MONGODB LEARNING SHEET

1. Introduction to MongoDB

Definition: MongoDB is a NoSQL database that stores data in JSON-like documents with dynamic schemas, making the integration of data in certain types of applications easier and faster.

Key Features:

- **Document-oriented**: Uses BSON format (binary JSON).
- **Schema-less**: Documents in a collection need not have the same schema.
- **Scalable**: Supports horizontal scaling via sharding.
- **Flexible**: Allows for complex data structures and nested documents.

2. Data Modeling

Concept: Designing how data is stored in MongoDB. Proper modeling ensures efficient data retrieval and storage.

Key Considerations:

- **Document Structure**: Embedding vs. Referencing.
    - **Embedding**: Nest related data within a single document. Ideal for one-to-few relationships and when atomic operations are needed.
    - **Referencing**: Use references to other documents. Ideal for one-to-many relationships and when document sizes can grow large.

```
QUERY-

// Embedding example

{

  _id: 1,

  name: "Alice",

  address: {

      street: "123 Main St",

      city: "Springfield",

      state: "IL"
```

```
  }
}


// Referencing example
{
  _id: 2,
  name: "Bob",
  address_id: 1
}


// Address document
{
  _id: 1,
  street: "123 Main St",
  city: "Springfield",
  state: "IL"
}
```

Schema Design Patterns:

- **One-to-One**: Typically embedded.
- **One-to-Many**: Embedding for small sub-documents, referencing for large or growing sets.
- **Many-to-Many**: Referencing with separate collection for relationships.

3. Indexing

Concept: Indexes support the efficient execution of queries in MongoDB by reducing the amount of data that must be scanned.

Types:

- **Single Field**: Index on a single field.
- **Compound**: Index on multiple fields.
- **Multikey**: Indexes array fields.
- **Text**: Supports text search.

- **Geospatial**: Supports queries for geospatial data.

Creating Indexes:

```
QUERY-

// Single field index

db.collection.createIndex({ fieldName: 1 }); // Ascending order



// Compound index

db.collection.createIndex({ field1: 1, field2: -1 }); // Mixed order
```

Index Usage:

- **Improve Query Performance**: Speed up data retrieval.
- **Ensure Uniqueness**: Use unique indexes to enforce field uniqueness.
- **Sorting**: Indexes can also be used for efficient sorting.

Monitoring Indexes:

- Use `explain()` to analyze query performance and index usage.

```
QUERY-

db.collection.find({ field: value }).explain("executionStats");
```

4. CRUD Operations

Create: Insert documents into a collection.

```
QUERY-

// Insert a single document

db.collection.insertOne({ name: "Alice", age: 30 });



// Insert multiple documents

db.collection.insertMany([{ name: "Bob", age: 25 }, { name: "Charlie", age:
35 }]);
```

Read: Query documents from a collection.

```
QUERY-

// Find all documents
```

```
db.collection.find();


// Find with filter

db.collection.find({ age: { $gt: 25 } });


// Find one document

db.collection.findOne({ name: "Alice" });
```

Update: Modify existing documents.

```
QUERY-

// Update a single document

db.collection.updateOne({ name: "Alice" }, { $set: { age: 31 } });


// Update multiple documents

db.collection.updateMany({ age: { $gt: 25 } }, { $set: { status: "active" }
});
```

Delete: Remove documents from a collection.

```
QUERY-

// Delete a single document

db.collection.deleteOne({ name: "Alice" });


// Delete multiple documents

db.collection.deleteMany({ age: { $lt: 30 } });
```

5. Aggregation

Concept: Aggregation operations process data records and return computed results. MongoDB provides an aggregation framework for transforming and combining data.

Stages:

- **$match**: Filters documents.
- **$group**: Groups documents by a specified field.
- **$project**: Reshapes documents.

- **$sort**: Sorts documents.
- **$limit**: Limits the number of documents.
- **$skip**: Skips over a number of documents.

```
QUERY-

db.collection.aggregate([

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

  { $group: { _id: "$age", total: { $sum: 1 } } },

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

  { $project: { age: "$_id", total: 1, _id: 0 } }

]);
```

Pipeline: Series of stages through which documents pass, allowing for complex transformations and computations.

6. Query Optimization

Concept: Enhancing the performance of queries by using efficient query techniques and proper indexing.

Tips:

- **Use Indexes**: Ensure fields used in queries are indexed.
- **Limit Result Set**: Use `limit()` to restrict the number of documents returned.
- **Avoid Unnecessary Fields**: Use projections to return only necessary fields.
- **Analyze with explain()**: Use `explain()` to understand query execution plans and optimize accordingly.

```
QUERY-

// Example of using explain()

db.collection.find({ age: { $gt: 25 } }).explain("executionStats");
```

7. Replication

Concept: Replication in MongoDB provides high availability and data redundancy by replicating data across multiple servers.

Replica Set:

- **Primary**: Receives all write operations.
- **Secondaries**: Replicate data from the primary and can serve read operations.
- **Arbiter**: Participates in elections but does not store data.

Commands:

- **rs.initiate()**: Initialize a replica set.
- **rs.add()**: Add a member to a replica set.
- **rs.status()**: Check the status of a replica set.

```
QUERY-

rs.initiate({

  _id: "rs0",

  members: [

      { _id: 0, host: "localhost:27017" },

      { _id: 1, host: "localhost:27018" },

      { _id: 2, host: "localhost:27019" }

  ]

});
```

8. Sharding

Concept: Sharding is a method for distributing data across multiple machines, ensuring horizontal scalability.

Components:

- **Shard**: A single MongoDB instance that holds a subset of the sharded data.
- **Config Servers**: Store metadata and configuration settings.
- **Query Routers (mongos)**: Routes queries to the appropriate shard.

Key Concepts:

- **Shard Key**: Field used to distribute documents across shards.
- **Chunks**: Subsets of data within a shard.

Setup:

- Enable sharding on a database: `sh.enableSharding("myDatabase")`.
- Shard a collection: `sh.shardCollection("myDatabase.myCollection", { shardKey: 1 })`.

```
QUERY-

sh.enableSharding("myDatabase");

sh.shardCollection("myDatabase.myCollection", { shardKey: 1 });
```

9. Transactions

Concept: Transactions in MongoDB allow for multi-document operations to be executed with ACID properties.

Usage:

- **Start a Session**: Begin a transaction.
- **Commit**: Complete the transaction.
- **Abort**: Rollback the transaction.

```
QUERY-

const session = client.startSession();

session.startTransaction();



try {

  await db.collection.insertOne({ name: "Alice" }, { session });

  await db.collection.updateOne({ name: "Bob" }, { $set: { age: 30 } }, {
session });

  await session.commitTransaction();

} catch (error) {

  await session.abortTransaction();

  throw error;

} finally {

  session.endSession();

}
```

10. Security

Concept: Ensuring the MongoDB deployment is secure from unauthorized access and vulnerabilities.

Key Practices:

- **Authentication**: Require users to authenticate.
- **Authorization**: Assign roles to users to control access.
- **Encryption**: Encrypt data at rest and in transit.
- **Network Security**: Configure firewalls and network rules.

Commands:

- **Enable Authentication**: `security.authorization: "enabled"` in the configuration file.
- **Create User**: `db.createUser()` to create users with specific roles.

```
QUERY-

db.createUser({

  user: "admin",

  pwd: "password",

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

});
```

1 Common Use Cases for MongoDB

1.1 Content Management Systems (CMS)

- **Use Case**: MongoDB is ideal for CMS applications where content types vary and evolve over time.
- **Example**: Blogs, news sites, and e-commerce product catalogs.
- **Reason**: Flexible schema allows easy adaptation to changing content structures without downtime.

1.2 Internet of Things (IoT)

- **Use Case**: Storing and analyzing large volumes of data generated by IoT devices.
- **Example**: Smart home systems, industrial sensors, and wearables.
- **Reason**: Scalability and ability to handle high write loads and real-time data processing.

1.3 Real-Time Analytics

- **Use Case**: Applications that require real-time analytics and data visualization.
- **Example**: Financial services, monitoring systems, and recommendation engines.
- **Reason**: Aggregation framework and in-memory storage capabilities for fast data access and computation.

1.4 Product Data Management

- **Use Case**: Managing complex product information and inventory systems.
- **Example**: E-commerce platforms, inventory management systems, and supply chain management.
- **Reason**: Ability to store nested documents and perform complex queries.

1.5 Social Networks

- **Use Case**: Storing and managing user-generated content and interactions.
- **Example**: Social media platforms, messaging apps, and community forums.
- **Reason**: Flexible data model for user profiles, posts, comments, and relationships.

1.6 Gaming

- **Use Case**: Handling player profiles, game state, and real-time analytics.
- **Example**: Online multiplayer games, mobile games, and game leaderboards.
- **Reason**: High availability, low latency, and ability to handle large volumes of concurrent connections.

2. Best Practices for Implementing MongoDB

2.1 Data Modeling Best Practices

- **Design Schema Based on Application Needs**: Consider query patterns, data access, and update frequency.
- **Use Embedding for Small, Frequently Accessed Data**: Embed related data within the same document when possible to avoid additional queries.
- **Use Referencing for Large or Growing Data Sets**: Use references for one-to-many relationships where sub-documents are large or frequently updated.
- **Avoid Large Documents**: Keep document sizes within MongoDB's 16MB limit to prevent performance issues.

2.2 Indexing Best Practices

- **Create Indexes on Frequently Queried Fields**: Ensure fields used in queries, sorting, and filtering have indexes.
- **Use Compound Indexes for Multiple Fields**: Optimize queries that filter by multiple fields using compound indexes.
- **Monitor Index Performance**: Use `explain()` to analyze query performance and adjust indexes as needed.
- **Avoid Over-Indexing**: Too many indexes can slow down write operations. Index only necessary fields.

2.3 Query Optimization Best Practices

- **Limit and Project**: Use `limit()` to restrict the number of documents returned and projections to return only necessary fields.
- **Use Covered Queries**: Ensure queries can be satisfied by indexes alone, without accessing the document data.
- **Analyze and Optimize Slow Queries**: Regularly use performance monitoring tools to identify and optimize slow queries.

2.4 Aggregation Best Practices

- **Break Down Complex Aggregations**: Simplify complex aggregation pipelines into multiple stages or smaller pipelines.

- **Use Appropriate Stages**: Choose the correct stages (`$match`, `$group`, `$project`, etc.) for your aggregation needs to improve performance.
- **Index Fields Used in Aggregations**: Ensure fields used in `$match` and `$group` stages are indexed for faster processing.

3. Best Practices for Managing MongoDB Databases

3.1 Backup and Recovery

- **Regular Backups**: Schedule regular backups using MongoDB tools like `mongodump` and `mongorestore`.
- **Test Restores**: Periodically test restoring backups to ensure data integrity and recovery procedures.
- **Automate Backup Processes**: Use automated backup solutions to reduce manual intervention and ensure consistency.

3.2 Monitoring and Performance Tuning

- **Use Monitoring Tools**: Leverage tools like MongoDB Atlas, MMS, or third-party solutions to monitor database performance and health.
- **Set Alerts**: Configure alerts for critical metrics such as CPU usage, memory consumption, disk I/O, and query performance.
- **Regularly Review Performance Metrics**: Analyze performance data and adjust configurations, indexes, or data models as needed.

3.3 Security Best Practices

- **Enable Authentication and Authorization**: Ensure that MongoDB is configured to require user authentication and proper role-based access control.
- **Encrypt Data**: Use TLS/SSL for data in transit and enable encryption at rest to protect sensitive data.
- **Network Security**: Configure firewalls to allow only trusted IP addresses to access MongoDB instances. Use VPNs or private networks for added security.
- **Regular Security Audits**: Conduct regular security audits and penetration testing to identify and mitigate vulnerabilities.

3.4 Scalability and High Availability

- **Replica Sets for High Availability**: Use replica sets to ensure high availability and automatic failover.
- **Sharding for Scalability**: Implement sharding to distribute data across multiple servers, enabling horizontal scaling.
- **Load Balancing**: Use load balancers to distribute client requests evenly across MongoDB instances.

3.5 Maintenance Best Practices

- **Regular Maintenance Windows**: Schedule regular maintenance windows for updates, backups, and other administrative tasks.

- **Monitor Disk Space**: Ensure sufficient disk space for data growth, log files, and backups.
- **Optimize Storage**: Use storage engines like WiredTiger for efficient data compression and better performance.