Replication in MongoDB

Replication in **MongoDB** is an essential **feature** that enables **high availability** and **redundancy by creating multiple copies of data across different nodes in a distributed system**.

What is Replication in MongoDB?

Replication in **MongoDB** involves **creating copies of the same dataset across multiple nodes**. This is achieved using a **structure** called a **replica set**. A **replica set** is a group of **MongoDB servers (nodes)** that maintain the **same dataset, providing data redundancy and failover support**.

Key Components of a MongoDB Replica Set

1. **Primary Node**: The **primary node** is the **main node** where **all write operations are directed**. It holds the **original, up-to-date copy of the dataset**.
2. **Secondary Nodes**: **Secondary nodes replicate data** from the **primary node**, keeping their **datasets synchronized**. These copies can be used for **read operations** and as **backups in case the primary node fails**.
3. **Arbiter**: An **arbiter** is a special type of **member in the replica** set that **does not store any data**. Its **role** is to participate in the **election process** when **choosing a new primary node**, ensuring **high availability** without additional data storage requirements.

How Replication Works in MongoDB

- **MongoDB** uses a **master-slave approach for replication**. The **primary node** acts as the "**master**" where all **write operations are initially applied**, while **secondary nodes** serve as "**slaves**," replicating the primary node's data.
- In case of **primary node failure**, an **election** is held among the **replica set members to promote one of the secondary nodes to primary, ensuring uninterrupted data access**.

Example: Replicating a Collection in MongoDB

Suppose we want to replicate a collection called Customers in a replica set. Here's how the data would be distributed:

- **Primary Node** (N1): Stores the **main copy** of the Customers collection.
- **Secondary Node** (N2): Stores a **replica** of the Customers collection.
- **Arbiter Node** (N3): Participates in **elections** but **does not store any data**.

This setup ensures that if **N1** fails, MongoDB can automatically promote **N2** as the primary, with **N3** helping in the election process.

Configuring a Replica Set

To set up a **replica set**, you must:

1. Start each MongoDB instance with the **--replSet** option to enable replication.
2. Initialize the replica set on the primary node.
3. Add secondary nodes to the replica set.

Syntax and Example

1. Start MongoDB Instances: Start MongoDB instances with replication enabled. Here, we start three nodes on different ports.

```
mongod --port 27017 --dbpath /data/db1 --replSet myReplicaSet
mongod --port 27018 --dbpath /data/db2 --replSet myReplicaSet
mongod --port 27019 --dbpath /data/db3 --replSet myReplicaSet
```

2. Initiate the Replica Set: Connect to the first instance (primary node) using the Mongo shell and initiate the replica set.

```
mongo --port 27017
rs.initiate(
  {
    _id: "myReplicaSet",
    members: [
      { _id: 0, host: "localhost:27017" },
      { _id: 1, host: "localhost:27018" },
      { _id: 2, host: "localhost:27019", arbiterOnly: true }
    ]
  }
)
```

- Explanation:
  - **_id**: Specifies the name of the replica set (myReplicaSet).
  - **members**: Lists all nodes in the replica set.
  - **host**: Defines the address of each node.
  - **arbiterOnly**: true: Sets the node on port 27019 as an arbiter, meaning it will not store data but will participate in elections.

3. Verifying the Replica Set Status: Run the following command in the Mongo shell to check the status of the replica set.

```
rs.status()
```

- This command shows the health of each node, including which node is the primary and which are secondaries.

Operations in a Replica Set

1. **Write Operations**

All **write operations** (such as inserting, updating, or deleting documents) are **directed** to the **primary node**. MongoDB then propagates these **writes to the secondary nodes**, **ensuring data consistency across the replica set**.

```
db.Customers.insertOne({ "name": "John Doe", "email": "johndoe@example.com" })
```

- Explanation:
  - This command inserts a document into the Customers collection.
  - The write is first applied to the **primary node** and then replicated to **secondary nodes**.

2. **Read Operations**

MongoDB allows you to **configure read preferences**, enabling applications to specify which **nodes to read from**. By default, **all reads go to the primary node to ensure the latest data**. However, you can change this to read from **secondary nodes**, which can help **distribute read load**.

```
db.getMongo().setReadPref("secondary")
db.Customers.find()
```

- Explanation:
  - **setReadPref("secondary")** configures the **MongoDB shell** to **read from a secondary node instead of the primary**.
  - This can be useful in **read-heavy applications** where slightly stale data is acceptable, helping to **reduce the load on the primary node**.

Automatic Failover and Election Process

In a MongoDB **replica set**, if the **primary node goes down**, **MongoDB automatically initiates an election process among the remaining nodes to select a new primary**. The **arbiter** helps in this **election**, ensuring that a **new primary can be elected if there is an even number of voting members**.

For example:

1. **Primary Node Failure**: If the primary node (**N1**) fails, the secondary nodes (**N2**) and the arbiter (**N3**) vote for a new primary.
2. **Secondary Promotion**: One of the secondary nodes (**N2**) is promoted to **primary**.

3. **Continuous Operation**: The **system** continues to **operate without downtime**, providing **high availability for both reads and writes**.

Key Points in MongoDB Replication

- **Primary Node**: The **main node** where **writes occur**.
- **Secondary Nodes**: **Replicate data** from the **primary for redundancy**.
- **Arbiter**: A lightweight member for **election purposes only**; **does not store data**.
- **Replication Mechanism**: **Write operations** go to the **primary** and are **propagated to the secondaries**.
- **Read Preferences**: Allow you to specify whether to **read** from the **primary or secondary nodes**.

This setup helps ensure **high availability**, **data redundancy**, and **automatic failover**. MongoDB replication is ideal for applications that need to **stay online and accessible, even in cases of individual node failures**.

Sharding in MongoDB

When a **MongoDB collection contains a massive number of documents or requires large storage space, placing all data on a single node can lead to performance issues**. To address this, **MongoDB** provides **sharding**, a **feature that partitions data across multiple nodes**. **Sharding** improves **performance by distributing data and workload across multiple servers**, thus achieving **load balancing and horizontal scalability**.

What is Sharding in MongoDB?

Sharding in MongoDB is the **process of dividing a large dataset into smaller parts, known as shards**. Each **shard** holds a **subset of data**, allowing for **parallel processing** and **optimized resource usage**. This technique is **essential** when dealing with **collections with large amounts of data or high concurrency needs**.

- **Horizontal Partitioning**: **MongoDB's** **sharding** is also known as **horizontal partitioning**. It distributes documents into disjoint partitions across different nodes, allowing each node to handle a fraction of the workload.
- **Horizontal Scaling**: **Sharding** enables the **system to add more nodes as needed**, distributing **data across these nodes** to **handle increased demand and maintain performance**.

Key Concepts in Sharding

1. **Shards**: Each **shard** is a **subset** of the **total data** and operates as a **separate MongoDB instance**.
2. **Shard Key**: The **shard key** is a field (or fields) in each document used to **partition the data into shards**.
3. **Chunks**: **Chunks** are **contiguous ranges of shard key values**, which are distributed across shards. MongoDB automatically balances chunks across shards to maintain even data distribution.
4. **Query Router (mongos)**: The **query router** directs client **requests to the appropriate shards**. It keeps **track of which shards contain which parts of the dataset**, ensuring **queries** are **efficiently routed**.

Types of Partitioning in MongoDB

MongoDB supports **two methods** for **partitioning collections** into **shards**:

1. **Range Partitioning**
2. **Hash Partitioning**

Both methods require a **shard key** to define the basis for partitioning. The shard key should:

- Exist in every document within the collection.
- Have an index to enable efficient querying.

1. **Range Partitioning**

Range partitioning divides data into ranges based on the **shard key values**. Each range represents a **chunk**, and **documents** are **distributed across shards based on the shard key values falling within specific ranges**.

- Example: If the shard key has values from 1 to 10 million, you could define ranges such as:
  - 1 to 1,000,000
  - 1,000,001 to 2,000,000
  - …
  - 9,000,001 to 10,000,000

Each **chunk** would contain **all documents** whose **shard key values fall within one of these defined ranges**.

2. **Hash Partitioning**

Hash partitioning uses a **hash function** applied to the **shard key values**. **Documents are assigned to chunks based on the hash value, leading to a more randomized distribution of data**.

- Example: A hash function, h(K), is applied to each shard key K. Documents with similar shard key values are distributed across multiple chunks, improving load balancing.

When to Use **Range Partitioning** vs. **Hash Partitioning**

- **Range Partitioning**: Best for collections frequently queried using range queries (e.g., fetching documents with shard key values between specific ranges).
- **Hash Partitioning**: Ideal for collections with one-document retrieval patterns. It ensures an even distribution of documents across shards by randomizing the placement of shard key values.

Setting Up Sharding in MongoDB

To enable sharding on a MongoDB collection, follow these steps:

1. Enable Sharding on the Database
2. Define a Shard Key
3. Shard the Collection

Example: Sharding a Collection in MongoDB

Let’s shard a collection called Orders using a field orderId as the shard key.

1. Connect to the Query Router: First, connect to the mongos instance (query router) in the Mongo shell.
2. Enable Sharding on the Database:

```
sh.enableSharding("myDatabase")
```

- This command enables sharding on myDatabase.

3. Create an Index on the Shard Key Field:

db.Orders.createIndex({ orderId: 1 })

- Shard key fields require an index. Here, we create an index on orderId in the Orders collection.

4. Shard the Collection:

sh.shardCollection("myDatabase.Orders", { orderId: 1 })

- This command shards the Orders collection using orderId as the shard key. MongoDB will automatically start creating chunks based on the values of orderId.

Query Routing and Load Balancing

In a **sharded MongoDB setup**, all **queries** (CRUD operations) are submitted to the **query router** (mongos), which **directs each query to the relevant shard(s)**.

1. **Efficient Query Routing**: The **query router** maintains **metadata** to know which **shard holds which chunks**. For example, if a query is searching for a document with orderId of 500000, the router will direct the query to the shard containing this chunk, making retrieval more efficient.
2. **Broadcasting Queries**: If the **system cannot determine which shards hold the required documents** (e.g., when the shard key is not specified), **the query is broadcast to all nodes**.

Combining Sharding and Replication

MongoDB can use **sharding** and **replication** together:

- **Sharding** focuses on **improving performance** by **distributing the data load**.
- **Replication** ensures **high availability** by **maintaining copies of data on multiple nodes**.

For example, each shard in a sharded cluster can be a replica set, allowing MongoDB to balance the data load and provide failover support in case of node failure.

Example Scenario: **Range Partitioning and Hash Partitioning**

Consider a collection of customer orders that you want to partition based on customerId:

1. **Range Partitioning**: Useful if you frequently need to retrieve all orders from a particular range of customerId values. For example:
   - customerId values 1 to 1000 could be assigned to Shard 1.
   - 1001 to 2000 to Shard 2, and so on.
   - This allows MongoDB to direct range queries to specific shards.

2. **Hash Partitioning**: Useful if you mainly retrieve individual orders by customerId. By hashing the customerId, MongoDB distributes documents more evenly across shards.

Key Points in MongoDB Sharding

- **Sharding**: MongoDB's approach to **horizontal scaling** by **partitioning collections into shards**.
- **Shard Key**: **Field(s)** used to determine **how documents are distributed across shards**.
- **Range Partitioning**: Creates **chunks** based on **specified ranges** of **shard key values**, ideal for **range-based queries**.
- **Hash Partitioning**: Applies a **hash function** to **shard key values**, distributing data more evenly, ideal for **random** or **one-document retrievals**.
- **Query Router (mongos)**: **Routes queries** to the **relevant shards** based on the **shard key** and **sharding configuration**.
- **Sharding and Replication Together**: Combine for **optimal performance**, **load balancing**, and **high availability**.