### MongoDB – Session 2 and 3

The CAP theorem and BASE principles are fundamental concepts in distributed systems and database design, often referenced when discussing MongoDB and other NoSQL databases.

**CAP Theorem**

The CAP theorem, also known as Brewer's theorem, states that it's impossible for a distributed computer system to simultaneously achieve all three of the following guarantees:

1. **Consistency (C)**: Every read receives the most recent write or an error. In other words, all nodes in the system have the same data at any given time.
2. **Availability (A)**: Every request receives a response, without guarantee that it contains the most recent write. In other words, the system remains operational despite node failures.
3. **Partition tolerance (P)**: The system continues to operate despite arbitrary message loss or failure of part of the system.

**Examples in MongoDB context:**

* **Consistency (C)**: MongoDB can provide strong consistency within a single replica set where all reads and writes go to the primary node. However, in a sharded cluster, eventual consistency is more common due to the distributed nature of data across shards.
* **Availability (A)**: MongoDB ensures high availability through replica sets. If a primary node fails, a secondary node can be elected as the new primary, ensuring continued availability of data.
* **Partition tolerance (P)**: MongoDB is designed to handle network partitions gracefully. Replica sets and sharded clusters are inherently partition-tolerant, allowing them to continue functioning even if some nodes are unreachable.

**BASE Principles**

BASE, standing for **Basically Available, Soft state, eventually consistent**, provides a set of principles that are often applied to NoSQL databases like MongoDB:

1. **Basically Available**: The system guarantees availability, even in the presence of network partitions or node failures. This principle prioritizes availability over strong consistency.
2. **Soft state**: The state of the system may change over time, even without input. This implies that data may not be consistent immediately but will converge over time.
3. **Eventually consistent**: The system will become consistent over a period, given that the system doesn't receive input during that period.

**Examples in MongoDB context:**

* **Basically Available**: MongoDB's architecture allows read and write operations to continue even during network partitions or node failures, ensuring basic availability.
* **Soft state**: MongoDB employs replication and sharding to distribute data across multiple nodes. As updates are propagated, different nodes may temporarily have different views of the data (soft state).
* **Eventually consistent**: MongoDB aims for eventual consistency, meaning that after a period of time where no updates are made, all replicas will converge to a consistent state.

**MongoDB Examples**

* **CAP**: In a MongoDB replica set, if the primary node fails, one of the secondary nodes will be elected as the new primary, ensuring availability (A) while maintaining partition tolerance (P). Consistency (C) can vary depending on the read concern level set by the client application.
* **BASE**: MongoDB's default behaviour, especially in sharded clusters, adheres to BASE principles. As data is distributed across shards, immediate consistency might not be guaranteed across the entire system, but it will converge over time (eventual consistency).

1. **Basically Available**: MongoDB ensures high availability through features like replica sets and sharded clusters:
   * **Replica Sets**: In a MongoDB replica set, there are multiple copies of data (nodes). If one node fails, others can continue to serve read and write requests, ensuring basic availability.
   * **Sharded Clusters**: MongoDB uses sharding to horizontally partition data across multiple shards. If one shard becomes unavailable, the other shards can continue to handle requests, maintaining availability.
2. **Soft state**: MongoDB allows for the concept of eventual consistency:
   * **Replication**: MongoDB's replication model involves primary and secondary nodes. When data is written to the primary node, it asynchronously replicates updates to secondary nodes. During this replication process, there may be a temporary period where different nodes have slightly different views of the data (soft state).
   * **Sharding**: In a sharded cluster, data is distributed across shards. As data is inserted or updated, it may take some time for these changes to propagate to all shards, leading to soft state characteristics where different shards may temporarily have different views of data.
3. **Eventually consistent**: MongoDB aims for eventual consistency across distributed systems:
   * **Read Concerns**: MongoDB allows configuring read concerns to specify the level of consistency required for read operations. For example, using "majority" as the read concern ensures that reads return data that has been acknowledged by a majority of nodes, achieving stronger consistency over time.
   * **Write Concerns**: Write operations in MongoDB can specify the level of acknowledgment required from nodes (like acknowledgement from a majority of nodes), ensuring that data consistency is eventually achieved across the system.

**Examples:**

* **Sharded Cluster Example**: Consider a MongoDB sharded cluster where data is distributed across multiple shards. When a write operation occurs, data is written to the primary shard and then propagated to secondary shards asynchronously. During this propagation period, different shards may temporarily have different versions of data. However, over time, all shards will converge to a consistent state (eventual consistency).
* **Replica Set Example**: In a MongoDB replica set, if the primary node fails, one of the secondary nodes is elected as the new primary. During this failover process, reads and writes can still be serviced by other nodes in the replica set, ensuring basic availability. As data is replicated to the new primary and other secondaries, eventual consistency is maintained across the replica set.

### Sample JSON Data of Student Records

[

{

"student\_id": 1,

"name": "Alice",

"age": 21,

"major": "Computer Science",

"courses": ["Databases", "Algorithms"],

"address": {

"street": "123 Main St",

"city": "Springfield",

"zip": "62701"

},

"enrollment\_date": "2021-09-01"

},

{

"student\_id": 2,

"name": "Bob",

"age": 22,

"major": "Mathematics",

"courses": ["Calculus", "Statistics"],

"address": {

"street": "456 Elm St",

"city": "Springfield",

"zip": "62702"

},

"enrollment\_date": "2020-09-01"

},

{

"student\_id": 3,

"name": "Charlie",

"age": 20,

"major": "Physics",

"courses": ["Quantum Mechanics", "Electromagnetism"],

"address": {

"street": "789 Oak St",

"city": "Springfield",

"zip": "62703"

},

"enrollment\_date": "2022-01-15"

}

]

### MongoDB Operations

#### **1. Import Data**

To import the JSON data into a MongoDB collection, you can use the mongoimport command:

mongoimport --db school --collection students --file students.json –-jsonArray

#### **2. Create (Insert) Documents**

db.students.insertMany([

{

"student\_id": 1,

"name": "Alice",

"age": 21,

"major": "Computer Science",

"courses": ["Databases", "Algorithms"],

"address": {

"street": "123 Main St",

"city": "Springfield",

"zip": "62701"

},

"enrollment\_date": ISODate("2021-09-01")

},

{

"student\_id": 2,

"name": "Bob",

"age": 22,

"major": "Mathematics",

"courses": ["Calculus", "Statistics"],

"address": {

"street": "456 Elm St",

"city": "Springfield",

"zip": "62702"

},

"enrollment\_date": ISODate("2020-09-01")

},

{

"student\_id": 3,

"name": "Charlie",

"age": 20,

"major": "Physics",

"courses": ["Quantum Mechanics", "Electromagnetism"],

"address": {

"street": "789 Oak St",

"city": "Springfield",

"zip": "62703"

},

"enrollment\_date": ISODate("2022-01-15")

}

]);

#### **3. Read (Find) Documents**

**Find all students:**

db.students.find().pretty();

**Find a student by student\_id:**

db.students.find({ "student\_id": 1 }).pretty();

**Find students enrolled in "Computer Science":**

db.students.find({ "major": "Computer Science" }).pretty();

**Find students older than 21:**

db.students.find({ "age": { $gt: 21 } }).pretty();

**Find students taking "Databases" course:**

db.students.find({ "courses": "Databases" }).pretty();

#### **4. Update Documents**

**Update a student's age:**

db.students.updateOne(

{ "student\_id": 1 },

{ $set: { "age": 22 } }

);

**Add a new course to a student:**

db.students.updateOne(

{ "student\_id": 1 },

{ $push: { "courses": "Machine Learning" } }

);

#### **5. Delete Documents**

**Delete a student by student\_id:**

db.students.deleteOne({ "student\_id": 3 });

**Delete all students in "Mathematics":**

db.students.deleteMany({ "major": "Mathematics" });

#### **6. Export Data**

To export the data from the MongoDB collection to a JSON file, you can use the mongoexport command:

mongoexport --db school --collection students --out students\_export.json --jsonArray

### MongoDB Operations: Collection and Database Management

#### 1. Create a Collection

Creating a collection in MongoDB is typically done implicitly when you insert a document into a collection that does not exist yet. However, you can also explicitly create a collection using the createCollection method.

**Create a collection explicitly:**

use school; // Switch to the database 'school' or create it if it doesn't exist

db.createCollection("students"); // Create the 'students' collection explicitly

**Create a collection implicitly by inserting a document:**

use school;

db.students.insertOne({

"student\_id": 4,

"name": "David",

"age": 23,

"major": "Chemistry",

"courses": ["Organic Chemistry", "Inorganic Chemistry"],

"address": {

"street": "101 Maple St",

"city": "Springfield",

"zip": "62704"

},

"enrollment\_date": ISODate("2023-02-01")

});

#### 2. Drop a Collection

You can drop a collection using the drop method.

**Drop the 'students' collection:**

use school;

db.students.drop();

#### 3. Create a Database

Databases in MongoDB are created implicitly when you switch to a database and insert a document into a collection. There is no specific command to create a database explicitly. However, to demonstrate the creation:

**Switch to a new database and create a collection by inserting a document:**

use new\_school; // Switch to or create the 'new\_school' database

db.new\_students.insertOne({

"student\_id": 1,

"name": "Eve",

"age": 19,

"major": "Biology",

"courses": ["Genetics", "Ecology"],

"address": {

"street": "202 Pine St",

"city": "Springfield",

"zip": "62705"

},

"enrollment\_date": ISODate("2023-03-01")

});

#### 4. Drop a Database

You can drop a database using the dropDatabase method.

**Drop the 'school' database:**

use school;

db.dropDatabase();

**Drop the 'new\_school' database:**

use new\_school;

db.dropDatabase();

### Verification

You can verify that the database and collection were created by listing the databases and collections:

#### List Databases

show dbs;

This command will list all the databases. The school database should be in the list if it contains data.

#### List Collections in the school Database

use school;

show collections;

### Sample Aggregation Pipeline Examples

#### 1. Group Students by Major and Count the Number of Students in Each Major

db.students.aggregate([

{

$group: {

\_id: "$major",

count: { $sum: 1 }

}

}

]);

#### 2. Calculate the Average Age of Students

db.students.aggregate([

{

$group: {

\_id: null,

averageAge: { $avg: "$age" }

}

}

]);

#### 3. Find the Oldest and Youngest Students

db.students.aggregate([

{

$group: {

\_id: null,

oldest: { $max: "$age" },

youngest: { $min: "$age" }

}

}

]);

#### 4. List All Courses Taken by Students

db.students.aggregate([

{

$unwind: "$courses"

},

{

$group: {

\_id: null,

allCourses: { $addToSet: "$courses" }

}

}

]);

#### 5. Get the Number of Students Enrolled Per Year

db.students.aggregate([

{

$group: {

\_id: { $year: "$enrollment\_date" },

count: { $sum: 1 }

}

}

]);

#### 6. Find Students Who Are Enrolled in Both "Databases" and "Algorithms" Courses

db.students.aggregate([

{

$match: {

courses: { $all: ["Databases", "Algorithms"] }

}

}

]);

#### 7. Get Students Count by City

db.students.aggregate([

{

$group: {

\_id: "$address.city",

count: { $sum: 1 }

}

}

]);

#### 8. Sort Students by Enrollment Date

db.students.aggregate([

{

$sort: { enrollment\_date: 1 } // Sort by enrollment date in ascending order

}

]);

#### 9. Project Specific Fields

db.students.aggregate([

{

$project: {

\_id: 0,

name: 1,

major: 1,

age: 1

}

}

]);

#### 10. Count the Number of Students in Each Course

db.students.aggregate([

{

$unwind: "$courses"

},

{

$group: {

\_id: "$courses",

count: { $sum: 1 }

}

}

]);

### Update Operations

#### Update One Document

**Example: Update the age of the student with student\_id 1 to 22**

js

Copy code

db.students.updateOne(

{ student\_id: 1 }, // Filter criteria

{ $set: { age: 22 } } // Update operation

);

#### Update All Documents

**Example: Increment the age of all students by 1**

db.students.updateMany(

{}, // Empty filter criteria matches all documents

{ $inc: { age: 1 } } // Increment operation

);

### Replace Operations

#### Replace One Document

**Example: Replace the document of the student with student\_id 2 with a new document**

db.students.replaceOne(

{ student\_id: 2 }, // Filter criteria

{

student\_id: 2,

name: "Bob",

age: 23,

major: "Statistics",

courses: ["Probability", "Statistics"],

address: {

street: "456 Elm St",

city: "Springfield",

zip: "62702"

},

enrollment\_date: ISODate("2020-09-01")

} // New document

);

### Delete Operations

#### Delete One Document

**Example: Delete the student with student\_id 3**

db.students.deleteOne(

{ student\_id: 3 } // Filter criteria

);

#### Delete All Documents

**Example: Delete all documents in the students collection**

db.students.deleteMany({});

Managing indexes and performance tuning are essential for optimizing the performance of MongoDB queries, especially as the dataset grows. Here are some strategies and examples for managing indexes and performance tuning for the students collection.

### Index Management

#### 1. Creating Indexes

**Example: Create an index on the student\_id field**

db.students.createIndex({ student\_id: 1 });

**Example: Create a compound index on major and age fields**

db.students.createIndex({ major: 1, age: -1 });

**Example: Create a text index on the name field for text search**

db.students.createIndex({ name: "text" });

#### 2. Viewing Indexes

**Example: View all indexes on the students collection**

db.students.getIndexes();

#### 3. Dropping Indexes

**Example: Drop an index on the student\_id field**

db.students.dropIndex({ student\_id: 1 });

**Example: Drop all indexes on the students collection**

db.students.dropIndexes();

### Performance Tuning

#### 1. Query Optimization

**Example: Optimize a query using an index on the major field**

Without an index:

db.students.find({ major: "Computer Science" });

With an index on the major field:

db.students.createIndex({ major: 1 });

db.students.find({ major: "Computer Science" });

#### 2. Use of Projections

**Example: Use projections to return only specific fields**

db.students.find({ major: "Computer Science" }, { name: 1, major: 1, \_id: 0 });

#### 3. Aggregation Pipeline Optimization

**Example: Optimize an aggregation pipeline by placing $match and $sort stages early**

db.students.aggregate([

{ $match: { major: "Computer Science" } },

{ $sort: { age: -1 } },

{ $project: { name: 1, age: 1, major: 1 } }

]);

#### 4. Using Covered Queries

**Example: Create an index to support a covered query**

db.students.createIndex({ student\_id: 1, name: 1 });

db.students.find({ student\_id: 1 }, { student\_id: 1, name: 1, \_id: 0 });

This query can be satisfied entirely using the index without scanning the documents.

### Analysing Query Performance

#### 1. Explain Plan

**Example: Use the explain method to analyze query performance**

db.students.find({ major: "Computer Science" }).explain("executionStats");

#### 2. Profiling

**Example: Enable the database profiler to collect performance data**

db.setProfilingLevel(2); // Enable profiling at level 2 (all operations)

db.system.profile.find(); // View the profiling data

### Example Data and Index Management

Let's combine these practices with examples:

#### Inserting Sample Data

db.students.insertMany([

{

"student\_id": 1,

"name": "Alice",

"age": 21,

"major": "Computer Science",

"courses": ["Databases", "Algorithms"],

"address": {

"street": "123 Main St",

"city": "Springfield",

"zip": "62701"

},

"enrollment\_date": ISODate("2021-09-01")

},

{

"student\_id": 2,

"name": "Bob",

"age": 22,

"major": "Mathematics",

"courses": ["Calculus", "Statistics"],

"address": {

"street": "456 Elm St",

"city": "Springfield",

"zip": "62702"

},

"enrollment\_date": ISODate("2020-09-01")

},

{

"student\_id": 3,

"name": "Charlie",

"age": 20,

"major": "Physics",

"courses": ["Quantum Mechanics", "Electromagnetism"],

"address": {

"street": "789 Oak St",

"city": "Springfield",

"zip": "62703"

},

"enrollment\_date": ISODate("2022-01-15")

}

]);

#### Creating Indexes

db.students.createIndex({ student\_id: 1 });

db.students.createIndex({ major: 1, age: -1 });

db.students.createIndex({ name: "text" });

#### Query Optimization

db.students.find({ major: "Computer Science" }).explain("executionStats");

db.students.find({ major: "Computer Science" }, { name: 1, major: 1, \_id: 0 }).explain("executionStats");

### Conclusion

Managing indexes and optimizing performance in MongoDB involves creating, viewing, and dropping indexes, as well as using projections, optimizing aggregation pipelines, and analyzing query performance using explain plans and profiling. These practices help ensure that queries run efficiently and the database performs well, even as the size of the dataset grows.

### Advanced Performance Optimization Techniques

#### 1. Indexing for Read and Write Operations

**Balancing Read and Write Performance**

* **Compound Index for Read Optimization**

Create a compound index for commonly queried fields.

db.students.createIndex({ major: 1, age: -1 });

* **Index for Write Optimization**

Avoid creating too many indexes, as each index incurs a write penalty. Drop unnecessary indexes.

db.students.dropIndex({ name: "text" });

#### 2. Covered Queries

**Example: Use Covered Queries to Ensure Index-Only Operations**

Create an index to support a covered query.

db.students.createIndex({ student\_id: 1, name: 1 });

Run a query that is fully covered by the index.

db.students.find({ student\_id: 1 }, { student\_id: 1, name: 1, \_id: 0 });

#### 3. Projection Optimization

**Example: Return Only Required Fields**

Use projection to limit the fields returned by the query, reducing the amount of data transferred.

db.students.find({ major: "Computer Science" }, { name: 1, age: 1, \_id: 0 });

#### 4. Indexing Array Fields

**Example: Create Indexes on Array Fields**

If you query array fields frequently, create indexes on those fields.

db.students.createIndex({ courses: 1 });

Query that benefits from the index.

db.students.find({ courses: "Databases" });

#### 5. Sharding

**Example: Shard the Collection for Scalability**

Sharding can distribute the load across multiple servers.

Enable sharding on the school database.

sh.enableSharding("school");

Shard the students collection using student\_id as the shard key.

sh.shardCollection("school.students", { student\_id: 1 });

#### 6. Query Optimization with $hint

**Example: Use $hint to Force Index Usage**

Force the query to use a specific index.

db.students.find({ major: "Computer Science" }).hint({ major: 1, age: -1 });

#### 7. Caching Frequent Queries

**Example: Cache Results of Frequent Queries**

Use a caching layer like Redis to cache the results of frequent queries.

// Pseudocode: Check cache before querying MongoDB

cachedResult = redis.get("frequentQuery");

if (cachedResult) {

return cachedResult;

} else {

result = db.students.find({ major: "Computer Science" }).toArray();

redis.set("frequentQuery", result);

return result;

}

#### 8. Avoiding Large Data Transfers

**Example: Use limit and skip for Pagination**

Limit the amount of data transferred by using pagination.

db.students.find({}).skip(0).limit(10);

#### 9. Avoiding $lookup in Large Collections

**Example: Denormalize Data to Avoid Expensive Joins**

Instead of using $lookup, denormalize the data to avoid expensive joins.

db.students.updateOne(

{ student\_id: 1 },

{ $set: { major: { name: "Computer Science", department: "Engineering" } } }

);

#### 10. Query Optimization with Aggregation Pipeline

**Example: Optimize Aggregation Pipelines by Placing $match Early**

Place the $match stage early in the pipeline to reduce the number of documents processed in subsequent stages.

db.students.aggregate([

{ $match: { major: "Computer Science" } },

{ $group: { \_id: "$age", count: { $sum: 1 } } },

{ $sort: { count: -1 } }

]);

#### 11. Use of TTL Indexes for Time-Based Data

**Example: Automatically Remove Documents After a Certain Time**

Create a TTL (Time-To-Live) index to automatically remove documents after a specified time period.

db.students.createIndex({ enrollment\_date: 1 }, { expireAfterSeconds: 31536000 }); // 1 year

#### 12. Profiling and Monitoring

**Example: Enable Profiling to Collect Performance Data**

Set the profiling level to capture slow queries.

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

Analyze the collected profiling data.

db.system.profile.find().sort({ ts: -1 }).limit(5).pretty();

### Example Data and Performance Optimization

Let's combine these practices with examples:

#### Inserting Sample Data

db.students.insertMany([

{

"student\_id": 1,

"name": "Alice",

"age": 21,

"major": "Computer Science",

"courses": ["Databases", "Algorithms"],

"address": {

"street": "123 Main St",

"city": "Springfield",

"zip": "62701"

},

"enrollment\_date": ISODate("2021-09-01")

},

{

"student\_id": 2,

"name": "Bob",

"age": 22,

"major": "Mathematics",

"courses": ["Calculus", "Statistics"],

"address": {

"street": "456 Elm St",

"city": "Springfield",

"zip": "62702"

},

"enrollment\_date": ISODate("2020-09-01")

},

{

"student\_id": 3,

"name": "Charlie",

"age": 20,

"major": "Physics",

"courses": ["Quantum Mechanics", "Electromagnetism"],

"address": {

"street": "789 Oak St",

"city": "Springfield",

"zip": "62703"

},

"enrollment\_date": ISODate("2022-01-15")

}

]);

#### Creating Indexes for Optimization

db.students.createIndex({ student\_id: 1 });

db.students.createIndex({ major: 1, age: -1 });

db.students.createIndex({ courses: 1 });

#### Optimized Query with Projection

db.students.find({ major: "Computer Science" }, { name: 1, age: 1, \_id: 0 });

### Conclusion

These advanced performance optimization techniques include creating and managing indexes, using projections, leveraging covered queries, sharding, caching, and optimizing aggregation pipelines. By applying these techniques, you can significantly improve the performance and scalability of your MongoDB database operations.

User management and security in MongoDB are critical for controlling access and ensuring the safety of your data. Here are examples of user management, including creating users with different roles, and implementing security at the database and collection levels.

### User Management

#### 1. Enabling Authentication

First, enable authentication in MongoDB. This typically involves editing the MongoDB configuration file (mongod.conf) to enable authorization.

# mongod.conf

security:

authorization: "enabled"

Then, restart the MongoDB server.

#### 2. Creating Admin User

Create an admin user that has privileges to manage other users.

use admin;

db.createUser({

user: "admin",

pwd: "adminPassword",

roles: [{ role: "userAdminAnyDatabase", db: "admin" }]

});

#### 3. Creating Database-Specific Users

**Example: Create a user for the school database with read and write permissions**

use school;

db.createUser({

user: "schoolUser",

pwd: "schoolUserPassword",

roles: [{ role: "readWrite", db: "school" }]

});

**Example: Create a user with read-only access to the school database**

use school;

db.createUser({

user: "readOnlyUser",

pwd: "readOnlyPassword",

roles: [{ role: "read", db: "school" }]

});

### Security at Database Level

**Example: Grant a user read-only access to a specific database**

use school;

db.createUser({

user: "readOnlyUser",

pwd: "readOnlyPassword",

roles: [{ role: "read", db: "school" }]

});

**Example: Grant a user read-write access to a specific database**

use school;

db.createUser({

user: "writeUser",

pwd: "writePassword",

roles: [{ role: "readWrite", db: "school" }]

});

### Security at Collection Level

MongoDB does not natively support fine-grained, collection-level access control directly through user roles. Instead, you can achieve similar control by creating specific roles with granular permissions.

**Example: Create a custom role with read-only access to the students collection**

use admin;

db.createRole({

role: "readStudents",

privileges: [

{ resource: { db: "school", collection: "students" }, actions: ["find"] }

],

roles: []

});

**Example: Assign the custom role to a user**

use school;

db.createUser({

user: "studentsReadUser",

pwd: "studentsReadPassword",

roles: [{ role: "readStudents", db: "school" }]

});

**Example: Create a custom role with read-write access to the students collection**

use admin;

db.createRole({

role: "readWriteStudents",

privileges: [

{ resource: { db: "school", collection: "students" }, actions: ["find", "insert", "update", "remove"] }

],

roles: []

});

**Example: Assign the custom role to a user**

use school;

db.createUser({

user: "studentsReadWriteUser",

pwd: "studentsReadWritePassword",

roles: [{ role: "readWriteStudents", db: "school" }]

});

### Auditing and Monitoring

Enable MongoDB auditing to track user activity and monitor for suspicious actions.

**Example: Enable auditing in MongoDB configuration**

# mongod.conf

auditLog:

destination: file

format: JSON

path: /var/log/mongodb/audit.log

### Conclusion

These examples demonstrate how to manage users and secure your MongoDB instance at both the database and collection levels. By enabling authentication, creating users with appropriate roles, and configuring custom roles for fine-grained access control, you can ensure that only authorized users can access or modify your data. Additionally, auditing and monitoring user activity provide an extra layer of security.

In MongoDB, collections are flexible and can store various types of data. While MongoDB doesn't enforce a strict schema, it does support different types of collections that can be optimized for specific use cases. Here are the primary collection types and their usage, along with examples for each type:

### 1. Standard Collections

Standard collections are the most commonly used in MongoDB. They store documents in BSON format and provide flexibility without any predefined schema.

**Usage**: General-purpose storage for any type of document.

**Example**:

db.createCollection("students");

db.students.insertOne({

student\_id: 1,

name: "Alice",

age: 21,

major: "Computer Science",

courses: ["Databases", "Algorithms"],

address: {

street: "123 Main St",

city: "Springfield",

zip: "62701"

}

});

### 2. Capped Collections

Capped collections are fixed-size collections that maintain insertion order and automatically remove the oldest documents when the collection reaches its maximum size.

**Usage**: Logging, caching, and real-time data processing.

**Example**:

db.createCollection("logs", { capped: true, size: 5242880, max: 5000 });

db.logs.insertOne({

log\_id: 1,

timestamp: new Date(),

message: "User logged in",

level: "INFO"

});

### 3. Time-Series Collections

Time-series collections are optimized for storing time-series data, such as IoT data, monitoring data, and financial data. They automatically manage time-based partitions and improve query performance for time-series workloads.

**Usage**: IoT, monitoring, financial data, and any time-series data.

**Example**:

db.createCollection("sensorData", {

timeseries: {

timeField: "timestamp",

metaField: "sensorId",

granularity: "seconds"

}

});

db.sensorData.insertOne({

sensorId: "sensor1",

timestamp: new Date(),

temperature: 22.5,

humidity: 30

});

### 4. Views

Views are virtual collections created from the results of an aggregation pipeline. They do not store data but provide a way to query transformed data.

**Usage**: Aggregated views, read-only data transformations, data abstraction layers.

**Example**:

db.createView("activeStudents", "students", [

{ $match: { status: "active" } },

{ $project: { student\_id: 1, name: 1, major: 1 } }

]);

db.activeStudents.find({});

### 5. GridFS Collections

GridFS is a specification for storing and retrieving large files, such as images, videos, and documents, by dividing them into smaller chunks and storing them in two collections: fs.files and fs.chunks.

**Usage**: Storing large files and media content.

**Example**:

var bucket = new Mongo().getDB("myDatabase").getGridFSBucket();

bucket.uploadFromStream("myImage.jpg", fs.createReadStream("path/to/image.jpg"));

### 6. Encrypted Collections (MongoDB 6.0+)

Encrypted collections provide automatic encryption of documents at rest using field-level encryption.

**Usage**: Storing sensitive data that requires encryption, such as personal information or financial records.

**Example**:

// Define the encryptedFields option

const encryptedFields = {

fields: [

{

path: "ssn",

bsonType: "string",

keyId: [UUID("your-key-id")]

}

]

};

// Create the collection with encryption

db.createCollection("customers", { encryptedFields });

db.customers.insertOne({

name: "John Doe",

ssn: "123-45-6789",

email: "john.doe@example.com"

});

### Summary of Collection Types and Usage

1. **Standard Collections**: General-purpose storage.
2. **Capped Collections**: Fixed-size collections for logging and real-time data.
3. **Time-Series Collections**: Optimized for time-series data like IoT and monitoring.
4. **Views**: Virtual collections for querying transformed data.
5. **GridFS Collections**: For storing large files and media content.
6. **Encrypted Collections**: For storing sensitive data with automatic encryption.

By choosing the appropriate collection type, you can optimize your MongoDB database for different use cases, ensuring better performance, scalability, and security.

Handling backup and restore operations in MongoDB is essential for data protection and disaster recovery. Below, I provide examples of how to back up and restore data for the types of collections mentioned above using mongodump and mongorestore.

### Backup with mongodump

mongodump creates a binary export of the contents of a database.

**Basic Syntax:**

mongodump --uri="mongodb://<username>:<password>@<host>:<port>/<database>"

**Example: Backup the school database:**

mongodump --uri="mongodb://admin:adminPassword@localhost:27017/school" --out /path/to/backup/school\_backup

### Restore with mongorestore

mongorestore restores data from a binary export created by mongodump.

**Basic Syntax:**

mongorestore --uri="mongodb://<username>:<password>@<host>:<port>" /path/to/backup

**Example: Restore the school database:**

mongorestore --uri="mongodb://admin:adminPassword@localhost:27017" /path/to/backup/school\_backup

### Detailed Steps for Backup and Restore

#### Step 1: Backup Standard Collections

**Backup the students collection:**

mongodump --uri="mongodb://admin:adminPassword@localhost:27017/school" --collection=students --out /path/to/backup

**Backup the logs capped collection:**

mongodump --uri="mongodb://admin:adminPassword@localhost:27017/school" --collection=logs --out /path/to/backup

#### Step 2: Backup Time-Series Collections

**Backup the sensorData time-series collection:**

mongodump --uri="mongodb://admin:adminPassword@localhost:27017/school" --collection=sensorData --out /path/to/backup

#### Step 3: Backup Encrypted Collections

**Backup the customers encrypted collection:**

mongodump --uri="mongodb://admin:adminPassword@localhost:27017/school" --collection=customers --out /path/to/backup

#### Step 4: Restore Collections

**Restore the students collection:**

mongorestore --uri="mongodb://admin:adminPassword@localhost:27017/school" /path/to/backup/school/students.bson

**Restore the logs capped collection:**

mongorestore --uri="mongodb://admin:adminPassword@localhost:27017/school" /path/to/backup/school/logs.bson

**Restore the sensorData time-series collection:**

mongorestore --uri="mongodb://admin:adminPassword@localhost:27017/school" /path/to/backup/school/sensorData.bson

**Restore the customers encrypted collection:**

mongorestore --uri="mongodb://admin:adminPassword@localhost:27017/school" /path/to/backup/school/customers.bson

### Backup and Restore GridFS Collections

GridFS collections store large files, such as images and videos, in chunks.

**Backup the GridFS collections:**

mongodump --uri="mongodb://admin:adminPassword@localhost:27017/myDatabase" --out /path/to/backup/gridfs\_backup

**Restore the GridFS collections:**

mongorestore --uri="mongodb://admin:adminPassword@localhost:27017/myDatabase" /path/to/backup/gridfs\_backup