Hue Learn

Wise Work

DNAi World

Mongo DB Guidelines

Best Practices to follow

by Hue Learn, Wise Work, DNAi World

Contents

1. Introduction

MongoDB has gained widespread popularity as a powerful NoSQL database solution, offering flexibility, scalability, and ease of use. However, to harness its full potential, developers must adhere to best practices that ensure optimal performance and reliability.

2. Code Practices

1. **Connection Management**: Reuse database connections rather than creating a new connection for each operation. Use connection pooling to manage connections efficiently.

Right Coding Style.

```
const { MongoClient } = require('mongodb');
const client = new MongoClient(uri, { poolSize: 10 });
```

2. **Error Handling**: Implement comprehensive error handling for database operations, **including retry logic for transient errors.**

Right Coding Style.

```
async function runQuery() {
    try {
        await client.connect();
        const db = client.db('test');
        const result = await db.collection('users').findOne({ email:
"john.doe@example.com" });
        console.log(result);
    }
    catch (error) {
        console.error("Query failed", error);

        // Retry logic here

    }
    finally {
        await client.close();
    }
}
```

3. **Validation and Sanitization**: Validate and sanitize all input data to prevent **injection attacks** and ensure data integrity.

Right Coding Style.

```
const Joi = require('joi');

const schema = Joi.object({
    name: Joi.string().min(3).required(),
    email: Joi.string().email().required()
});

const { error, value } = schema.validate({ name: "John", email:
"john.doe@example.com" });

if (error) {
    console.error("Validation failed", error.details);
}
else {
    // Proceed with value
}
```

4. **Use Mongoose (for Node.js)**: If you're using Node.js, consider using **Mongoose for schema validation**, middleware, and easier interaction with MongoDB.

Right Coding Style.

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: String,
  email: String
});

const User = mongoose.model('User', userSchema);

const user = new User({ name: "John Doe", email: "john.doe@example.com" });
user.save();
```

3. Data Management

1. **Data Archival**: Implement a strategy for archiving old or less frequently accessed data to keep your working set small.

Right Coding Style.

- Archive old data to another collection.

```
const oldOrders = db.orders.find({ createdAt: { $lt: new
Date("2022-01-01") } }).toArray();
db.archived_orders.insertMany(oldOrders);
db.orders.deleteMany({ createdAt: { $lt: new Date("2022-01-01") } });
```

2. **Backup and Restore**: Regularly back up your data and test your restore procedures to ensure data availability and integrity.

Right Coding Style.

- Use **mongodump** and **mongorestore** for backups.

```
mongodump --db mydb --out /backup/mydb
mongorestore --db mydb /backup/mydb
```

3. **Sharding**: For very large datasets, consider sharding your database to distribute data across multiple servers and improve performance.

Right Coding Style.

- Distribute data across multiple servers.

```
sh.enableSharding("mydb");
sh.shardCollection("mydb.orders", { orderId: 1 });
```

4. Indexes in MongoDB

In any database, **indexes support the efficient execution of queries**. Without them, the database must scan every document in a collection or table to select those that match the query statement. If an appropriate index exists for a query, the database can use the index to limit the number of documents it must inspect.

1. **Compound Indexes**: Create compound indexes for queries that filter on multiple fields. Ensure the index fields are in the correct order based on the query.

Right Coding Style.

```
db.products.createIndex({ category: 1, price: 1 });

With this compound index in place, queries like the following will benefit from
it:

// Querying by category and price range
db.products.find({ category: "Electronics", price: { $gte: 100, $lte: 500 } });
```

Follow the ESR(Equality, Sort, Range) rule

For compound indexes, this rule of thumb is helpful in deciding the order of fields in the index:

- First, add those fields against which Equality queries are run.
- The next fields to be indexed should reflect the Sort order of the query.
- The last fields represent the Range of data to be accessed.

2. **Unique Indexes**: Use unique indexes to enforce uniqueness constraints on fields, such as email addresses or usernames.

Right Coding Style.

```
db.users.createIndex({ email: 1 }, { unique: true });
```

3. **TTL Indexes**: Use Time-To-Live (TTL) indexes to automatically delete documents after a certain period, useful for session data or logs.

Right Coding Style.

```
db.users.createIndex({ email: 1 }, { unique: true });
```

4. **Sparse and Partial Indexes:** Use sparse indexes for fields that do not exist in every document, and partial indexes to index a subset of documents based on a filter.

Right Coding Style.

```
// Sparse index:
db.users.createIndex({ nickname: 1 }, { sparse: true });

// Partial index:
db.users.createIndex({ status: 1 }, { partialFilterExpression: { status:
{ $exists: true } } });
```

4.1. Covering Indexes:

Include all necessary fields in an index to prevent the need for additional data fetching.

Consider a collection of Orders where you often need to retrieve order numbers and their total amounts. Instead of fetching additional data from the document, you can create a covering index.

4.2. Avoid Over Indexing

Unnecessary indexes consume storage and slow down write operations. Avoid creating indexes for every single field.

Wrong Coding Style.

```
db.users.createIndex({ username: 1 });
db.users.createIndex({ email: 1 });
```

5. Query Optimization

1. Analyze Query Performance

Right Coding Style.

Use `.explain()` to understand query execution.

```
db.users.find({ email: "john.doe@example.com" }).explain("executionStats");
```

2. Avoid $ne and $nin Operators, use other operators for efficiency.

Wrong Coding Style.

```
db.users.find({ status: { $ne: "inactive" } });
```

Right Coding Style.

```
db.users.find({ status: "active" });
```

3. Limit Projection Fields, only return necessary fields.

Right Coding Style.

```
db.users.find({ email: "john.doe@example.com" }, { name: 1, email: 1 });
```

4. Use Aggregation Pipeline for complex queries.

Right Coding Style.

```
db.orders.aggregate([
   { $match: { status: "shipped" } },
   { $group: { _id: "$customer_id", total: { $sum: "$amount" } } },
   { $sort: { total: -1 } }
]);
```

6. Schema Design

1. **Understand Your Queries**: Design your schema according to the queries you'll run most often. Optimize for read-heavy or write-heavy operations based on your use case.

Right Coding Style.

- If you frequently query by `user_id`, design your schema accordingly.

```
{
  user_id: ObjectId("someId"),
  name: "John Doe",
  email: "john.doe@example.com"
}
```

2. **Avoid Embedding Too Deeply**: While MongoDB supports nested documents, embedding too deeply can lead to inefficiencies. Aim for a balance between embedding and referencing.

Wrong Coding Style.

- Instead of deeply nested documents like below,

```
{
  user: {
    profile: {
      settings: {
        notifications: true
      }
    }
  }
}
```

Right Coding Style.

- Use this balanced approach

```
{
  user_id: ObjectId("someId"),
  profile: {
    settings: {
      notifications: true
    }
  }
}+
```

3. **Use Proper Indexing**: Create indexes on fields that are frequently used in queries, but avoid over-indexing as it can slow down write operations.

Right Coding Style.

- Create indexes on fields used in queries.

```
db.users.createIndex({ email: 1 });
```

4. **Consider Data Growth**: Design your schema to accommodate future data growth and scalability requirements.
   - Plan your schema to accommodate future data.

Right Coding Style.

```
// Initial schema:
{
   name: "John Doe",
   email: "john.doe@example.com"
}
// Future-proof schema:
{
  name: "John Doe",
  email: "john.doe@example.com",
  created_at: new Date(),
  updated_at: new Date()
}
```

5. **Schema Validation**: Use MongoDB's schema validation to enforce data integrity and ensure documents conform to the expected structure.

Right Coding Style.

- Use schema validation to enforce data integrity.

```
db.createCollection("users", {
     validator: {
       $jsonSchema: {
         bsonType: "object",
         required: ["name", "email"],
         properties: {
           name: {
             bsonType: "string",
             description: "must be a string and is required"
           },
           email: {
             bsonType: "string",
             pattern: "@mongodb.com$",
             description: "must be a string and match the regular expression"
          }
        }
      }
    }
});
```

7. Scaling and Sharding:

MongoDB's horizontal scalability is one of its strengths. Plan for future growth with sharding. Choose a Sharding Key Carefully. The sharding key affects how data is distributed across shards. Select a key that evenly distributes data and avoids hotspots.

Right Coding Style.

- Enable sharding on the database.

```
sh.enableSharding("social_media_db");
```

- Shard the Posts collection using the userId field as the sharding key.

```
sh.shardCollection("social_media_db.Posts", { "userId": 1 });
```

- You can check the status of shard balancing using the following commands.

```
sh.getBalancerState();
```

- View information about chunks and their distribution.

```
sh.status();
```

By carefully choosing a sharding key and actively monitoring shard distribution, you can effectively harness MongoDB's horizontal scalability to accommodate growing data while maintaining optimal performance.

8. Monitoring and Maintenance

Monitoring and profiling your MongoDB database helps you identify performance issues and bottlenecks.

1. **Monitor Performance**: Use monitoring tools like **MongoDB Atlas** or third-party solutions to keep an eye on database performance and identify potential issues.
2. **Regular Maintenance**: Regularly update your MongoDB version and apply patches to ensure you have the latest features and security updates.
3. **Optimize Storage**: Periodically run the `compact` command or use the **WiredTiger** storage engine's built-in compression to optimize disk space usage.
4. **Query profiler**: MongoDB's query profiler is a powerful tool for identifying slow queries and understanding the performance characteristics of your operations. By **enabling profiling**, examining the **system.profile** collection, and using tools like **MongoDB Compass,** you can analyze query performance and optimize your database operations accordingly.

8.1. Enabling Profiling:

MongoDB provides a profiling feature that records performance statistics for database operations.

Right Coding Style.

- Enable profiling to capture slow operations

```
db.setProfilingLevel(1, { slowms: 100 });
```

By setting the profiling level to 1 and specifying a threshold of 100 milliseconds, MongoDB will capture operations taking longer than 100ms.

8.2. Viewing Profiling Data

8.2.1. system.profile collection

Profiling data is stored in the **system.profile** collection, which is capped and stores detailed information about database operations. You can query this collection to view profiling data.

Right Coding Style.

View all profiling data

```
db.system.profile.find().pretty()
```

View slow queries only

```
db.system.profile.find({ millis: { $gt: 100 } }).pretty()
```

This query shows operations that took more than 100 milliseconds to complete.

8.2.2. MongoDB Compass

- If you prefer a GUI tool, MongoDB Compass provides an easy way to view profiling data visually.
- In Compass, navigate to the Performance tab. Compass will show you slow queries and their execution statistics, helping you identify performance bottlenecks.

9. Linting

MongoDB doesn't have direct linter tools like you would find in programming languages, but there are ways to lint or validate MongoDB schemas and queries.

9.1. ESLint with MongoDB Plugins

If you're working with MongoDB in a **Node.js environment** (using Mongoose or native MongoDB driver), you can use ESLint plugins to lint your MongoDB code.

9.2. ESlint-plugin-mongoose

This helps enforce best practices when using Mongoose, a popular ODM (Object Data Modeling) library for MongoDB.

9.3. MongoDB Query Linter

Tools like Compass (MongoDB's GUI) allow you to validate and test queries, though they do not provide automatic linting for query structure or optimization. You can manually optimize queries by reviewing query plans.

10. Formatters

While there isn't a MongoDB-specific formatter, you can format your queries and documents using general JSON formatters because MongoDB uses BSON (a binary JSON format).

10.1. MongoDB Compass

MongoDB Compass has built-in formatting tools for visualizing documents in a structured and readable manner. It also allows you to format queries for better readability.

10.2. JSON Formatter Plugins

If you're working in a text editor like **VS Code or Sublime Text**, you can use JSON formatter plugins to format MongoDB queries and documents. **Examples**: Prettier, JSON Viewer

10.2.1. Prettier

A popular formatter that can format JSON, including MongoDB documents and queries in JavaScript co, de.

10.2.2. JSON Viewer

Browser plugins or text editor extensions that allow you to view JSON in a more readable and formatted manner.

11. Analyzers:

MongoDB offers tools to analyze query performance, schema structure, and other optimizations.

11.1. MongoDB Compass

Provides schema analysis, query analysis, and performance insights. You can visually explore and analyze the structure of your MongoDB collections and check the efficiency of your queries using the query planner.

11.2. Atlas Performance Advisor

In MongoDB Atlas, the Performance Advisor suggests indexes and other optimizations for your MongoDB cluster based on query patterns. It analyzes slow queries and proposes improvements.

11.3. Query Profiler

The MongoDB profiler allows you to log and analyze query performance. It gives you insights into slow-running queries so that you can improve their efficiency.

11.4. mtools

A collection of helper scripts to analyze and parse MongoDB log files and perform query analysis. It can help find slow queries and inefficient query patterns.

11.5. Studio 3T

A commercial tool for MongoDB, it provides advanced query building, schema analysis, and data migration features. It includes a Query Profiler and Schema Explorer for analyzing query performance and structure.

12. Security

12.1. Authentication and Authorization:

- **Enable authentication** and define roles and permissions to control access to your database. Require authentication for all connections and implement **role-based access control (RBAC).**
- MongoDB supports authentication and role-based access control (RBAC) to ensure that only authorized users can access and perform actions on the database.

12.1.1. Enabling Authentication:

Right Coding Style.

- To enable authentication, you need to start your MongoDB instance with the **–auth flag.**

```
mongod --auth --dbpath /path/to/data
```

- Creating Users and Assigning Roles, after enabling authentication, you can create users and assign specific roles to control their access.
- Connect to the admin database as a user with root privileges.

```
mongo admin -u admin -p
```

- Create a user with read and write access to a specific database.

```
db.createUser({
  user: "myappuser",
  pwd: "mypassword",
  roles: [{ role: "readWrite", db: "myappdb" }]
});
```

2. **Encryption**: Use **TLS/SSL** to encrypt data in transit and enable encryption at rest to protect sensitive data.
3. **Network Security**: Use **firewalls and VPNs** to secure access to your MongoDB instances and avoid exposing them directly to the internet. Restricting incoming connections to your MongoDB instance using firewalls adds an extra layer of security.
   - **Firewall Configuration**: Configure your firewall to allow connections **only from trusted IP addresses.**

13. Conclusion

By following these best practices, you can ensure your MongoDB application is efficient, secure, and scalable. Linters, Formatters, Analyzers help ensure that your MongoDB queries are efficient, well-formatted, and optimized for performance.