UNIT -III

MongoDB Use cases

MongoDB Use Cases: Use Case 1 -Performance Monitoring, Schema Design, Operations, Sharding, Managing the Data, Use Case 2 – Social Networking, Schema Design, Operations, Sharding

PYQ : (Previous Year Mumbai University Question)

Nov – 18

-

Apr -19

Nov – 19

1. Define Monitoring . Explain the factors to be considered while using Monitoring services

Nov – 22

Dec – 23

1. Write a short note on performance monitoring of MongoDB Query

Nov – 24

1. Explain Monitoring Mechanism in Mongodb

Use Case 1 -Performance Monitoring, Schema Design, Operations, Sharding :

Certainly! Let's break down each topic in more detail with simple explanations and examples.

Use Case 1: Performance Monitoring in MongoDB

Performance monitoring involves tracking and analyzing the performance of systems, like servers or databases. In MongoDB, we can manage performance data using schema design, operations, sharding, and data management. Here's how each of these works:

1. **Schema Design**

****What is Schema Design?****

- Schema design refers to the way we organize and structure the data in a database. In MongoDB, the schema is flexible and can be tailored to

fit the type of data being collected. For performance monitoring, the schema will depend on the specific data points that the monitoring tool collects, such as timestamps, server identifiers, and performance metrics like CPU usage.

Simple Explanation:

- Imagine you are monitoring the performance of a computer. You might want to track the time (`timestamp`), which computer you're monitoring (`host`), and how busy the computer is (`cpu_usage`).

Example:

- Here's a simple structure (schema) for this data in MongoDB:

```
{
  "timestamp": "2024-08-10T12:34:56Z",
  "host": "Server_1",
  "cpu_usage": 75
}
```

- In this example, the schema is designed to store the time the data was recorded, the server's name, and the CPU usage percentage. This structure allows you to efficiently store and retrieve the performance data.

---

2. **Operations**

What are Operations?

- Operations in MongoDB are actions we perform on the data. For performance monitoring, key operations include:
  - **Inserting Data**: Adding new records (pieces of data) into the database.
  - **Bulk Insert**: Adding multiple records at once, which is faster when you have a lot of data.
  - **Querying Data**: Retrieving or looking up data to analyze the performance.

Simple Explanation:

- If you're monitoring several computers, you might collect data every minute. You need to add this data to your database and then later retrieve it to analyze how each computer is performing.

Example:

- **Inserting Data:**

```
db.performance.insertOne({
  "timestamp": "2024-08-10T12:34:56Z",
  "host": "Server_1",
  "cpu_usage": 75
})
```

- **Bulk Insert:** If you have data from 3 servers at once:

```
db.performance.insertMany([
  { "timestamp": "2024-08-10T12:34:56Z", "host": "Server_1", "cpu_usage": 75 },
  { "timestamp": "2024-08-10T12:34:56Z", "host": "Server_2", "cpu_usage": 60 },
  { "timestamp": "2024-08-10T12:34:56Z", "host": "Server_3", "cpu_usage": 80 }
])
```

- **Querying Data:** To find all records for `Server_1`:

```
db.performance.find({ "host": "Server_1" })
```

- This query will give you all the performance records for `Server_1`.

---

3. **Sharding**

What is Sharding?

- Sharding is a method used to distribute data across multiple servers. This is especially useful when dealing with large amounts of data, like performance monitoring, where you collect data frequently from many servers.

Simple Explanation:

- Imagine you're monitoring thousands of computers. If you tried to store all that data on one server, it would be too much. Instead, you

split (or shard) the data across several servers. You can do this based on time, server ID, or some other method.

Example:

- **Time-based Sharding:** You could store data from different time periods on different servers.
- **Hash-based Sharding:** You could distribute data evenly across servers by using a hash function on the server's ID.

- **Implementation Example:**

```
db.performance.createIndex({ "host": "hashed" })
db.performance.createIndex({ "timestamp": 1 })
```

- The first index distributes data based on the server ID (using hashing), and the second index organizes data by time. This ensures that your data is evenly spread across multiple servers, improving performance and storage.

---

4. **Managing the Data**

What is Data Management?

- Managing data involves strategies to store, maintain, and delete old data efficiently. MongoDB offers several tools to manage performance data:
  - **Capped Collection**: A fixed-size collection that automatically deletes the oldest records when it reaches its size limit.
  - **TTL (Time-To-Live) Collection**: A collection where records are automatically deleted after a certain period.
  - **Multiple Collections**: Splitting data into different collections based on criteria like time or data type.

Simple Explanation:

- If you only need to keep the last week's performance data, you don't want old data cluttering your database. A TTL collection can automatically remove data older than 7 days. Alternatively, you might use a capped collection to keep only the most recent data, up to a certain size.

Example:

- **TTL Collection:** To keep only 7 days of data:

```
db.performance.createIndex({ "timestamp": 1 }, { expireAfterSeconds: 604800 })
```

- This setup ensures that any data older than 7 days (604800 seconds) is automatically deleted.

- **Capped Collection:** To store up to 1GB of data:

```
db.createCollection("performance", { capped: true, size: 1073741824 })
```

- This configuration ensures that only the most recent 1GB of data is kept, and older data is automatically removed.

Summary

In performance monitoring using MongoDB, each component—schema design, operations, sharding, and data management—plays a crucial role. By carefully designing the schema, efficiently performing operations, distributing data using sharding, and managing data with capped or TTL collections, you can handle large volumes of performance data effectively and ensure that your system remains efficient and scalable.

Use Case 2 – Social Networking, Schema Design, Operations, Sharding :

Social Networking in MongoDB

Social Networking applications involve platforms like Facebook, Twitter, or Instagram, where users interact with each other by posting updates, commenting, liking, and sharing content. MongoDB is often

used for social networking due to its flexible schema and ability to handle large amounts of unstructured data.

Schema Design for Social Networking

Schema Design is the process of organizing the data in a database in a structured way. For social networking applications, you need to design a schema that can efficiently store and retrieve user posts, comments, likes, and user relationships.

Example Explanation:
- **Users Collection:** Stores user information such as username, email, profile picture, and friends list.

- **Posts Collection:** Stores posts made by users. Each post may include fields like `user_id` (reference to the user who made the post), `content` (text or media), `timestamp`, and `comments`.

- **Comments Collection:** Stores comments on posts. Each comment might include fields like `post_id` (reference to the post being commented on), `user_id` (who made the comment), `content`, and `timestamp`.

This schema allows you to easily retrieve all posts by a user, view the comments on a post, and see the user's profile.

Operations - Viewing Posts and Creating Comments

Operations refer to the actions performed on the data stored in the database. In a social networking app, common operations include viewing posts and creating comments.

- **Viewing Posts:** When a user views their feed, the app queries the `Posts` collection to retrieve recent posts made by the user's friends or other users they follow. The query might filter posts based on the user's friends list and order them by `timestamp`.

Example Explanation:

```
db.posts.find({ user_id: { $in: friends_list } }).sort({ timestamp: -1 })
```

This MongoDB query retrieves all posts made by users in the `friends_list`, sorted by the most recent posts first.

- **Creating Comments:** When a user comments on a post, the app inserts a new document into the `Comments` collection.

Example Explanation:**

```
db.comments.insert({
  post_id: "post123",
  user_id: "user456",
  content: "Great post!",
  timestamp: new Date()
})
```

This operation creates a new comment associated with a specific post and user.

Sharding for Social Networking

Sharding is a method of distributing data across multiple servers (or shards) to ensure the database can scale horizontally. In social networking, sharding is essential to manage large amounts of data efficiently.

Example Explanation:

- **Sharding by User ID:** You might shard the `Posts` and `Comments` collections based on the `user_id`. This way, all posts and comments made by a specific user are stored on the same shard, which can optimize read and write operations for that user.

```
shardKey: { user_id: "hashed" }
```

This ensures that user data is evenly distributed across the shards, avoiding hotspots where too much data is concentrated on a single shard.

Benefits of Sharding:

- **Scalability:** As your social networking app grows, you can add more shards to handle the increased load.
- **Performance:** By distributing data across multiple servers, read and write operations can be performed more quickly, improving the user experience.

Summary

- **Social Networking**: Involves platforms where users interact by posting, commenting, and sharing content.
- **Schema Design**: Organizes data into collections like `Users`, `Posts`, and `Comments` for efficient retrieval and storage.
- **Operations**: Include actions like viewing posts and creating comments, involving queries and insert operations.
- **Sharding**: Distributes data across multiple servers for scalability and performance, often based on user identifiers.

This setup allows social networking applications to handle large volumes of data while maintaining fast, efficient access for users.