Comprehensive Lecture Notes: Distributed MongoDB

Architecture, Replication, Sharding, and Consistency

Based on Course Material Enriched with External Technical Context

Contents

1 Introduction to Distributed Systems in MongoDB

MongoDB transforms from a single-server database into a distributed system through two primary mechanisms: **Replica Sets** and **Sharded Clusters**. While a single instance suffices for development or low-traffic applications, production environments almost invariably require distributed deployments to ensure data safety and performance.

1.1 Core Concepts

- **High Availability (Replica Sets):** Ensures the system remains operational even if hardware fails. It involves copying data across multiple servers.
- **Horizontal Scalability (Sharded Clusters):** Solves the problem of data volume exceeding the storage or processing capacity of a single server by distributing portions of data across multiple machines.

2 High Availability: Replica Sets

A **Replica Set** is a group of 'mongod' processes that maintain the same data set. It is the foundation of data protection in MongoDB.

2.1 Architecture

A standard Replica Set consists of:

1. **Primary Node:** The single entry point for all *write* operations. It records all changes to its datasets in its operation log (oplog).
2. **Secondary Nodes:** These nodes replicate the Primary's oplog and apply the operations to their own data sets to stay synchronized. By default, they serve read operations only if explicitly configured (Read Preference).

2.2 The Oplog (Operations Log)

The **Oplog** is a special capped collection that keeps a rolling record of all operations that modify the data stored in your database.

- **Mechanism:** When a client writes to the Primary, the operation is logged in the Primary's oplog. Secondaries asynchronously query the Primary's oplog and apply those operations to themselves.
- **Idempotency:** Operations in the oplog are idempotent. This means applying the same oplog entry multiple times yields the same result, which is crucial for data consistency during network retries.
- **Example:**

  ```
  { op: "i", ns: "test.users", o: { _id: 1, name: "Ali" } }
  ```

  Here, 'op: "i"' indicates an insert operation into the 'users' namespace.

2.3 Automatic Failover and Elections

High availability is achieved through automatic failover.

- **Heartbeats:** Members of the set send heartbeats (pings) to each other every 2 seconds.
- **Election Trigger:** If a Primary does not respond within a specific timeout (usually 10 seconds), the Secondaries mark it as inaccessible.
- **Consensus Protocol:** MongoDB uses a consensus algorithm (inspired by Raft) to elect a new Primary. Eligible Secondaries vote, and the node with the most up-to-date data (and majority votes) becomes the new Primary.
- **Client Recovery:** MongoDB drivers automatically detect the new Primary and reroute operations. The downtime typically lasts 2 to 5 seconds.

2.4 Specialized Node Types

- **Arbiter:** A node that holds no data and exists solely to provide a vote in elections. Useful for maintaining an odd number of voting members to break ties without the storage cost of a full Secondary.
- **Hidden Node:** A Secondary that maintains a copy of the data but is invisible to client applications. It cannot become Primary. Often used for heavy reporting or analytics workloads to isolate them from operational traffic.
- **Delayed Node:** A Secondary that intentionally lags behind the Primary by a fixed amount of time (e.g., 1 hour). It serves as protection against human error (e.g., accidental 'DROP DATABASE'), allowing administrators to recover data from a state prior to the error.

3 Consistency and Durability

In distributed systems, there is a trade-off between consistency (all nodes see the same data at the same time) and performance/availability. MongoDB allows developers to tune this via **Write Concerns** and **Read Concerns**.

3.1 Write Concern

Write concern describes the level of acknowledgement requested from MongoDB for write operations.

- `w: 1` **(Default):** Acknowledged by the Primary only. Fast, but data could be lost if the Primary crashes before replicating to Secondaries.
- `w: majority` Acknowledged only after the data has been written to the Primary *and* a majority of Secondaries. This guarantees data safety against rollback in case of failover.
- `w: 0` Fire-and-forget. No acknowledgement is returned. Highest performance, lowest reliability (rarely used in production).

3.2 Read Concern

Read concern controls the consistency and isolation properties of the data read from replica sets.

- `local`**:** Returns the most recent data available on the node. No guarantee that the data has been written to a majority of nodes (subject to rollback).

- `available`**:** Similar to local but for sharded clusters; lowest latency but lowest consistency.
- `majority`**:** Returns data that has been acknowledged by a majority of the replica set. Guarantees the data is durable and will not be rolled back.
- `linearizable`**:** Guarantees that the read returns data reflecting all successful writes that completed before the read started. This is the strongest consistency level but incurs a performance penalty as the Primary must verify with Secondaries that it is still the Primary.

4 Horizontal Scaling: Sharding

Sharding partitions data across multiple machines. It is required when the dataset exceeds the storage capacity of a single machine or when the write throughput exceeds the I/O capacity of the Primary.

4.1 Sharding Architecture

1. **Mongos (Query Router):** A lightweight proxy service. Applications connect to ‘mongos‘, which analyzes the query and routes it to the specific Shard(s) containing the relevant data.
2. **Config Servers (CSRS):** A dedicated Replica Set that stores the metadata of the cluster. This includes the mapping of data chunks to specific shards. If Config Servers are down, the cluster metadata becomes read-only (no chunk migrations).
3. **Shards:** Each Shard is a Replica Set that holds a subset of the total data. Collectively, all shards hold the complete dataset.

4.2 Data Distribution Mechanics

- **Shard Key:** A specific field (or fields) in a document chosen to partition the data. This is the most critical decision in sharding strategy.
- **Chunks:** MongoDB divides the shard key range into contiguous ranges called "chunks" (default size 64MB).
- **Balancer:** A background process that monitors the number of chunks on each shard. If the distribution becomes uneven, it automatically migrates chunks from overloaded shards to underutilized ones to ensure even load distribution.

4.3 Shard Key Strategies

Choosing the right shard key is vital for performance.

4.3.1 Ranged Sharding

Data is partitioned based on ranges of the shard key values.

- **Pros:** Efficient for range queries (e.g., "Find users with IDs between 100 and 200") because the router can target specific shards.
- **Cons: Monotonic Keys** (e.g., timestamps or auto-incrementing IDs) cause "Hotspotting." All new writes go to the shard holding the upper range (the "max" chunk), creating a write bottleneck on a single shard.

4.3.2 Hashed Sharding

Data is partitioned based on an MD5 hash of the shard key.

- **Pros:** Ensures an even distribution of writes across shards, preventing hotspots even with monotonic keys.
- **Cons:** Range queries become inefficient (Scatter-Gather). A query for a range of keys is broadcast to all shards because hashed values of sequential keys are not sequential.

4.4 Query Routing

- **Targeted Query:** If the query includes the Shard Key, 'mongos' routes it directly to the specific shard holding that data. This is efficient.
- **Scatter-Gather Query:** If the query does not include the Shard Key, 'mongos' must broadcast the query to *all* shards and merge the results. This increases latency and load.

5 Distributed Transactions

Prior to version 4.0, ACID transactions were limited to a single document. Modern MongoDB (v4.2+) supports multi-document ACID transactions across shards.

5.1 Two-Phase Commit (2PC)

MongoDB implements transactions using a variant of the Two-Phase Commit protocol to ensure atomicity across distributed components.

1. **Phase 1 (Prepare):** All participating shards prepare to commit the transaction. They lock the relevant data and ensure they can persist the changes.
2. **Phase 2 (Commit):** If all participants vote "Yes", the transaction is committed. If any participant fails, the transaction is aborted (rolled back) across all shards.

5.2 Considerations

While powerful, distributed transactions incur significant performance overhead due to network coordination and locking.

- **Latency:** Increased due to coordination between shards.
- **Usage:** They should be reserved for scenarios requiring strict atomicity (e.g., financial transfers) rather than standard operations.

6 Operational Management

6.1 Backup and Restore

Backing up distributed databases requires capturing a consistent state across multiple nodes.

- `mongodump/mongorestore`: Logical backups. Reads data and writes it to BSON files. Good for small datasets but slow for recovery.
- **Filesystem Snapshots:** Physical backups. Taking snapshots of the underlying storage volume (e.g., EBS snapshots). Fast and suitable for large datasets. For sharded clusters, snapshots must be coordinated across all shards and config servers to ensure cluster-wide consistency.

6.2 Monitoring Tools

- `mongostat`: Provides a real-time view of database status (insert/query/update/delete rates, memory usage).
- `mongotop`: Tracks the amount of time a MongoDB instance spends reading and writing data, broken down by collection.
- **Key Metrics:**
  - **Replication Lag:** The time delay between the Primary and Secondaries. High lag puts data at risk during failover.
  - **Chunk Imbalance:** Indicates if the Balancer is failing to distribute data evenly.
  - **Page Faults:** Indicates if the working set exceeds available RAM (disk thrashing).