UNIT -II

The MongoDB Data Model

MongoDB Data Model: The Data Model, JSON and BSON, The Identifier (_id), Capped Collection, Polymorphic Schemas, ObjectOriented Programming, Schema Evolution

PYQ : (Previous Year Mumbai University Question)

Nov – 18

1. Write a Short note on Capped Collection

Apr – 19

1. Explain Capped Collection
2. Explain the concept of inserting explicitly Specifying _id.

Nov – 19

1. Explain Binary JSON (BSON)

Nov – 22

1. Explain _id,Capped Collection ,Binary java Script Notation(BSON)
2. What is Polymorphic Schema ? Explain various reason for using a polymorphic schema

Dec – 23

Nov – 24

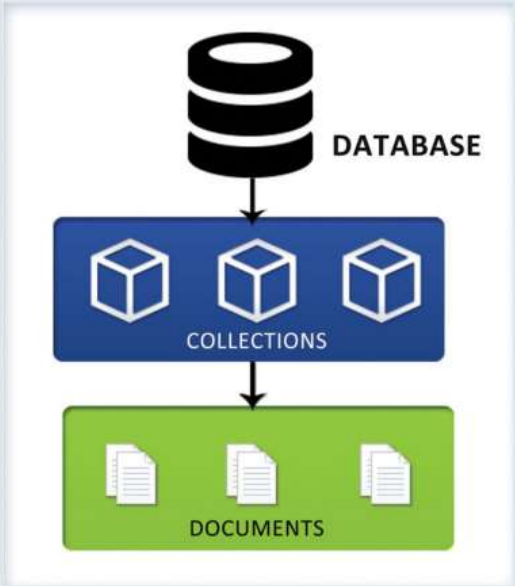
1. Explain the MongoDB database Model and Binary JSON (BSON)

Question 1 : Explain the MongoDB database Model and Binary JSON (BSON)

"MongoDB is designed to work with documents without any need of predefined columns or data types (unlike relational databases), making the data model extremely flexible."

The Data Model :

MongoDB uses a flexible and dynamic data model that is different from traditional relational databases. Here's a simple explanation with examples:



1. **Document Model:**

MongoDB stores data in documents, which are JSON-like objects.

Each document contains key-value pairs, and the structure can vary from one document to another.

Example:

```
json
{
  "name": "John Doe",
  "age": 30,
  "email": "john.doe@example.com"
}
```

2. **Collections:**

Documents are grouped into collections.

A collection is like a table in a relational database, but documents within a collection can have different structures.

Example:

Collection: `users`

Document 1:

```
json
{
  "name": "Alice",
  "age": 25,
  "email": "alice@example.com"
}
```

```
Document 2:
```

```
json
{
  "name": "Bob",
  "age": 28,
  "address": "123 Main St"
}
```

3. **Embedded Documents:**

Documents can contain sub-documents (nested documents) to represent related data.

This reduces the need for joins, making data retrieval faster.

Example:

```
json
```

```
{
  "name": "Jane Smith",
  "age": 27,
  "contact": {
    "email": "jane.smith@example.com",
    "phone": "555-1234"
  }
}
```

4. **Arrays:**

Documents can include arrays to store lists of values or sub-documents.

Useful for storing multiple related items in a single document.

Example:

```
json
{
  "name": "Mark Johnson",
  "hobbies": ["reading", "swimming", "traveling"]
}
```

5. **Denormalization:**

MongoDB often uses denormalization, embedding related data within a single document.

This improves read performance but can increase storage requirements.

Example:

```
json
{
  "title": "MongoDB Basics",
  "author": "John Doe",
  "publisher": {
    "name": "Tech Books",
    "location": "New York"
  }
}
```

6. **References:**

Sometimes, documents reference other documents using unique identifiers.

This is similar to foreign keys in relational databases.

Example:

```
json
{
  "title": "Learning Python",
  "author_id": "60d5f9f3f8b0b2c8c7e77e44"  // Reference to an author's document
}
```

7. **Dynamic Schema:**

MongoDB allows you to change the schema at any time without affecting existing documents.

This flexibility makes it easy to adapt to changing requirements.

Example:

Initially, a document might look like this:

```
json
{
  "name": "Emily Clark",
  "age": 22
}
```

Later, you can add a new field without altering the existing structure:

```
json
{
  "name": "Emily Clark",
  "age": 22,
  "email": "emily.clark@example.com"
}
```

8. Polymorphism:

Different documents in a collection can represent different types of data.

This allows storing diverse data in a single collection.

Example:

Collection: `products`

Document 1:

```
json
{
  "type": "book",
  "title": "MongoDB Guide",
```

```
  "author": "John Doe"
}
```

Document 2:

json

```
{
  "type": "electronic",
  "name": "Laptop",
  "brand": "TechBrand"
}
```

The data models in MongoDB provide great flexibility, allowing you to store and retrieve data efficiently. By using documents, collections, embedded documents, arrays, and dynamic schemas, MongoDB can handle a wide variety of data types and structures, making it a powerful choice for modern applications

---

Question 1 : Explain _id,Capped Collection ,Binary java Script Notation(BSON)

Question 2 : Explain Binary JSON (BSON)

Question 3 : Explain the MongoDB database Model and Binary JSON (BSON)

JSON and BSON :

****JSON (JavaScript Object Notation)****

1. **Overview:**

JSON is a lightweight, text-based format for representing structured data.

It is easy for humans to read and write, and easy for machines to parse and generate.

2. **Structure:**

JSON consists of key-value pairs.

Keys are strings, and values can be strings, numbers, objects, arrays, true/false, or null.

3. **Example:**

```
json
{
  "name": "Alice",
  "age": 30,
  "email": "alice@example.com"
}
```

4. **Usage:**

JSON is commonly used for data exchange between a server and a web application.

In MongoDB, you write queries and interact with data using JSON format.

5. **Advantages:**

Human-readable: Easy to understand and debug.

Language-agnostic: Supported by many programming languages.

Interoperable: Can be easily used for communication between different systems.

6. **Limitations:**

Limited Data Types: Only supports basic data types.

Size: Text-based format can be larger compared to binary formats.

****BSON (Binary JSON)****

1. **Overview:**

BSON stands for Binary JSON.

It is a binary-encoded serialization of JSON-like documents.

Used internally by MongoDB for storing data.

2. **Structure:**

BSON extends JSON by adding more data types (e.g., Date, Binary, int32, int64, etc.).

It includes length prefixes to make parsing faster.

3. **Example:**

The JSON document:

```
json
{
  "name": "Bob",
  "age": 28,
  "birthdate": "1995-06-15T00:00:00Z"
}
```

In BSON, it might look like (in binary format, which is not human-readable):

```
0x16 0x00 0x00 0x00 0x02 name 0x00 0x04 0x00 0x00 0x00 Bob
0x00 0x10 age 0x00 0x1c 0x00 0x00 0x00 0x09 birthdate 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x01 0x00
```

4. **Usage:**

MongoDB stores all documents in BSON format for efficiency.

Data is converted from JSON to BSON when stored and from BSON to JSON when retrieved.

5. **Advantages:**

Efficient Storage: More compact than JSON due to binary format.

Rich Data Types: Supports more data types than JSON (e.g., date, binary data).

Fast Parsing: Length prefixes make it faster to parse and traverse documents.

6. **Limitations:**

Not Human-readable: Binary format is not meant for direct human reading.

Size: Although more compact than JSON, BSON can sometimes be larger due to extra metadata.

Comparison:

Readability:

JSON: Human-readable.

BSON: Binary format, not human-readable.

Data Types:

JSON: Limited to basic types.

BSON: Supports additional types like date, binary, etc.

Efficiency:

JSON: Larger in size, slower to parse.

BSON: More efficient in storage and parsing.

Conclusion:

JSON is used for data interchange and human-readable queries.

BSON is used for efficient storage and fast data retrieval in MongoDB.

MongoDB converts JSON to BSON for storage and back to JSON for queries, combining the strengths of both formats to provide flexibility and performance.

Question 1 : Explain _id,Capped Collection ,Binary java Script Notation(BSON)The Identifier (_id) :

In MongoDB, each document in a collection is uniquely identified by a field called `_id`. This field is essential for the database's operations and ensures that every document can be uniquely and efficiently accessed.

Key Points:

1. **Unique Identifier:**

The `_id` field serves as a unique identifier for each document within a collection.

No two documents in the same collection can have the same `_id` value.

2. **Automatic Generation:**

If you do not explicitly provide an `_id` value when inserting a document, MongoDB will automatically generate one.

The default type for this automatically generated `_id` is an `ObjectId`.

3. **ObjectId:**

`ObjectId` is a 12-byte identifier that is unique across the collection.

It consists of a timestamp, machine identifier, process ID, and a counter.

4. **Custom _id:**

You can specify your own value for the `_id` field. It can be any data type, such as a string, number, or any other BSON type.

However, the value must be unique within the collection.

5. **Indexing:**

The `_id` field is automatically indexed, which means queries that search by `_id` are very fast.

This index is a primary key for the collection, ensuring quick lookups and operations.

Example:

1. **Automatic _id Generation:**

When you insert a document without specifying `_id`, MongoDB generates it:

```
json
db.users.insert({
  "name": "Alice",
  "age": 25
```

```
})
```

The resulting document might look like:

```
json
{
  "_id": ObjectId("60c72b2f9b1e8e1a3d8b4567"),
  "name": "Alice",
  "age": 25
}
```

2. **Custom _id:**

You can specify your own `_id`:

```
json
db.users.insert({
  "_id": "user123",
  "name": "Bob",
  "age": 30
})
```

The document is stored as:

```
json
{
  "_id": "user123",
  "name": "Bob",
  "age": 30
}
```

3. **Querying by _id:**

Queries using `_id` are efficient due to the index:

```
json
db.users.find({ "_id": ObjectId("60c72b2f9b1e8e1a3d8b4567") })
```

This query quickly retrieves the document with the specified `_id`.

4. **Advantages:**

Uniqueness: Ensures each document can be uniquely identified.

Performance: Indexed `_id` provides fast query performance.

Flexibility: Allows for custom identifiers that can be meaningful to the application.

Question 1 : Write a Short note on Capped Collection
Question 2 : Explain Capped Collection
Question 3 : Explain _id,Capped Collection ,Binary java Script Notation(BSON)

Capped Collection :

A capped collection in MongoDB is a special type of collection with fixed size and order

1. **Fixed Size:**

- A capped collection has a pre-defined maximum size.

- Once this size limit is reached, the oldest documents are automatically overwritten by new ones, following a circular buffer model.

2. **Fixed Order:**

- Documents in a capped collection are stored in the order they are inserted.

- The insertion order is preserved, and documents cannot be updated in a way that increases their size.

3. **High Performance:**

- Capped collections offer high performance for insert and read operations.

- They are optimized for scenarios where data is inserted and read in a predictable order.

4. **Use Cases:**

- Ideal for logging, caching, or real-time data applications where only the most recent data is important.

- Examples include storing logs, sensor data, or messages in a chat application.

Example:

1. **Creating a Capped Collection:**

- You can create a capped collection with a maximum size and an optional maximum number of documents.

```
json
db.createCollection("logs", { capped: true, size: 5000 })
```

- This creates a capped collection named `logs` with a maximum size of 5000 bytes.

2. **Inserting Documents:**

- Insert documents as usual:

```
json
db.logs.insert({ "message": "System started", "timestamp": new Date() })
db.logs.insert({ "message": "User logged in", "timestamp": new Date() })
```

3. **Behavior When Full:**

- When the collection reaches its size limit, older documents are automatically removed to make room for new ones.

- For example, if the `logs` collection is full and you insert a new document:

```
json
db.logs.insert({ "message": "New entry", "timestamp": new Date() })
```

- The oldest document (e.g., "System started") will be overwritten by the new document.

4. **Querying:**

- You can query documents in a capped collection just like any other collection:

```
json
db.logs.find()
```

- This will return documents in the order they were inserted.

Advantages:

1. **Predictable Storage:**

- The fixed size ensures that the collection will not grow indefinitely, which is useful for managing storage space.

2. **Efficient Operations:**

- Insert and read operations are very fast due to the fixed order and size constraints.

3. **Automatic Management:**

- MongoDB automatically handles the deletion of old documents, so you don't need to manually manage the collection size.

Limitations:

1. **No Document Growth:**

- Documents cannot be updated in a way that increases their size, as this would disrupt the fixed size constraint.

2. **Limited Flexibility:**

- Once a capped collection is created, its size cannot be changed.

Capped collections in MongoDB are a powerful tool for managing data that requires fixed storage size and order. They provide high performance for specific use cases like logging and real-time data processing. Understanding capped collections can help you efficiently handle scenarios where only the most recent data is of interest.

Polymorphic Schema:

Polymorphic Schemas As you are already conversant with the schemaless nature of MongoDB data structure, let's now explore polymorphic schemas and use cases. A polymorphic schema is a schema where a collection has documents of different types or schemas. A good example of this schema is a collection named Users. Some user documents might have an extra fax number or email address, while others might have only phone numbers, yet all these documents coexist within the same Users collection. This schema is generally referred to as a polymorphic schema. In this part of the chapter, you'll explore the various reasons for using a polymorphic schema .

Object-Oriented Programming :

Object-oriented programming enables you to have classes share data and behaviors using inheritance. It also lets you define functions in the parent class that can be overridden in the child class and thus will function differently in a different context. In other words, you can use the same function name to manipulate the child as well as the parent class, although under the hood the implementations might be different. This feature is referred to as polymorphism. The requirement in this case is the ability to have a schema wherein all of the related sets of objects or objects within a hierarchy can fit in together and can also be retrieved identically.

Let's consider an example. Suppose you have an application that lets the user upload and share different content types such as HTML pages, documents, images, videos, etc. Although many of the fields are common across all of the above-mentioned content types (such as Name, ID, Author, Upload Date, and Time), not all fields are identical. For example, in the case of images, you have a binary field that holds the image content, whereas an HTML page has a large text field to hold the HTML content. In this scenario, the MongoDB polymorphic schema can be used wherein all of the content node types are stored in the same collection, such as LoadContent, and each document has relevant fields only.

```
// "Document collections" - "HTMLPage" document
{
_id: 1,
title: "Hello",
type: "HTMLpage",
text: "<html>Hi..Welcome to my world</html>"
}
...
// Document collection also has a "Picture" document
{
_id: 3,
title: "Family Photo",
type: "JPEG",
sizeInMB: 10,........
}
```

This schema not only enables you to store related data with different structures together in a same collection, it also simplifies the querying. The same collection can be used to perform queries on common fields such as fetching all content uploaded on a particular date and time as well as queries on specific fields such as finding images with a size greater than X MB. Thus object-oriented programming is one of the use cases where having a polymorphic schema makes sense.

Schema Evolution :

When you are working with databases, one of the most important considerations that you need to account for is the schema evolution (i.e. the change in the schema's impact on the running application). The design should be done in a way as to have minimal or no impact on the application, meaning no or minimal downtime, no or very minimal code changes, etc. Typically, schema evolution happens by executing a migration script that upgrades the database schema from the old version to the new one. If the database is not in production, the script can be simple drop and recreation of the database. However, if the database is in a production environment and contains live data, the migration script will be complex because the data will need to be preserved. The script should take this into consideration. Although MongoDB offers an Update option that can be used to update all the documents' structure within a collection if there's a new addition of a field, imagine the impact of doing this if you have thousands of documents in the collection. It would be very slow and would have a negative impact on the underlying application's performance. One of the ways of doing this is to include the new structure to the new documents being added to the collection and then gradually migrating the collection in the background while the application is still running. This is one of the many use cases where having a polymorphic schema will be advantageous.

Question 1. What is Polymorphic Schema ? Explain various reason for using a polymorphic schema

Question 1. What is a Polymorphic Schema?**

A polymorphic schema in MongoDB is a flexible data model where a single collection can store documents of different types and structures. This concept is similar to polymorphism in Object-Oriented Programming (OOP), where objects of different classes can be handled through a common interface or superclass.

Question 2. Reasons for Using a Polymorphic Schema**

1. **Flexibility:**

- A polymorphic schema allows storing different types of data in a single collection, providing flexibility in handling varying data structures.

- For instance, in a media library application, you can store books, movies, and music albums in the same collection, even though they have different attributes.

2. **Simplified Data Management:**

- It reduces the complexity of managing multiple collections for different data types.

- With a polymorphic schema, you can perform operations (e.g., queries, updates) on a single collection, simplifying data management.

3. **Scalability:**

- Adding new types of documents is easier and doesn't require altering the existing schema or creating new collections.

- This scalability is beneficial in rapidly evolving applications where new data types may be introduced frequently.

4. **Efficient Queries:**

- Queries can be more efficient as you can retrieve related documents from a single collection without the need for joins or multiple queries across collections.

- For example, finding all media items with a certain title can be done in one query.

5. **Consistent Data Access:**

- Access patterns remain consistent as you interact with a single collection, regardless of the document type.

- This consistency simplifies application logic and code maintenance.

Example of Polymorphic Schema in MongoDB:

Consider a media library storing books, movies, and music albums. Each type of media has some common attributes (e.g., title, release date) and some unique attributes.

1. **Storing Different Types of Media:**

- All media items are stored in a single collection called `media`.

json

```
// Example documents in the media collection
{
  "_id": ObjectId("60c72b2f9b1e8e1a3d8b4567"),
  "type": "book",
  "title": "The Great Gatsby",
  "release_date": "1925-04-10",
```

```
  "author": "F. Scott Fitzgerald",
  "pages": 218
}
{
  "_id": ObjectId("60c72b2f9b1e8e1a3d8b4568"),
  "type": "movie",
  "title": "Inception",
  "release_date": "2010-07-16",
  "director": "Christopher Nolan",
  "duration": 148
}
{
  "_id": ObjectId("60c72b2f9b1e8e1a3d8b4569"),
  "type": "music",
  "title": "Abbey Road",
  "release_date": "1969-09-26",
  "artist": "The Beatles",
  "tracks": 17
}
```

2. **Querying Polymorphic Data:**

- To find all books:

```
json
db.media.find({ "type": "book" })
```

- To find all media items with the title "Inception":

```
json
```

```
db.media.find({ "title": "Inception" })
```

Polymorphism in Object-Oriented Programming:

In OOP, polymorphism allows methods to process objects differently based on their class. Here's how it relates to our example:

1. **Base Class and Derived Classes:**

```java
// Base class
public class Media {
  private String title;
  private String releaseDate;
  // Getters and setters
}

// Derived class for books
public class Book extends Media {
  private String author;
  private int pages;
  // Getters and setters
}

// Derived class for movies
public class Movie extends Media {
  private String director;
  private int duration;
  // Getters and setters
```

```
}

// Derived class for music
public class Music extends Media {
  private String artist;
  private int tracks;
  // Getters and setters
}
```

2. **Handling Different Media Types:**

```
java
public void printMediaDetails(Media media) {
  System.out.println("Title: " + media.getTitle());
  if (media instanceof Book) {
    Book book = (Book) media;
    System.out.println("Author: " + book.getAuthor());
  } else if (media instanceof Movie) {
    Movie movie = (Movie) media;
    System.out.println("Director: " + movie.getDirector());
  } else if (media instanceof Music) {
    Music music = (Music) media;
    System.out.println("Artist: " + music.getArtist());
  }
}
```

A polymorphic schema in MongoDB, similar to polymorphism in OOP, allows for flexible and scalable data modeling. It enables storing

different types of related data in a single collection, simplifying data management and enhancing efficiency. This approach is particularly useful in applications where data types can vary but share a common context.