University of Bologna

Dipartimento di Informatica –
Scienza e Ingegneria (DISI)

Engineering Bologna Campus

Class of Computer Networks M or Infrastructures for Cloud Computing and Big Data

Global Data Storage

Luca Foschini

Academic year 2017/2018

Outline

Modern global systems need new tools for data storage with the necessary quality

We have seen

- **Distributed** file systems
  - – Google File System GFS
  - – Hadoop file system HDFS

But we need less conventional

- **NoSQL** Distributed storage systems
  - ❑ Cassandra
  - ❑ MongoDB

Distributed Storage Systems: 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

This abstraction is a **dictionary data structure organized for easing the operations by key I/O**

giving the key, you get the content fast

Via **insert, lookup, and delete** by key

e.g., hash table, binary tree

The main property is the **requirement** of **being distributed in deployment, and scalable**

Distributed Hash tables (DHT) in P2P systems

It is not surprising that **key-value stores** reuse many techniques from DHTs and tuple spaces

Isn’t that just a database?

Yes, sort of… but not exactly

Relational Database Management Systems (RDBMSs) have been around for ages

where MySQL is the most popular among them

- Data stored in tables
- Schema-based, i.e., structured complete tables
- Each row (data item) in a table has a primary key that is unique within that table
- Queries by 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 workloads

- Data are extremely **large and unstructured**
- Lots of **random reads and writes**
- Sometimes **write-heavy**
- **Foreign keys** rarely needed
- **Joins** rare

Typically **not regular queries** and sometimes very **forecastable** (so you can **prepare for them**)

In other terms, you can prepare data for the usage you want to optimize

Requirements of Today Workloads

- **Speed in answering**
- **No Single point of Failure** (**SPoF**)
- **Low TCO** (**T**otal **C**ost of **O**peration)
- **Fewer system administrators**
- **Incremental Scalability**
- **Scale out, not up**
  - What?

Scale out, not Scale up

Scale up = grow **your cluster capacity** by replacing **more powerful machines (vertical scalability)**

- 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) (the so-called **horizontal scalability**)

- **Cheaper and more effective**
- **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 = "**N**ot **o**nly **SQL**"

Necessary API operations:

get(key) and put(key, value)

- And some extended operations, e.g., "**CQL L**anguage" in Cassandra key-value store

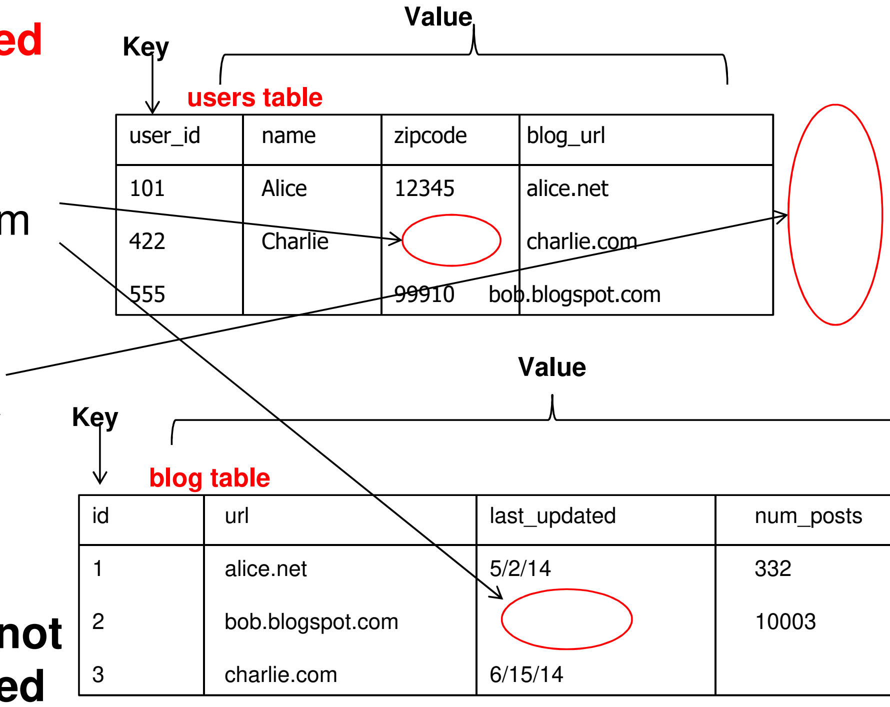
Tables

- Similar to RDBMS tables, but they …
- **May be unstructured: do not have schemas**
  - Some columns may be missing from some rows
- **Do not always support joins or have foreign keys**
- **Can have index tables**, just like RDBMSs

  "Column families" in Cassandra, "Table" in HBase, "Collection" in MongoDB

Key-value/NoSQL Data Model

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



user_id	name	zipcode	blog_url
101	Alice	12345	alice.net
422	Charlie		charlie.com
555		99910	bob.blogspot.com

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

Column-Oriented Storage

NoSQL systems can use column-oriented storage

- **RDBMSs store an entire row together (on a disk)**
- **NoSQL systems typically store a column together (also a group of columns)**
  - Entries within a column are indexed and easy to locate, given a key (and vice-versa)
- **Why?**
  - 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, without fetching the other columns

Cassandra

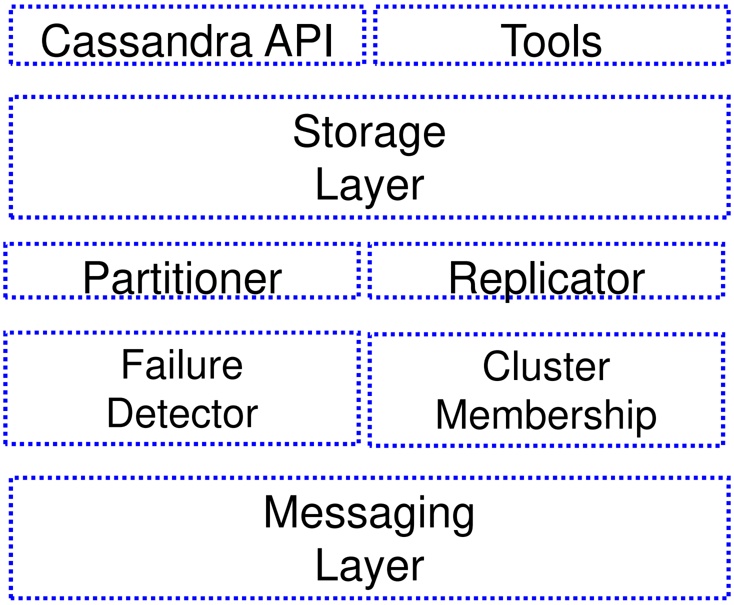
A distributed key-value store intended to run in a datacenter (and also across DCs)

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

Cassandra Architecture



Cassandra API

Tools

Storage Layer

Partitioner

Replicator

Failure Detector

Cluster Membership

Messaging Layer

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

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

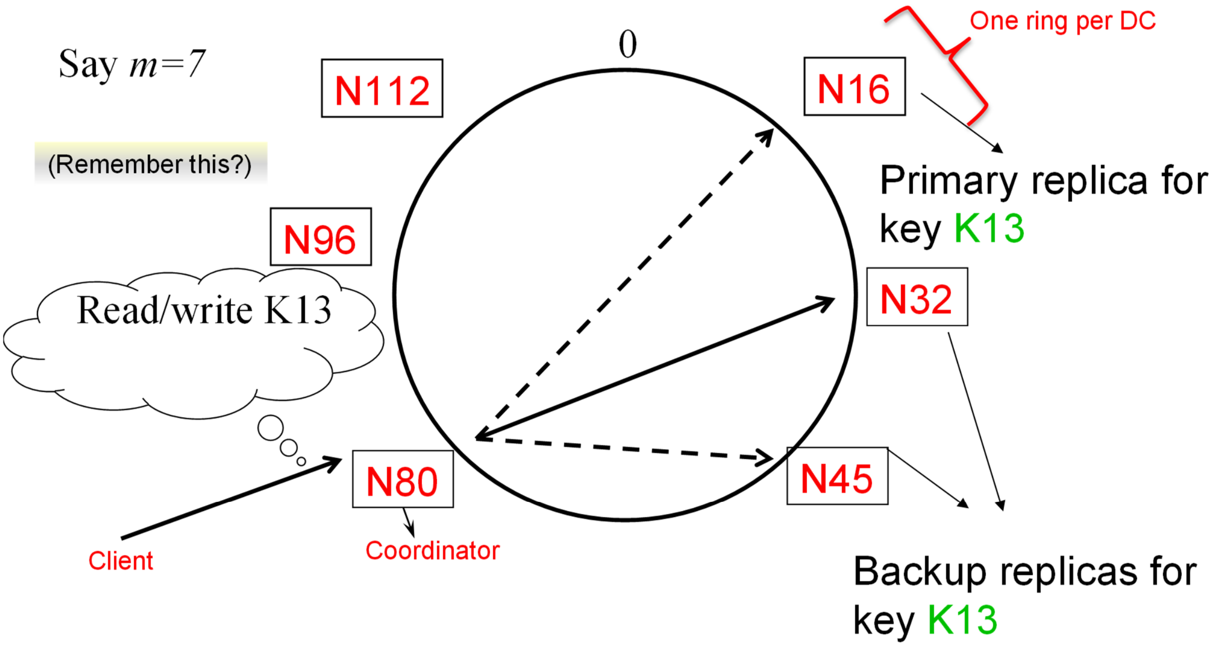
The main point is to **map efficiently and in a very suitable way for the current configuration based on different data centers and on the placement of replicas there**

So that it can change and adapt fast to needs and variable requirements and configurations

Cassandra Key -> Server Mapping

Say *m=7*

(Remember this?)



0

N112

N16

One ring per DC

Primary replica for key K13

N96

Read/write K13

N32

N80

N45

Client

Coordinator

Backup replicas for key K13

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

Key →server mapping is the "Partitioner"

Data Placement Strategies

Two different Replication Strategies based on partition policies

1. ***SimpleStrategy***
2. ***NetworkTopologyStrategy***

1. **SimpleStrategy**: in one Data Center with two kinds of Partitioners
   a. *RandomPartitioner*: Chord-like hash partitioning
   b. *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
   a. Two replicas per DC
   b. Three replicas per DC
   c. Per Data Center
      - First replica placed according to Partitioner
      - Then go clockwise around ring until you hit a different rack

Snitches

Snitches must **map IPs** to racks and DCs
they are 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

Write operations

Write operations must 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 that 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 is the majority

Write Policies

Always writable: Hinted Handoff mechanism

- If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until the crashed replica comes back up
- When all replicas are down, the Coordinator (front end) buffers writes (defers it for up to a few hours)

One ring per datacenter

- Per-DC coordinator elected to coordinate with other DCs
- Election done via Zookeeper, which implements distributed synchronization and group services (similar to JGroups reliable multicast)

Writes at a replica node

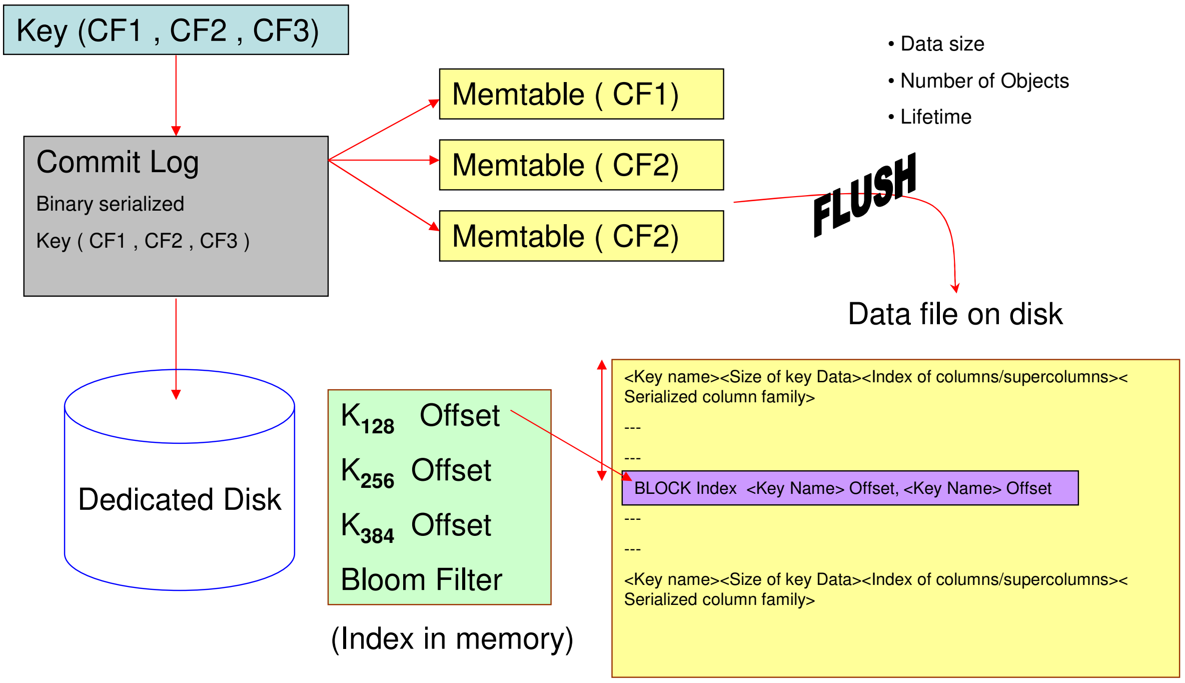
On receiving a write

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 pairs: (key, position in data sstable)
- Also employs a Bloom filter (for efficient search) – next slide

Writes: distributed architecture



Key (CF1 , CF2 , CF3)

Commit Log

Binary serialized

Key (CF1 , CF2 , CF3)

Dedicated Disk

Memtable (CF1)

Memtable (CF2)

Memtable (CF2)

- Data size
- Number of Objects
- Lifetime

FLUSH

Data file on disk

K_{128} Offset

K_{256} Offset

K_{384} Offset

Bloom Filter

(Index in memory)

<Key name><Size of key Data><Index of columns/supercolumns>< Serialized column family>

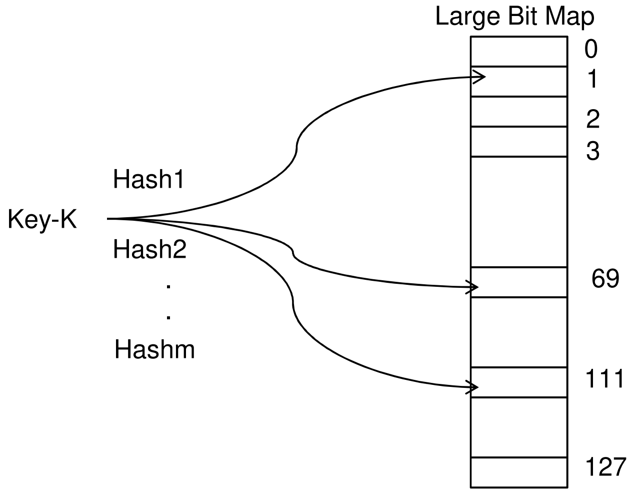
BLOCK Index <Key Name> Offset, <Key Name> Offset

<Key name><Size of key Data><Index of columns/supercolumns>< Serialized column family>

Bloom Filter

A compact table to hint for location

- **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 rate low

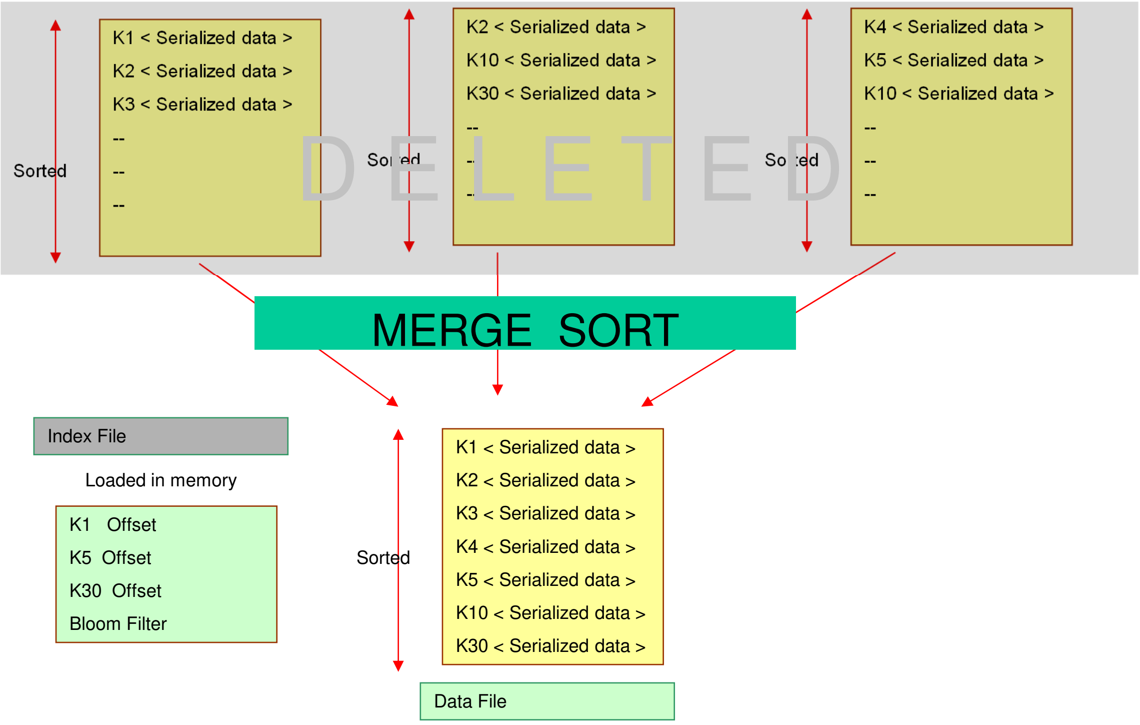
- m=4 hash functions
- 100 items
- 3200 bits
- FP rate = 0.02%

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
- Compaction runs periodically and locally at each server

Compaction at work



Sorted

K1 < Serialized data >
K2 < Serialized data >
K3 < Serialized data >
--
--
--

Sorted

K2 < Serialized data >
K10 < Serialized data >
K30 < Serialized data >
--
--
--

Sorted

K4 < Serialized data >
K5 < Serialized data >
K10 < Serialized data >
--
--
--

DELETED

MERGE SORT

Index File

Loaded in memory

K1 Offset
K5 Offset
K30 Offset
Bloom Filter

Sorted

K1 < Serialized data >
K2 < Serialized data >
K3 < Serialized data >
K4 < Serialized data >
K5 < Serialized data >
K10 < Serialized data >
K30 < Serialized data >

Data File

Deletes

Delete: do not delete items right away

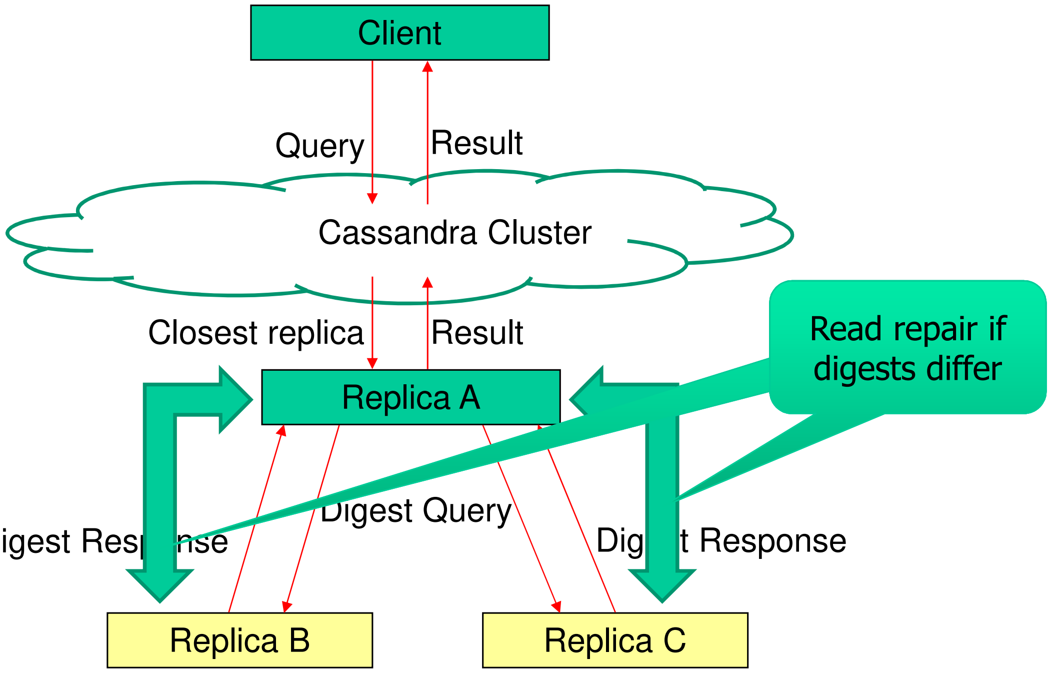
- Add a **tombstone** to the log
- Eventually, when compaction encounters tombstone it will delete item

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)

Reads: distributed architecture



Read repair if digests differ

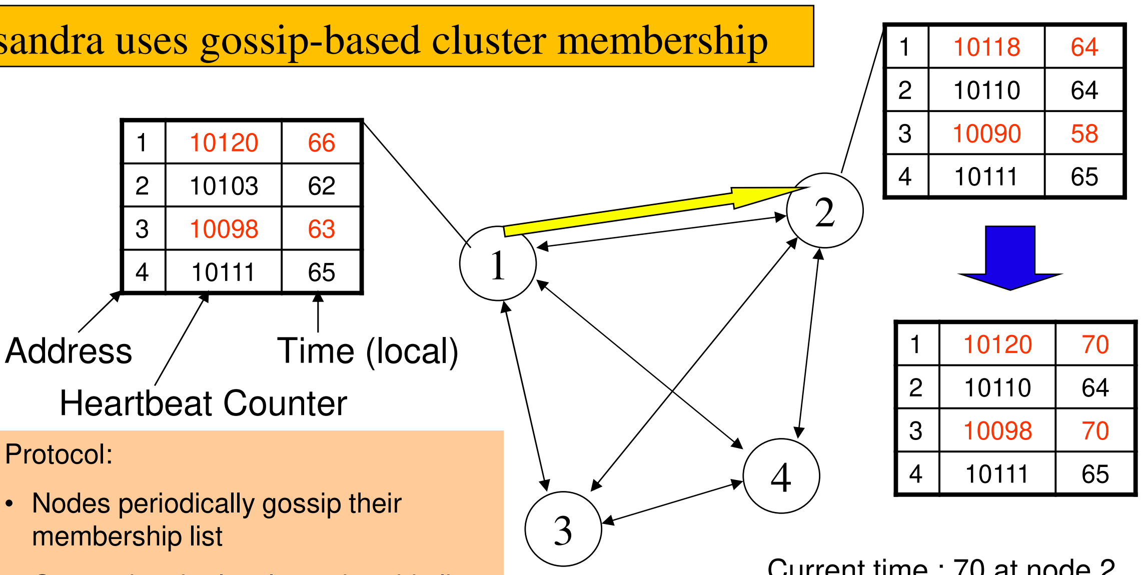
Membership

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



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

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

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

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

Cassandra orders of magnitude faster

- What is the catch? What did we lose?

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

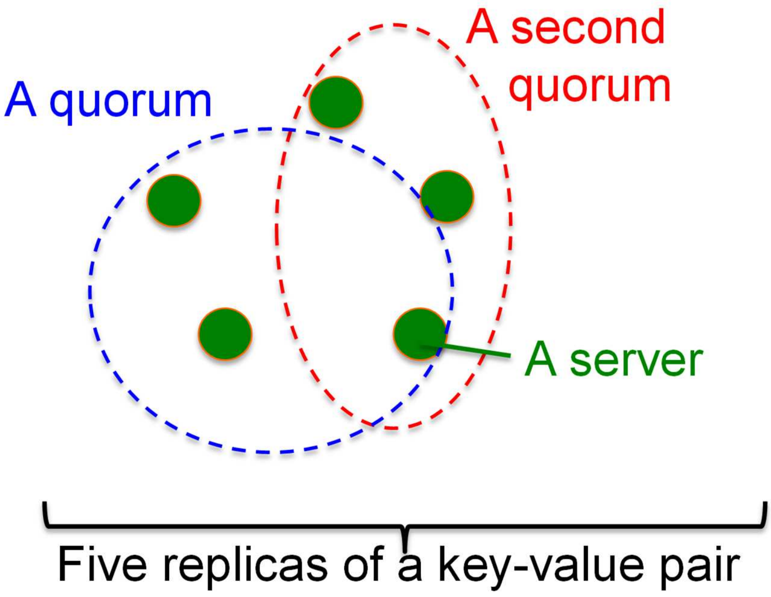
Client is allowed to choose a **consistency level** for each operation (read/write)

- 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 Operations

Several key-value/NoSQL stores use quorums

Reads

The Client specifies value of R (≤ N = number of replica)

R = read consistency level.

- The coordinator waits for R replicas to respond before sending result to client and
- In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed

Writes come in two flavors

- The Client specifies W (≤ N) W = write consistency level.
- The 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

- 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

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

MongoDB

MongoDB is Document-oriented NoSQL tool

Typically MongoDB

Open source NoSQL DB

In **memory access to data**

Native replications toward **reliability and high availability (CAP)**

Collection partitioning by using **sharding key** so to **keep the information fast available and also replicated**

Designed in **C++**

MongoDB in a nutshell

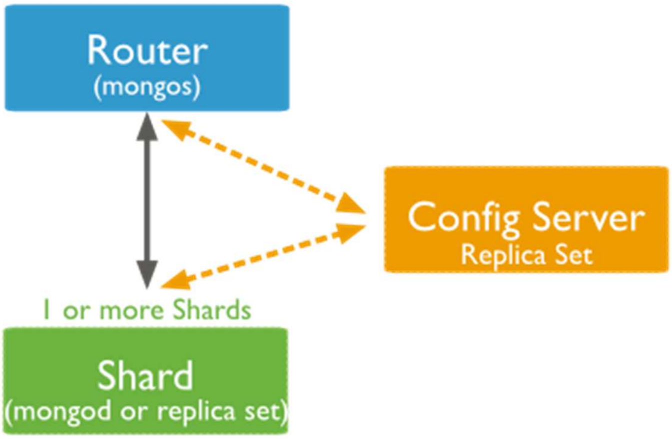
Collection partitioning by using a **shard key**:

Hashed-based to obtain a (not always) balanced distribution

Distributed architecture:

- **Router** to accept and route incoming requests coordinating with **Config Server**
- **Shard** to store data

- Pros
  - **Adding/removing shards**
  - **Automatic balancing**
- Cons
  - Max document size 16Mb



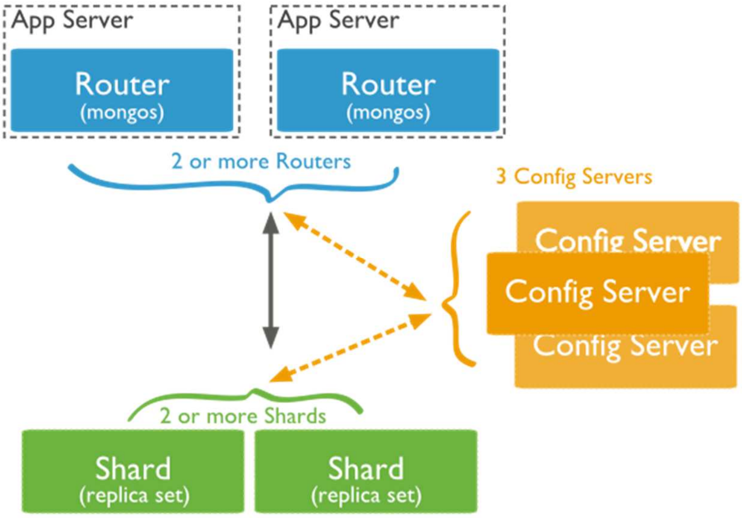
MongoDB in a deployement

The configuration can grant different properties
In a distributed architecture you may employ replication

Distributed architecture:

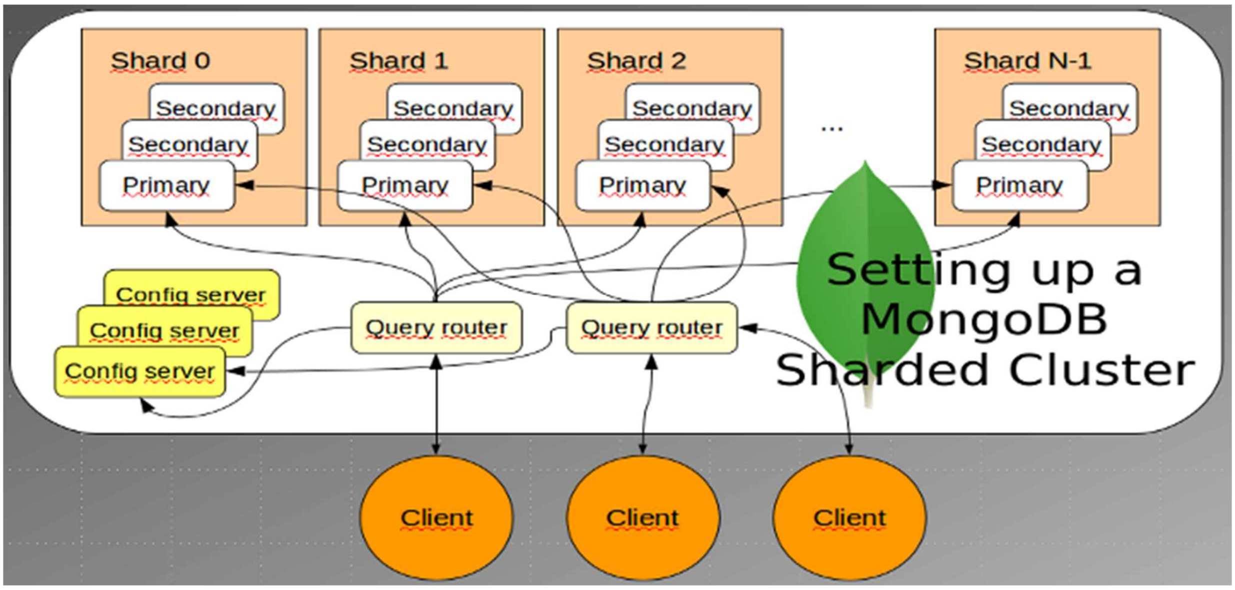
- **Several Router2** to accept incoming requests
- **Config Server to give access to requests**
- **Shards** to store data

The system is capable of supporting dynamic access to documents



MongoDB in a nutshell

The configuration can grant different properties
In a distributed architecture you may define better



Mongo Data Model

Based on collections of documents

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

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

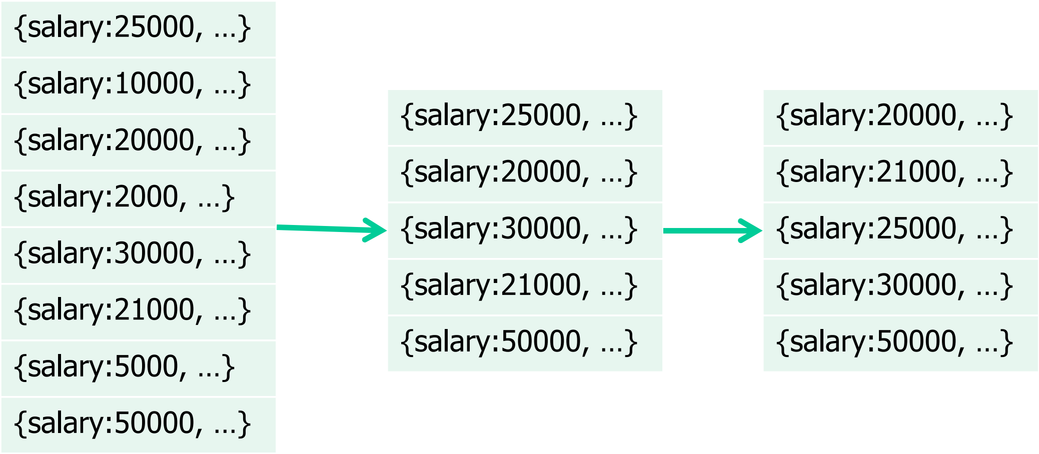
- Group of related *documents* with a shared common index is a ***collection***

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



{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

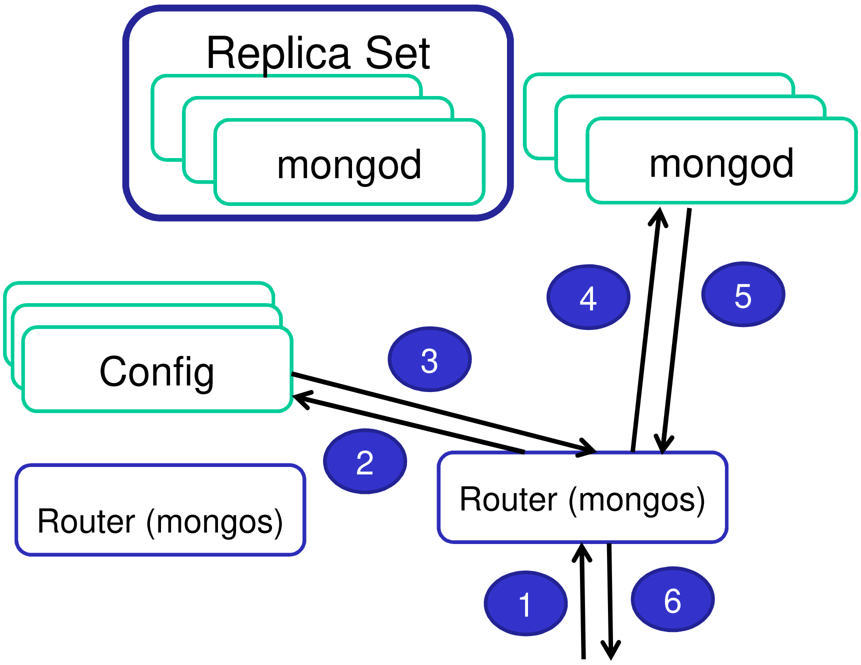
Delete

Remove all employees who earn less than 10000

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

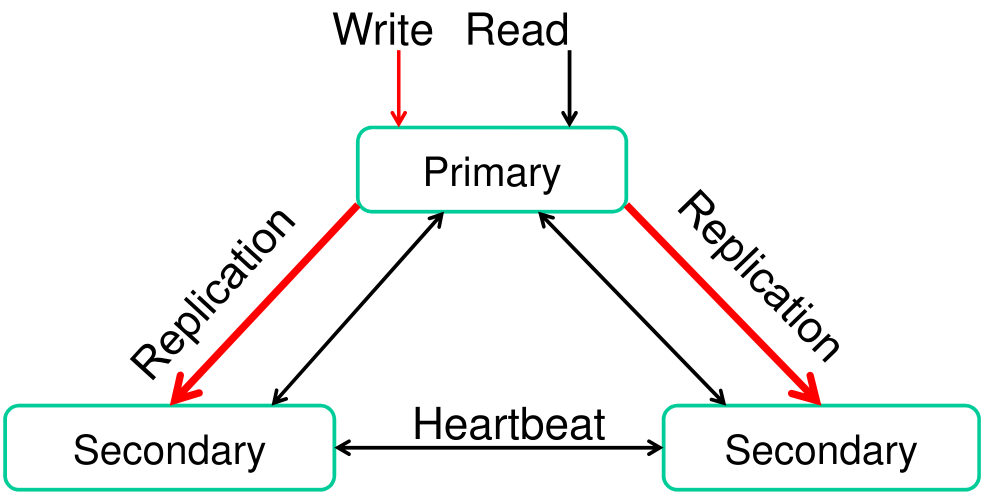
Can accept a flag to limit the number of documents removed

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 lo**g) 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