CS 425 / ECE 428 Distributed Systems Fall 2015

Indranil Gupta (Indy)

Oct 27-Nov 3

Lecture 19-21: Key-value/NoSQL Stores

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 (2)

- 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.

Isn't that just a database?

- Yes, sort of
- 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

Relational Database Example

users table

user_id	name	zipcode	blog_url	blog_id
101	Alice	12345	alice.net	1
422	Charlie	45783	charlie.com	3
555	Bob	99910	bob.blogspot.com	2

Primary keys

Foreign keys

blog table

id	url	last_updated	num_posts
1	alice.net	5/2/14	332
2	bob.blogspot.com	4/2/13	10003
3	charlie.com	6/15/14	7

Example SQL queries

1. SELECT zipcode
   FROM users
   WHERE name = “Bob”

2. SELECT url
   FROM blog
   WHERE id = 3

3. SELECT users.zipcode, blog.num_posts
   FROM users JOIN blog
   ON users.blog_url = blog.url

Mismatch with 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 (SPoF)
- Low TCO (Total cost of operation)
- Fewer system administrators
- Incremental Scalability
- Scale out, not up
  - What?

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

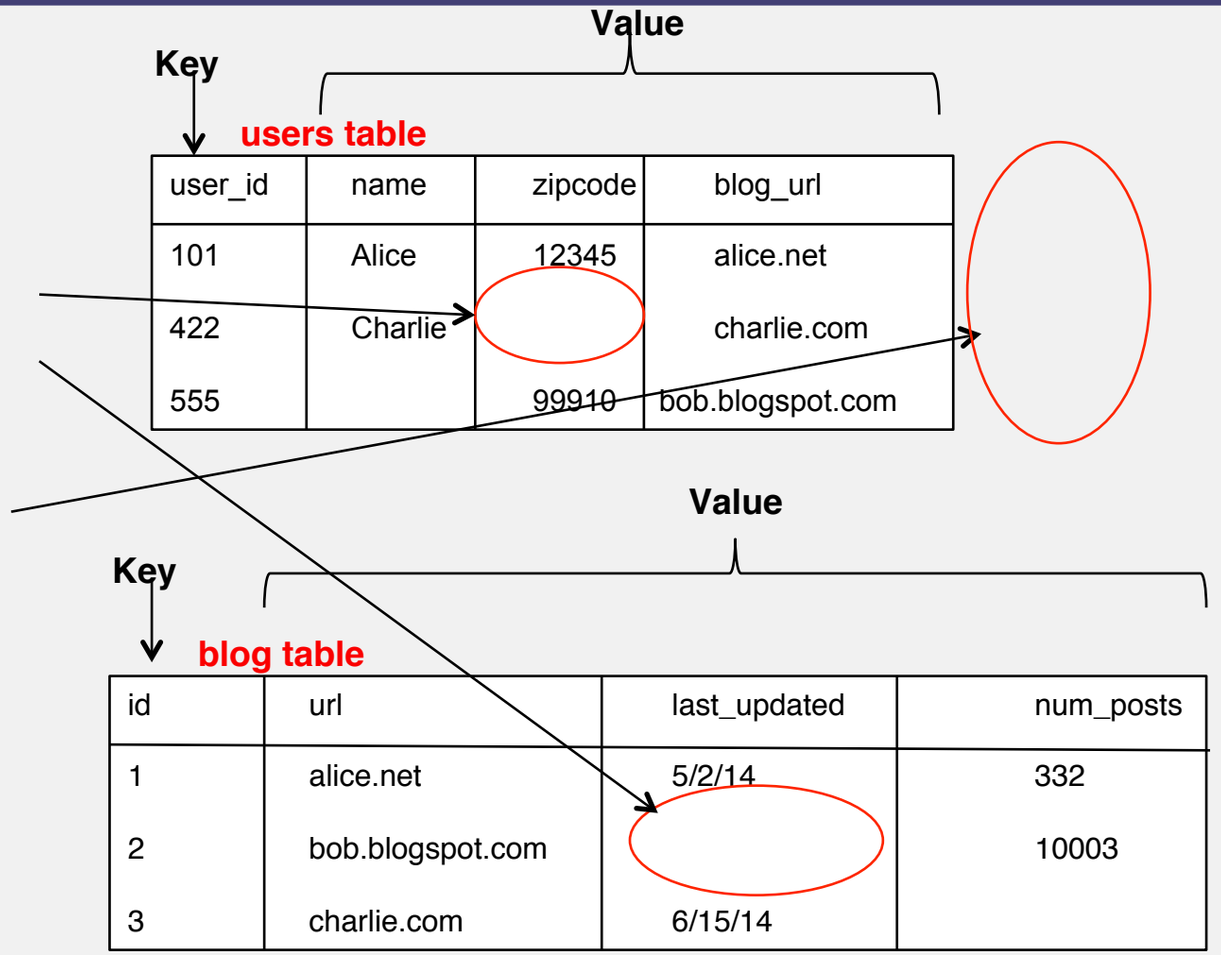
Key-value/NoSQL Data Model

- NoSQL = "Not Only SQL"
- Necessary API operations: get(key) and put(key, value)
  - And some extended operations, e.g., "CQL" in Cassandra key-value store

- Tables
  - "Column families" in Cassandra, "Table" in HBase, "Collection" in MongoDB
  - Like RDBMS tables, but …
  - May be unstructured: May not have schemas
    - Some columns may be missing from some rows
  - Don't always support joins or have foreign keys
  - Can have index tables, just like RDBMSs

Key-value/NoSQL Data Model

- Unstructured
- Columns Missing from some Rows
- No schema imposed
- No foreign keys, joins may not be supported



Column-Oriented Storage

NoSQL systems often use 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).
  - Entries within a column are indexed and easy to locate, given a key (and vice-versa)
- 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

Next

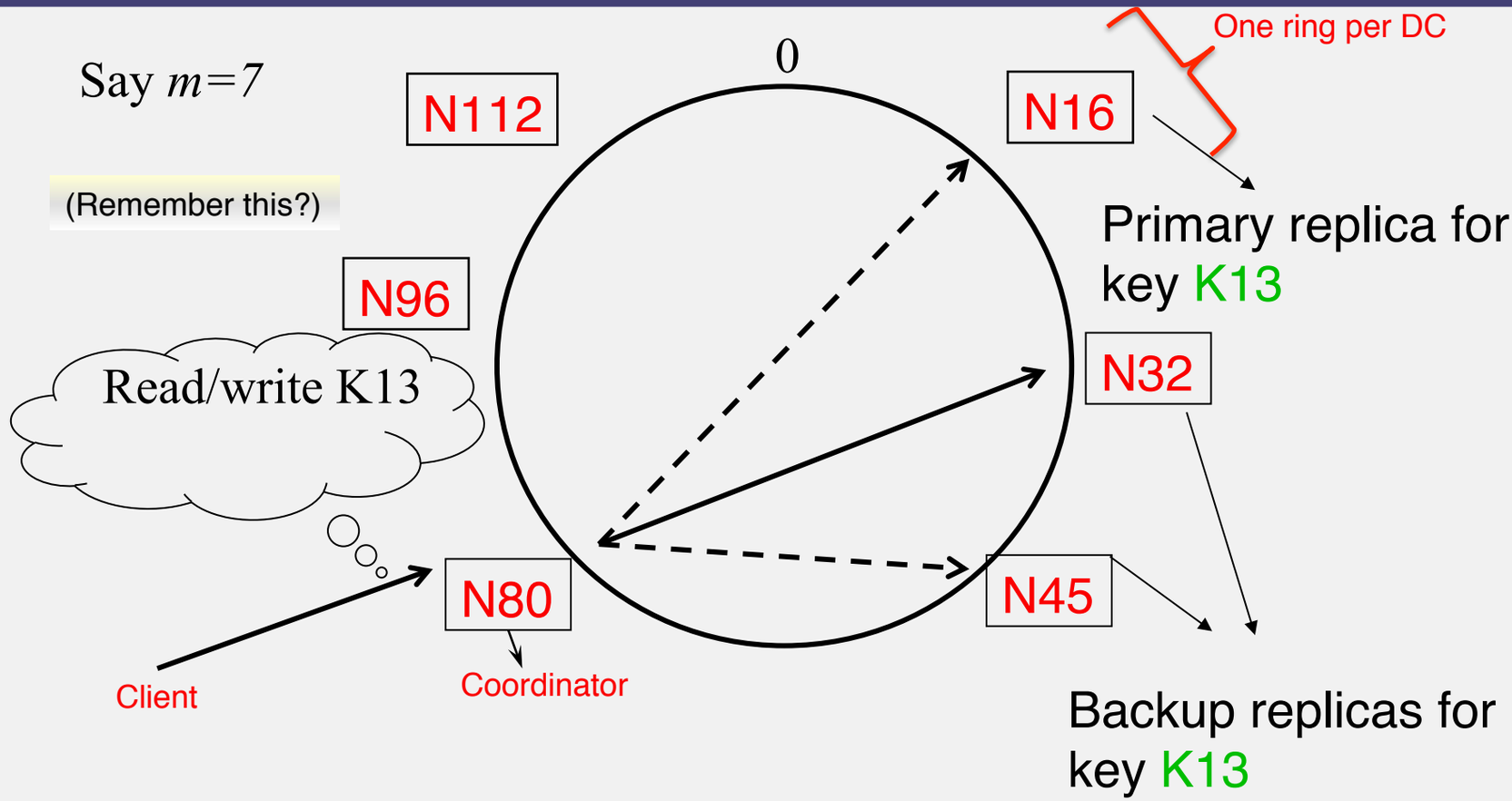
Design of a real key-value store, Cassandra.

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
  - PBS Kids
  - Netflix: uses Cassandra to keep track of your current position in the video you're watching

Let's go Inside Cassandra: Key -> Server Mapping

- How do you decide which server(s) a key-value resides on?



Cassandra uses a Ring-based DHT but without finger tables or routing

Key→server mapping is the "Partitioner"

Data Placement Strategies

- Replication Strategy: two options:
  1. *SimpleStrategy*
  2. *NetworkTopologyStrategy*

1. SimpleStrategy: uses the Partitioner, of which there are two kinds
   1. *RandomPartitioner*: Chord-like hash partitioning
   2. *ByteOrderedPartitioner*: Assigns ranges of keys to servers.
      - Easier for *range queries* (e.g., Get me all twitter users starting with [a-b])
2. NetworkTopologyStrategy: for multi-DC deployments
   - Two replicas per DC
   - Three replicas per DC
   - Per DC
     - First replica placed according to Partitioner
     - Then go clockwise around ring until you hit a different rack

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 EC2.
    - EC2 Region = DC
    - Availability zone = rack
- Other snitch options available

Writes

- Need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one coordinator node in Cassandra cluster
  - Coordinator may be per-key, or per-client, or per-query
  - Per-key Coordinator ensures writes for the key are serialized
- 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.

Writes (2)

- 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).
- One ring per datacenter
  - Per-DC coordinator elected to coordinate with other DCs
  - Election done via Zookeeper, which runs a Paxos (consensus) variant
    - Paxos: elsewhere in this course

Writes at a replica node

On receiving a write

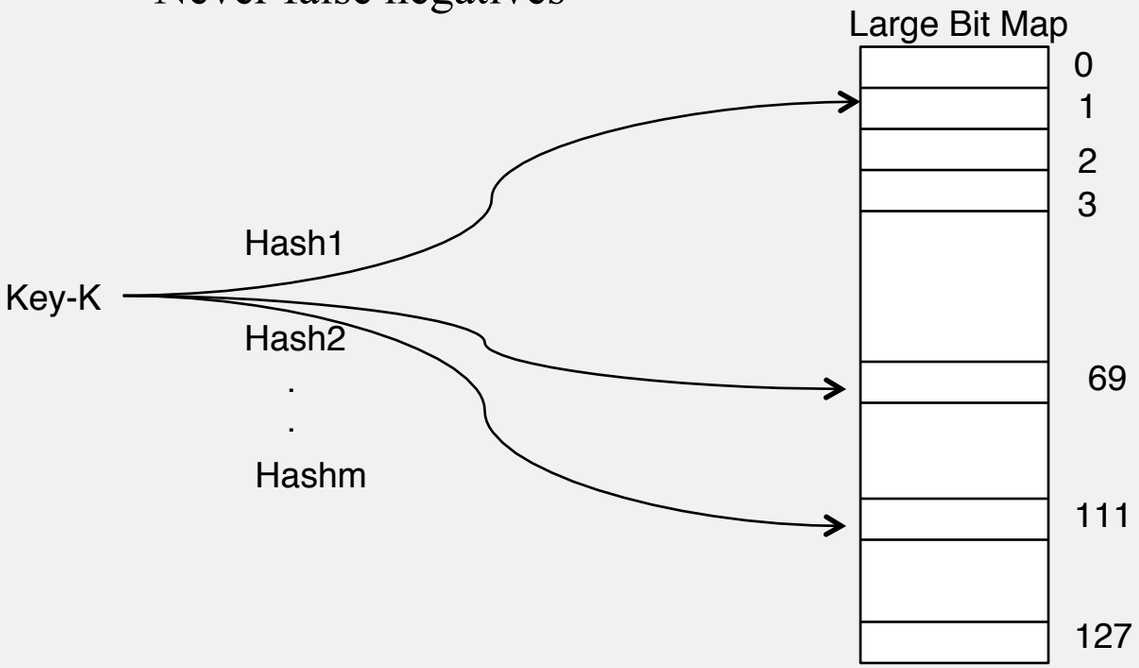
1. Log it in disk commit log (for failure recovery)
2. Make changes to appropriate memtables
    - **Memtable** = In-memory representation of multiple key-value pairs
    - *Typically append-only datastructure (fast)*
    - Cache that can be searched by key
    - Write-back cache as opposed to write-through

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) – next slide

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

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

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

Deletes

Delete: don’t delete item right away

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

Reads

Read: Similar to writes, except

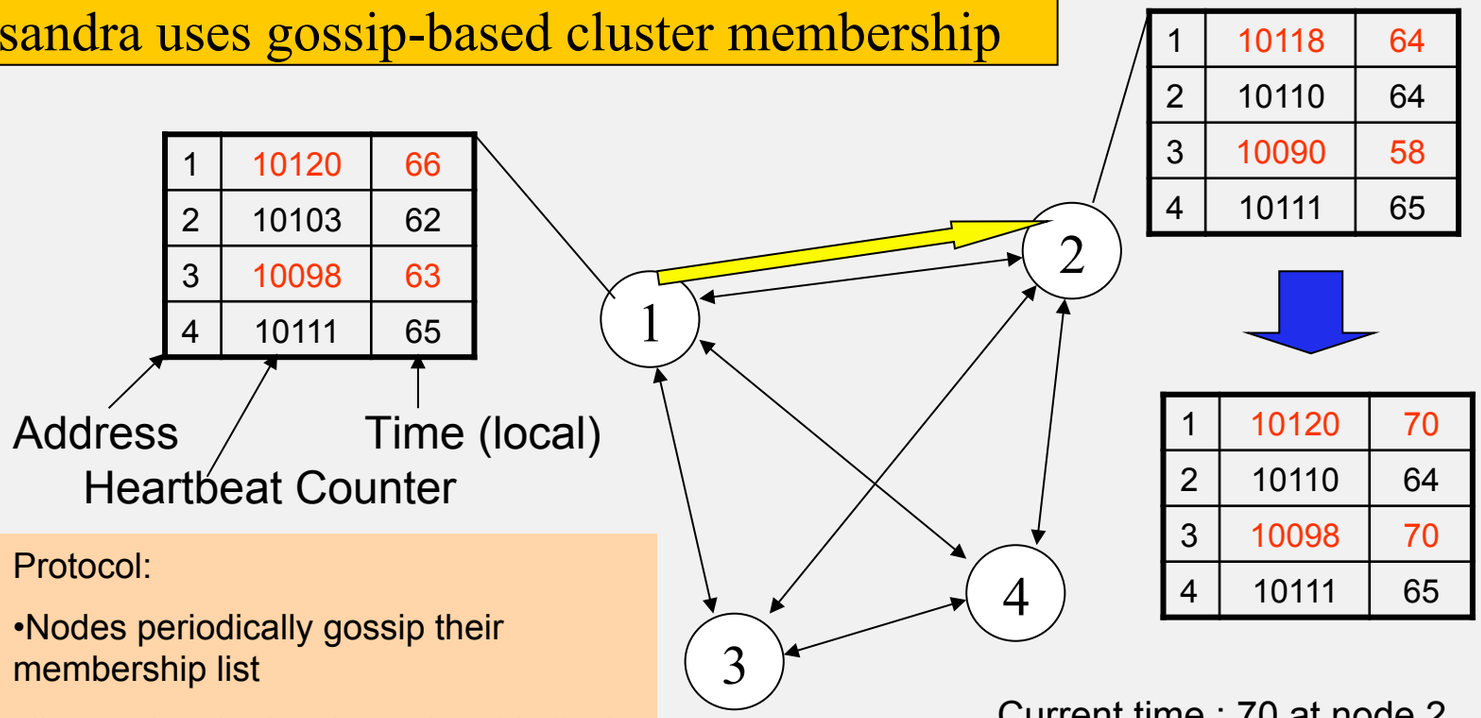
- Coordinator can contact X replicas (e.g., in same rack)
  - Coordinator sends read to replicas that have responded quickest in past
  - When X replicas respond, coordinator returns the latest-timestamped value from among those X
  - (X? We'll see later.)
- Coordinator also fetches value from other 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 server
- List needs to be updated automatically as servers join, leave, and fail

Cluster Membership – Gossip-Style

Cassandra uses gossip-based cluster membership



Protocol:

- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated, as shown
- If any heartbeat older than Tfail, node is marked as failed

Current time : 70 at node 2

(asynchronous clocks)

(Remember this?)

Suspicion Mechanisms in Cassandra

- Suspicion mechanisms to adaptively set the timeout based on underlying network and failure behavior
- Accrual detector: Failure Detector outputs a value (PHI) representing suspicion
- Apps set an appropriate threshold
- 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 takes into account historical inter-arrival time variations for gossiped heartbeats
- In practice, PHI = 5 => 10-15 sec detection time

Cassandra Vs. RDBMS

- MySQL is one of 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
- Orders of magnitude faster
- What's the catch? What did we lose?

Mystery of "X": 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

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.
- At Amazon, each added millisecond of latency implies a $6M yearly loss.
- User cognitive drift: If more than a second elapses between clicking and material appearing, the user's mind is already somewhere else
- SLAs (Service Level Agreements) written by providers predominantly deal with latencies faced by clients.

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

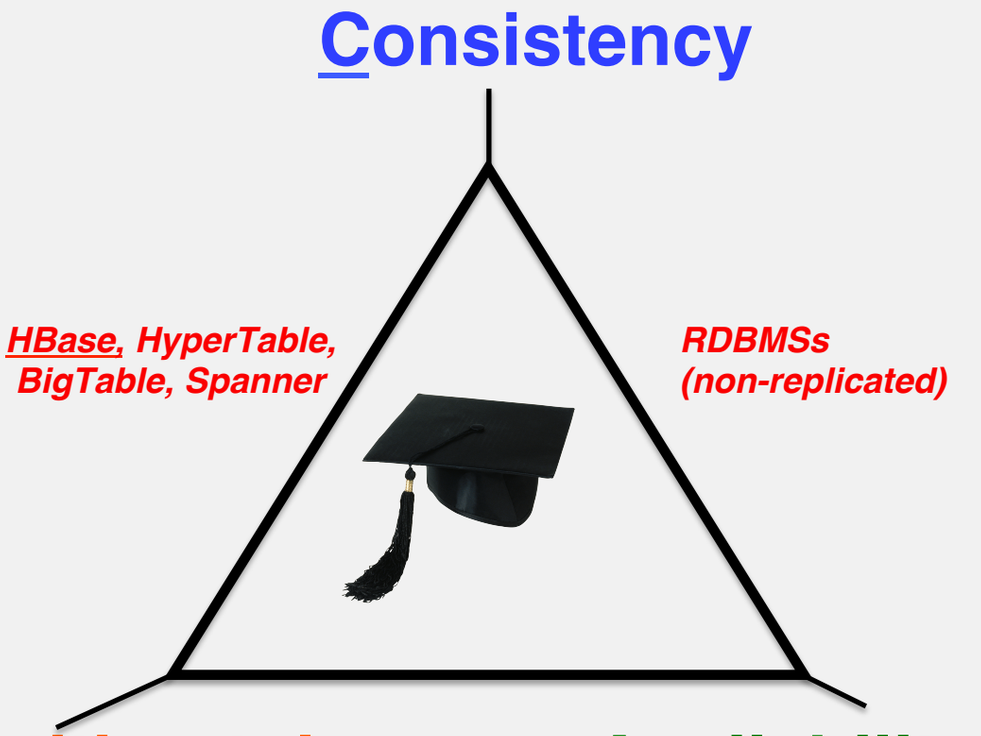
CAP Theorem Fallout

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

- Cassandra
  - Eventual (weak) consistency, Availability, Partition-tolerance
- Traditional RDBMSs
  - Strong consistency over availability under a partition

CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve at most two of C, A, and P.
- When partition-tolerance is important, you have to choose between consistency and availability



Eventual Consistency

- If all writes 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.
- May still return stale values to clients (e.g., if many back-to-back writes).
- But works well when there a few periods of low writes – system converges quickly.

RDBMS vs. Key-value stores

- While RDBMS provide ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Key-value stores like Cassandra provide BASE
  - Basically Available Soft-state Eventual Consistency
  - Prefers Availability over Consistency

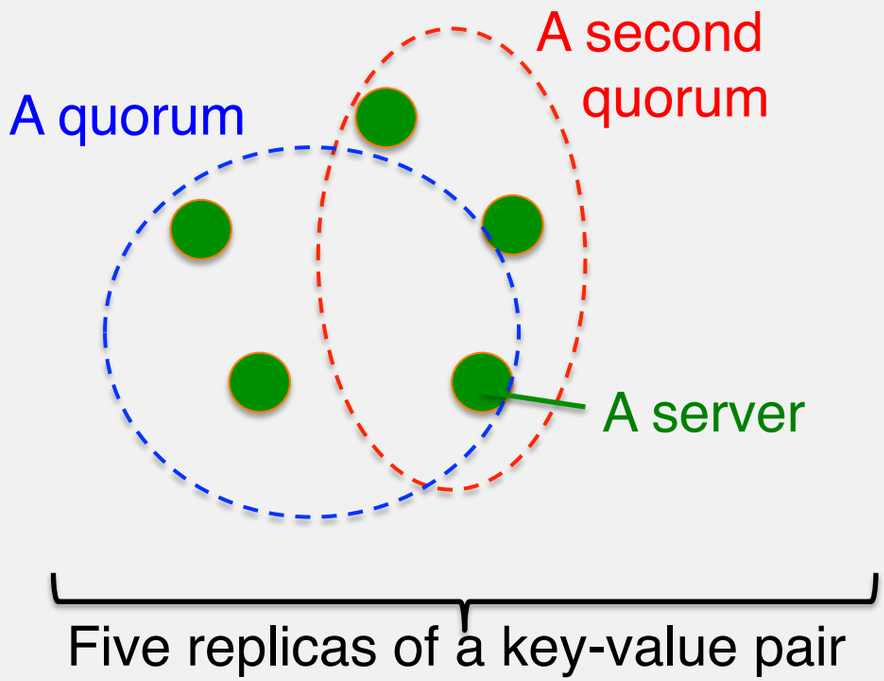
Back to Cassandra: Mystery of X

- Cassandra has consistency levels
- 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, but cannot tolerate a failure
  - QUORUM: quorum across all replicas in all datacenters (DCs)
    - What?

Quorums?

In a nutshell:

- Quorum = majority
  - > 50%
- Any two quorums intersect
  - Client 1 does a write in red quorum
  - Then client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums faster than ALL, but still ensure strong consistency



Quorums in Detail

- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.
- Reads
  - Client specifies value of R (≤ 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.

Quorums in Detail (Contd.)

- 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: Just write and return.

Quorums in Detail (Contd.)

- R = read replica count, W = write replica count
- Two necessary conditions:
  1. W+R > N
  2. W > N/2
- Select values based on application
  - (W=1, R=1): very few writes and reads
  - (W=N, R=1): great for read-heavy workloads
  - (W=N/2+1, R=N/2+1): great for write-heavy workloads
  - (W=1, R=N): great for write-heavy workloads with mostly one client writing per key

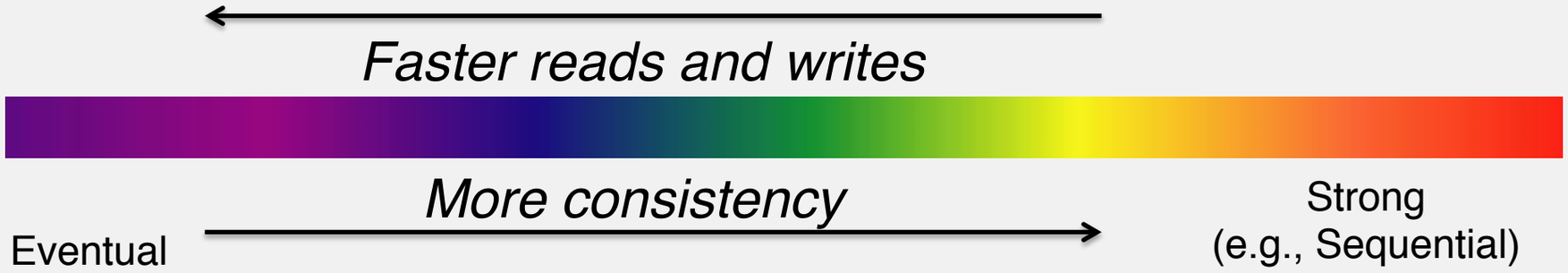
Cassandra Consistency Levels (Contd.)

- Client is allowed to choose a consistency level for each operation (read/ write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator may cache write and reply quickly to client
  - ALL: all replicas
    - Slowest, but ensures strong consistency
  - ONE: at least one replica
    - Faster than ALL, and ensures durability without failures
  - 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 first DC client contacts
  - EACH_QUORUM: quorum in every DC
    - Lets each DC do its own quorum: supports hierarchical replies

Types of Consistency

- Cassandra offers Eventual Consistency
- Are there other types of weak consistency models?

Consistency Spectrum



Spectrum Ends: Eventual Consistency

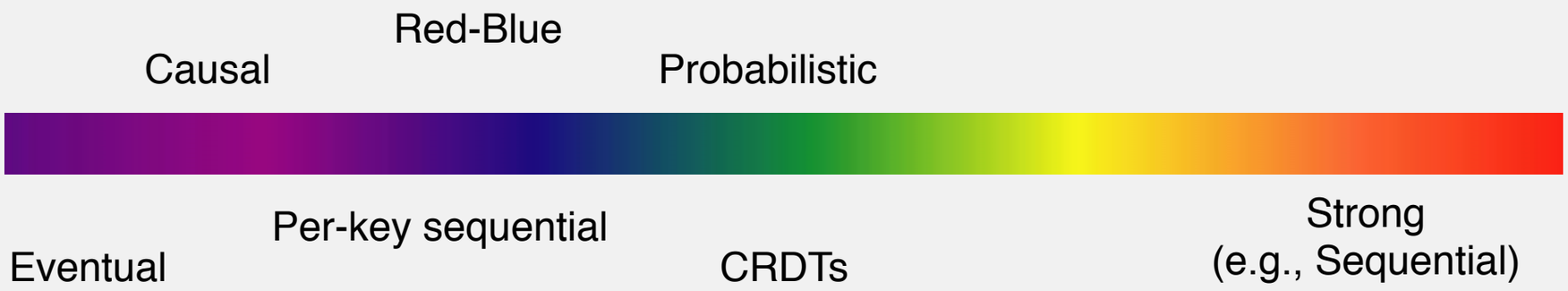
- Cassandra offers Eventual Consistency
  - If writes to a key stop, all replicas of key will converge
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems

Spectrum Ends: Strong Consistency Models

- **Linearizability**: Each operation by a client is visible (or available) instantaneously to all other clients
  - Instantaneously in real time
- **Sequential Consistency** [Lamport]:
  - *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
  - After the fact, find a "reasonable" ordering of the operations (can re-order operations) that obeys sanity (consistency) at all clients, and across clients.
- Transaction ACID properties, e.g., newer key-value/NoSQL stores (sometimes called "NewSQL")
  - Hyperdex [Cornell]
  - Spanner [Google]
  - Yesquel [Microsoft Research]

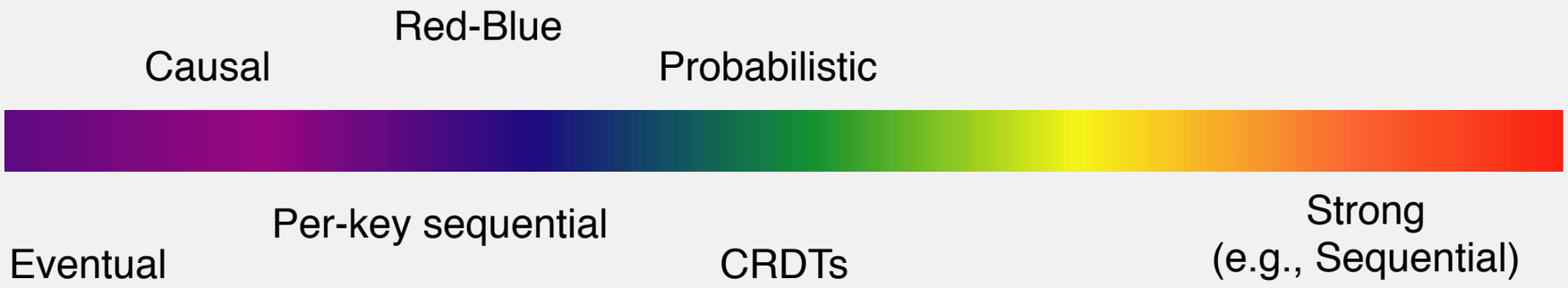
Newer Consistency Models

- Striving towards strong consistency
- While still trying to maintain high availability and partition-tolerance



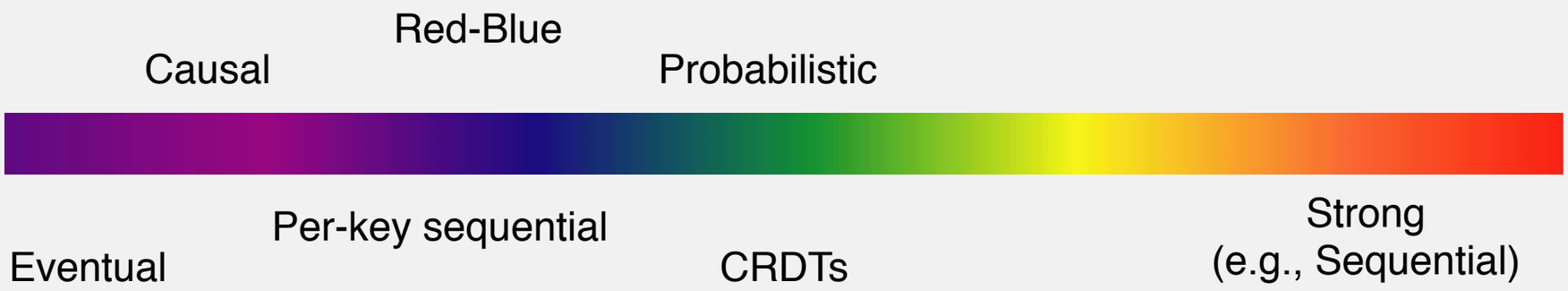
Newer Consistency Models (Contd.)

- **Per-key sequential**: Per key, all operations have a global order
- **CRDTs** (Commutative Replicated Data Types): Data structures for which commutated writes give same result [INRIA, France]
  - E.g., value == int, and only op allowed is +1
  - Effectively, servers don't need to worry about consistency



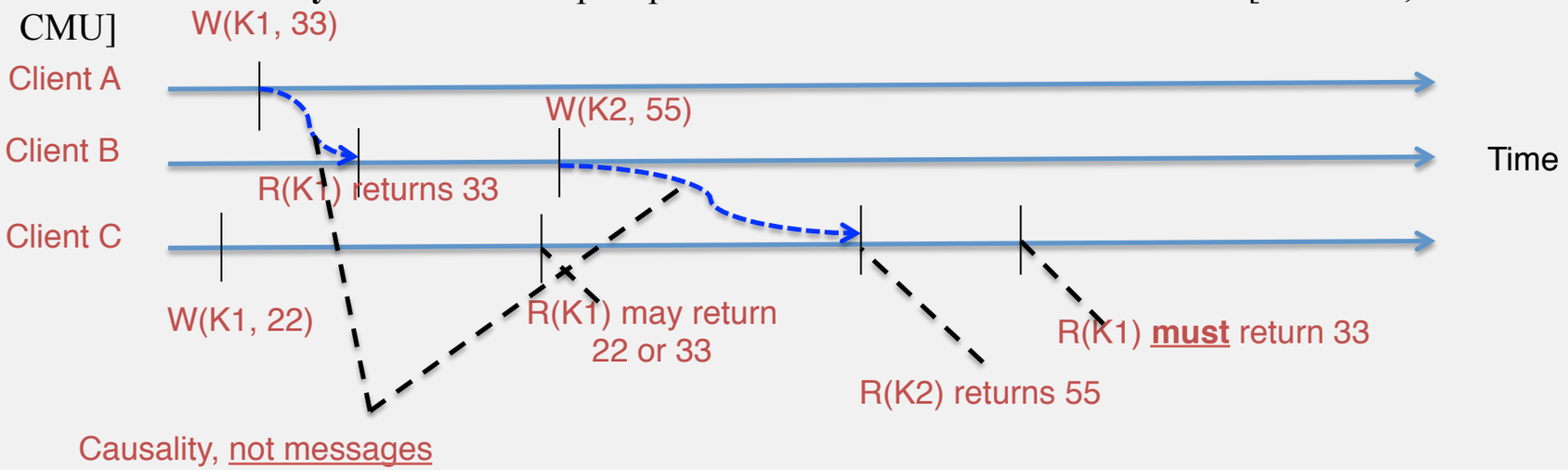
Newer Consistency Models (Contd.)

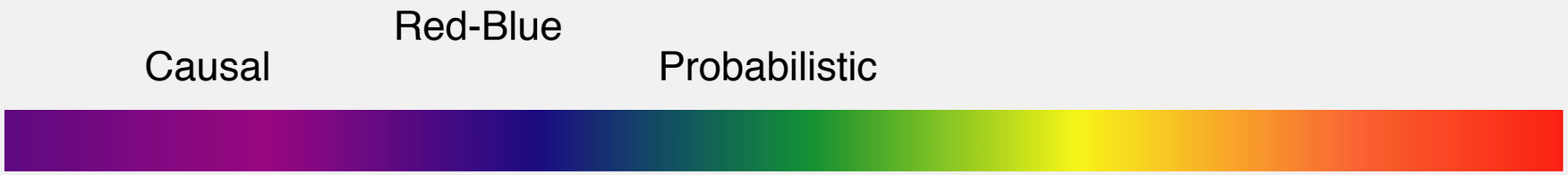
- **Red-blue Consistency**: Rewrite client transactions to separate ops into red ops vs. blue ops [MPI-SWS Germany]
  - Blue ops can be executed (commutated) in any order across DCs
  - Red ops need to be executed in the same order at each DC



Newer Consistency Models (Contd.)

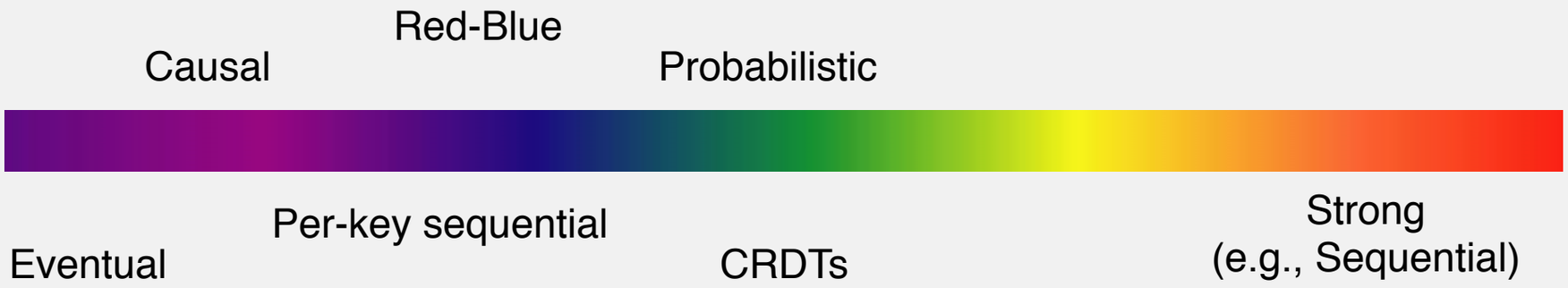
Causal Consistency: Reads must respect partial order based on information flow [Princeton, CMU]





Which Consistency Model should you use?

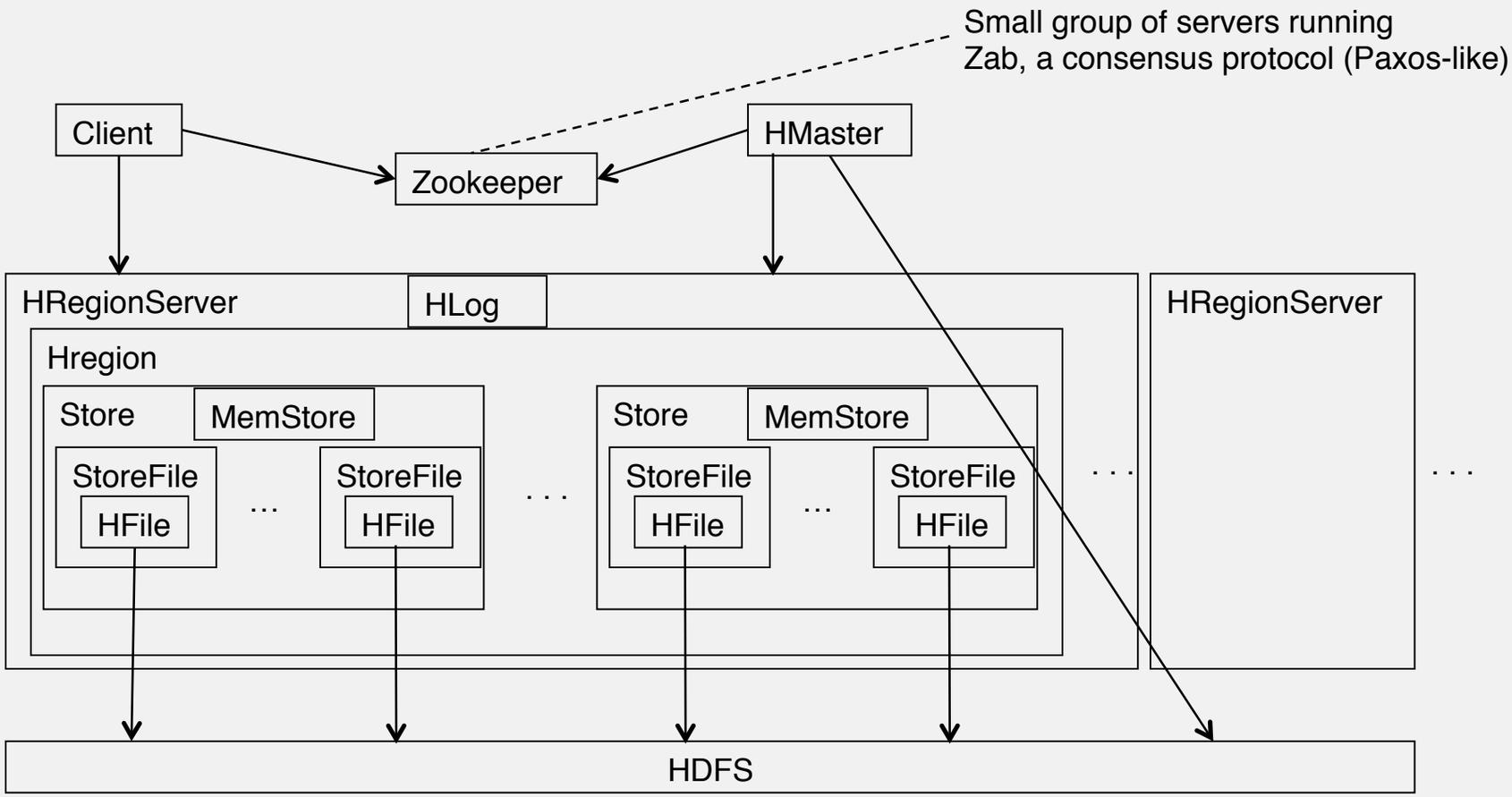
- Use the lowest consistency (to the left) consistency model that is “correct” for your application
  - Gets you fastest availability



HBase

- Google's BigTable was first "blob-based" storage system
- Yahoo! Open-sourced it → HBase
- Major Apache project today
- Facebook uses HBase internally
- API functions
  - Get/Put(row)
  - Scan(row range, filter) – range queries
  - MultiPut
- Unlike Cassandra, HBase prefers consistency (over availability)

HBase Architecture

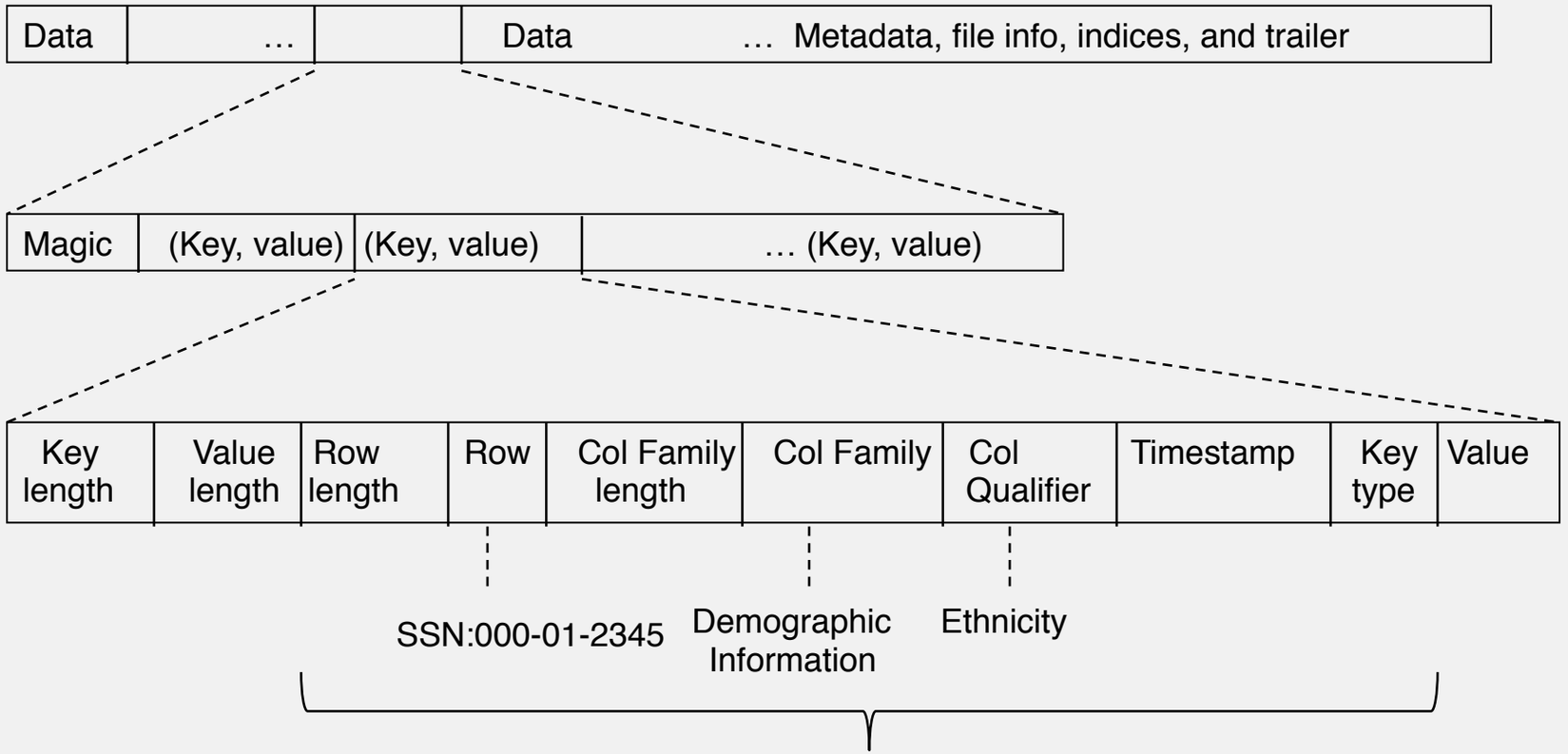


HBase Storage hierarchy

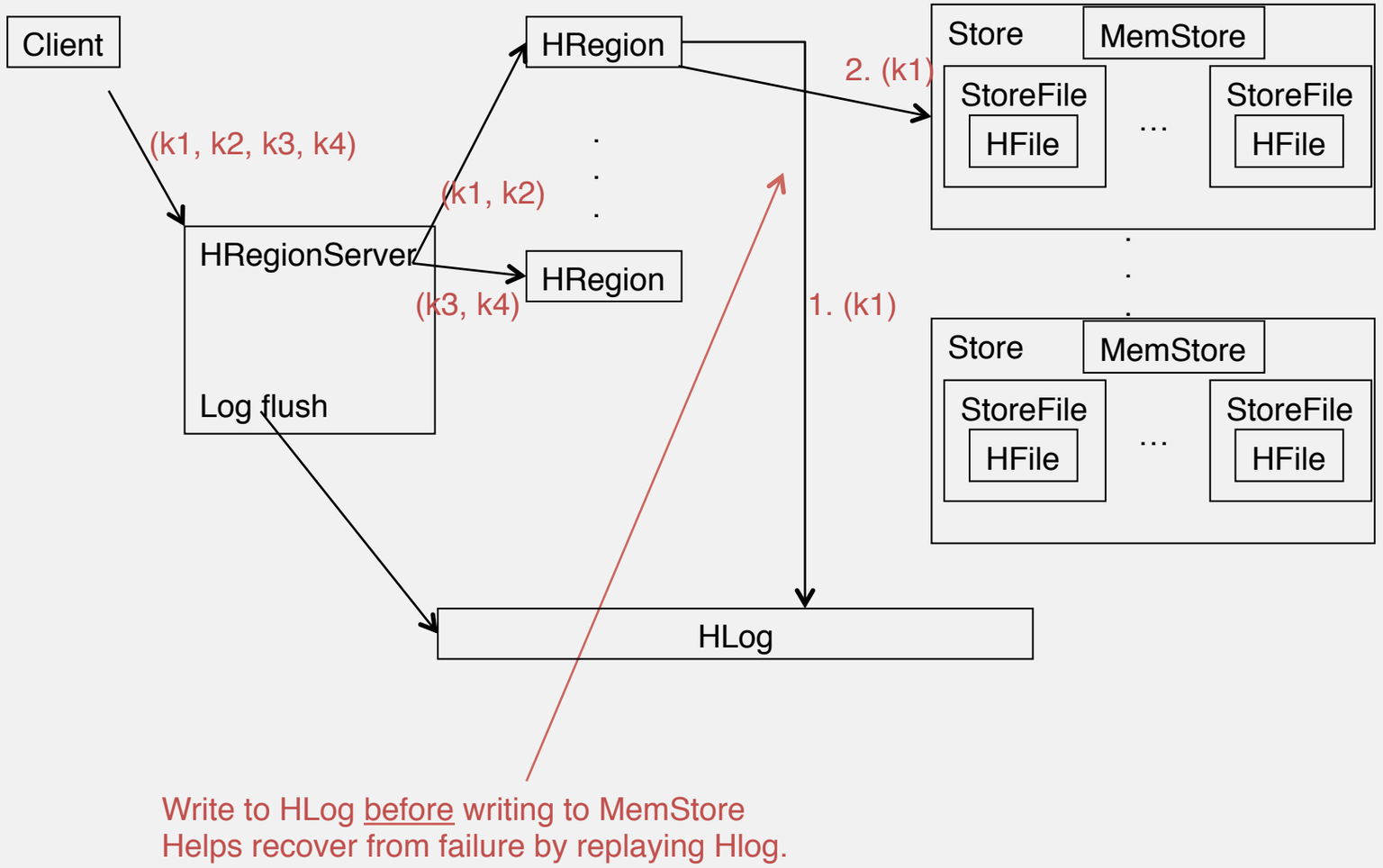
- HBase Table
  - Split it into multiple regions: replicated across servers
    - ColumnFamily = subset of columns with similar query patterns
    - One Store per combination of ColumnFamily + region
      - Memstore for each Store: in-memory updates to Store; flushed to disk when full
        - StoreFiles for each store for each region: where the data lives
          - 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



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

Cross-Datacenter Replication

- Single “Master” cluster
- Other “Slave” clusters replicate the same tables
- Master cluster synchronously sends HLogs over to slave clusters
- Coordination among clusters is via Zookeeper
- Zookeeper can be used like a file system to store control information

1. */hbase/replication/state*
2. */hbase/replication/peers/<peer cluster number>*
3. */hbase/replication/rs/<hlog>*

MongoDB: A NoSQL System Installation

- http://www.mongodb.org/downloads
- http://docs.mongodb.org/manual/installation
- mongod --dbpath <path-to-data>
- Mongo

- (MongoDB slides adapted from Mainak Ghosh's slides)

Data Model

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

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

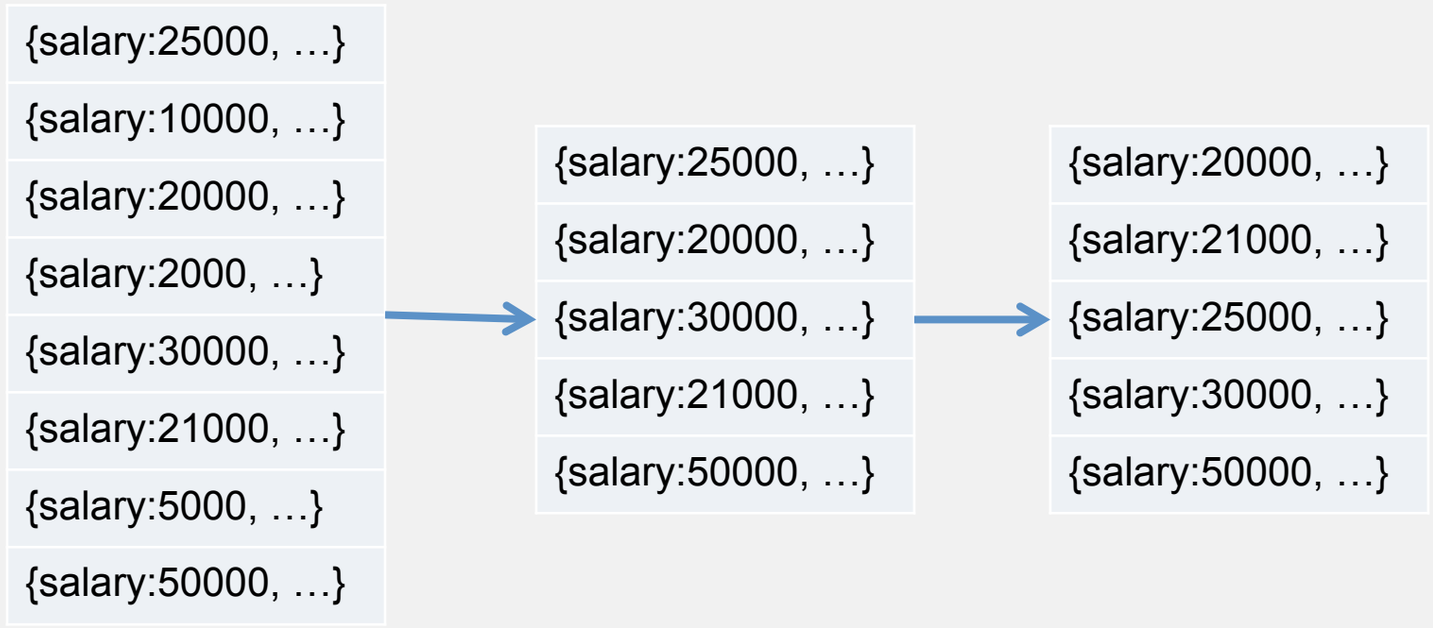
- Group of related *documents* with a shared common index is a *collection*

MongoDB: Typical Query

Query all employee names with salary greater than 18000 sorted in ascending order

```
db.employee.find({salary:{$gt:18000}}, {name:1}).sort({salary:1})
```

Collection — Condition — Projection — Modifier



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

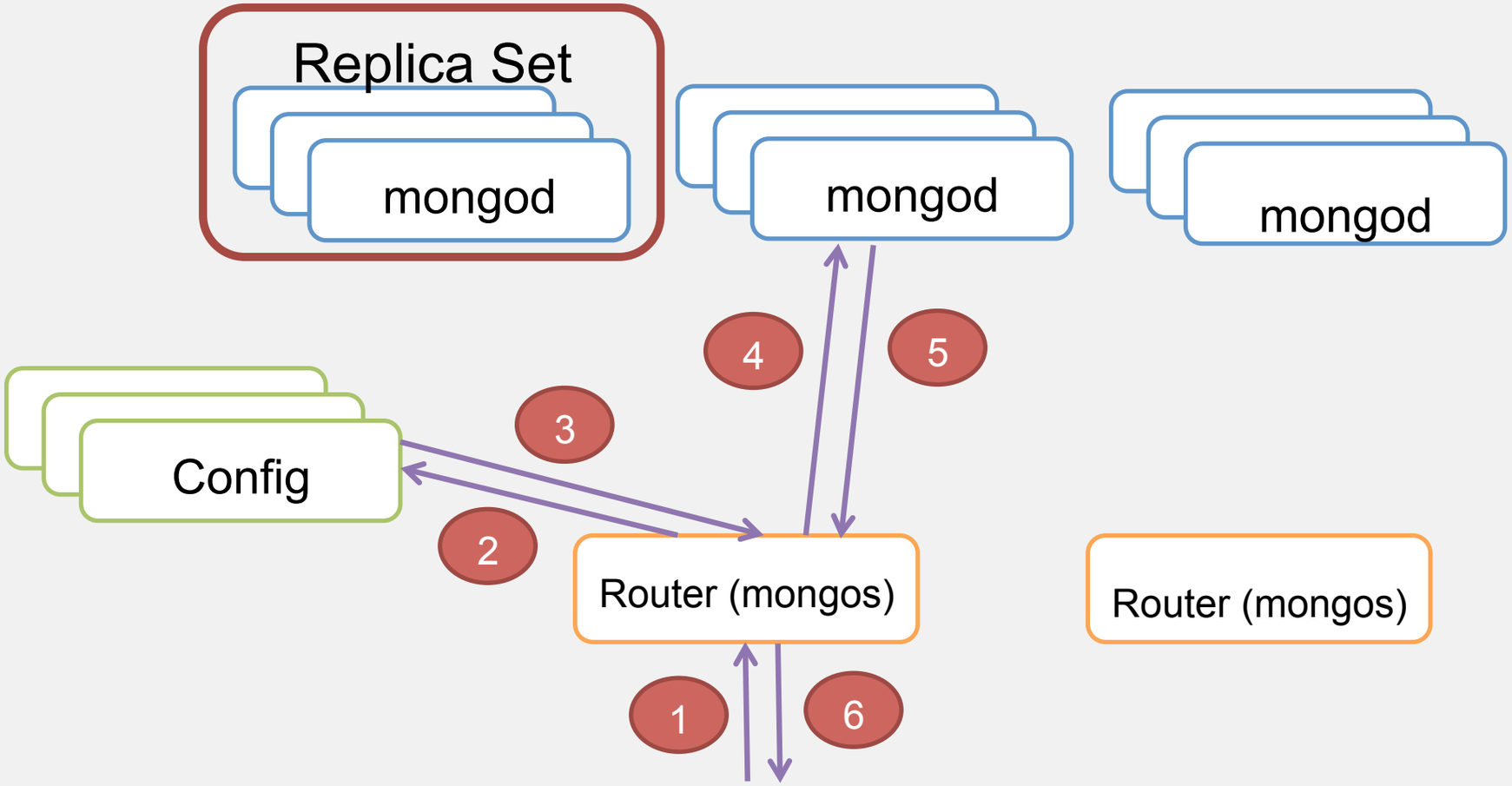
Remove all employees who earn less than 10000

Remove Criteria

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

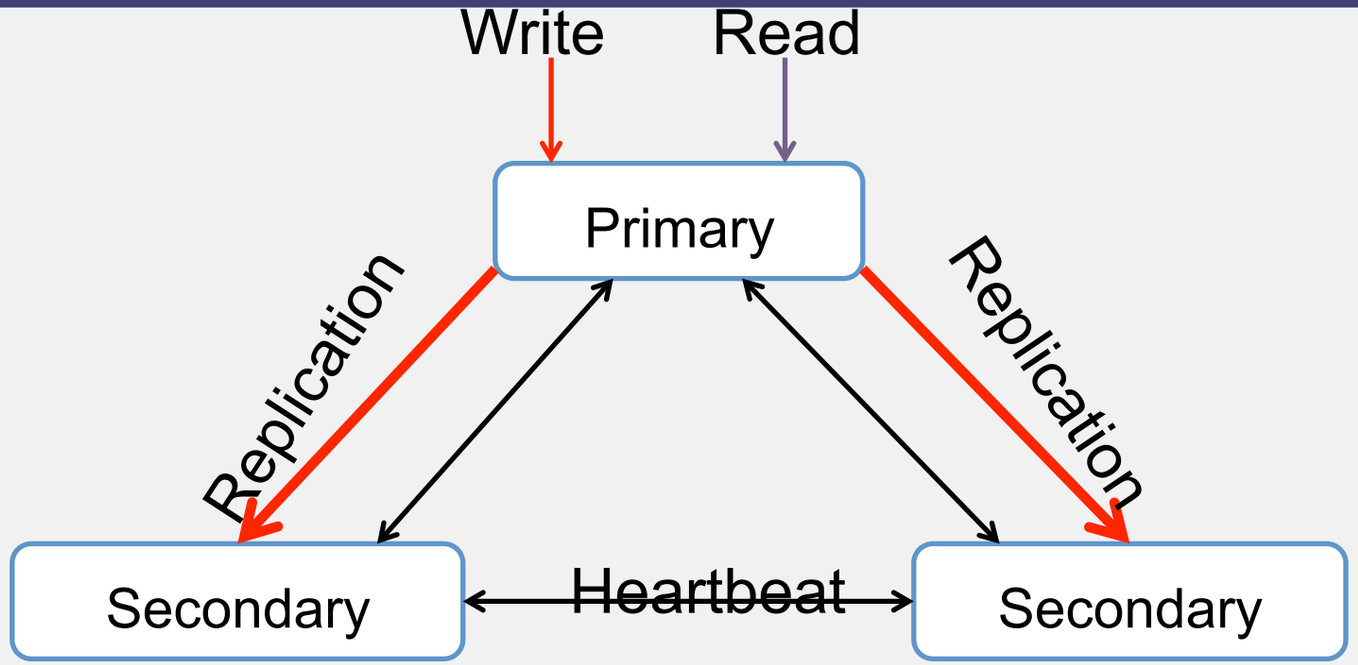
Can accept a flag to limit the number of documents removed

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.

Replication



Replication

- Uses an oplog (operation log) for data sync up
  - 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
- Journal may be memory-mapped
- 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 or Tertiary)
- CAP Theorem: With Strong consistency, under partition, MongoDB becomes write-unavailable thereby ensuring consistency

Performance

- 30 – 50x faster than MySQL Server 2008 for writes [1]
- At least 3x faster for reads [1]
- MongoDB 2.2.2 offers slower throughput for different YCSB workloads compared to Cassandra [2]

[1] http://blog.michaelckennedy.net/2010/04/29/mongodb-vs-sql-server-2008-performance-showdown/

[2] http://hyperdex.org/performance/

Summary

- Traditional Databases (RDBMSs) work with strong consistency, and offer ACID
- Modern workloads don't need such strong guarantees, but do need fast response times (availability)
- Unfortunately, CAP theorem
- Key-value/NoSQL systems offer BASE
  - Eventual consistency, and a variety of other consistency models striving towards strong consistency
- We discussed design of
  - Cassandra
  - HBase
  - MongoDB

Optional: Some more MongoDB queries

Insert

Insert a row entry for new employee Sally

```
use records                          -- Creates a database

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

Also can use save instead of insert

Bulk Load

```
people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy",
"Greg", "Steve", "Kristina", "Katie", "Jeff"];
money = [10000, 5000, 8000, 2000];
position = ["MTS", "Computer Scientist", "Manager", "Director"];
groups = ["cluster-management", "human-resource", "backend",
"ui"];

for(var i=0; i<10000; i++){
                name = people[Math.floor(Math.random()*people.length)];
                salary = money[Math.floor(Math.random()*money.length)];
                designation = position[Math.floor(Math.random()*position.length)];
                teams = groups[Math.floor(Math.random()*groups.length)];
                db.employee.save({name:name, salary:salary,
designation:designation, teams:teams});
}
```

Query

- db.employee.find()
- db.employee.find({name: "Sally"})
- var cursor = db.employee.find({salary: {$in: [5000, 2000] } })
- Use next() to access the rest of the records

Query

- db.employee.find({name: "Steve", salary: {$lt: 3000} })
- db.employee.find({ $or: [{ name: "Bill" }, { salary: { $gt: 9000 } }] })
- Find records of all managers who earn more than 5000
- db.employee.find({designation:"Manager", salary: {$gt: 5000}})

Aggregation Commands

- db.employee.count()
- How many employees with name Steve?
- db.employee.find({name: "Steve"}).count()
- db.employee.find({name: "Steve"}).skip(10)
- db.employee.find({name: "Steve"}).limit(10)

Modify

- Increment salary of all managers by 1000
- db.employee.update({ designation : "Manager" }, { $inc : { salary : 1000 } })
- db.employee.update({ designation : "Manager" }, { $inc : { salary : 1000 } } , { multi: **true** })
- Increment salary of all managers working in cluster-management group by 5000
- db.employee.update({ designation : "Manager", teams: "cluster-management"}, { $inc : { salary : 5000 } } , { multi: **true** })

Remove

- db.employee.remove({ name : "Sally" })
- Remove all Computer Scientist in the ui division
- db.employee.remove({teams: "ui", designation: "Computer Scientist"})