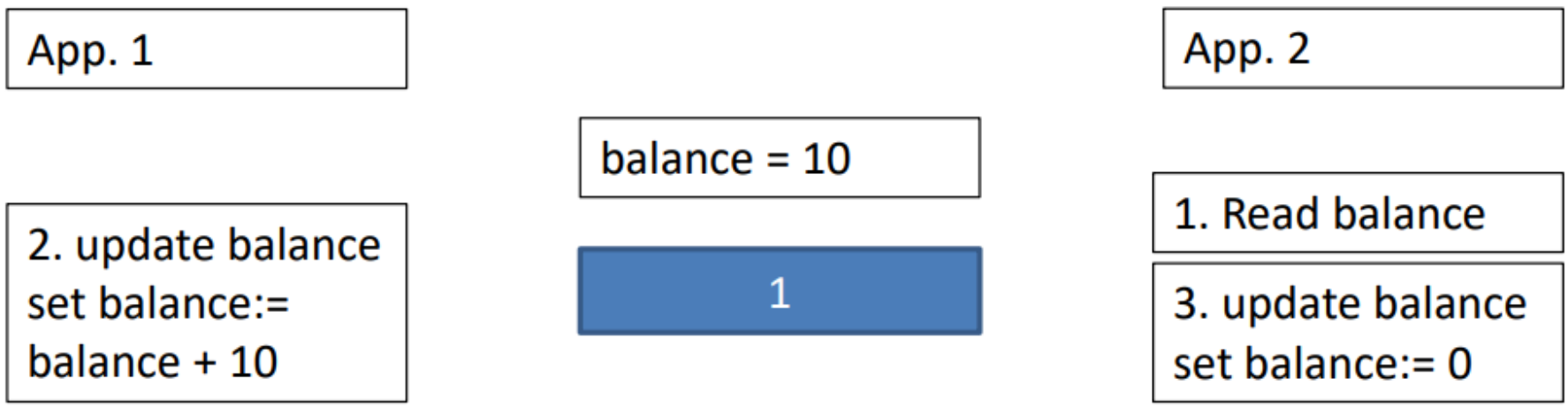
Lecture 13

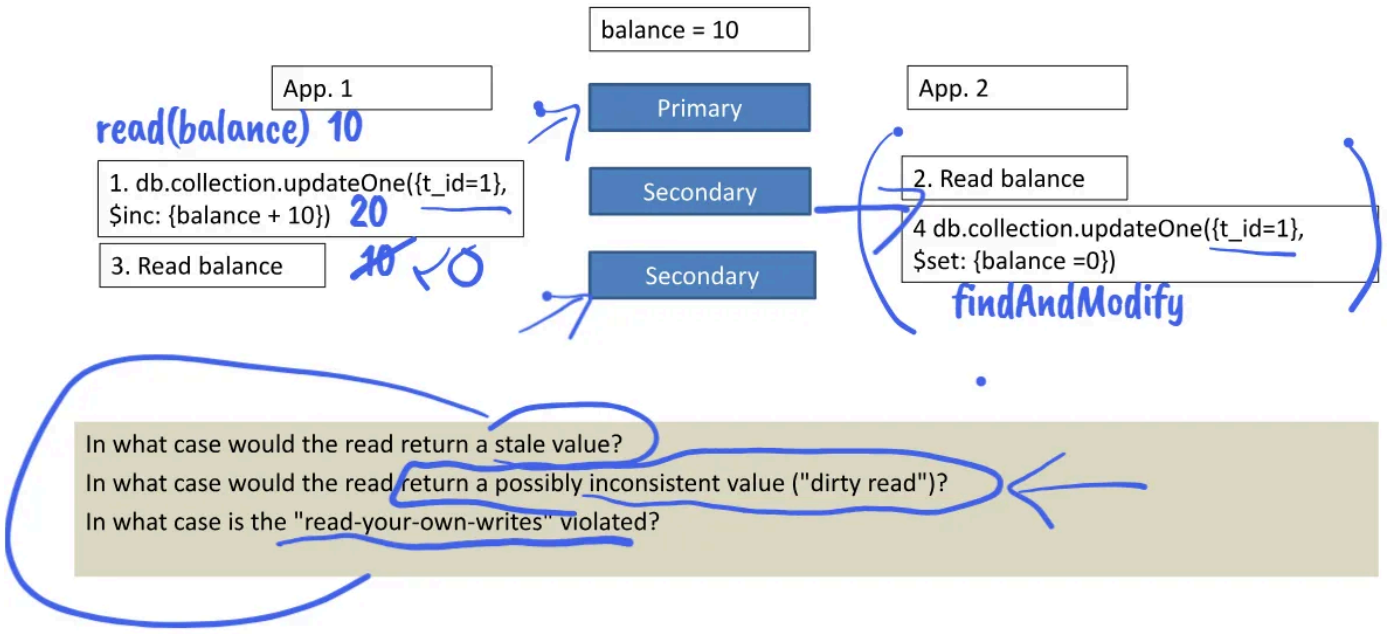
Type: Lecture

Concurrent Writes in a Relational Single-Node Database



1. **What problem could happen and how can it be prevented?**
   a. **Lost Update Problem**:
      - App. 2's write overwrites App. 1's update, causing data loss.
      - Root Cause: No proper isolation between transactions.
   b. **Race Condition**:
      - The final state depends on the execution order of the transactions.
2. **What to do on the application side?**
   - **Pessimistic Locking**:
     - App. 2 blocks until App. 1's transaction commits.
     - Example: `SELECT ... FOR UPDATE` (row-level lock).
   - **Optimistic Concurrency Control**:
     - Use version numbers/timestamps to detect conflicts before committing.
   - **Isolation Levels**:
     - `SERIALIZABLE` ensures strict ordering but reduces performance.
     - `REPEATABLE READ` (in some databases) prevents dirty reads but not lost updates.
3. **How are the writes processed?**
   - In a single-node database, writes are **serialized** (executed one after another).
   - Without proper concurrency control, the database may interleave operations, leading to anomalies.
   - With locking/isolation, the database ensures atomicity and consistency (e.g., App. 1's update fully completes before App. 2's write).

Concurrent Writes in Single-Leader Replicated Database (MongoDB)



App 1

1. Executes: `db.collection.updateOne({t_id:1}, {$inc: {balance: +10}})`

   ➤ Increments balance by 10 on the primary.

App 2

1. Executes: `db.collection.updateOne({t_id:1}, {$set: {balance: 0}})`

   ➤ Sets balance to 0 on the primary.

App 1 (again)

1. Reads: `db.collection.find({t_id:1})`

   ➤ But from a secondary replica.

1. **In what case would the read return a stale value?**

   A read from a secondary can return stale data if that secondary has not yet applied the latest changes from the primary's oplog. The reads from secondaries are eventually consistent, not immediately consistent.

2. **In what case would the read return a possibly inconsistent value ("dirty read")?**

   This can happen if:

   - App 1 and App 2 perform concurrent writes, but the read is done during a window where only one of the writes is applied.
   - For example, if `$inc +10` is applied but `$set 0` is not yet.

   In general, MongoDB avoids true dirty reads at the document level since single-document operations are atomic. But in more complex cases, especially with multi-document transactions or if secondaries lag, it can seem inconsistent.

3. **In what case is the "read-your-own-writes" violated?**

   This happens if a client:

   - Writes to the primary,
   - Then immediately reads from a secondary that hasn't caught up yet.

   The client won't see its own write. This breaks "Read-Your-Own-Writes" consistency, a form of session consistency.

MongoDB: Primary-Secondary Replication

1. **Single-Document Writes Are Atomic**

   - Atomicity is guaranteed only within a single document.
   - MongoDB uses locks internally to serialize updates to a document.
   - This means two updates to the same document will not interleave, but instead occur one after the other.

   We don't need transactions for updates like `{$inc: 1}` or `{$set: "value"}` on a single document—they're safe.

2. **All Writes Go to the Primary Node**

- MongoDB uses single-leader replication:
  - The primary node handles all writes.
  - Writes are recorded in the Oplog (operation log).
  - Secondaries read the oplog and replay the operations.

This design enforces a single ordering of writes, which simplifies consistency (compared to peer-to-peer replication).

3. **Replication is Asynchronous by Default**
   - Writes do not wait for secondaries to apply the oplog.
   - This increases performance but allows:
     - Stale Reads (secondaries behind primary).
     - Lost Writes on Failover (see below).
   - You can configure write concern to enforce synchronous replication if desired.

     ```
     db.collection.insert({ ... }, { writeConcern: { w: "majority" } })
     ```

4. **Possible Inconsistencies**
   - **Stale Reads -** You read from a secondary that hasn't caught up to the latest write on the primary.
   - **Dirty Reads -** A secondary might partially apply operations, especially with multi-document transactions, leading to temporarily invalid views.
   - **Read-Your-Own-Writes Violation -** If a client writes to the primary, then reads from a lagging secondary, it won't see its own write.

   MongoDB's read preference system must be carefully configured to avoid this.

5. **Rollbacks and Data Loss**

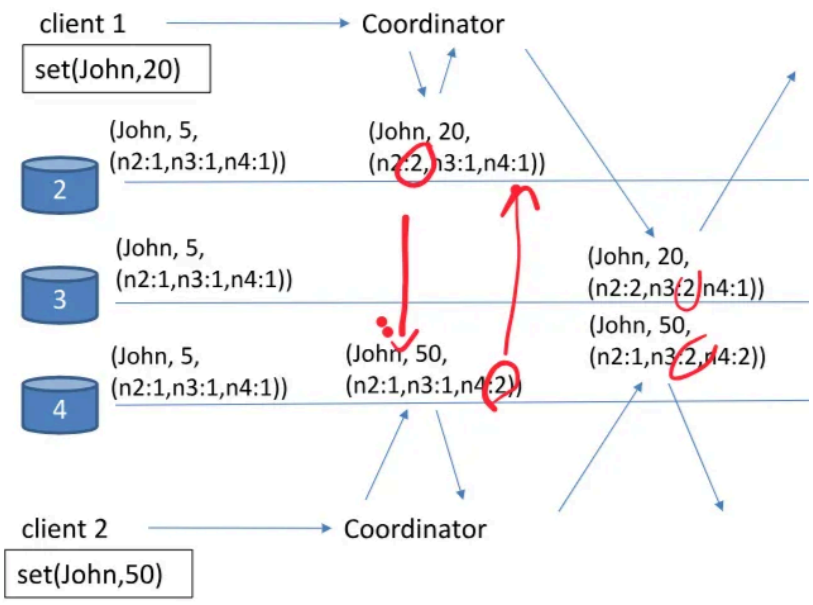
   This is critical in failover scenarios:
   - If a primary accepts writes, but fails before replication, those writes are not durable.
   - When the former primary rejoins as a secondary, it rolls back those unreplicated writes to match the new primary's state.

   MongoDB trades off strict durability for availability unless you use strong write concerns. Use `writeConcern: "majority"` *so a write only succeeds when replicated to a majority of the replica set.*

6. **Multi-Document Operations Require Distributed Transactions**

   These require more coordination and are costly in distributed settings, especially across multiple shards.

Concurrent Writes in P2P Distributed Databases



- **Initial State**: All nodes have:

`John = 5` , `VV = {n2:1, n3:1, n4:1}`

- Client 1 sends the set `set(John, 20)` operation and Client 2 send the set `set(John, 50)` operation, coordinator nodes send this operation to the respective nodes
- Client 1's coordinator sends to n2 and increments its counter → `VV_2 = {n2: 2, n3: 1, n4: 1}`

  Client 2's coordinator sends to n4 and increments its counter → `VV_4 = {n2: 1, n3: 1, n4: 2}`

The VV's are not comparable since $VV_2 > VV_4$ and $VV_2 < VV_4$ on different indexes → Concurrent writes.

- Client 1's coordinator sends to n3 and increments its counter → `VV_2 = {n2: 2, n3: 2, n4: 1}`

  Client 2's coordinator sends to n3 and increments its counter → `VV_4 = {n2: 1, n3: 2, n4: 2}`

Are both writes successul?

Both writes are successful, because both clients reached a quorum (2 out of 3 nodes: nodes 2, 3, 4).

Are the VVs comparable?

VV's are not comparable, because the version vectors are concurrent. Neither version is causally after the other ⇒ it is a true write conflict.

What happens to the VVs?

If `.allow_mult(true)` is enabled, both VVs are stored as siblings with their respective values.

```
[
  { "value": 20, "VV": {n2:2, n3:2, n4:1} },
  { "value": 50, "VV": {n2:1, n3:2, n4:2} }
]
```

What is returned to the clients?

Both values are returned to the client on read

(Client 1 gets `{John, 20}` and Client 2 gets `{John, 50}`):

- The client application receives all siblings.
- The app must then merge or choose one value.

Once resolved, the application writes back a new value:

- The chosen value is written with an updated VV.
- All replicas replace the siblings with this new version.

Failover Process

A failover process is needed for **Primary-Secondary Database Systems**

In a primary-secondary architecture (e.g., MongoDB, PostgreSQL replication):

- Only the primary can accept writes.
- If the primary node fails, no writes can occur until a new primary is elected.
- This creates a single point of failure unless automatic failover is in place.

Therefore, a failover process is essential to:

- Detect primary failure.
- Elect/promote a new primary.
- Reconfigure clients to write to the new leader.

In primary-secondary systems, the primary is a bottleneck and a single point of failure. Systems like MongoDB require a replica set with majority quorum to maintain high availability.

Summary Replication

Single leader replication:

- Primary node (the leader) handles all write requests.
- Followers/secondaries replicate the data by applying the writes in order.
- If the leader fails, a failover process elects a new leader.

Advantages:

- Provides strong consistency if secondaries are read-only.

Disadvantages:

- Write availability is limited to the leader.
- Failover is complex and can be slow.
- May not scale well under heavy write loads.

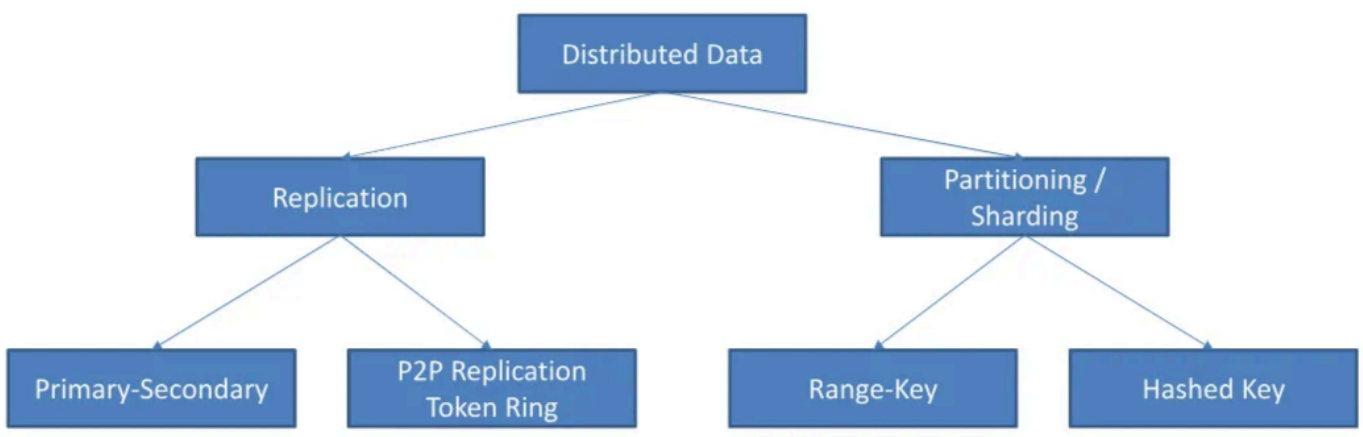
Peer-to-peer replication:

- Any available replica can accept writes (no central leader).
- Writes are accepted if a quorum of nodes responds.
- Unavailable replicas may miss writes temporarily and catch up later.
- *"No failover"* → there's no single point of failure.
- Concurrent Write Conflicts:
  - Since any replica can write independently, concurrent writes to the same key can result in conflicts.
  - Requires conflict detection and resolution mechanisms (like version vectors or CRDTs).
- Eventual Consistency:
  - Eventually each node sees every operation. Eventually all nodes have the same state.
- Strong Eventual Consistency (SEC):
  - Using data structures that guarantee a conflict-resolution merge.

Finally:

- Single-leader replication is easier for strict consistency but limits availability.
- P2P replication prioritizes availability and fault tolerance, but requires mechanisms (like CRDTs) to ensure convergence.

Distributed Databases



Replication

> Replication involves maintaining multiple copies of the same data across different nodes. This helps ensure data availability and fault tolerance.

Common with both SQL and NoSQL databases:

- In SQL, replication is often used for read scaling and failover.
- In NoSQL, replication is used to enhance availability and data redundancy across nodes.

Partitioning/Sharding

> Sharding is the process of dividing a database into smaller, more manageable parts called shards, which are distributed across multiple nodes.

More common in NoSQL databases:

- NoSQL databases like MongoDB, Cassandra, and Couchbase use sharding to scale out across multiple nodes as the dataset grows.
- Sharding is usually not done in traditional SQL databases due to the complexity of joins and transactions that are difficult to manage across distributed parts of the data.

Goals of Sharding:

1. **Even Distribution of Workload**:

   The goal is to ensure that the data and queries are evenly distributed across the available nodes to avoid overloading any single node. This allows the system to handle increased load and scale more efficiently.

2. **Adding/Removing Nodes Incrementally and Easily**:

   Sharding should allow for flexibility in scaling the system. Nodes can be added or removed from the system without significant disruption to operations. This is a critical feature for systems that need to handle growing data volumes or traffic.

Why Sharding is More Common in NoSQL:

- **SQL Use Joins**: SQL databases often rely on joins to combine related data from different tables. Joins become challenging in a distributed environment, where related data might not reside on the same node.
- NoSQL databases are designed to be schema-less and can scale horizontally. They favor denormalized data models that are optimized for quick reads and write-heavy applications.

  NoSQL solutions typically avoid joins by embedding related data in the same document (e.g., MongoDB) or by using techniques like eventual consistency to simplify data distribution.

Shard Key Strategies

1. **Hashed Key**

   The shard key is hashed, and the hash value determines the distribution of data across nodes.

   How it works:

   - Each key is hashed, and the resulting hash value is used to decide which node stores the data. This method distributes the data as evenly as possible across all nodes.

   Advantages:

   - Even distribution of data across nodes, reducing the chances of hotspots (nodes that are overloaded with data).

   Disadvantages:

   - **Breaks locality**: Since data is distributed randomly based on the hash value, related data might end up on different nodes. This can be problematic if related records need to be queried together, as it can result in more expensive cross-node operations.

   Example systems: Rlak, Cassandra, Dynamo, Couchbase, Redis, and MongoDB.

2. **Range Key**

   The shard key is used to define **ranges**, and each node holds a specific range of data.

   How it works:

- For example, if the shard key is a postal code, the ranges could be `10000-30000`, `30000-60000`, etc. Each range of values is assigned to a node.

Advantages:

- Related data is stored together on the same node. This makes it easier to retrieve consecutive records (e.g., range queries like "get all records between postal codes `10000` and `30000`").

Disadvantages:

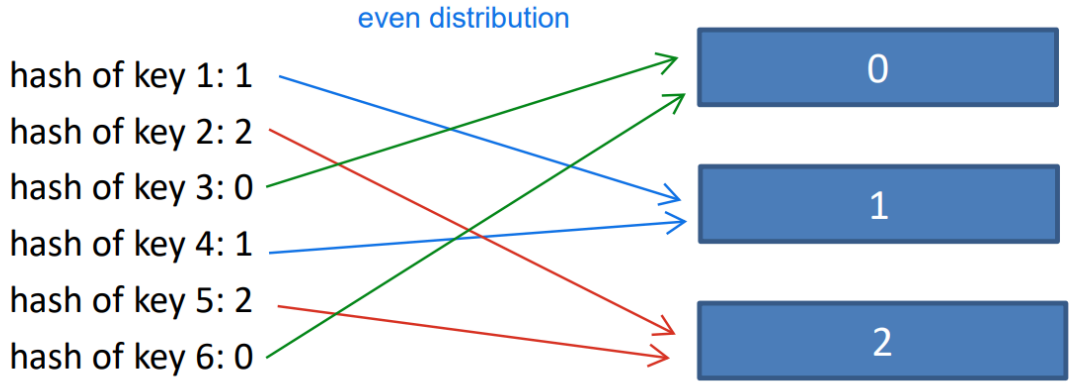
- **Uneven distribution**: If certain ranges have significantly more data than others (e.g., a popular postal code range), it could lead to hotspots or an imbalance in data distribution.

Example systems: MongoDB, HBase, CockroachDB.

Hashed Shard Key in P2P distributed database: Why Token Ring?

Example with 3 nodes and 3 hash, Keys: 1, 2, 3, 4, 5, 6, 7, 8, 9.

Naive hashing → $hash(key) \bmod n$



Adding another server node 4:

Key	key % 3	key % 4	Result
3	0	3	Moved to N3
4	1	0	Moved to N0
5	2	1	Moved to N1

Key	key mod 4	Node
7	3	Node 3
8	0	Node 0
9	1	Node 1

Naive hashing using `hash(key) % N` causes **most or all keys to move** when the number of nodes `N` changes.

This leads to a **major redistribution of data**, which is inefficient and unscalable.

Basic Consistent Hashing

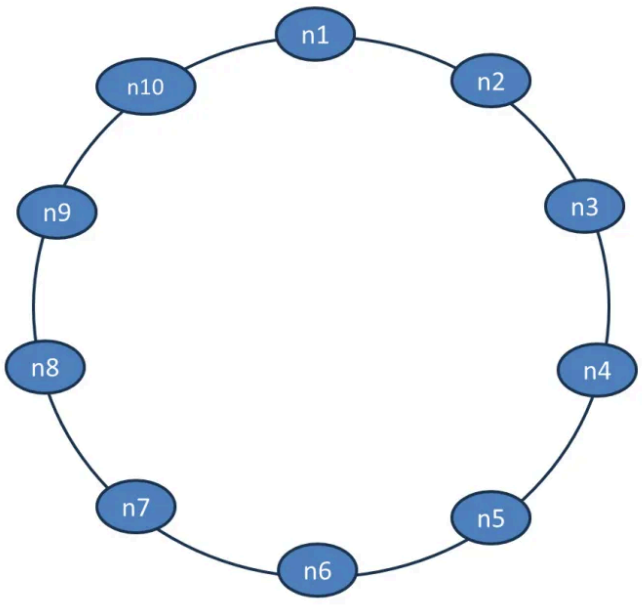
Treat the output of the hash function as a circular space (a “ring” or “token ring”), rather than a linear list of buckets.

- **Hash Space → Ring**:
  - Imagine the hash function output as a circular ring (e.g., values from `0` to `2³² - 1`).
  - The maximum hash value wraps around to the minimum.
- **Nodes Get Positions on the Ring**:
  - Each server (or virtual node) is assigned a position on this ring using a hash of its ID (e.g., `hash("node1")`).
  - These become the anchor points for data responsibility.
- **Keys Get Hashed Too**:
  - Each key is hashed to a position on the same ring.
  - A key is stored on the first node clockwise from its hashed position.

Advantages of Consistent Hashing:

- **Minimal Movement on Change**:
  - When a node is added/removed, only a small portion of keys are remapped (those between the new/old node and its neighbor).
  - This allows incremental scaling with minimal disruption.
- **Even Distribution** (with virtual nodes):
  - If physical nodes are unevenly distributed, virtual nodes help spread the load evenly.
- **Fault Tolerance**:
  - Data can be replicated to the next k clockwise nodes (replication factor), making the system resilient.

Hashed Shard Key on the Token Ring – Consistent Hashing



1. Define hash space: the output range of the hash function, from minimum value to maximum value. Map the hash space onto the logical ring (hash ring).

   Example: Hash Space: 1 - 10000

2. Map each server node to a position on the ring.

   Example: n1 → 1000, n2 → 2000

3. Position the nodes onto the correct position on the ring.

4. Calculate hash for each key.

   Example: John hashes to 1500

5. Locate hashed key on the token ring.

   Example: Key John is placed between node1 and node2

6. Each key is stored on the node that is closest to the hashed key in clockwise direction. Traverse ring clockwise to locate the host node.

   Example: Key John is stored on node 2

Remove (decommission) of n2 from the ring:

- assume that databases (key spaces) have a rf = 3:
- Hash space affected: `1001–2000` (since `n2` was responsible for that range).
- When node `n2` is removed:
  - All keys in its range must be transferred to the next clockwise node on the ring (`n3`).
  - This is efficient because only the data for the segment `1001–2000` is moved.

    - This is a key benefit of consistent hashing—only a small portion of the total data is affected.

Replication Factor (rf = 3)

The system replicates each piece of data to 3 nodes:

- Primary node: owns the key's range.
- Two successors (clockwise on the ring): replicas.

This ensures fault tolerance and high availability.

Example: Key in range 1500 originally replicated on n2, n3, and n4. After n2 is removed, now likely on n3, n4, and n5.

Add server node7a to position 7500

- Hash space `7001–7500` must be moved from node 8 to node 7a.
- This is a localized change, again showing the efficiency of consistent hashing.
- However, it introduces some replica rebalancing:
    - If data in `7001–7500` was replicated to `n10`, that's no longer necessary.
    - Now `n7a` takes over, and replication chain may shift (e.g., `n7a → n8 → n9`).

Even though consistent hashing minimizes overall movement, the nodes directly affected still carry a temporary load spike.

That's why newer systems (like Cassandra) use virtual nodes (vnodes), where each physical server takes many positions on the ring. This reduces hotspots during redistribution.

Virtual Nodes (vnodes)

Instead of assigning one hash position per physical node, you assign multiple positions (virtual nodes) to each physical server.

- Each physical server is split into many virtual nodes, each owning a portion of the hash ring.
- Vnodes are mapped randomly around the ring.
- A node can thus be responsible for multiple, non-contiguous hash ranges.

> "The system generates random tokens (positions) for each vnode on the ring."

- These tokens represent unique hash ranges assigned to each vnode.
- Important that each vnode has a unique position, so each key maps to exactly one vnode.

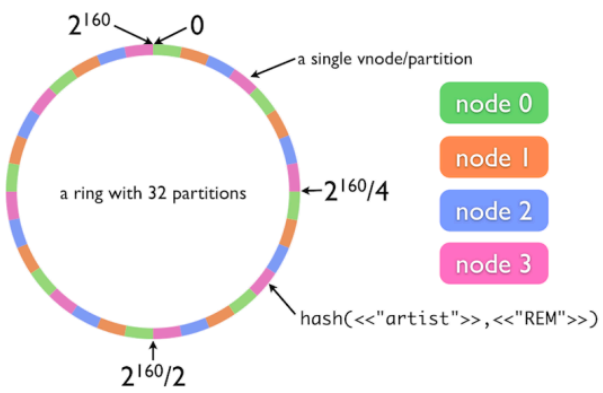
> "positions of vnodes need to be unique..."

Otherwise, multiple nodes might claim ownership of the same key range—breaking determinism.

Benefits:

1. **Smoother Data Redistribution**:
    - When a node is added/removed, its vnodes are spread across the ring.
    - This causes small pieces of data to move to/from many different servers, reducing overload on any single node.
2. **Better Load Balancing**:
    - Vnodes make it easier to evenly distribute data and replication responsibilities, even when physical servers differ in capacity.
3. **Simpler Scaling**:

- Adding a new server = assign it several vnodes → less impact per existing server.
- The system avoids the "few overloaded nodes".



hash space: 2^x

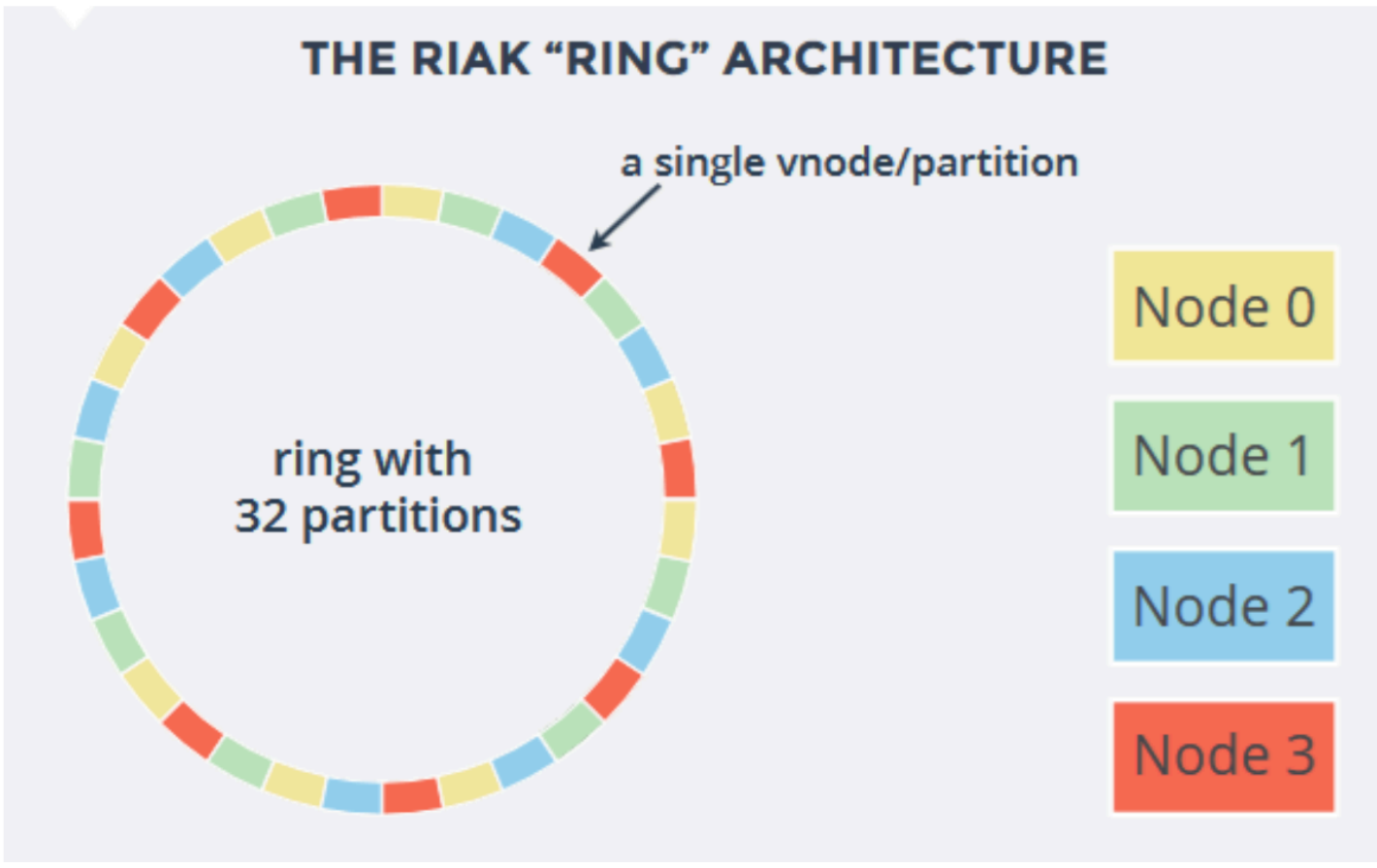
2^{20} = 1.048.576 hash values
2^{160} hash values

in this case we have 8 Virtual nodes, 2^{160} hash values guarantees that there is no 2 virtual nodes that collide.

What is wrong with the vnode ring picture?

green vnode is always followed by orange vnode so if we remove green vnode then the orange one will have to take all the load, or if we skip the orange one and it is replicated on the green and blue node, then the pink one will take all the workload.

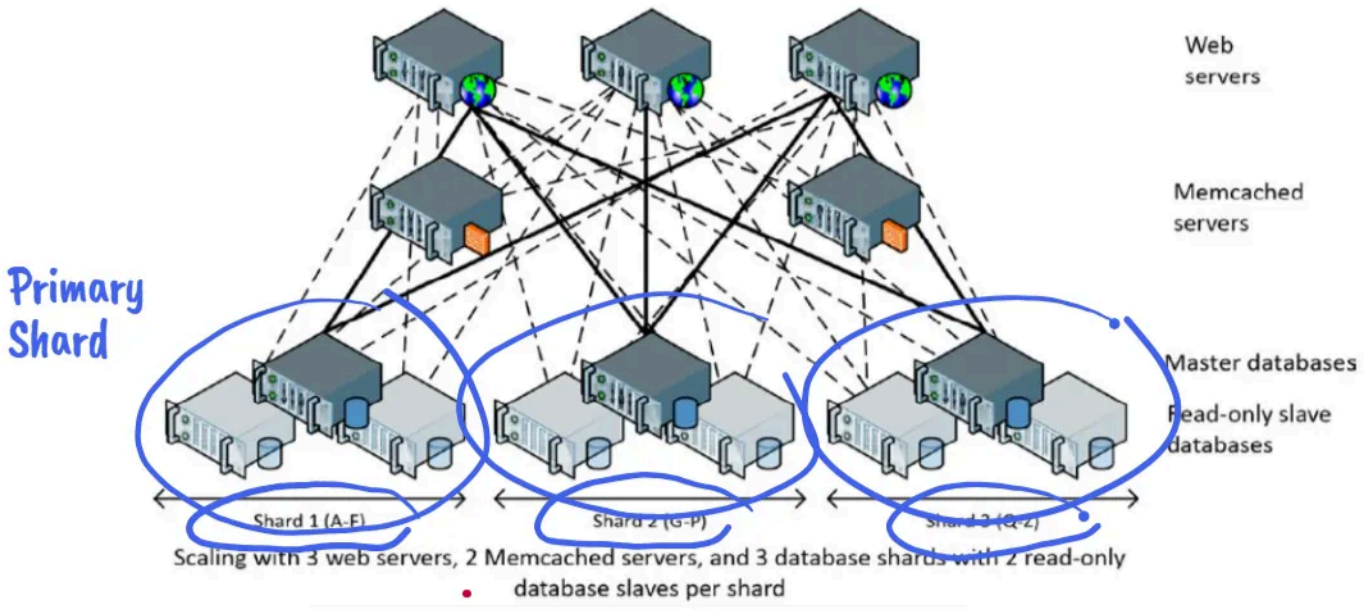
RIAK Cluster



we want virtual nodes to be distributed randomly, if we take down blue node then firstly red node takes workload, then yellow, then red, then yellow agian and so on.

this is the correct implementation.

Sharding



Even though replication and sharding are different concepts, they typically are combined together.

In production systems (like MongoDB clusters):

- Each shard is a replicated unit (a replica set).
- So the system scales with sharding but remains resilient due to replication.

Example: MongoDB

- A shard = a replica set (usually 3 nodes: 1 primary + 2 secondaries)
- Documents are distributed across shards based on a shard key
- Each shard internally replicates its portion of the data

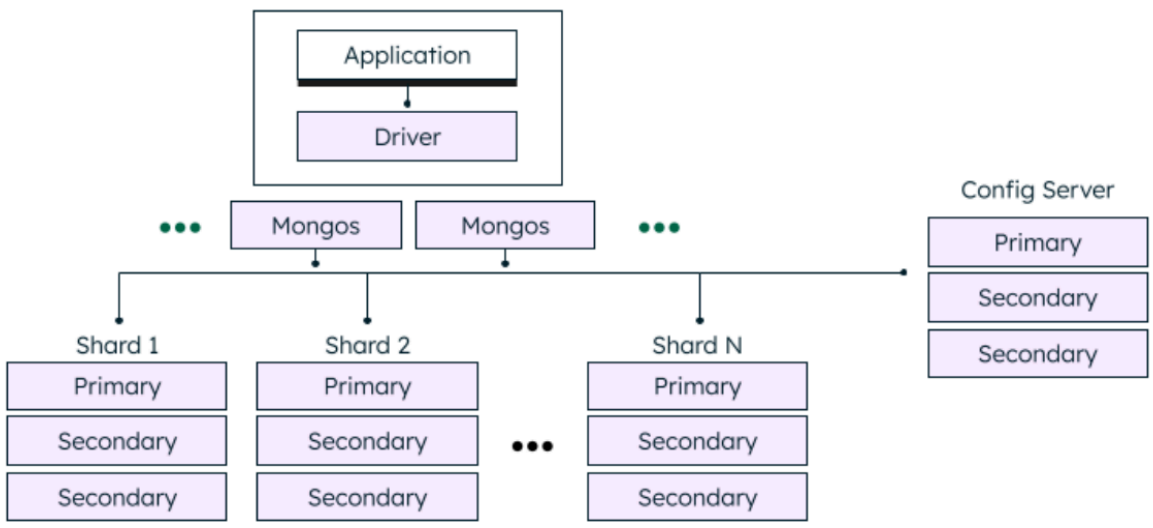
MongoDB Sharding

1. **Data is Partitioned into Subsets (Chunks)**
   - In MongoDB, the collection's data is divided into chunks.
   - A chunk is a range of documents defined by a shard key (e.g., a range of `postalCode` values).
   - These chunks are the units of data movement between shards.
2. **Nodes = Shards (Physical or Virtual Servers)**
   - A shard in MongoDB is either:
     - A physical machine
     - A virtual server
   - Each shard contains multiple chunks.
   - MongoDB's design ensures that chunks are spread across shards for balance.
3. **Each Shard is a Replica Set**
   - To ensure high availability, each shard is not a single server, but a replica set.
   - A typical shard might include:
     - 1 Primary (handles writes)
     - 2+ Secondaries (replicas for failover and read scaling)
4. **More Chunks than Shards (Best Practice)**

   > “The recommendation is to have more chunks than shards.”

   Why?
   - Enables finer-grained balancing:
     - The balancer process in MongoDB can move chunks between shards to maintain even data distribution.

- Having more chunks:
  - Helps prevent data skew.
  - Improves the system's ability to rebalance dynamically as data grows or shards are added.



The Config Server stores all metadata and configuration information for the sharded cluster.

It knows:

- Which chunks exist
- What shard key ranges they cover
- On which shard each chunk currently resides

Because the config server tracks the global cluster state, it is crucial for routing and balancing.

Examples of what it supports:

- When a query comes in, the mongos router consults the config server to find out which shard(s) to query.
- When chunks need to be moved (e.g., for load balancing), the balancer uses config server data to plan the migration.

> "The Config Server must also be deployed as a replica set."

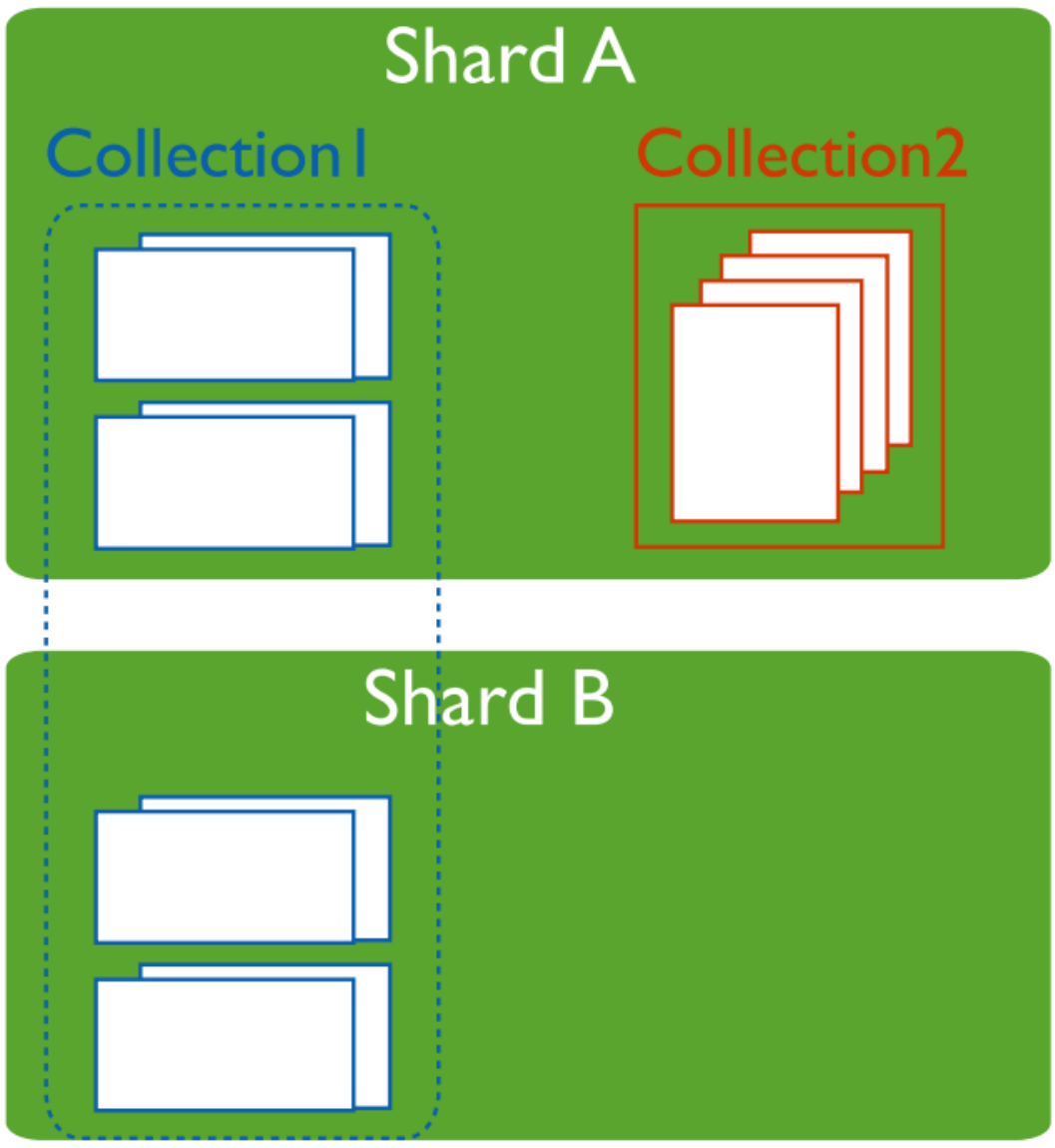
This is to ensure high availability and fault tolerance:

Previously, config servers were a single point of failure.

Now, MongoDB requires a minimum of 3 config server nodes in a replica set to:

- Prevent downtime
- Avoid metadata corruption
- Ensure consistent reads

Mongo Shards – Primary Shard – Chunks and Shards



1. **Each Database Has a "Primary Shard"**
   - In a MongoDB sharded cluster, each **database** (not just each collection) is assigned a **primary shard**.
   - This shard is responsible for:
     - Holding all **unsharded collections** within that database.
     - Acting as the default storage location unless explicitly sharded.

   Primary node *= leader of a replica set (for replication).*

   Primary shard *= home for unsharded collections of a specific database (for partitioning).*
2. **MongoDB Partitions Collections into Chunks**
   - Once a collection is sharded, MongoDB splits it into chunks.
   - Each chunk:
     - Covers a range of shard key values.
     - Is the smallest unit of distribution and balancing.
     - Can be migrated between shards for load balancing.

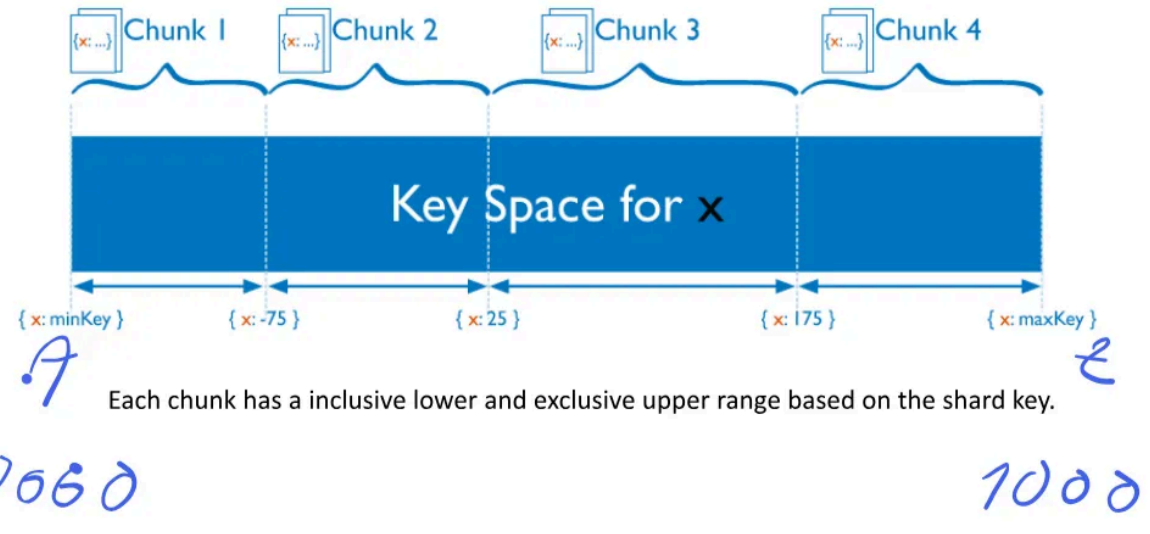
   Example:

   ```
   postalCode: 10001–20000 → Chunk A → Shard 1
   postalCode: 20001–30000 → Chunk B → Shard 2
   ```

   Chunks are tracked by the config server and moved automatically as needed.

For write-heavy or large collections, always shard them early, or you risk overloading the database's primary shard.

Range-Key example



Each Chunk Covers a Key Range

- MongoDB divides data into chunks, and each chunk has:
  - An inclusive lower bound (included)
  - An exclusive upper bound (not included)

This is expressed as: `[minValue, maxValue)`

This notation ensures no overlaps between chunks and complete coverage of the shard key space.

Let's assume the shard key is `postalCode`:

Chunk	Range
Chunk A	`[10000, 20000)`
Chunk B	`[20000, 30000)`
Chunk C	`[30000, 40000)`

A document with `postalCode = 40000` → outside current range → MongoDB will create a new chunk if needed.

Range-based sharding:

- Enables efficient range queries (e.g., `postalCode > 30000 AND < 40000`)
- Keeps related data physically together

But:

- If values are not uniformly distributed, one chunk (or shard) can become a hotspot.

Example:

If 90% of data has postalCode = 1000–1999, then the chunk covering that range will grow rapidly, requiring chunk splits and rebalancing.

Selection of a Range-Shard Key

In MongoDB, a shard key is a field (or set of fields via a compound index) that determines:

- How documents in a collection are divided into chunks
- How chunks are distributed across shards

A well-chosen shard key ensures even distribution of documents and supports efficient queries.

MongoDB offers two ways to determine chunk ranges:

1. **Automatic Splitting (MongoDB-Determined)**

   MongoDB:

   - Monitors insert volume and document size.
   - Automatically splits chunks when they exceed a size threshold (default: ~64MB).
   - Generates non-overlapping ranges based on observed shard key values.

   Good for: simple use cases, or when data arrives uniformly.

2. **Manual Range Planning (Developer/DBA)**
    - The developer or DBA predefines shard key ranges.
    - This gives more control over distribution.
    - Useful when:
        - You know the data distribution pattern (e.g., certain postal codes are very frequent).
        - You want to avoid hotspots manually.

The shard key is either a single indexed field or multiple fields covered by a compound index that determines the distribution of the collection's documents among the cluster's shards.

A ‘good’ shard key needs to distribute documents evenly and fit query patterns.

Key characteristics of a good shard key:

Trait	Explanation
High Cardinality	Many unique values → allows more chunks
Uniform Frequency	Values appear with similar frequency → balanced load
Query Targeting	Frequently used in filters/indexes → improves routing efficiency

A **bad shard key** leads to:

- **Chunk imbalance** (some shards overloaded)
- **Inefficient queries** (when queries don’t use the shard key)
- **Chunk migration bottlenecks**

MongoDB Sharding: Code

MongoDB does not automatically shard data—you need to explicitly define what to shard and how.

1. Create an Index on the Shard Key Field:

    ```
    db.<collection>.createIndex({ "<shard key field>": 1 })
    ```

    - You must create an index on the field you want to use as the shard key.
    - This can be:
        - A single field, e.g., `{ postalCode: 1 }`
        - Or a compound key, e.g., `{ region: 1, date: -1 }`
    - The shard key must be indexed to support efficient routing and chunk splitting.
2. Enable Sharding on the Database:

    ```
    sh.enableSharding("<database>")
    ```

    - This tells MongoDB to allow sharding for the specified database.
    - It doesn’t shard collections yet—just prepares the cluster to accept sharded collections.

    Internally, this:

    - Registers the database with the config server
    - Assigns a primary shard for its unsharded collections
3. Shard the Collection:

    ```
    sh.shardCollection("<database>.<collection>", { "<shard key field>": 1 })
    ```

- This command tells MongoDB to **partition the collection** based on the chosen shard key.
- MongoDB will:
  - Create the first chunks using the shard key range
  - Distribute those chunks across shards based on the balancer's logic

- Once a shard key is set, it cannot be changed (without re-creating the collection).
- The field used as the shard key must exist in all documents.
- If your application grows, it's best to design the shard key early based on expected access patterns.

Good Range-Key for Teacher Collection

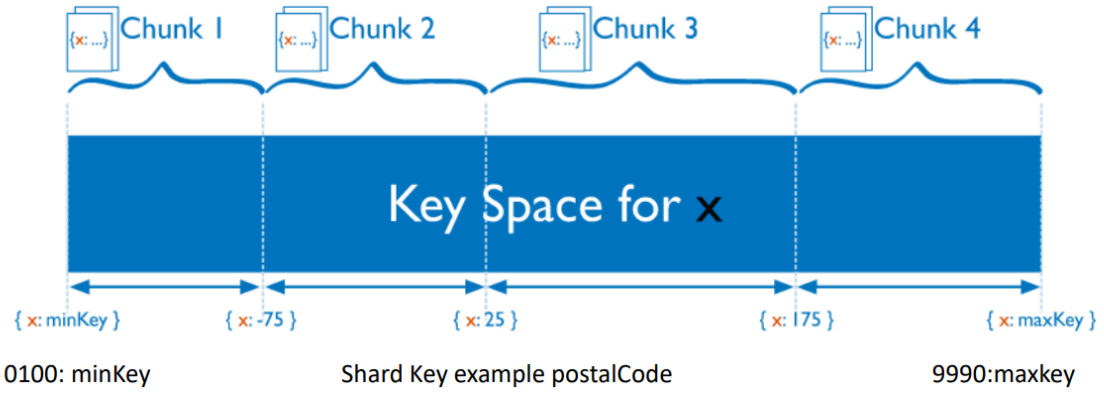

```
{
  "_id": {  "$oid": "640d683998ac446f58fe76f4" },
  "teacherID": "2",
  "name": "Doe",
  "mail": "doe@we-are.xx",
  "DoB": "1998-01-15",
  "gender": "f",
  "Country": "Georgia",
  "postalCode": 4600,
  "remark": null,
  "subjects": ["Java", "Neo4J"],
  "references": [
    { "fName": "Donald", "recommendation": "Great teacher" },
    { "fName": "Flower", "recommendation": "learned a lot" }
  ]
}
```


Attribute	Why It Could Be Good	Why It Could Be Bad
`_id`	- Globally unique, good for hashed key sharding	- Not suitable for range queries - Value is opaque, meaningless for sorting or filtering
`teacherID`	- High cardinality - Easy to use as a unique identifier	- Incremental values → new inserts cluster at one end → hotspot under range-based sharding
`name`	- Often indexed and queried	- Alphabetical order rarely aligns with actual query needs - Low randomness → skewed ranges
`mail`	- High cardinality (likely unique)	- Rarely filtered or sorted on - Not meaningful for range-based access patterns
`DoB`	- Suitable for time-based range queries (e.g., filter by age or year)	- If not aligned with common queries, becomes irrelevant - Clustered cohorts can create data imbalance
`gender`	- Always present	- Extremely low cardinality (does not have enough different values) → 1–2 shards overloaded with data
`Country`	- Could help divide data geographically	- Likely dominated by one value (e.g., Georgia) - Low cardinality
`postalCode`	- Commonly used in filters - Supports range queries	- Requires careful chunk planning to avoid hotspots in dense urban codes
`remark`	- Technically present in schema	- Mostly null or unused, No value as a shard key
`subjects`	- Relevant to business logic	- Is an array, which MongoDB does not allow as a shard key

Shard key for teacher collection that distributes data roughly evenly?

compound range shard key: `{country: 1, postalCode: 1}`, If data is extremely skewed, switch to: `{country: 1, postalCode: "hashed"}` or just `{postalCode: "hashed"}` to maximize even distribution.

Shard Key for Teacher Collection



1. **Shard Key Cardinality**

   > "The cardinality of a shard key determines the maximum number of chunks the balancer can create."

   - Cardinality = number of unique values in a field.
   - Higher cardinality → more fine-grained partitioning → more chunks possible.
   - Lower cardinality → fewer chunks → risk of uneven distribution.

   It would be Ideal to have a high cardinality field with reasonably uniform frequency of values.

   Each unique shard key value can exist on no more than a single chunk at any given time.

2. **Shard Key Frequency**

   > "The frequency of the shard key represents how often a given shard key value occurs in the data."

   - Even if cardinality is high, if a few values dominate, you get data skew.
   - MongoDB assigns chunks to ranges of values, but if many documents fall into one range, that chunk grows quickly and becomes a hotspot.

   Example:

   > "Postal codes with 01XX (Tbilisi) are much more frequent than other postal codes."
   >
   > High cardinality (many postal codes), But uneven frequency (most teachers live in one city) → leading to hotspots and chunk imbalance.

Consequnces of data skew:

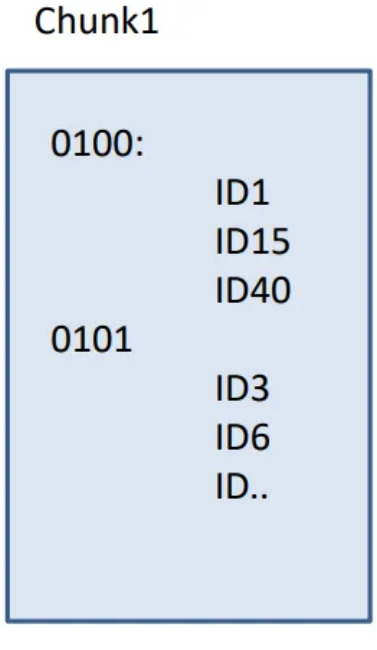
Chunks corresponding to Tbilisi (`01XX`) grow faster than others. Shard with that chunk will:

- Consume more disk
- Handle more queries
- Become a performance bottleneck

MongoDB's balancer tries to split and move chunks, but frequent splits can cause:

- Increased I/O
- Cluster-wide rebalancing load

Shard Key Modifications



What happens,

if the value of a shard key changes (e.g. teacher moves to a different location, gets a different postalCode)?

- MongoDB does not allow updates to the shard key value once the document is inserted.
- The only way to “change” the shard key is to:
  1. Delete the document
  2. Insert a new document with the new shard key value

If a value for the shard key is missing because shard key is not a mandatory field?

- MongoDB requires the shard key field to be present in every document.
- If it's missing, the document cannot be routed to a shard.

Outcome would be that the insert will fail:

- The `insert()` or `update()` command will throw an error like: `Missing shard key field`

Hashed Shard Key

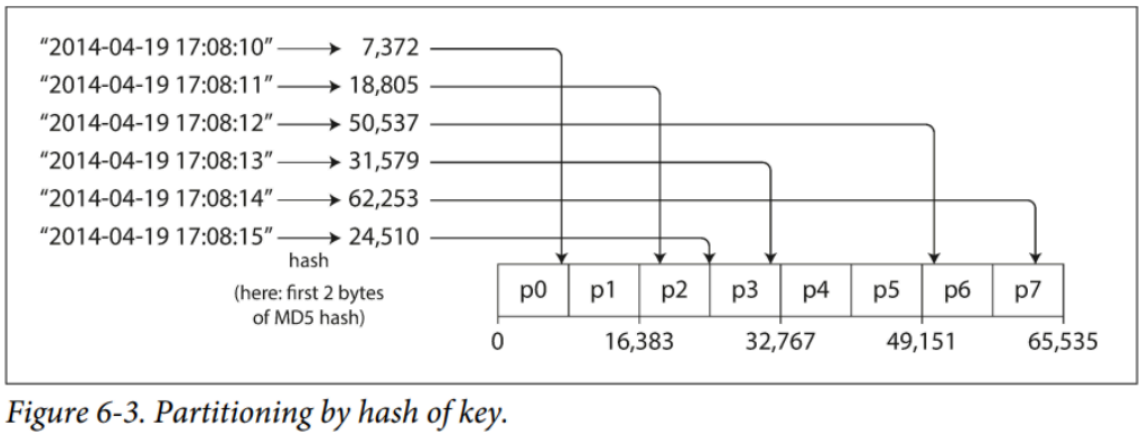


Figure 6-3. Partitioning by hash of key.

Used when primary concern is even distribution:

- Hashing ensures that shard key values are evenly spread across the hash space.
- This avoids hotspots, where one shard handles most of the traffic.
- It’s ideal for workloads with:
  - Heavy write throughput
  - Common values (e.g., same postal codes, countries, etc.)

Used for automatic sharding:

- MongoDB can automatically:
  - Split the hash space into chunks
  - Distribute chunks across shards
- No need for the database administrator to define manual ranges.

Hash ranges are assigned to partitions (shards):

- Internally, MongoDB:
  - Hashes each shard key value.
  - Divides the hash space (e.g., `minHash` to `maxHash`) into chunks.

- Assigns each chunk to a shard.

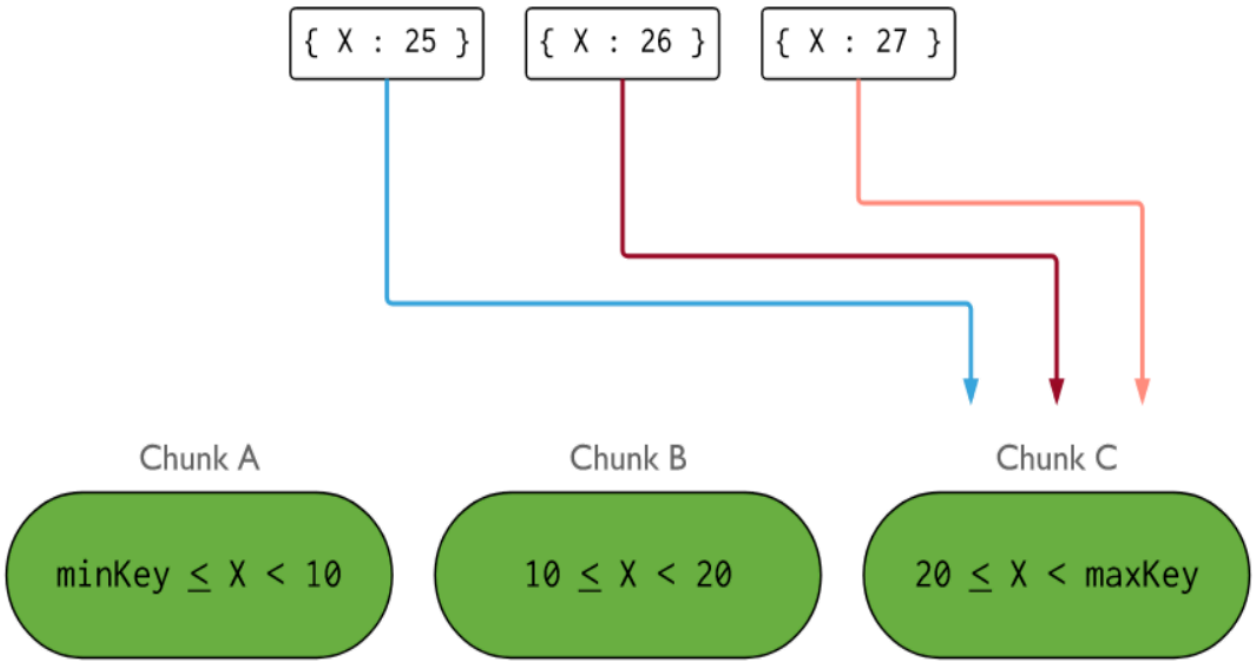
This means data is partitioned by hashed value, not the original value.

Objects that with a range key would be stored close together are now randomly distributed:

- This is the main trade-off:
  - You lose locality.
  - For example, postal codes `0100` and `0101`, which are close together, may hash to very different values and land on different shards.
- Result: range queries are no longer efficient—they must broadcast to all shards.

Range vs. Hashed Shard Key

Ranged Shard Key:

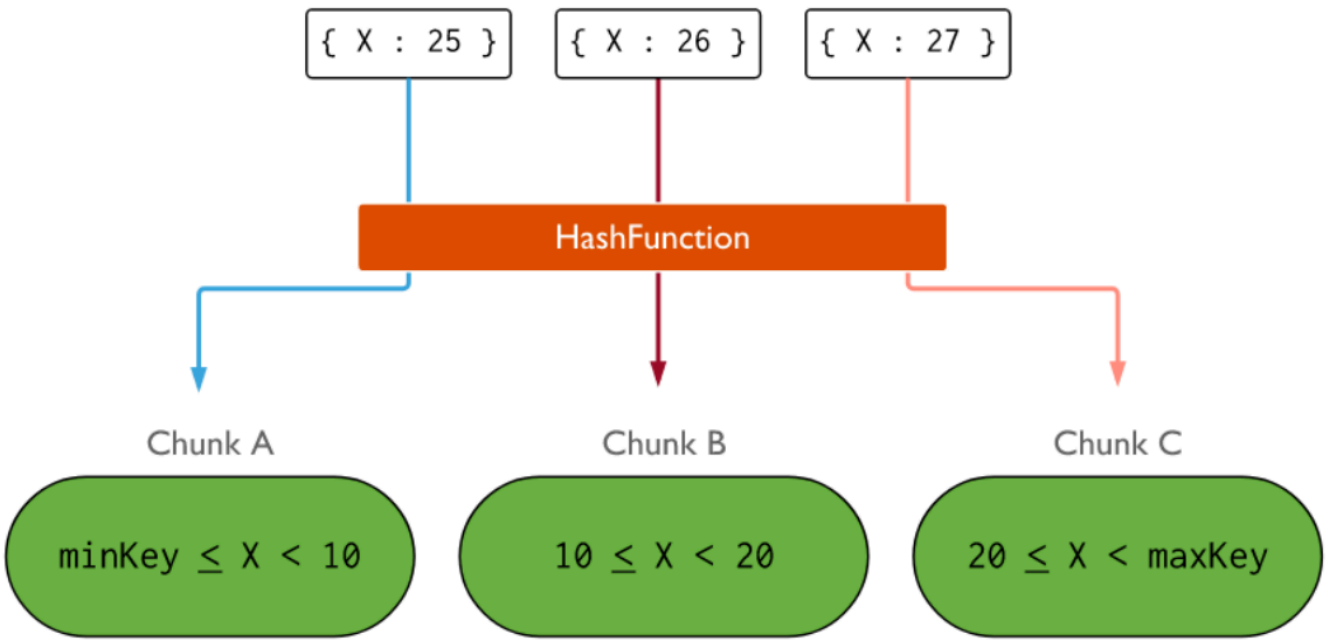


- MongoDB uses the raw value of `x` to determine which chunk a document belongs to.
- It compares the value directly against defined chunk ranges:
  - Chunk A: `minKey ≤ x < 10`
  - Chunk B: `10 ≤ x < 20`
  - Chunk C: `20 ≤ x < maxKey`

Result:

- All documents `{x: 25}`, `{x: 26}`, `{x: 27}` fall into Chunk C.
- That means:
  - Chunk C gets overloaded
  - Risk of a hotspot
  - Load is not balanced across shards

Hashed Shard Key:



- Before checking chunk ranges, MongoDB first applies a **hash function** to the value of `x`.
- `{x: 25}` → `hash(25)` → lands in, say, Chunk A
- `{x: 26}` → `hash(26)` → lands in Chunk B
- `{x: 27}` → `hash(27)` → lands in Chunk C

Result:

- Documents are evenly distributed across chunks (and therefore shards), regardless of how close the original values were.
- This prevents hotspots and ensures balanced load.