upGrad

NoSQL DATABASES AND MONGODB -II

In the previous module, you were introduced to one of the most popular NoSQL databases being used in the industry - MongoDB and learnt how to perform basic CRUD operations, run simple and advanced queries as well as perform aggregation stages to analyse the data that you have. Now it's time to take your knowledge of NoSQL a step further and understand the various aspects involved in building the databases, indexing them and performing more advanced operations.

DATA MODELLING IN MONGODB

NoSQL databases, in general, possess a dynamic schema and therefore may lead you to believe that you don't have to pay much attention to the data modelling part. However, if you want to build robust systems which can scale as per your requirement and perform seamlessly, you do have to understand certain nuances involved in schema design.

Data Modelling in MongoDB essentially boils down to the following question —

"Do you want to create references or embedded documents for your database?"

Embedding documents is one of the most essential features of MongoDB that enables you to store one document inside another. This way, you don't have to create multiple tables and join them to get to a specific result. All the necessary information that is needed can be kept in a single document. This form of storing data is also known as a **denormalized** data model.

Here's an example — Let's say you are storing order related information for multiple customers. Rather than storing the **Order** details in one collection and **Customer** information in a separate collection, you can embed the order information in the Customer document directly as shown below:

```
{

    "_id": 1,
"Cust_id": 10101,
   "Name": "Vishwa",
 "Orders":

        {
            "order_id" : 101,
            "Product Name" : "Plantronics",
            "Sales": 2400
        },

        {
            "order_id" : 102,
            "Product Name" : "Novimex",
            "Sales" : 1800
        }
}
```

Creating references is similar to how you used to store information in multiple tables in SQL and each of the tables would have a common field that would help you to relate between them.

Only here in case of MongoDB, you have multiple collections, and one of the fields would be connected. The same data that you saw earlier can be stored in two separate collections as shown below. This form of storing data is also known as a **normalised** data model.

```
#Customers Collection
{
      "_id": 1,
  "Cust_id": 10101,
     "Name": "Vishwa",
  "Orders" : [101,102,....]
}
```

```
#Orders Collection
{
 "_id" : 101,
 "Product Name": "Plantronics",
 "Sales": 2400
}

{
 "_id" : 102,
 "Product Name": "Novimex",
 "Sales": 1800
}
```

Modelling Relationships

- *One-to-One relationship*:

  The one-to-one relationship implies if you currently have two collections, each document from one collection relates to exactly one other document from the second collection and vice-versa. In such cases, it's better to use an **embedded document** to store one document in another rather than utilizing 2 different collections and then using references.

- *One-to-Many relationship*:

  In this case, one document can relate to many other documents but each of the other documents relates back to only one document.in the case of a one-to-many relationship, you can have both the embedded documents as well as references to store the same information. The only basic rule that you need to follow is that —

  - If the number of documents related to one document is less, you can go for embedded documents.
  - Else if the number of orders is more, you can store them in separate collections and use references to create relationships between them.

- *Many-to-Many relationship*:

  A many-to-many relationship happens when one document from a collection can relate to multiple other documents from the same or another collection and vice-versa. In such cases, it's best to use **references** to create relationships between the multiple documents rather than use any kind of embedded documents as it might lead to huge redundancy of data.

upGrad

INDEXING IN MONGODB

Indexes are defined as "*special data structures that store a small portion of the collection's data set in an easy-to-traverse form*."Indexes in MongoDB help in optimising query operations and give us faster results by reducing the number of documents the query needs to go through the relevant results.

Creating an index in MongoDB is pretty straightforward and is given by the following syntax

db.collection.createIndex({<field_name>:<order>})

Similarly, you can drop an index using the following syntax

db.collection.dropIndex({<field_name>:<order>})

You can even analyse your query performance after indexing has happened using a special function called explain() whose syntax is given below

db.purchases.find({<query>}).explain("executionStats")

Types of Indexes

There are following types of indexes that can be created in MongoDB

- **Single Field Index:** Indexes based on one field.
- **Compound Field Index:** Indexes based on multiple fields.
- **Multi-key Index:** Used to index content in arrays with separate indexes for each element.
- **Geospatial Indexes:** Index geospatial coordinate data using 2d indexes and 2dsphere indexes
- **Text Index:** Used for searching strings in a collection.
- **Hashed Index:** Mostly utilized for hash-based Sharding. Support equality matches only.

Single Field and Compound Indexes are two of the most commonly used indexes in MongoDB. Their syntax is as follows

Single Field - *db.collection.createIndex({<field_name>:<order>})*

Compound - *db.collection.createIndex({<field_1>:<order1>,<field_2>:<order2>,....})*

Text indexes are one of the special index types which can be used for searching strings in a given collection.

You can create a text index using the following syntax
db.collection.createIndex({<field>: "text"})

You can search for terms in a field using the following syntax
db.collection.find({$text: { $search: "<term>" } })

Even though indexing makes queries faster and thereby improves the performance of our code, there are some considerations that you must keep in mind before performing indexing. Some practical considerations and best practices when it comes to indexing in MongoDB are :

- Create indexes only for databases and servers that are read-heavy
- Don't over create indexes as they take around 8KB of space and may clog up your RAM and decrease the processing speeds
- Create indexes strategically for fields that are frequently queried together.

AGGREGATION IN MONGODDB - II

From the previous module, the stages of $group and $match is known to us. In this session, we learnt two additional stages which are given as

- **$project**: This will only project the necessary fields that you want in your documents to the next stage in the aggregation pipeline. The syntax for the same looks as follows.

  db.collection.aggregate([{$project:{ specifications }])

  You just need to set the necessary fields that you want to show and only those fields will get shown in the documents to the next stage in the aggregation pipeline.

- **$unwind**: It breaks down the embedded documents and you get a separate list of documents for each of the elements in the embedded document list. The syntax for the same is as follows.

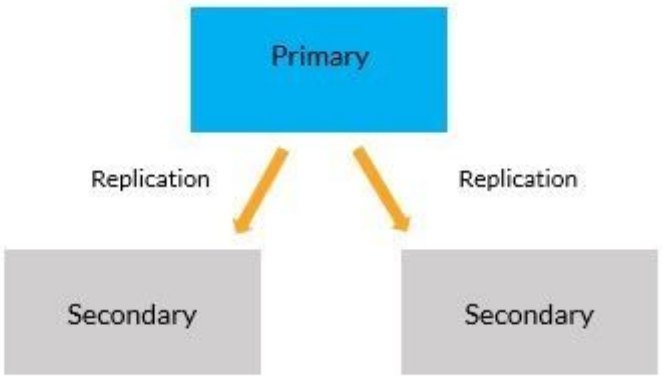
  db.collection.aggregate([{$unwind:{field_name}])

upGrad

REPLICATION

Replication, as the name suggests, implies that we want to create a replica or duplicate of the given data that we have. Replication in MongoDB has a significant impact on the way you can ensure data availability and data redundancy.

Essentially replication in MongoDB refers to the creation of new secondary nodes of data, from a single primary node. All the nodes or servers would contain the same number of documents and would receive different kinds of reading/write requests based on their availability. The replica creation process looks something like this.



The above combination (one primary node and multiple secondary nodes) is called a **replica set**.

Replication gives a significant boost to your data delivery needs. Two of its major advantages are —

- **Data availability:** If a server goes down, another server can take the workload. It improves the fault tolerance and if a server goes down, you have backups of the same data on other servers as well.

- **Improved efficiency:** In case of requests for collections containing a huge number of records (say in millions), the client can send requests to multiple replica sets instead of a single server to improve efficiency.

Replica set architecture

- Only the primary node receives the write operations.
- The secondary nodes only accept read operations.
- The replication of data from primary to secondary happens after the write operations are done. The primary records all the write operations in the operations log or **oplog**.
- The secondary nodes then replicate the primary's operation logs and make the exact same changes to their own replica sets so that they also reflect the primary data set.

Obviously, since only the primary node receives the write operation, it's of paramount importance that this node remains active at all the time to make any updates. However, there may be times when the primary node goes down and therefore, **elections** take place to replace one of the secondary nodes and promote it to a primary.

SHARDING

There are two ways of scaling your database systems to improve its performance.

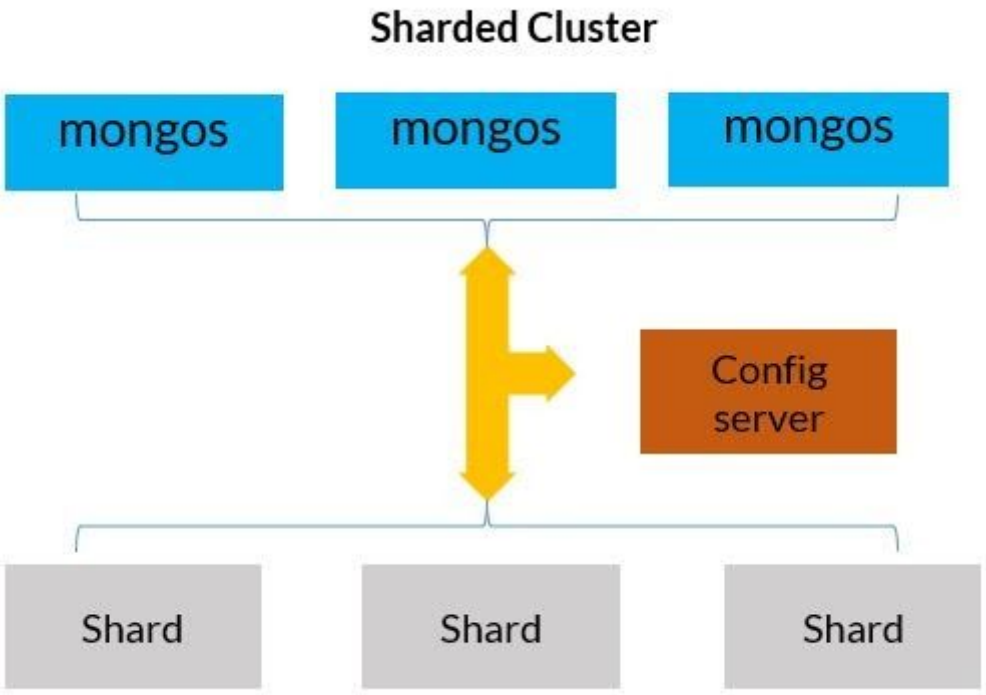
- **Vertical scaling:** Increasing the RAM, CPU, storage space of a single server.
- **Horizontal scaling:** This involves dividing the entire data and workload over multiple servers to keep up with the demand, which can lead to better performance than just improving a single server.

Sharding belongs to the horizontal scaling methodology. Here, the entire collection of a database is divided into several chunks and then each of these chunks is stored in separate replica sets or **shards**. This way, the querying performance improves as it runs only on a specific shard and not the entire collection.

A sharded cluster in MongoDB consists of the following 3 components —

- **Shard:** A portion of the entire data is collected in a shard that can be deployed as a replica set.
- **Mongos:** An interface between the client applications and the sharded cluster and routes the queries from the client to the corresponding shards that are required for operation. It is similar to the **mongo** and **mongod** command-line utilities for the Mongo DB shell and the MongoDB database server respectively that you have been using in the past two modules. The **mongos** utility will act as an intermediate between the client application and the shards.
- **Config server:** Store metadata and configuration settings for the entire sharded cluster.

The interaction between the three is shown in the diagram below.



MongoDB uses a Shard key to distribute the collection's documents across shards. The shard key can be created in the following ways.

- **Hash-based shard key**: Creates a hash value from the given field chosen.
- **Range-based shard key**: This involves dividing data into ranges based on the shard key values.

Querying in a sharded collection now takes place in the following way -

- The query is made by the client application which should contain the shard key
- When a query includes a shard key, the **mongos** uses the config database to route the queries to the corresponding shard instead of searching the entire collection.
- The query results are then routed back to the client through the **mongos**.

The fundamental advantages of sharding are as follows:

- The read/write operations can scale horizontally
- Helps in handling storage capacity for future use.
- High availability of data