University of Bologna
Dipartimento di Informatica –
Scienza e Ingegneria (DISI)
Engineering Bologna Campus

Class of

Computer Networks M

Global Data Storage

Luca Foschini

Academic year 2015/2016

Outline

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

- **Distributed** file systems:
  – Google File System
  – Hadoop file system
- **NoSQL** Distributed storage systems
  – Cassandra
  – MongoDB

Google File System (GFS)

- GFS exploits **Google** hardware, data, and application properties to improve performance
  - **Large scale**: thousands of machines with thousands of disks
  - **Component failures** are ‘normal’ events
    - Hundreds of thousands of machines/disks
    - MTBF of 3 years/disk → 100 disk failures/day
    - Additionally: network, memory, power failures
  - Files are **huge** (multi-GB file sizes are the norm)
    - Design decision: difficult to manage billions of small files
  - **File access** model: read/append
    - Random writes practically non-existent
    - Most reads sequential

Design criteria

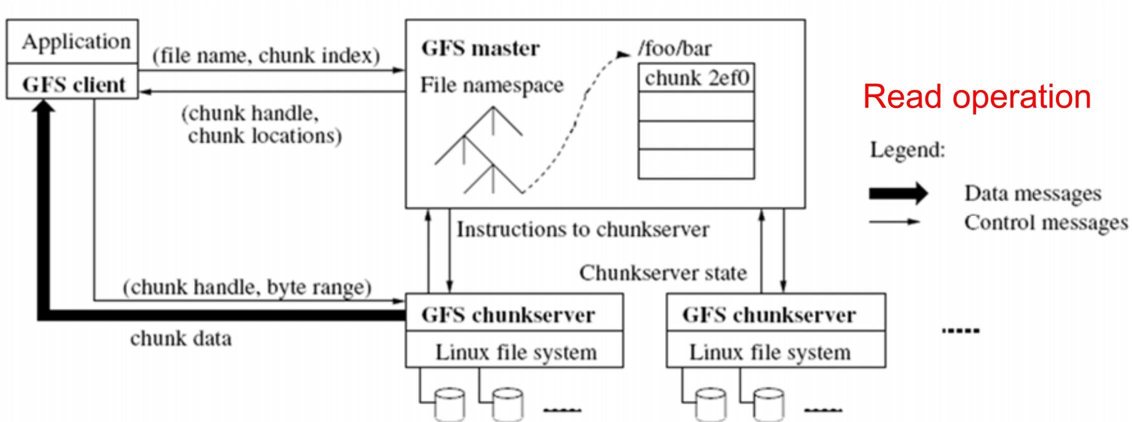
- Detect, tolerate, and recover from failures **automatically**
- “**Modest**” number of large files
  - Just a few millions
  - Each 100MB – multi-GB
  - Few small files
- **Read-mostly** workload
  - Large streaming reads (multi-MB at a time)
  - Large sequential append operations
    - Provide atomic consistency to parallel writes with low overhead
- High sustained **throughput** more important than low **latency**

Design decisions

- Files stored as **chunks**
  - Stored as local files on Linux file system
- Reliability through **replication** (3+ replicas)
- **Single master** to coordinate access, keep metadata
  - Simple centralized design (one master per GFS cluster)
  - Can make better chunk placement and replication decisions using global knowledge
- **No caching**
  - Large data set/streaming reads render caching useless
  - Linux buffer cache to keep data in memory
  - Clients cache meta-data (e.g., chunk location)

GFS architecture

- One master server (state replicated on backups)
- Many chunk servers (100s – 1000s)
  - Spread across racks for better throughput & fault tolerance
  - Chunk: 64 MB portion of file, identified by 64-bit, globally unique ID
- Many clients accessing files stored on same cluster
  - Data flow: client <-> chunk server (master involved just in control)



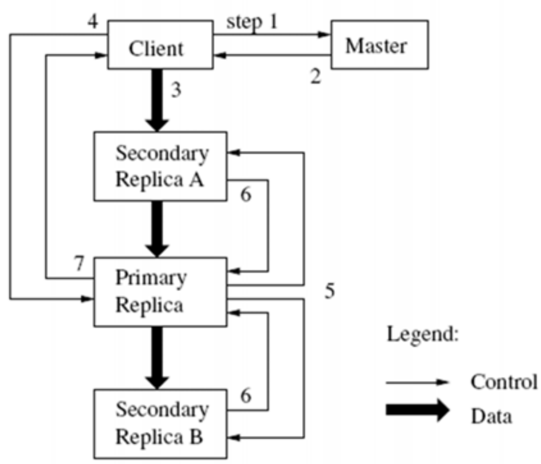
More on metadata & chunks

- **Metadata**
  - **3 types**: file/chunk namespaces, file-to-chunk mappings, location of any chunk replicas
  - **All in memory** (< 64 bytes per chunk)
    - **GFS capacity limitation**
- **Large chunk have many advantages**
  - Fewer **client-master** interactions and **reduced size metadata**
  - Enable persistent **TCP connection** between **clients and chunk servers**

Mutations, leases, version numbers

- **Mutation**: operation that changes the contents (write, append) or metadata (create, delete) of a chunk
- **Lease**: mechanism used to maintain consistent mutation order across replicas
  - Master grants a **chunk lease** to one replica (primary chunk server)
  - Primary picks a **serial order to all mutations to the chunk** (many clients can access chunk concurrently)
  - All replicas follow **this order when applying mutations**
- Chunks have **version numbers** to distinguish between up-to-date and stale replicas
  - Stored on disk at master and chunk servers
  - Each time master grants new lease, increments version & informs all replicas

Mutations step-by-step



1. Identities of primary chunk server holding lease and secondaries holding the other replicas
2. Reply
3. Push data to all replicas for consistency (see next slide for details)
4. Send mutation request to primary, which assigns it a serial number
5. Forward mutation request to all secondaries, which apply it according to its serial number
6. Ack completion
7. Reply (an error in any replica results in an error code & a client retry)

Data flow

Client can push **the data** to any replica

Data is pushed linearly along a carefully picked chain of **chunk servers**

- Each machine forwards data to "closest" machine in network topology that has not received it
  - Network topology is simple enough that "distances" can be accurately estimated from IP addresses
- **Method introduces delay**, but offers good **bandwidth** utilization
- **Pipelining**: servers receive and send data at the same time

Consistency model

- File **namespace mutations** (create/delete) are **atomic**
- State of a **file region** depends on
  - Success/failure of mutation (write/append)
  - Existence of concurrent mutations
- **Consistency states of replicas and files**:
  - **Consistent**: all clients see same data regardless of replica
  - **Defined**: consistent & client sees the mutation in its entirety
    - Example of consistent but undefined: initial record = AAAA; concurrent writes: _B_B and CC__; result = CCAB (none of the clients sees the expected result)
  - **Inconsistent**: due to a failed mutation
    - Clients see different data function of replica

How to avoid the undefined state?

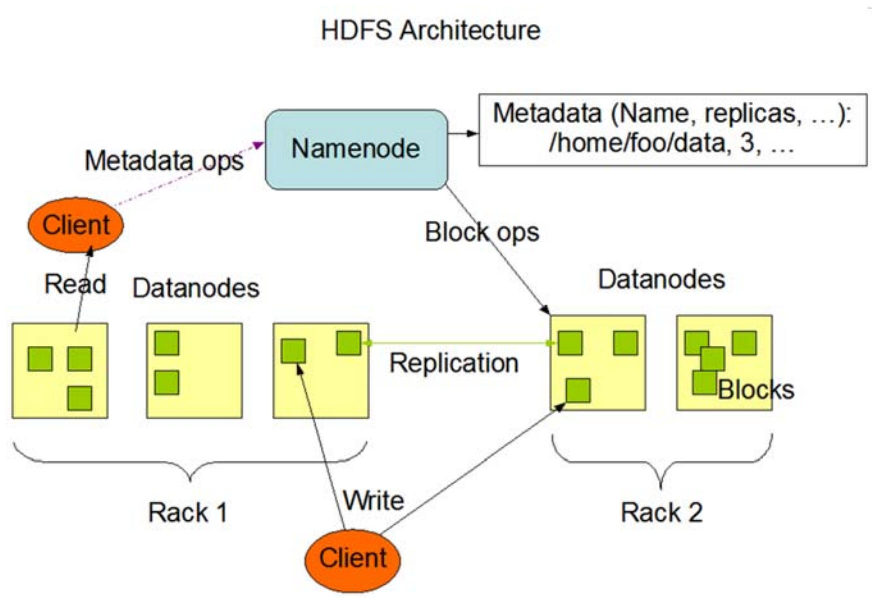
- Traditional **random writes** require expensive synchronization (e.g., lock manager)
  - Serializing writes does not help (see previous slide)
- Atomic *record append*: allows **multiple clients to append data to the same file concurrently**
  - Serializing append operations at primary solves the problem
  - The result of successful operations is defined
  - **"At least once" semantics**
    - **Data is written at least once at the same offset by all replicas**
    - If one operation fails at any replica, the client retries; as a result, replicas may contain duplicates or fragments
  - If not enough space in chunk, add padding and return error
    - Client retries

How can the applications deal with record append semantics?

- Applications should include checksums in records they write using *record append*
  - Reader can **identify padding/record fragments** using **checksums**
- If application cannot tolerate duplicated records, should include unique ID in record
  - Readers can use **unique IDs** to filter duplicates

HDFS (another distributed file system)

Inspired by GFS



- Master/slave architecture
  - NameNode is master (meta-data operations, access control)
  - DataNodes are slaves: one per node in the cluster

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 (2)

- It's a dictionary data structure organization insert, lookup, and delete by key
  - E.g., hash table, binary tree
- But distributed
- Sound familiar?
  Recall Distributed Hash tables (DHT) in P2P systems
- It is 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 workloads

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

Needs of Today 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

Key / Value

users table

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

Key / Value

blog table

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

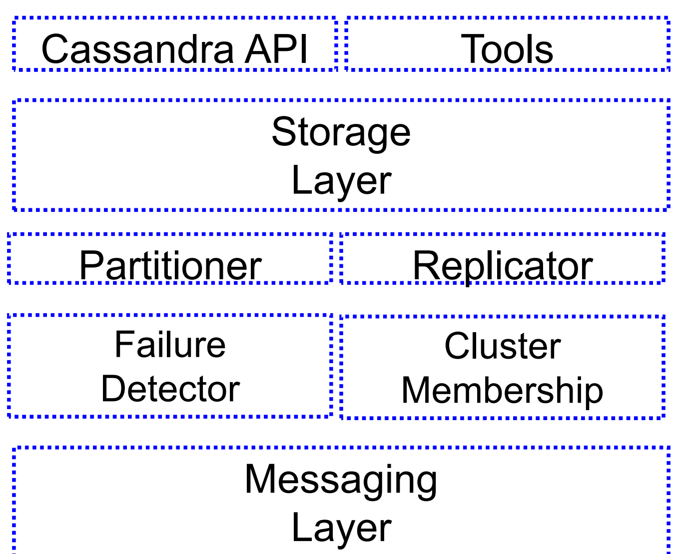
NoSQL systems often use column-oriented storage

- **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

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

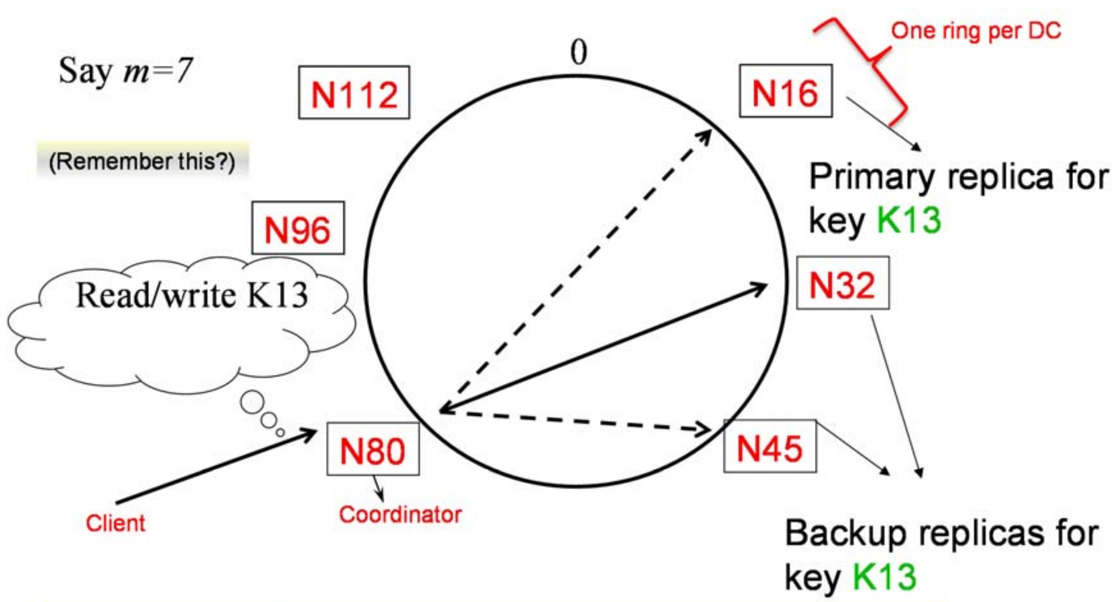
Cassandra Architecture



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

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

Cassandra Key -> Server Mapping



Cassandra uses a Ring-based DHT but without finger tables or routing
Key→server mapping is the "Partitioner"

Data Placement Strategies

- Replication Strategies, two possibilities:
  1. *SimpleStrategy*
  2. *NetworkTopologyStrategy*

1. <u>SimpleStrategy</u>: 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 *<u>range queries</u>* (e.g., Get me all twitter users starting with [a-b])
2. <u>NetworkTopologyStrategy</u>: 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)

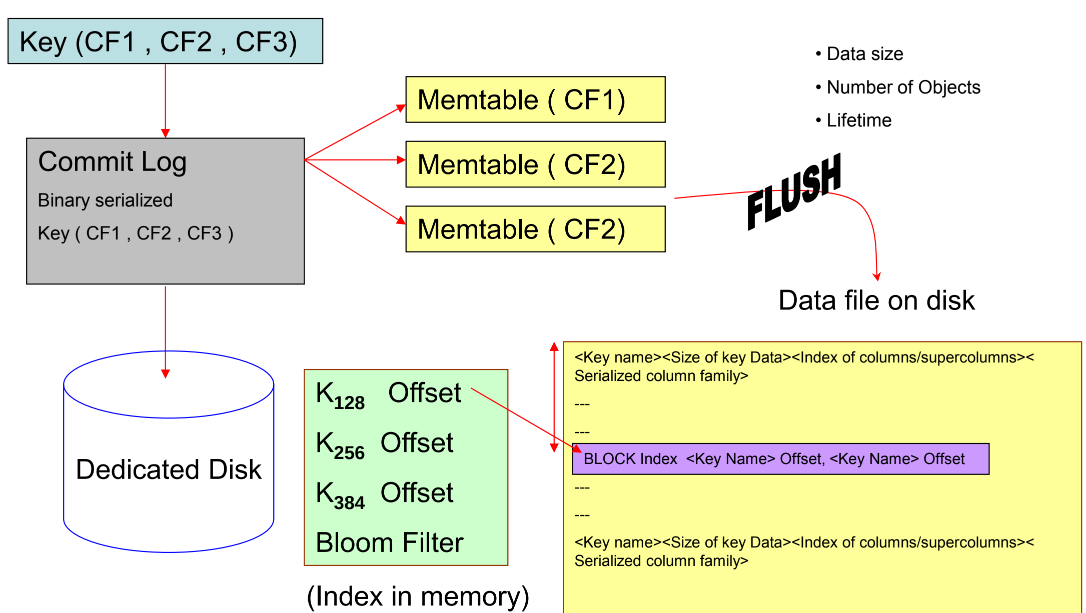
- Always writable: <u>Hinted Handoff mechanism</u>
  - If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until down replica comes back up.
  - When all replicas are down, the Coordinator (front end) buffers writes (for up to a few hours).
- One ring per datacenter
  - Per-DC coordinator elected to coordinate with other DCs
  - Election done via Zookeeper, which 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

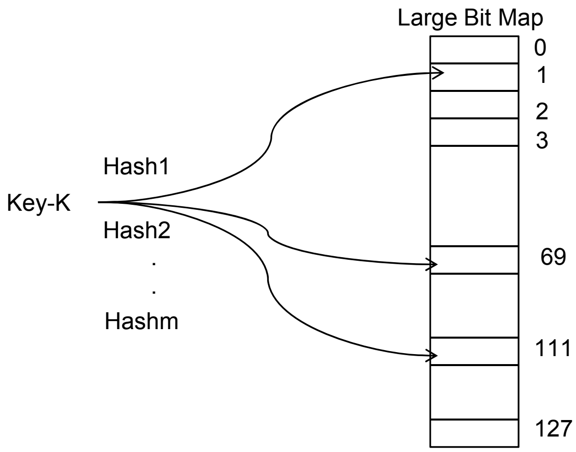
Later, when memtable is full or old, flush to disk
   - Data File: An **SSTable** (Sorted String Table) – list of key-value pairs, sorted by key
   - *SSTables are immutable (once created, they don't change)*
   - Index file: An SSTable of (key, position in data sstable) pairs
   - And a Bloom filter (for efficient search) – next slide

Writes: distributed architecture



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

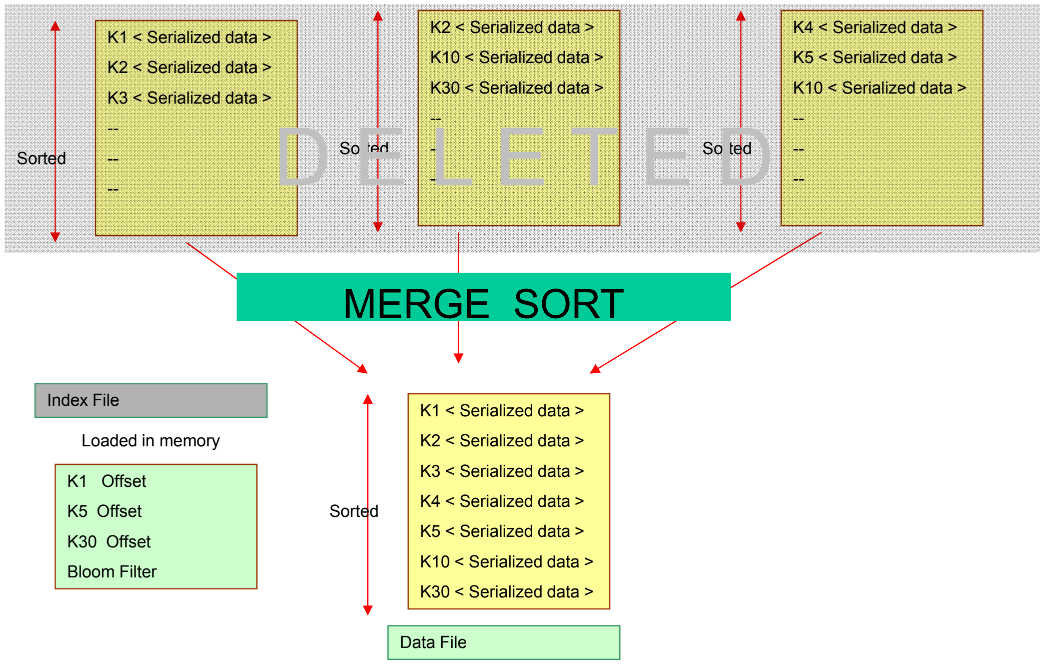
False positive rate low
- m=4 hash functions
- 100 items
- 3200 bits
- FP rate = 0.02%

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

Compaction at work



Deletes

Delete: don’t delete item right away

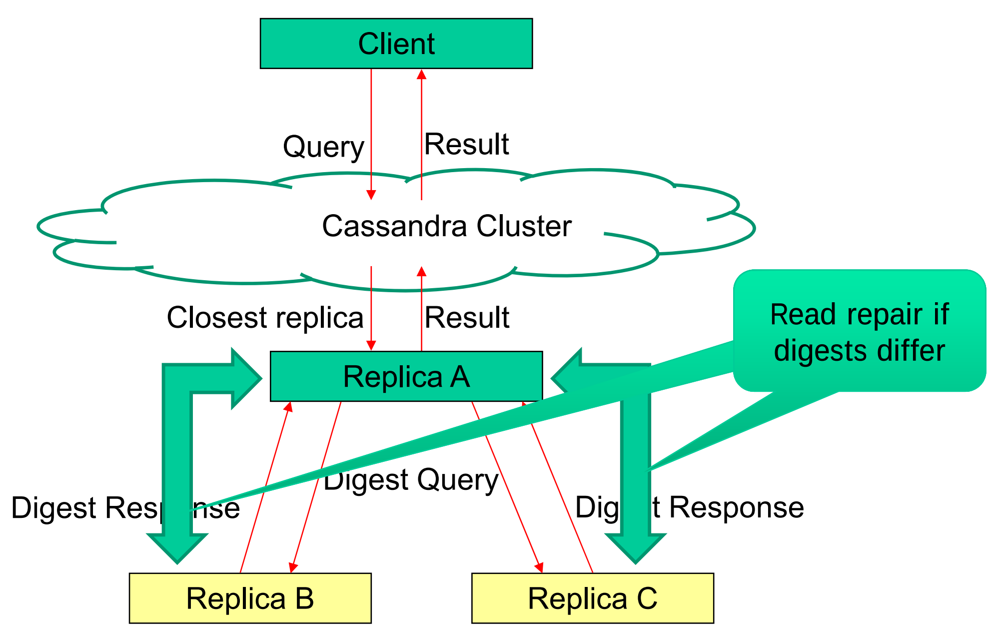
- Add a **tombstone** to the log
- Eventually, when compaction encounters tombstone it will delete item

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)

Reads: distributed architecture



Membership

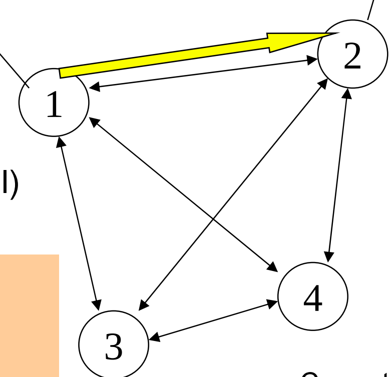
- Any server in cluster could be the coordinator
- So every server needs to maintain a list of all the other servers that are currently in the server
- List needs to be updated automatically as servers join, leave, and fail

Cluster Membership – Gossip-Style

Cassandra uses gossip-based cluster membership

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

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

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?

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

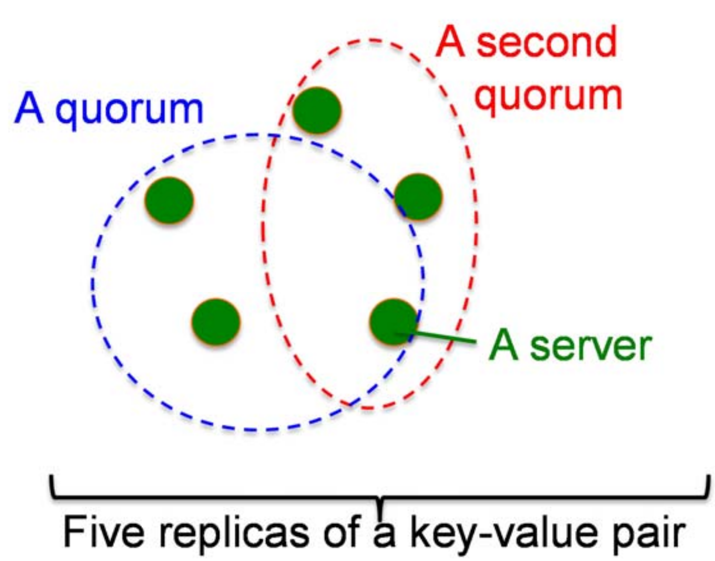
Back to Cassandra: Mystery of X

- 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

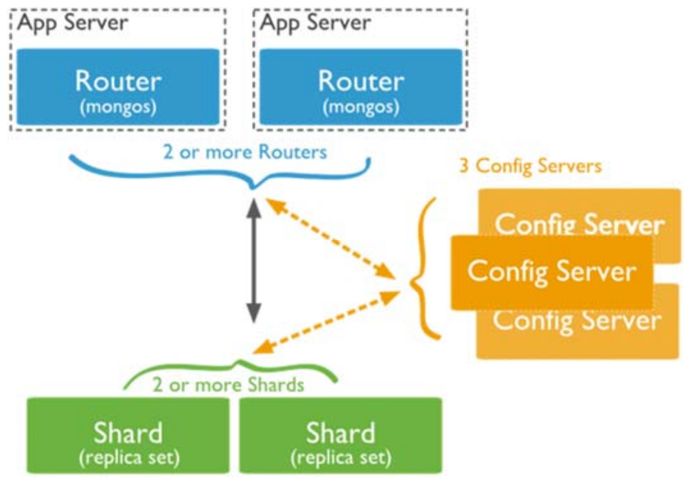
Cassandra Consistency Levels (Contd.)

- 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

MongoDB in a nutshell

- Document-oriented
- Collection partitioning using a **shard key**:
  - o 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

Data Model

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

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

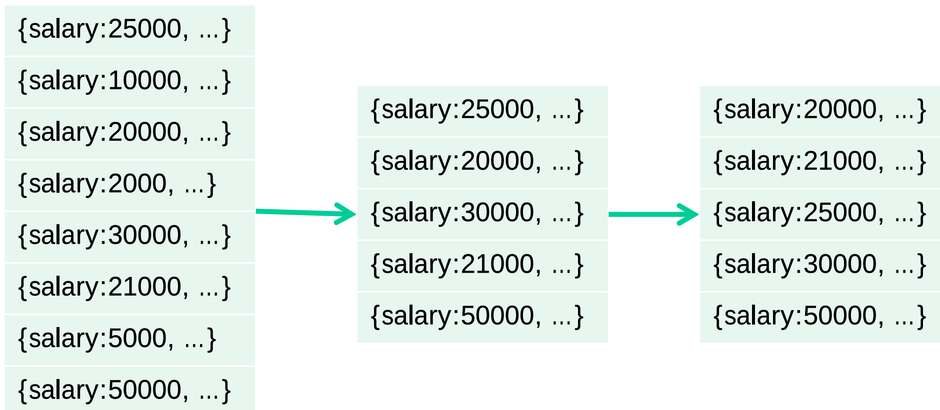
- Group of related *documents* with a shared common index is a *collection*

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

Update Criteria
Update Action
Update Option

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

Multi-option allows multiple document update

Delete

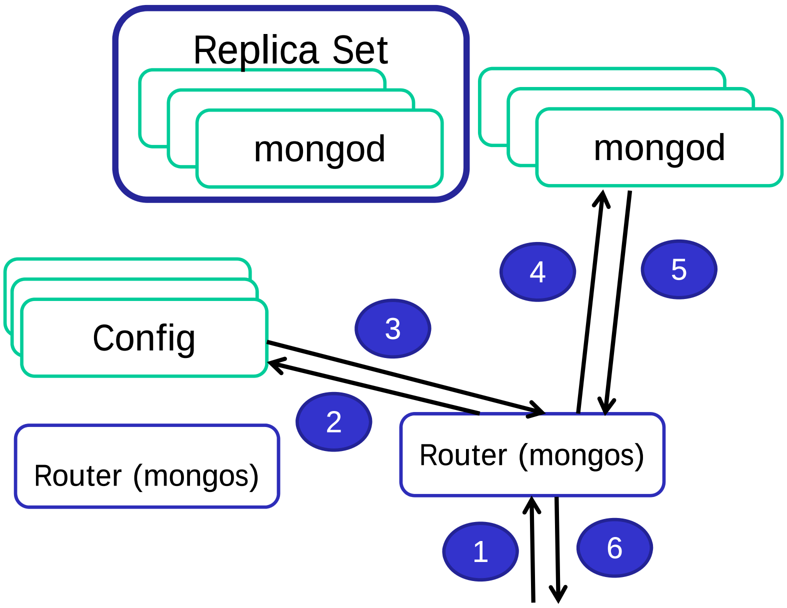
Remove all employees who earn less than 10000

Remove Criteria

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

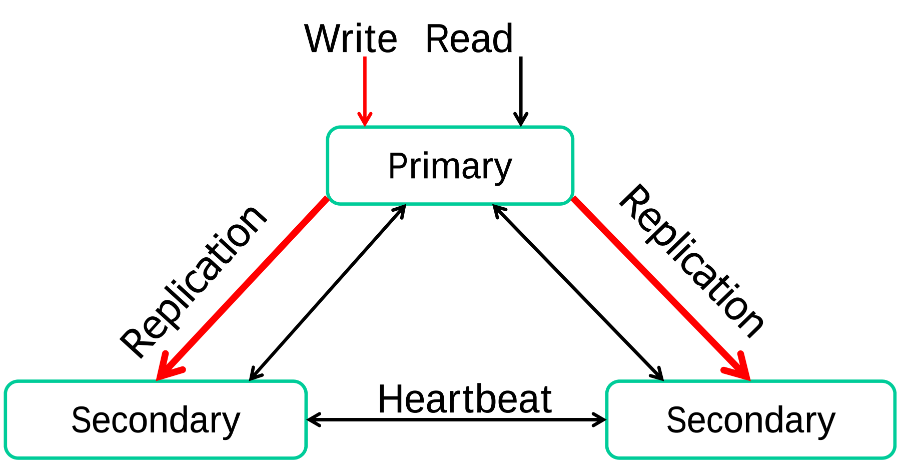
Can accept a flag to limit the number of documents removed

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