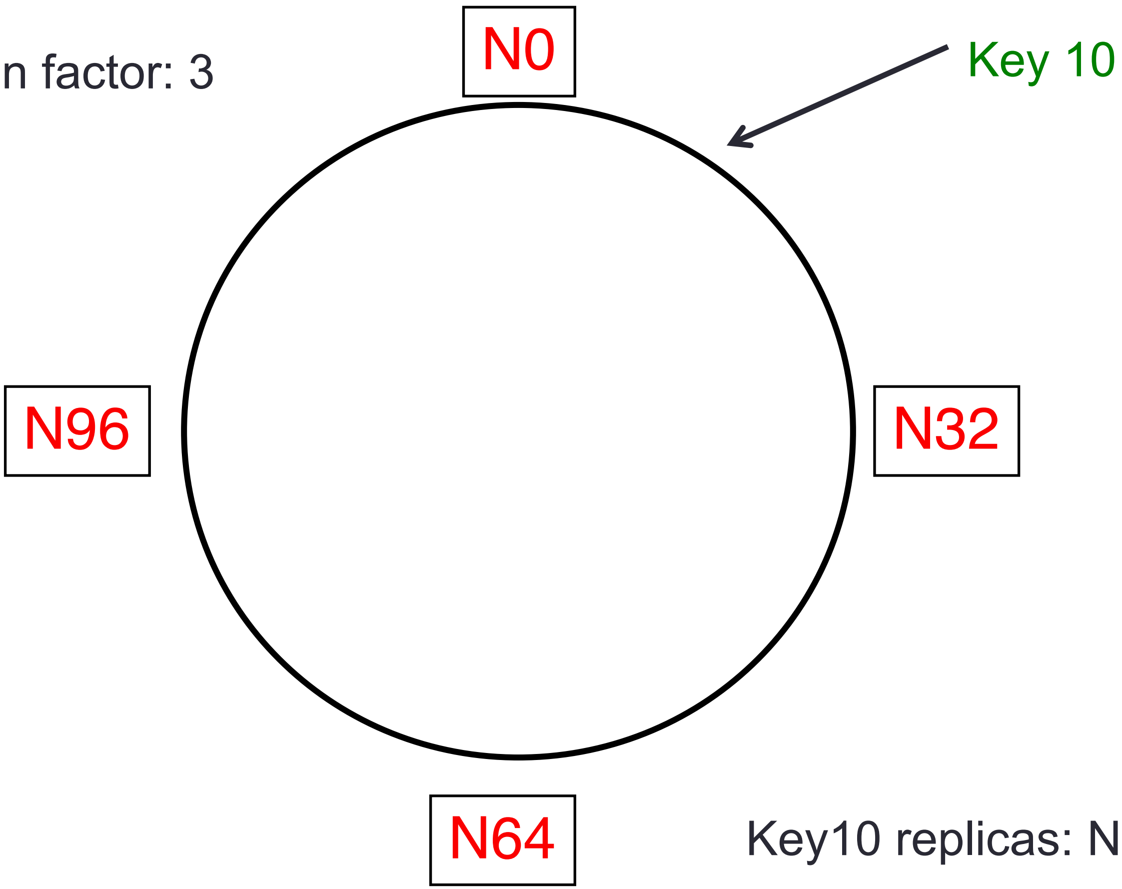
- A function for computing the key → server mapping

Data replication strategies

- Two replication strategies are available:
  - `SimpleStrategy`
  - `NetworkTopologyStrategy`
- SimpleStrategy: uses Partitioner
  - *RandomPartitioner*: Chord-like hash partitioning
  - *ByteOrderedPartitioner*: Assigns ranges of keys to servers.
    - Easier for *range queries* (e.g., return all twitter users starting with [a-b])
  - places the first replica on a node determined by the partitioner. Additional replicas are placed on the next nodes clockwise in the ring without considering topology (rack or datacenter location)
- NetworkTopologyStrategy: for multi-datacenter deployments
  - Two replicas per DC
  - Three replicas per DC
  - Within DC: first replica placed according to Partitioner, then go clockwise around ring until reaching the first node in another rack

SimpleStrategy

Replication factor: 3

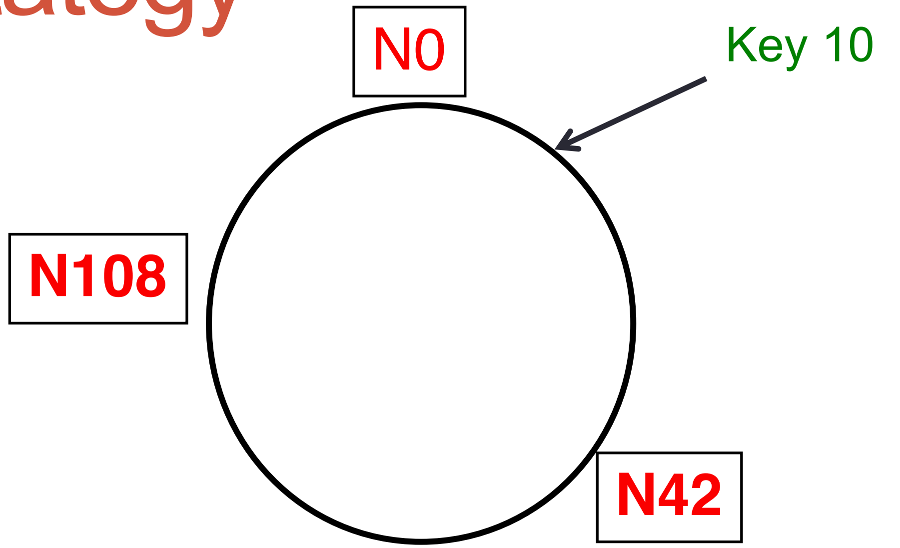


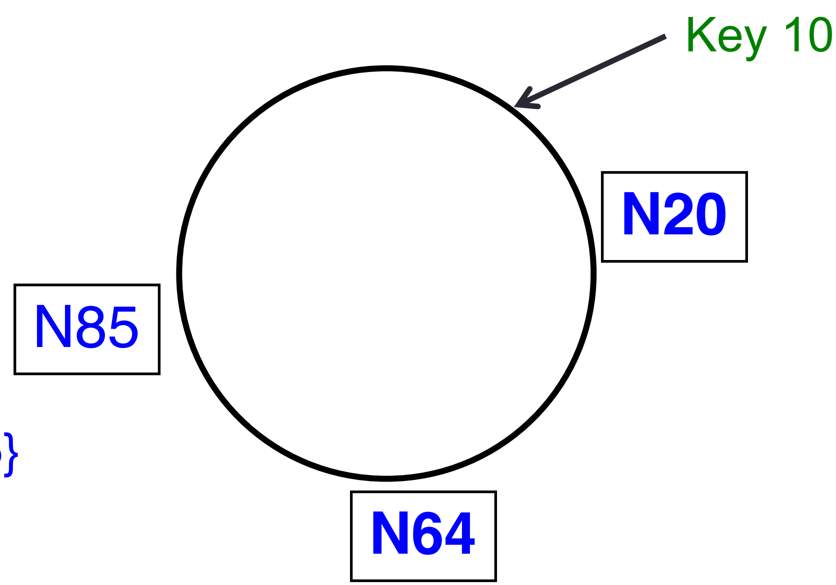
Key10 replicas: N32, N64, N96

NetworkTopologyStategy

Replication factor: {DC1:2, DC2:2}

Node	DC	Rack
N0	DC1	Rack2
N42	DC1	Rack1
N108	DC1	Rack1
N20	DC2	Rack1
N64	DC2	Rack1
N85	DC2	Rack2





Key10 replicas: DC1:{N42,N0}, DC2:{N20,N85}

Snitches

- Maps: IPs to racks and DCs. Configured in cassandra.yaml config file
- Some options:
  - `SimpleSnitch`: Unaware of topology (rack-unaware)
  - `RackInferring`: Assumes topology of network by octet of server's IP address
    - 101.201.202.203 = x.<DC octet>.<rack octet>.<node octet>
  - `PropertyFileSnitch`: uses a config file
  - `EC2Snitch`: uses Amazon EC2.
    - EC2 Region = DC
    - Availability zone = rack
- Other snitch options available

Write

- Need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one coordinator node in Cassandra cluster
- Coordinator uses Partitioner to send query to all replica nodes responsible for key
- When X replicas respond, coordinator returns an acknowledgement to the client
  - X? We'll see later.

Write

- Always writable: Hinted Handoff mechanism
  - If any replica is down, the coordinator writes to all other replicas and keeps the write locally until down replica comes back up.
  - When all replicas are down, the Coordinator (front end) buffers writes (for up to a few hours).
- Multi-datacenter write
  - The coordinator forwards the write request to one replica in each of other datacenters, who will further forward the write to local replicas

Write on a replica node

On receiving a write

1. Log it in **disk commit log** (for failure recovery)
2. Make changes to appropriate memtables
   - **Memtable** = In-memory representation of multiple key-value pairs
   - *Typically append-only datastructure (fast)*
   - Cache that can be searched by key
   - Write-back cache as opposed to write-through

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
- *SSTables are immutable (once created, they don't change)*
- Index file: An SSTable of (key, position in data SSTable) pairs
- And a Bloom filter (for efficient search)

Column-oriented storage

- RDBMSs store an entire row together (on disk or at a server)
- NoSQL systems typically store a column together (or a group of columns).
- Why useful?
  - Range searches within a column are fast since you don't need to fetch the entire database
  - e.g., get me all the blog_ids from the blog table that were updated within the past month
    - Search in the the last_updated column, fetch corresponding blog_id column
    - Don't need to fetch the other columns

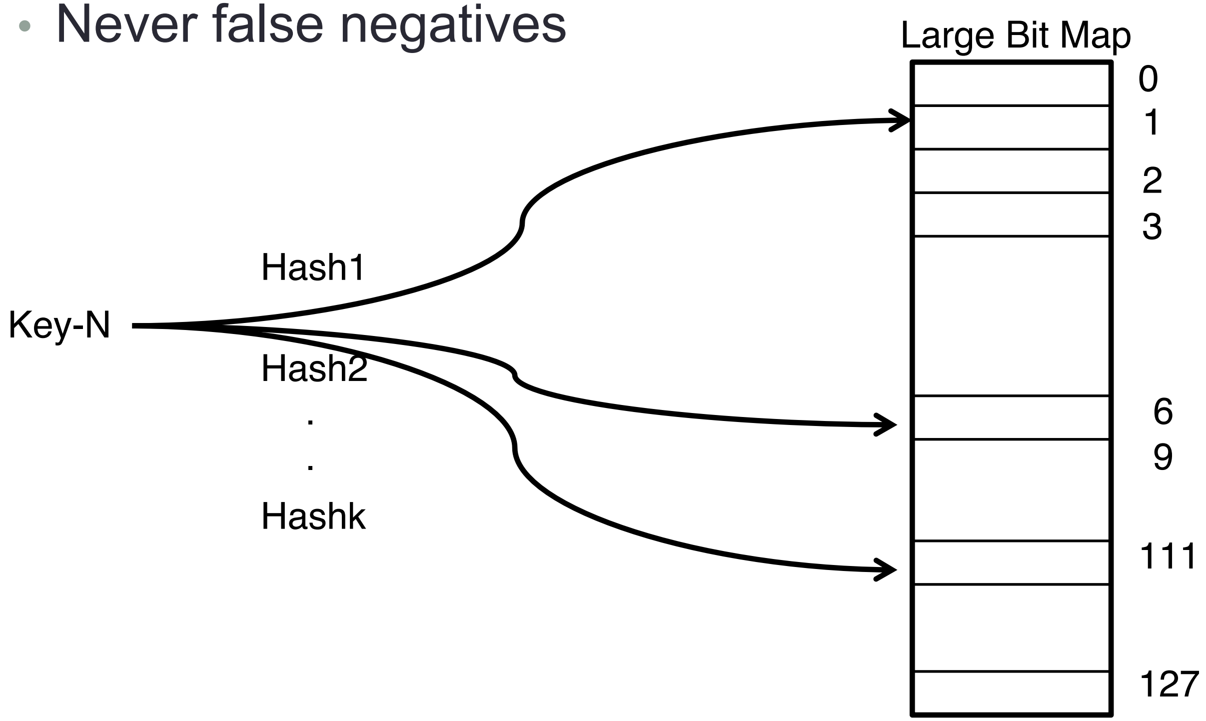
Compaction

Data updates accumulate over time and SSTables and logs need to be compacted

- The process of compaction merges SSTables, i.e., by merging updates for a key
- Run periodically and locally at each server

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
- n = 100 items
- m = 3200 bits
- FP rate = 0.0191%

http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html

Delete

Delete: don’t delete item right away

- Add a **tombstone**
- Eventually, when compaction encounters tombstone it will delete item

Read

Read: Similar to writes, except

- Coordinator can contact X replicas
  - Coordinator sends read to replicas that have responded quickest in past
  - When X replicas respond, coordinator returns the **latest-timestamped value** from among those X
  - (X? We'll see later.)
- Coordinator also fetches value from other replicas
  - Checks consistency in the background, initiating a **read repair** if any two values are different
  - This mechanism seeks to eventually bring all replicas up to date
- At a replica
  - Read looks at **Memtables** first, and then **SSTables**
  - A row may be split across multiple SSTables → reads need to touch multiple SSTables → reads slower than writes (but still fast)

Membership

- Any server in cluster could be the coordinator
- So every server needs to maintain a list of all the other servers that are currently in the cluster
- List needs to be updated automatically as servers join, leave, and fail
- Gossip-based
  - Nodes periodically gossip their membership list
  - On receipt, the local membership list is updated
  - If any heartbeat older than Tfail, node is marked as failed

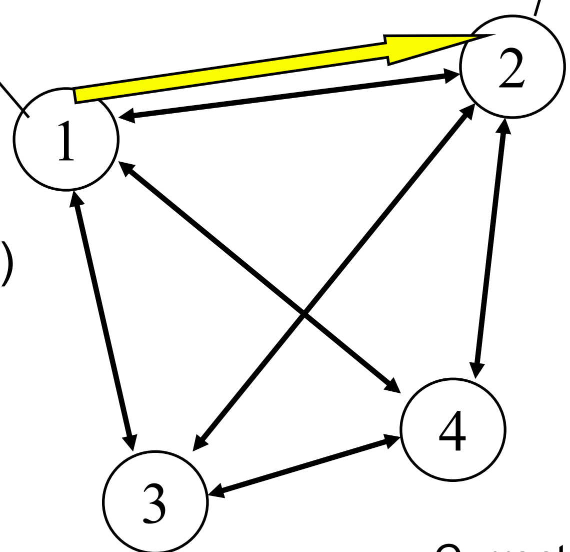
Gossip-based cluster membership

1	10120	66
2	10103	62
3	10098	63
4	10111	65

Address

Heartbeat Counter

Time (local)



1	10118	64
2	10110	64
3	10090	58
4	10111	65

1	10120	70
2	10110	64
3	10098	70
4	10111	65

Current time : 70 at node 2

Failure detection

- Does not use a fixed threshold for marking failing nodes.
- Accrual failure detector
  - designed to adapt to changing network conditions
- The value output, PHI, represents a suspicion level.
- Applications set an appropriate threshold, trigger suspicions and perform appropriate actions.

Cassandra vs. RDBMS

- MySQL is one of the most popular RDBMS(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
- Orders of magnitude faster

Consistency levels in Cassandra

- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator caches write and replies quickly to client
  - ALL: all replicas
    - Ensures strong consistency, but slowest
  - ONE: at least one replica
    - Faster than ALL
  - QUORUM: quorum across all replicas in all datacenters (DCs)
    - Global consistency, but still fast
  - LOCAL_QUORUM: quorum in coordinator's DC
    - Faster: only waits for quorum in the first DC the client contacts
  - EACH_QUORUM: quorum in every DC
    - Lets each DC do its own quorum: supports hierarchical replies

Quorum in detail

- Reads
  - Client specifies value of R ($\leq$ N = total number of replicas of that key).
  - R = read consistency level.
  - Coordinator waits for R replicas to respond before sending result to client.
  - In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed.

Quorum in detail

- Writes come in two flavors
  - Client specifies W (≤ N)
  - W = write consistency level.
  - Client writes new value to W replicas and returns. Two flavors:
    - Coordinator blocks until quorum is reached.
    - Asynchronous: push the data to appropriate nodes but return immediately

Reading

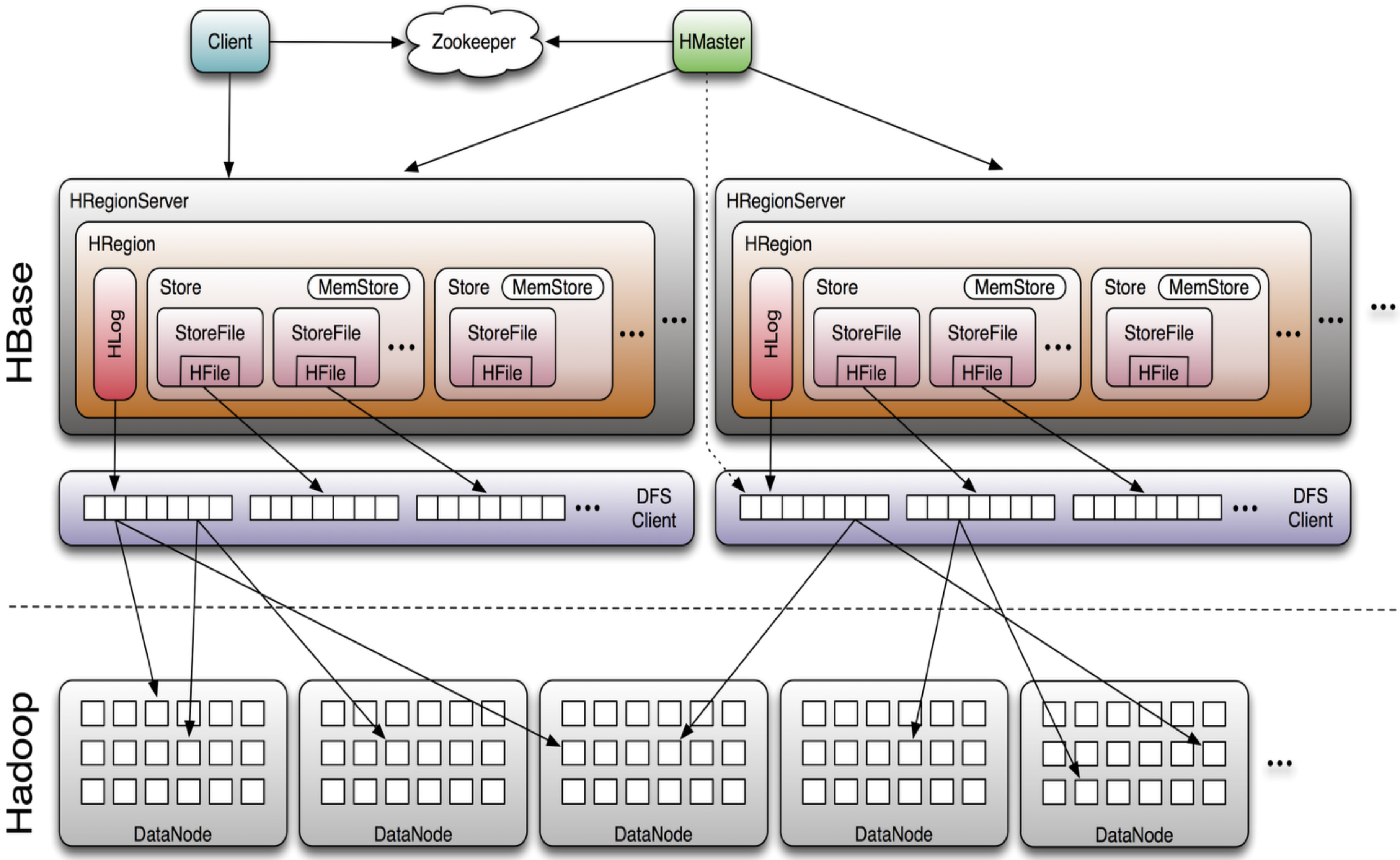
- Cassandra
- https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf
- http://www.slideshare.net/Eweaver/cassandra-presentation-at-nosql
- http://cassandra.apache.org/doc/latest/architecture/index.html

HBase

HBase

- BigTable is a distributed system for storing structured data at Google
- Yahoo! Open-sourced it → HBase
- Major Apache project today
- Facebook uses HBase internally
- API functions
  - Get/Put(row)
  - Scan(row range, filter) – range queries
  - MultiAction for batch processing
- Unlike Cassandra, HBase prefers consistency (over availability)

HBase architecture



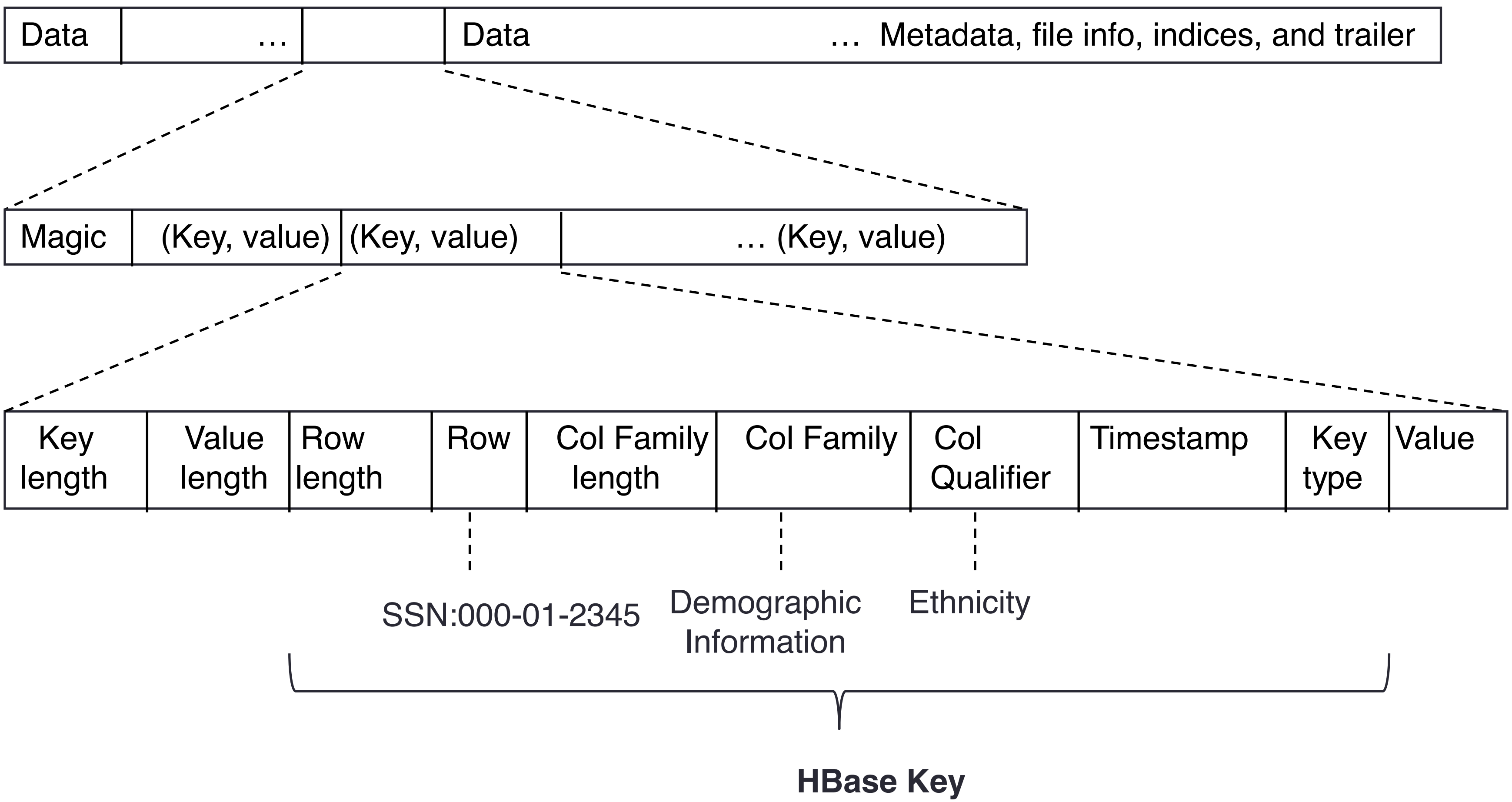
HBase storage hierarchy

- HBase Table
  - Split it into multiple regions:
    - a region is only served by one single regionserver
    - a row can only belong to one region

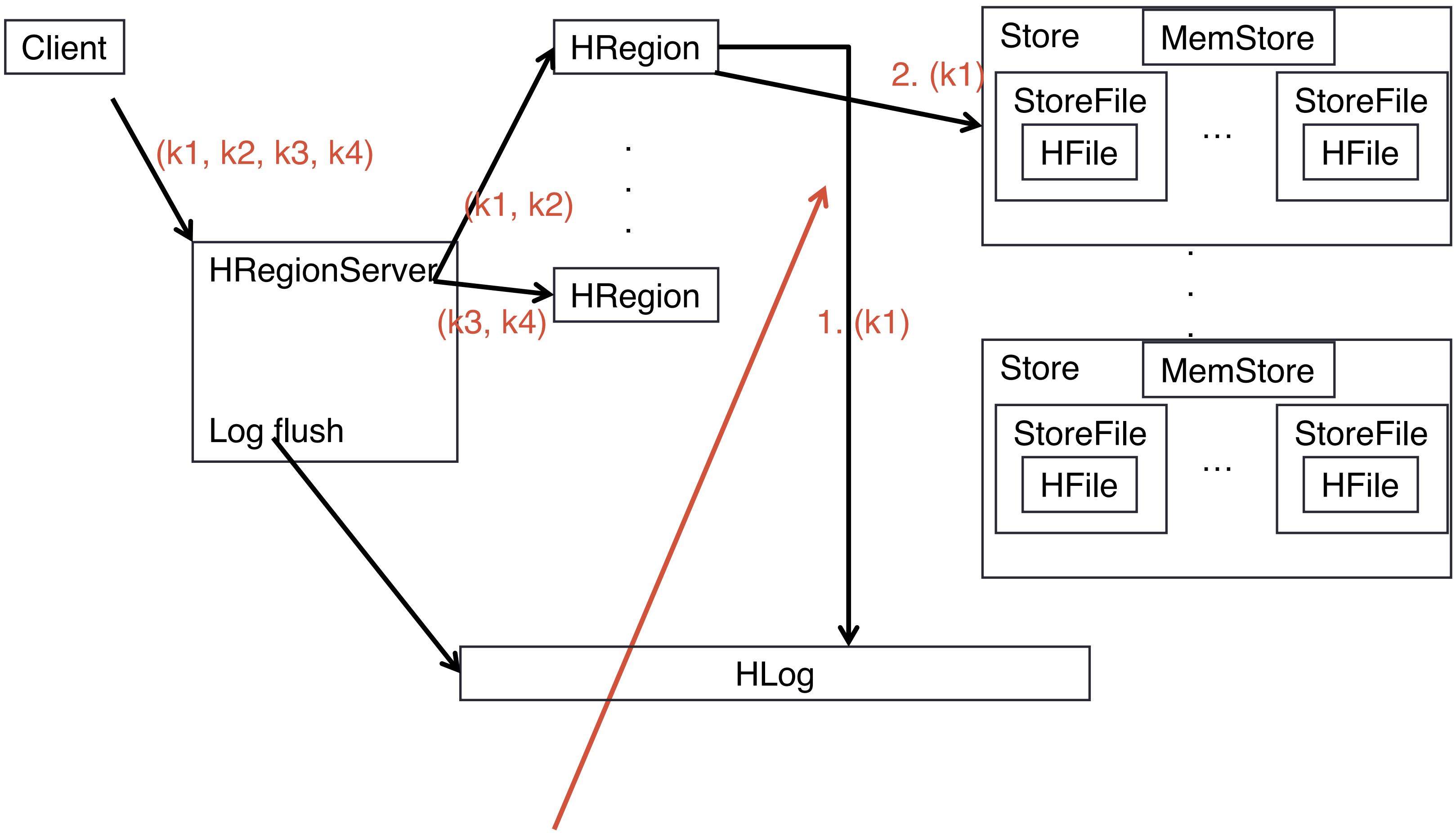
    - ColumnFamily = subset of columns with similar query patterns
    - One Store per combination of ColumnFamily + region
      - Memstore for each store: in-memory updates to Store; flushed to disk when full
      - StoreFiles for each store for each region: where the data lives

        - also called HFile
- HFile
  - SSTable from Google's BigTable

HFile

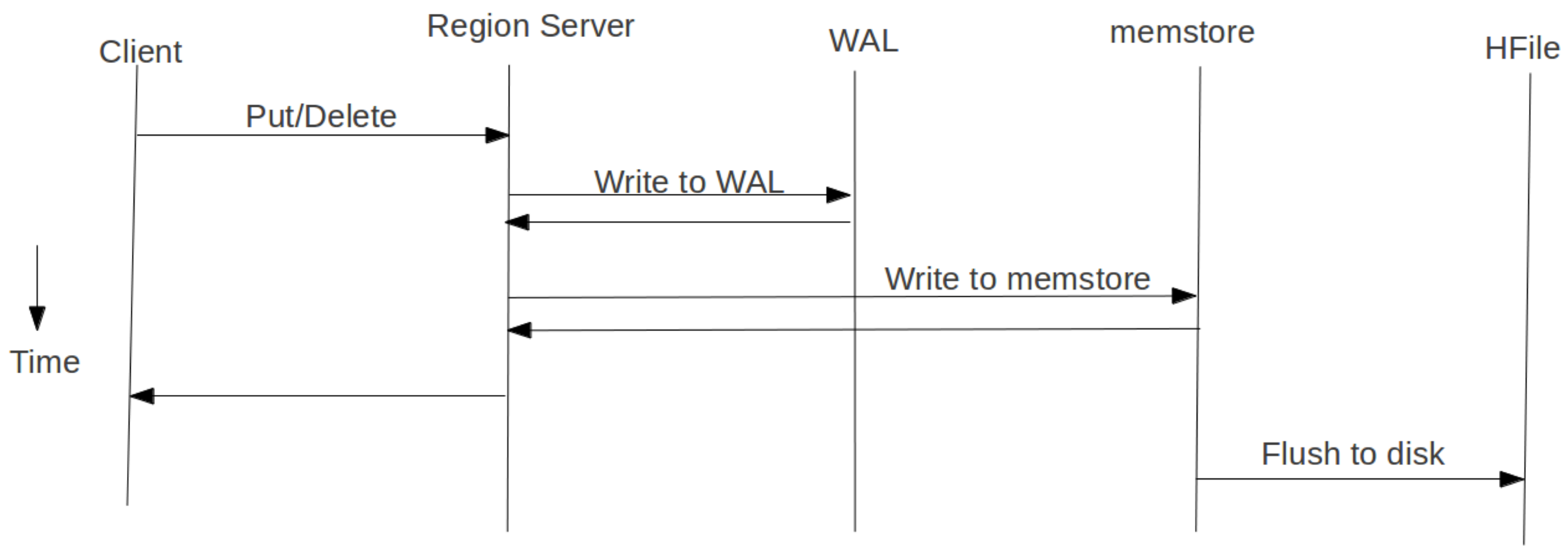

Data
…
Data
… Metadata, file info, indices, and trailer
Magic
(Key, value)
(Key, value)
… (Key, value)
Key length
Value length
Row length
Row
Col Family length
Col Family
Col Qualifier
Timestamp
Key type
Value
SSN:000-01-2345
Demographic Information
Ethnicity
HBase Key

Strong consistency: HBase write-ahead log (WAL)



Write to HLog before writing to MemStore
Helps recover from failure by replaying HLog.

HBase write path



HBase Write Path

When a regionserver fails…

- The regions immediately become <u>unavailable</u> because the RegionServer is down.
- The Master will detect that the RegionServer has failed.
- The region assignments will be considered invalid and will be re-assigned just like boot up.

Log replay

- After recovery from failure, or upon boot up (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

Reading

- BigTable:
- http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf
- HBase:
- http://hbase.apache.org/book.html

MongoDB

Data model

- Stores data in form of BSON (Binary JavaScript Object Notation) *documents*

```
{
    name: "travis",
    salary: 30000,
    designation: "Computer Scientist",
    teams: [ "front-end",  "database" ]
}
```

- Group of related *documents* with a shared common index is a *collection*

MongoDB: typical query

- Query all employee names with salary greater than 18000 sorted in ascending order
- $\underbrace{\texttt{db.employee}}_{\text{collection}}\texttt{.find(}\underbrace{\texttt{\{salary:\{\$gt:18000\}}}_{\text{condition}}\texttt{, }\underbrace{\texttt{\{name:1\}}}_{\text{projection}}\texttt{\}).sort(}\underbrace{\texttt{\{salary:1\}}}_{\text{modifier}}\texttt{)}$

{salary:25000, …}
{salary:10000, …}
{salary:20000, …}
{salary:2000, …}
{salary:30000, …}
{salary:21000, …}
{salary:5000, …}
{salary:50000, …}

{salary:25000, …}
{salary:20000, …}
{salary:30000, …}
{salary:21000, …}
{salary:50000, …}

{salary:20000, …}
{salary:21000, …}
{salary:25000, …}
{salary:30000, …}
{salary:50000, …}

Insert

Insert a row entry for new employee Sally

```
db.employee.insert({
        name: "sally",
        salary: 15000,
        designation: "MTS",
        teams: [ "cluster-management" ]
        })
```

Update

All employees with salary greater than 18000 get a designation of Manager

	db.employee.update(
Update Criteria	{salary:{$gt:18000}},
Update Action	{$set: {designation: "Manager"}},
Update Option	{multi: true}
	)

Multi-option allows multiple document update

Delete

Remove all employees who earn less than 10000

Remove Criteria

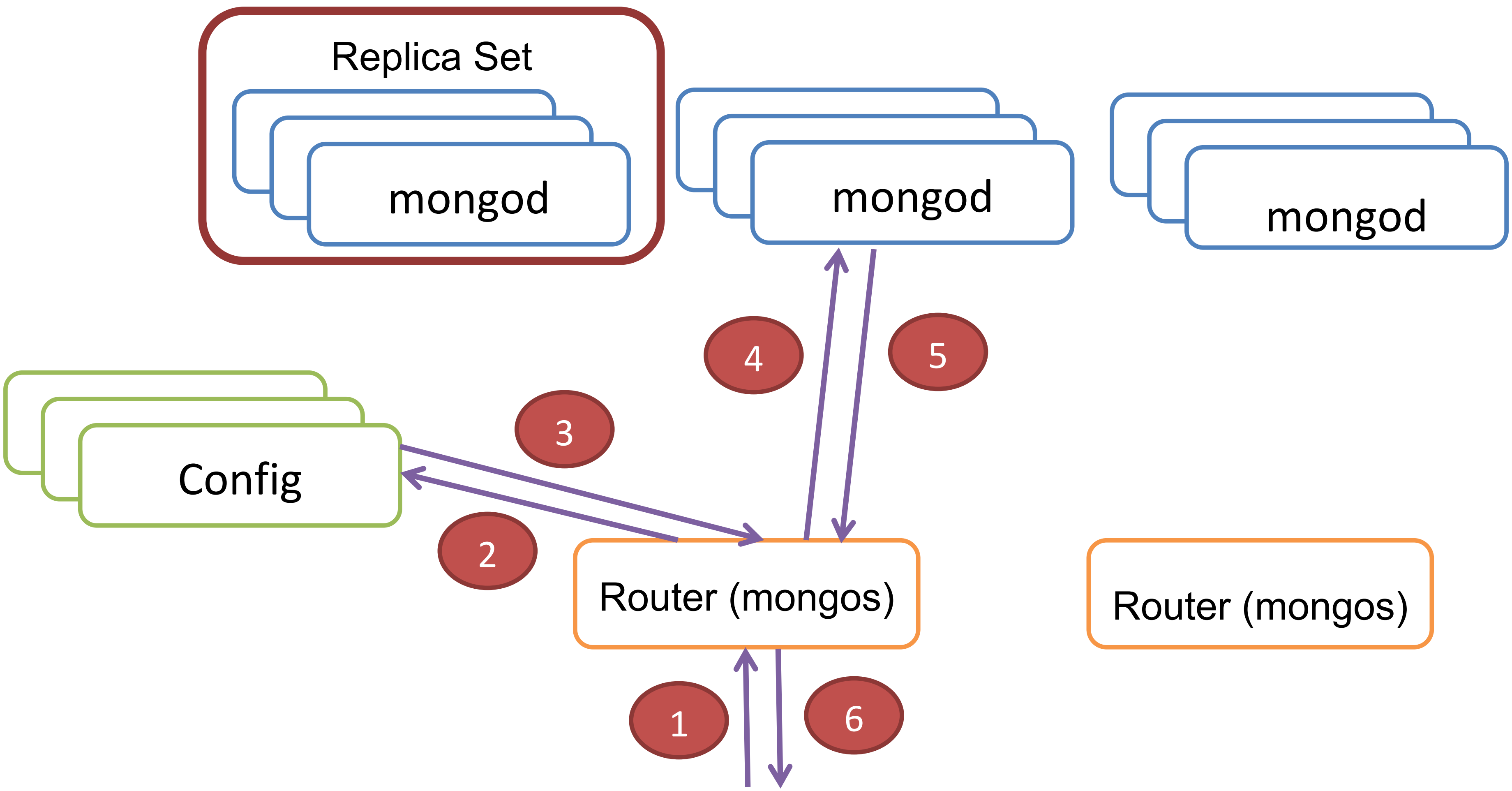
```
db.employee.remove(
        {salary:{$lt:10000}},
)
```

Can accept a flag to limit the number of documents removed

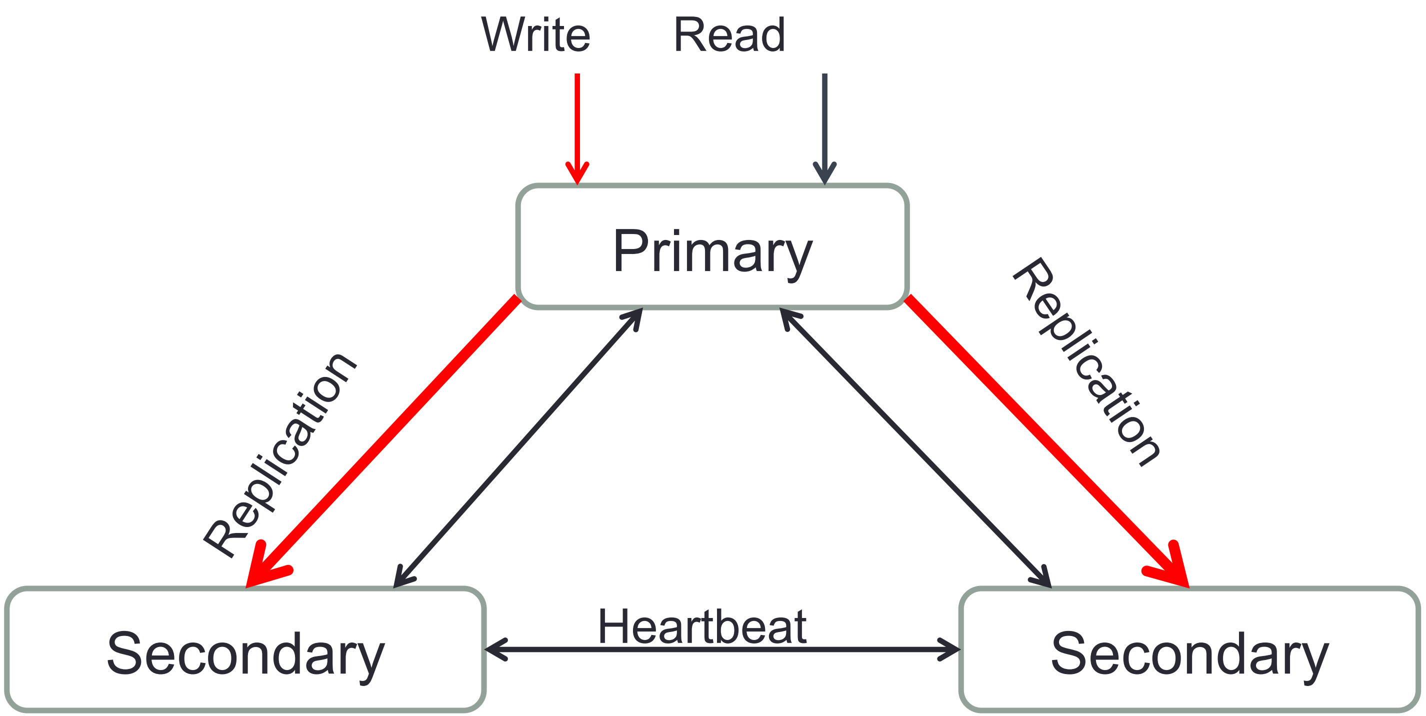
Typical MongoDB deployment

- Data split into chunks, based on shard key (~ primary key)
  - Either use hash or range-partitioning
- Shard: collection of chunks
- Shard assigned to a replica set
- Replica set consists of multiple mongod servers (typically 3 mongod’s)
- Replica set members are mirrors of each other
  - One is primary
  - Others are secondaries
- Routers: mongos server receives client queries and routes them to right replica set
- Config server: stores collection level metadata

Typical MongoDB deployment



Replication



Replication

- Uses an oplog (operation log) for data sync
  - Oplog maintained at primary, delta transferred to secondary continuously/every once in a while
- When needed, leader election protocol elects a master
- Some mongod servers do not maintain data but can vote – called as Arbiters

Read preference

- Determine where to route read operation
- Default is primary. Some other options are
  - primary-preferred
  - secondary
  - nearest
- Helps reduce latency, improve throughput
- Reads from secondary may fetch stale data

Write concern

- Determines the guarantee that MongoDB provides on the success of a write operation
- Default is *acknowledged* (primary returns answer immediately).
  - Other options are
    - journaled (typically at primary)
    - replica-acknowledged (quorum with a value of W), etc.
- Weaker write concern implies faster write time

Write operation performance

- Journaling: Write-ahead logging to an on-disk journal for durability
- Indexing: Every write needs to update every index associated with the collection

Balancing

- Over time, some chunks may get larger than others
- Splitting: Upper bound on chunk size; when hit, chunk is split
- Balancing: Migrates chunks among shards if there is an uneven distribution

Consistency

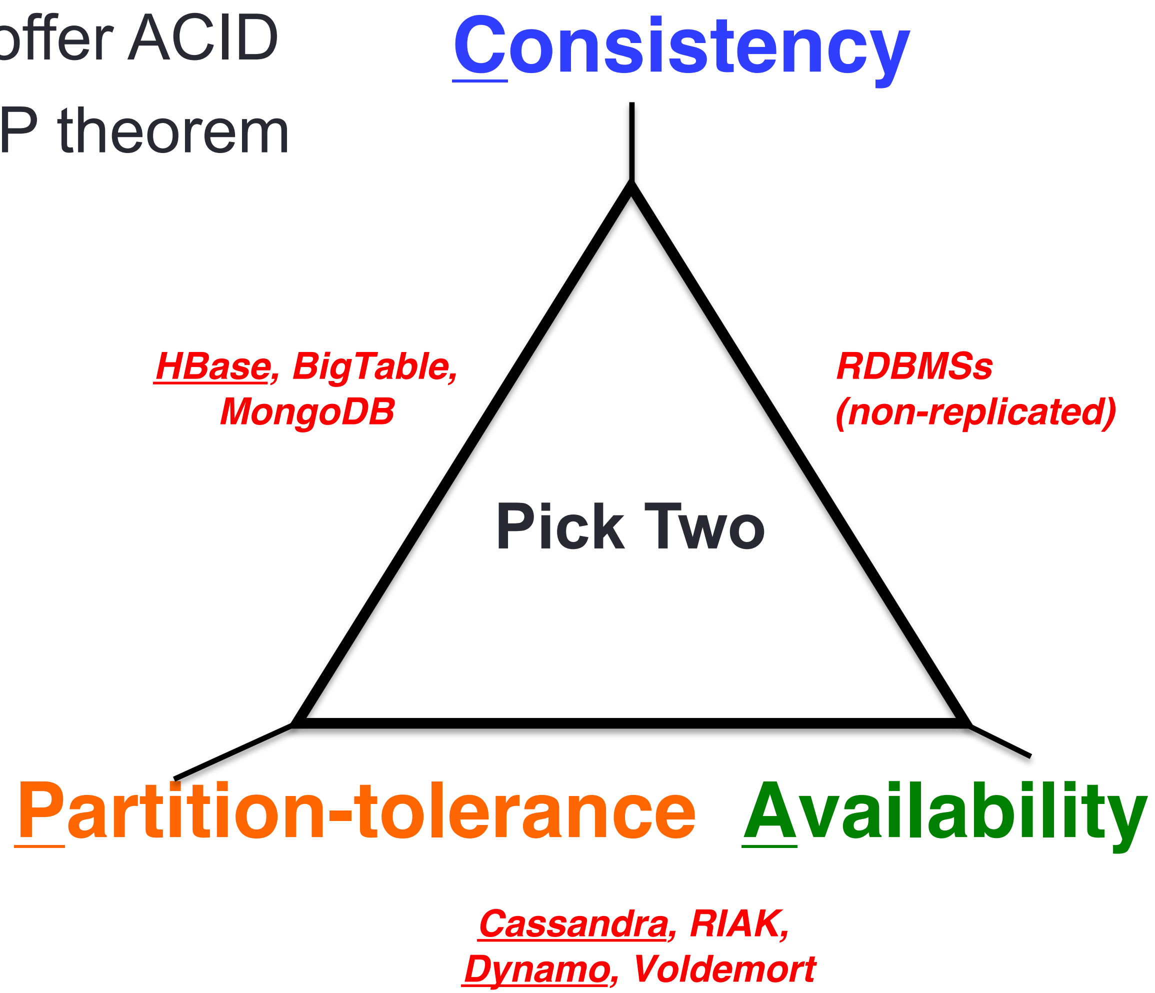
- Strongly Consistent: Read Preference is Master
- Eventually Consistent: Read Preference is Slave (Secondary)
- CAP Theorem: With strong consistency, under partition, MongoDB becomes write-unavailable thereby ensuring consistency

Reading

- https://docs.mongodb.org/manual/

Summary

- Traditional Databases (RDBMSs) work with strong consistency, and offer ACID
- Unfortunately, CAP theorem



Acknowledgement

- These slides contain material developed by Indranil Gupta (UIUC), Steve Ko (Buffalo), and Mainak Ghosh (UIUC).