23. MongoDb

Step 1: Outline Use Cases and Constraints

Before designing a MongoDB schema, it is crucial to understand the use cases and define constraints to ensure the design meets performance, scalability, and reliability requirements.

1.1 Use Cases

Clearly define the operations your system will support. We'll scope the problem to handle only the following use cases:

- Read-heavy system: The application retrieves a high volume of data frequently.
- Write-heavy system: Frequent inserts, updates, and deletions are performed.
- Real-time analytics: Queries should be optimized for quick aggregations.
- Complex queries: The system should support filtering, sorting, and searching efficiently.
- Relationships between data: Determine whether embedded documents or references should be used.

1.2 Out of Scope

- Transaction-heavy operations: If strict ACID transactions are needed, a relational database might be more suitable.
- Joins across multiple collections: MongoDB does not support traditional SQL joins efficiently.
- Highly structured and normalized data: MongoDB is not ideal for highly relational data models.

1.3 Constraints and Assumptions

To properly design MongoDB, we must state our assumptions and calculate expected system usage.

Assumptions

- The dataset will grow at a predictable rate over time.
- The system will experience X reads and Y writes per second.
- Indexing will be used to optimize read performance.
- Data sharding might be required to handle large-scale datasets.
- Some level of data denormalization will be used to optimize performance.

Calculate Usage

- Total dataset size: If we store X million documents, estimate the total storage requirement.
- Read vs. Write ratio: Determine how frequently data is read vs. written.
- Indexing overhead: Consider the storage cost of indexes.
- Replication factor: Decide the number of replicas needed for high availability.

Step 2: Create a High-Level Design

Now that we understand the use cases and constraints, we need to architect the database at a high level. This step involves designing:

- Data models (Collections, Documents, Relationships)
- Indexes
- Sharding strategy

- Replication strategy

2.1 Data Modeling Strategy

MongoDB offers two primary approaches to storing data:

1. Embedding (Denormalization)

- Store related data together in a single document.

- Useful when data is frequently queried together.

- Example:

```
{
  "_id": "12345",
  "name": "John Doe",
  "email": "john@example.com",
  "orders": [
    { "orderId": "001", "amount": 100, "status": "delivered" },
    { "orderId": "002", "amount": 200, "status": "pending" }
  ]
}
```

- Pros: Fast reads, reduced need for joins.

- Cons: Can lead to large documents and duplication.

2. Referencing (Normalization)

- Store related data in separate collections and link via foreign keys.

- Useful when related data changes frequently.

- Example:

```
// User collection
{
  "_id": "12345",
  "name": "John Doe",
  "email": "john@example.com",
  "orders": ["001", "002"]
}

// Orders collection
{
  "_id": "001",
  "userId": "12345",
  "amount": 100,
  "status": "delivered"
}
```

- Pros: Avoids document bloat, better for large datasets.

- Cons: Requires additional queries (lookup).

2.2 Indexing Strategy

- Primary Index: `_id` field (default).
- Secondary Indexes: Indexes on frequently queried fields, such as `email` or `status`.
- Compound Indexes: Multi-field indexes for optimized searches.
- TTL Indexes: To automatically expire data (e.g., session logs).

2.3 Sharding Strategy

- Range-based Sharding: Divides data based on a field's value range (e.g., user ID ranges).
- Hash-based Sharding: Uses a hash function to distribute data randomly.
- Zone-based Sharding: Distributes data based on geographical location.

2.4 Replication Strategy

- Primary-Secondary Replication: One primary, multiple secondary nodes.
- Read Preference: Configure some replicas to serve read traffic.
- Write Concern: Define durability requirements (e.g., `majority` for stronger consistency).

Step 3: Design Core Components (According to the Use Cases)

Now, we implement the core database features based on the use cases identified in Step 1.

3.1 Optimizing for Reads

- Use denormalization to avoid joins.

- Create proper indexes on frequently queried fields.

- Enable caching to reduce direct database calls.

3.2 Optimizing for Writes

- Use bulk inserts instead of multiple single inserts.

- Partition write-heavy collections across multiple shards.

- Use write concerns to balance speed vs. consistency.

3.3 Handling Relationships

- One-to-One: Embed or reference based on retrieval needs.

- One-to-Many: Use arrays for small lists, references for large lists.

- Many-to-Many: Use separate collections for lookup efficiency.

3.4 Aggregations

- Use the aggregation pipeline for analytics-heavy queries.

- Optimize pipeline stages (match → project → group → sort).

- Consider materialized views for complex queries.

You're right to question whether the Java code fully aligns with the MongoDB design principles I outlined earlier. Let's break it down properly and design MongoDB according to best practices while keeping the code structured accordingly.

MongoDB Design Approach Based on the Explanation

3.1 Optimizing for Reads

- Avoid Joins: Use denormalization where necessary.

- Indexing: Create indexes on frequently queried fields.

- Caching: Use Redis or application-level caching.

3.2 Optimizing for Writes

- Bulk Inserts: Instead of multiple single inserts, use `insertMany()`.

- Sharding: Partition collections for high-write workloads.

- Write Concerns: Adjust based on consistency vs. performance needs.

3.3 Handling Relationships

- One-to-One: Embed if frequently retrieved together, otherwise reference.

- One-to-Many: Use an array for small lists, references for large lists.

- Many-to-Many: Use a separate collection to optimize lookups.

3.4 Aggregations

- Use Aggregation Pipelines: Optimize queries for analytics.

- Pipeline Order: `$match → $project → $group → $sort` for efficiency.

- Materialized Views: Precompute results for complex queries.

MongoDB Schema Design Example

Let's assume we're building a User Management System with Users, Orders, and Products.

User Collection (Embedded for small relationships)

```
{
 "_id": "user123",
 "name": "John Doe",
 "email": "john@example.com",
```

```
  "age": 28,
  "address": {
    "street": "123 Main St",
    "city": "New York"
  },
  "orders": [
    {
      "orderId": "order001",
      "totalAmount": 500,
      "status": "Completed"
    }
  ]
}
```

> Why embed orders?

> If a user has only a few orders, we can embed them inside the user document. But for frequent order history retrieval, we should reference them.

Orders Collection (Reference for large relationships)

```
{
  "_id": "order001",
  "userId": "user123",
  "products": [
```

```
   {"productId": "prod101", "quantity": 2},
   {"productId": "prod102", "quantity": 1}
 ],
 "totalAmount": 500,
 "status": "Completed",
 "orderDate": "2024-02-05"
}
```

> Why use references?

> If users have a large number of orders, embedding them would increase document size and retrieval complexity.

Products Collection

```
{
 "_id": "prod101",
 "name": "Laptop",
 "price": 1000,
 "stock": 50
}
```

> Why separate?

> Products are independent and should not be embedded inside orders or users.

Java Implementation Based on This Design

Now, let's implement this properly in Java.

Java Code for MongoDB with Optimized Design

```
import com.mongodb.client.*;
import com.mongodb.client.model.*;
import org.bson.Document;
import java.util.Arrays;


public class MongoDBDesignExample {
    public static void main(String[] args) {
        String uri = "mongodb://localhost:27017";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
            MongoDatabase database = mongoClient.getDatabase("shopDB");


            // Collections
            MongoCollection<Document> users = database.getCollection("users");
            MongoCollection<Document> orders = database.getCollection("orders");
            MongoCollection<Document> products = database.getCollection("products");


            // Insert Product
            Document product1 = new Document("_id", "prod101")
                    .append("name", "Laptop")
```

```
                .append("price", 1000)
                .append("stock", 50);
        products.insertOne(product1);


        // Insert User
        Document user1 = new Document("_id", "user123")
                .append("name", "John Doe")
                .append("email", "john@example.com")
                .append("age", 28)
                .append("address", new Document("street", "123 Main St").append("city", "New York"));
        users.insertOne(user1);


        // Insert Order (Referencing user & products)
        Document order1 = new Document("_id", "order001")
                .append("userId", "user123")
                .append("products", Arrays.asList(
                        new Document("productId", "prod101").append("quantity", 2)
                ))
                .append("totalAmount", 2000)
                .append("status", "Completed");
        orders.insertOne(order1);
```

```
        // Indexing for Optimization
        users.createIndex(Indexes.ascending("email")); // Email indexing for fast lookup
        orders.createIndex(Indexes.ascending("userId")); // Fast order retrieval for a user


        // Query Optimization - Fetch user with their orders (Denormalized Approach)
        FindIterable<Document> userWithOrders = users.find(new Document("_id", "user123"));
        for (Document doc : userWithOrders) {
            System.out.println(doc.toJson());
        }


        // Aggregation Example: Get total sales per product
        AggregateIterable<Document> totalSales = orders.aggregate(Arrays.asList(
                Aggregates.unwind("$products"),
                Aggregates.group("$products.productId", Accumulators.sum("totalSold", "$products.quantity"))
        ));
        for (Document doc : totalSales) {
            System.out.println(doc.toJson());
        }
    }
}
```

```
}
```

How This Matches the Optimization Principles

Optimizing Reads

✅ Indexes: Created indexes on `email` and `userId`.

✅ Denormalization for Small Data: Embedded address in the user document.

✅ Referencing for Large Data: Orders reference user and product documents.

Optimizing Writes

✅ Bulk Insert: Used `insertOne()` but could use `insertMany()` for bulk writes.

✅ Sharding Readiness: Partitioning orders per user is easy.

✅ Write Concerns: The application can configure write concerns when needed.

Handling Relationships

✅ One-to-One: Address is embedded in the user document.

✅ One-to-Many: Orders reference the user instead of embedding large orders.

✅ Many-to-Many: Orders reference multiple products for scalability.

Aggregation Optimization

✅ Match → Group → Unwind: Used `$unwind` and `$group` to calculate total sales per product efficiently.

✅ Index-Based Queries: Indexed `userId` for faster lookup in orders.

Conclusion

This MongoDB schema design follows best practices:

✔ Denormalization & Referencing Based on Needs

✔ Indexes for Faster Queries

✔ Bulk Inserts & Write Concerns for Performance

✔ Aggregation Pipelines for Efficient Analytics

Step 4: Scale the Design

As the database grows, we need to scale MongoDB efficiently to handle increased traffic and data.

4.1 Horizontal Scaling (Sharding)

- Distribute data across multiple nodes to avoid overloading a single machine.

- Choose a shard key that prevents hotspots (e.g., userId instead of timestamps).

4.2 Vertical Scaling

- Increase CPU, RAM, and disk space if necessary.

- Optimize queries to reduce resource usage.

4.3 Load Balancing

- Use read replicas to distribute read traffic.

- Route queries efficiently to minimize latency.

4.4 Monitoring and Maintenance

- Use MongoDB Atlas or Prometheus for performance monitoring.

- Set up alerts for slow queries and replication lag.

- Perform regular indexing and data compaction.

Final Thoughts

By following this step-by-step MongoDB design approach, you can build a scalable, high-performance database system tailored to your specific use cases.