Intermediate MongoDB Concepts

Aggregation Framework ....

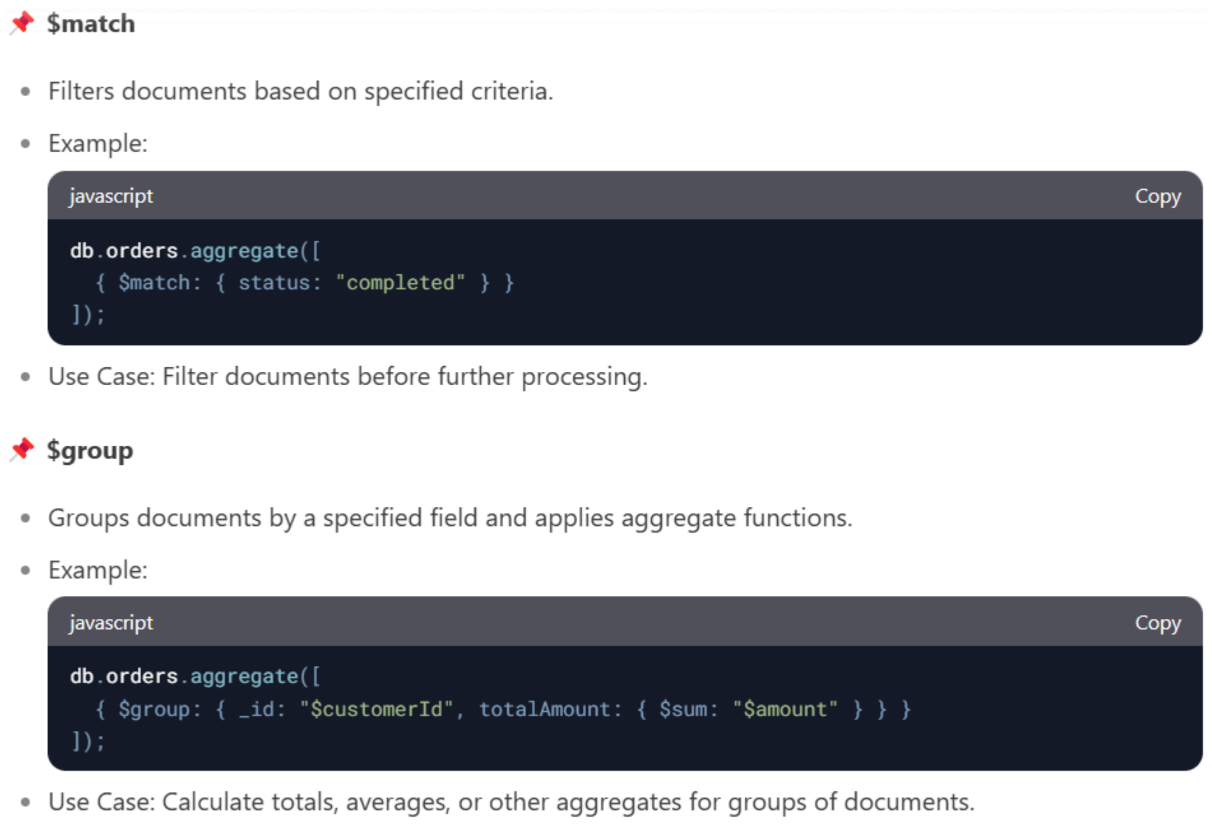
The **Aggregation Framework** in MongoDB is a powerful tool for performing complex data transformations and analysis. It processes data records and returns computed results using a **pipeline** of stages. Each stage transforms the documents as they pass through the pipeline.

Aggregation Syntax:

```
db.collection.aggregate([
  { stage1 },
  { stage2 },
  { stage3 },
]);
```

📌 1. Aggregation Pipeline

The aggregation pipeline consists of multiple stages, where each stage performs a specific operation on the data. Common stages include:



📌 $match

- Filters documents based on specified criteria.
- Example:

javascript Copy

```javascript
db.orders.aggregate([
  { $match: { status: "completed" } }
]);
```

- Use Case: Filter documents before further processing.

📌 $group

- Groups documents by a specified field and applies aggregate functions.
- Example:

javascript Copy

```javascript
db.orders.aggregate([
  { $group: { _id: "$customerId", totalAmount: { $sum: "$amount" } } }
]);
```

- Use Case: Calculate totals, averages, or other aggregates for groups of documents.

📌 $sort

- Sorts documents by specified fields.
- Example:

javascript Copy

```javascript
db.orders.aggregate([
  { $sort: { amount: -1 } } // Sort by amount in descending order
]);
```

- Use Case: Order results for reporting or further processing.

📌 $project

- Selects specific fields or computes new fields.
- Example:

javascript Copy

```javascript
db.orders.aggregate([
  { $project: { customerId: 1, amount: 1, _id: 0 } }
]);
```

- Use Case: Include or exclude fields, or create computed fields.

📌 $limit

- Limits the number of documents passed to the next stage.
- Example:

javascript Copy

```javascript
db.orders.aggregate([
  { $limit: 5 } // Limit to 5 documents
]);
```

- Use Case: Restrict results for pagination or sampling.

📌 $skip

- Skips a specified number of documents.
- Example:

javascript Copy

```javascript
db.orders.aggregate([
  { $skip: 10 } // Skip the first 10 documents
]);
```

- Use Case: Implement pagination or skip initial records.

📌 2. Operators

Aggregation operators are used within stages like $group or $project to perform calculations or transformations.

📌 $sum

- Calculates the sum of numeric values.
- Example:

```javascript
db.orders.aggregate([
  { $group: { _id: "$customerId", totalAmount: { $sum: "$amount" } } }
]);
```

📌 $avg

- Calculates the average of numeric values.
- Example:

```javascript
db.orders.aggregate([
  { $group: { _id: "$customerId", averageAmount: { $avg: "$amount" } } }
]);
```

📌 $min

- Finds the minimum value in a group.
- Example:

```javascript
db.orders.aggregate([
  { $group: { _id: "$customerId", minAmount: { $min: "$amount" } } }
]);
```

📌 $max

- Finds the maximum value in a group.
- Example:

```javascript
db.orders.aggregate([
  { $group: { _id: "$customerId", maxAmount: { $max: "$amount" } } }
]);
```

📌 **$push**

- Adds values to an array for each group.
- Example:

javascript Copy

```javascript
db.orders.aggregate([
  { $group: { _id: "$customerId", orderIds: { $push: "$_id" } } }
]);
```

📌 **$addToSet**

- Adds unique values to an array for each group.
- Example:

javascript Copy

```javascript
db.orders.aggregate([
  { $group: { _id: "$customerId", uniqueProducts: { $addToSet: "$product" } } }
]);
```

📌 3. $lookup (Joining Collections in MongoDB)

Performs a left outer join between two collections.

Example:

javascript Copy

```javascript
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",         // Collection to join
      localField: "customerId", // Field from the input documents
      foreignField: "_id",       // Field from the "from" collection
      as: "customerDetails"      // Output array field
    }
  }
]);
```

Use Case: Process individual elements of an array (e.g., items in an order).

📌 4. $unwind (Working with Arrays)

Deconstructs an array field, creating a separate document for each element.

Example:

```javascript
db.orders.aggregate([
  { $unwind: "$items" } // Unwind the items array
]);
```

Use Case: Process individual elements of an array (e.g., items in an order).

📌 5. Example: Full Aggregation Pipeline

Here's an example that combines multiple stages:

```javascript
db.orders.aggregate([
  { $match: { status: "completed" } }, // Filter completed orders
  { $unwind: "$items" },               // Unwind the items array
  { $group: {                          // Group by product and calculate totals
      _id: "$items.product",
      totalQuantity: { $sum: "$items.quantity" },
      totalRevenue: { $sum: { $multiply: ["$items.quantity", "$items.price"] } }
  }},
  { $sort: { totalRevenue: -1 } },     // Sort by total revenue in descending order
  { $limit: 5 }                        // Limit to top 5 products
]);
```

Key Takeaways

- **Aggregation Pipeline**: A sequence of stages (`$match`, `$group`, `$sort`, etc.) to process data.
- **Operators**: Use operators like `$sum`, `$avg`, `$push`, and `$addToSet` for calculations and transformations.
- **$lookup**: Join data from multiple collections.
- **$unwind**: Process individual elements of an array.

Schema Design and Relationships ....

Designing an effective schema is crucial for building scalable and efficient MongoDB applications.
Unlike relational databases, MongoDB allows flexible schema designs, but this flexibility requires careful planning. Let's explore the key concepts and best practices:

📌 1. Embedding vs. Referencing

MongoDB supports two primary ways to structure data: embedding and referencing.

Embedding

Store related data within a single document.

Example:

```
{
  "_id": 1,
  "name": "John",
  "address": {
    "city": "New York",
    "zip": "10001"
  },
  "orders": [
    { "orderId": 101, "product": "Laptop", "price": 1200 },
    { "orderId": 102, "product": "Phone", "price": 800 }
  ]
}
```

Advantages:

- Fast read operations (all data is in one document).
- Atomic operations on a single document.

Use Cases:

- One-to-few relationships (e.g., user and their address).
- Data that is frequently accessed together.

Referencing

Store related data in separate documents and reference them using IDs.

Example:

```
// Users Collection
{
  "_id": 1,
  "name": "John",
  "addressId": 101
}

// Addresses Collection
{
  "_id": 101,
  "city": "New York",
  "zip": "10001"
}
```

Advantages:

- ○ Avoids duplication of data.
- ○ Better for large or frequently updated related data.

Use Cases:

- ○ One-to-many or many-to-many relationships (e.g., users and orders).
- ○ Data that grows over time or is updated frequently.

📌 2. One-to-One, One-to-Many, Many-to-Many Relationships

One-to-One Relationship

- A single document in one collection is related to a single document in another collection.
- Example: A user and their profile picture.
- **Design**:
  - Use **embedding** if the data is small and frequently accessed together.
  - Use **referencing** if the data is large or updated frequently.

One-to-Many Relationship

- A single document in one collection is related to multiple documents in another collection.
- Example: A user and their orders.
- **Design**:

    - Use **embedding** if the number of related documents is small (e.g., user and their addresses).
    - Use **referencing** if the number of related documents is large (e.g., user and their orders).

Many-to-Many Relationship

- Multiple documents in one collection are related to multiple documents in another collection.
- Example: Students and courses (a student can enroll in many courses, and a course can have many students).
- **Design**:
    - Use **referencing** with an array of IDs in one or both collections.
    - Example:

```
// Students Collection
{
  "_id": 1,
  "name": "Alice",
  "courseIds": [101, 102]
}

// Courses Collection
{
  "_id": 101,
  "name": "Math",
  "studentIds": [1, 2]
}
```

📌 3. Best Practices for Schema Design

Understand Your Data Access Patterns

- Design your schema based on how your application queries and updates data.
- Example: If you frequently query a user and their orders together, consider embedding orders in the user document.

Avoid Large, Deeply Nested Documents

- Large documents can lead to performance issues and storage inefficiencies.
- Example: Avoid embedding thousands of orders in a user document.

Use Indexes Wisely

- Create indexes on fields that are frequently queried or used in joins.

- Example: Index customerId in the orders collection for faster lookups.

Balance Embedding and Referencing

- Use embedding for small, related data that is frequently accessed together.
- Use referencing for large or frequently updated data.

Plan for Growth

- Design your schema to handle future growth in data size and query complexity.
- Example: Use referencing for relationships that may grow over time (e.g., user and their orders).

Leverage MongoDB Features

- Use **arrays** and **subdocuments** for flexible schema designs.
- Use **$lookup** for joining collections when referencing is necessary.

Example: Schema Design for an E-commerce Application

- **Users Collection**:

```
{
  "_id": 1,
  "name": "John",
  "email": "john@example.com",
  "addresses": [
    { "city": "New York", "zip": "10001" },
    { "city": "Los Angeles", "zip": "90001" }
  ]
}
```

- **Orders Collection**:

```
{
  "_id": 101,
  "userId": 1,
  "products": [
    { "productId": 201, "quantity": 2 },
    { "productId": 202, "quantity": 1 }
  ],
  "totalAmount": 2000
}
```

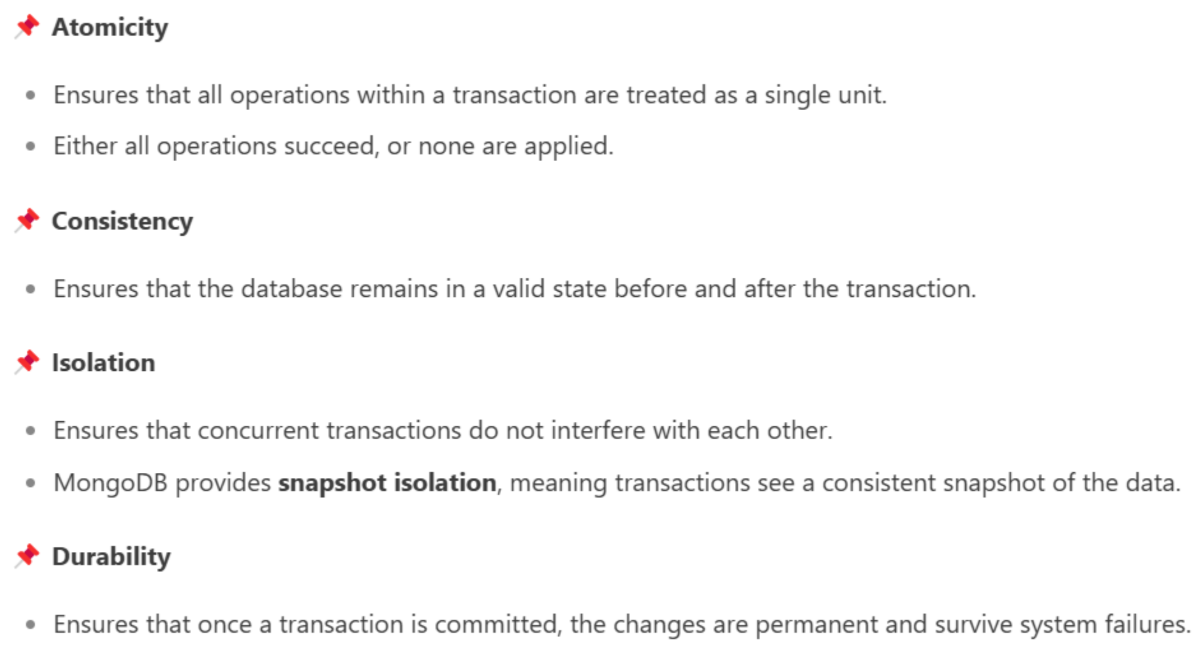
- **Products Collection**:

```
{
  "_id": 201,
  "name": "Laptop",
  "price": 1200
}
```

MongoDB Transactions (ACID Compliance) ……………

📌 1. What are ACID Transactions?

MongoDB transactions follow **ACID properties**:



📌 **Atomicity**

- Ensures that all operations within a transaction are treated as a single unit.
- Either all operations succeed, or none are applied.

📌 **Consistency**

- Ensures that the database remains in a valid state before and after the transaction.

📌 **Isolation**

- Ensures that concurrent transactions do not interfere with each other.
- MongoDB provides **snapshot isolation**, meaning transactions see a consistent snapshot of the data.

📌 **Durability**

- Ensures that once a transaction is committed, the changes are permanent and survive system failures.

📌 2. When to Use Transactions in MongoDB?

Transactions are **useful when multiple collections need consistency**.
For example, an **e-commerce system** where:
1️⃣**Deduct stock from the inventory**.
2️⃣**Create an order for the user**.
3️⃣**Process the payment**.

These actions **must either all succeed or all fail**.

📌 3. Key Methods for Transactions

📌 startSession()

- Starts a new session, which is required to create a transaction.
- Example:

```javascript
const session = db.getMongo().startSession();
```

📌 withTransaction()

- Executes a transaction within a session.
- Automatically handles committing or aborting the transaction.
- Example:

```javascript
session.withTransaction(async () => {
  const ordersCollection = session.getDatabase("test").orders;
  const usersCollection = session.getDatabase("test").users;

  // Transfer $100 from user1 to user2
  await usersCollection.updateOne(
    { _id: "user1" },
    { $inc: { balance: -100 } },
    { session }
  );
  await usersCollection.updateOne(
    { _id: "user2" },
    { $inc: { balance: 100 } },
    { session }
  );
});
```

📌 commitTransaction()

- Commits the transaction, making all changes permanent.
- Example:

```javascript
session.commitTransaction();
```

📌 abortTransaction()

- Aborts the transaction, discarding all changes.
- Example:

```javascript
session.abortTransaction();
```

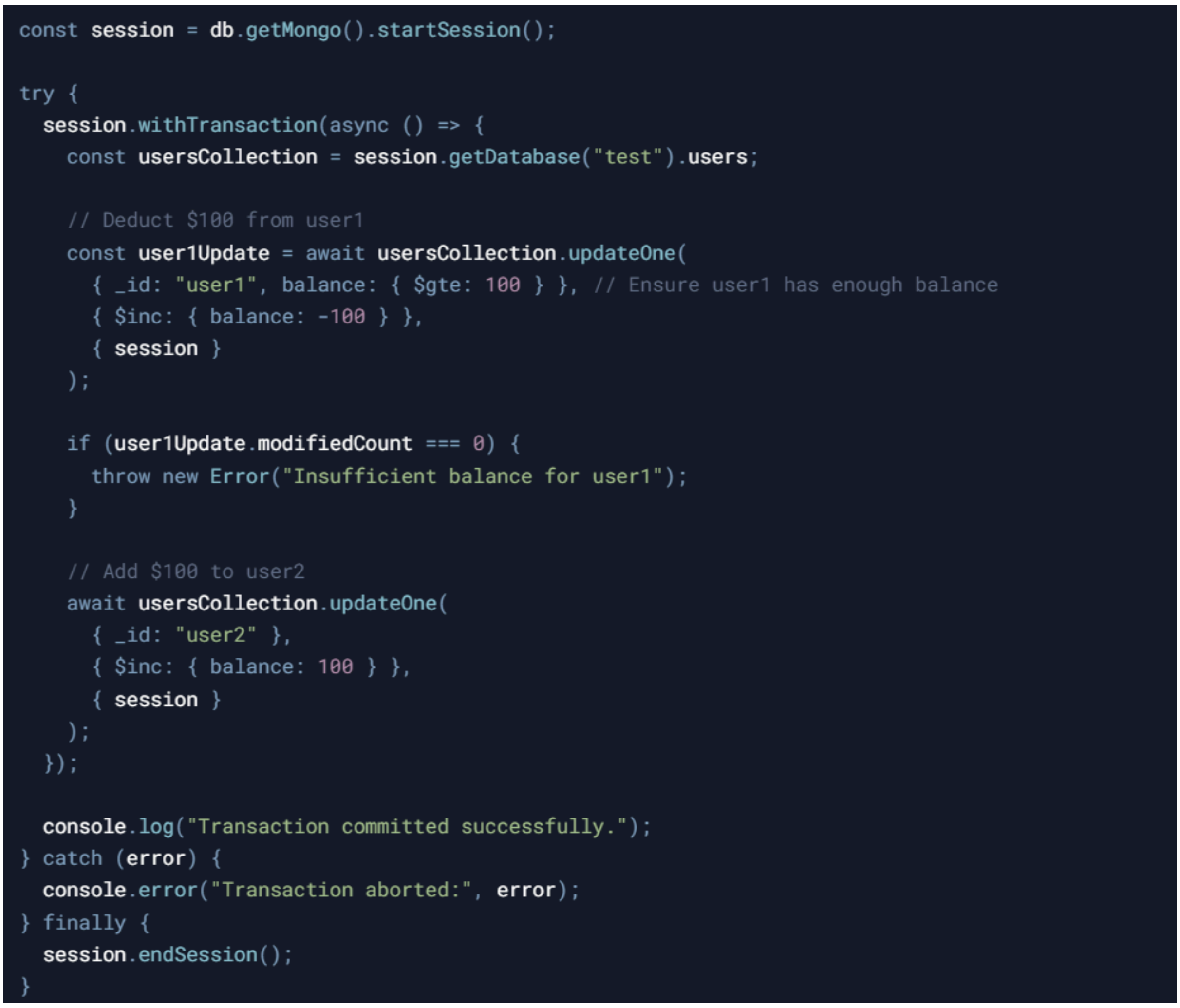
Handling Transactions with with `Transaction()` (Best Practice)

The `withTransaction()` method simplifies transaction handling by automatically **retrying** in case of failures.

```
async function runTransactionWithRetry(client) {
  const session = client.startSession();

  try {
    await session.withTransaction(async () => {
      const db = client.db("ecommerce");
      const orders = db.collection("orders");
      const inventory = db.collection("inventory");

      await inventory.updateOne(
        { product: "Laptop" },
        { $inc: { stock: -1 } },
        { session }
      );

      await orders.insertOne(
        { user: "Alice", product: "Laptop", status: "Processing" },
        { session }
      );

    }, { readConcern: { level: "majority" }, writeConcern: { w: "majority" } });

    console.log("Transaction committed successfully");

  } catch (error) {
    console.error("Transaction failed:", error);
  } finally {
    session.endSession();
  }
}

const client = new MongoClient("mongodb://localhost:27017");
client.connect().then(() => runTransactionWithRetry(client));
```

Example: **Transfer Funds Between Users**

Here’s a complete example of a transaction to transfer funds between two users:



```
const session = db.getMongo().startSession();

try {
  session.withTransaction(async () => {
    const usersCollection = session.getDatabase("test").users;

    // Deduct $100 from user1
    const user1Update = await usersCollection.updateOne(
      { _id: "user1", balance: { $gte: 100 } }, // Ensure user1 has enough balance
      { $inc: { balance: -100 } },
      { session }
    );

    if (user1Update.modifiedCount === 0) {
      throw new Error("Insufficient balance for user1");
    }

    // Add $100 to user2
    await usersCollection.updateOne(
      { _id: "user2" },
      { $inc: { balance: 100 } },
      { session }
    );
  });

  console.log("Transaction committed successfully.");
} catch (error) {
  console.error("Transaction aborted:", error);
} finally {
  session.endSession();
}
```

MongoDB Security ....

📌 1. Authentication & Authorization

Authentication

- Verifies the identity of users or applications connecting to the database.
- MongoDB supports several authentication mechanisms:
  - **SCRAM (Default)**: Salted Challenge Response Authentication Mechanism.
  - **x.509 Certificates**: For certificate-based authentication.
  - **LDAP**: For integrating with LDAP directories.
  - **Kerberos**: For enterprise-level authentication.

Authorization

- Controls what authenticated users can do within the database.
- MongoDB uses **Role-Based Access Control (RBAC)** to manage permissions.

Enabling Authentication in MongoDB

By default, MongoDB allows **unauthenticated access**. To enable authentication:

1️⃣ Start MongoDB with authentication enabled:

sh Copy Edit

```sh
mongod --auth --port 27017 --dbpath /data/db
```

2️⃣ Connect to MongoDB without authentication:

sh Copy Edit

```sh
mongo --port 27017
```

3️⃣ Switch to the `admin` database and create an **admin user**:

sh Copy Edit

```sh
use admin
db.createUser({
  user: "admin",
  pwd: "securePassword123",
  roles: ["root"]
})
```

4️⃣ Restart MongoDB with authentication:

```sh
mongod --auth --port 27017 --dbpath /data/db
```

5️⃣ Connect with authentication:

```sh
mongo --port 27017 -u "admin" -p "securePassword123" --authenticationDatabase "admin"
```

📌 2. Role-Based Access Control (RBAC)

RBAC controls **what actions a user can perform** in the database.

Default Roles in MongoDB

Role	Permissions
read	Read data from collections
readWrite	Read & write data
dbAdmin	Manage indexes & collections
userAdmin	Manage users & roles
root	Full access to everything

For example, creating a user who **can only read and write to the `ecommerce` database**:

```sh
use ecommerce
db.createUser({
  user: "ecomUser",
  pwd: "ecomPass123",
  roles: [{ role: "readWrite", db: "ecommerce" }]
})
```

Login as `ecomUser`:

```sh
mongo --port 27017 -u "ecomUser" -p "ecomPass123" --authenticationDatabase "ecommerce"
```

📌 3. Data Encryption

📌 Encryption at Rest

- Protects data stored on disk.
- MongoDB supports **WiredTiger Encryption** (available in MongoDB Enterprise).
- Example: Enable encryption at rest in the configuration file.

```yaml
storage:
  wiredTiger:
    engineConfig:
      encryption:
        keyFile: /path/to/keyfile
```

📌 Encryption in Transit

- Protects data transmitted over the network.
- MongoDB uses **TLS/SSL** to encrypt communication between clients and servers.
- Example: Enable TLS in the configuration file.

```yaml
net:
  tls:
    mode: requireTLS
    certificateKeyFile: /path/to/certificate.pem
```

📌 Client-Side Field-Level Encryption

- Encrypts specific fields in a document before sending data to the server.
- Example: Use the MongoDB Client-Side Field-Level Encryption library.

```javascript
const client = new MongoClient(uri, {
  autoEncryption: {
    keyVaultNamespace: "encryption.__keyVault",
    kmsProviders: { local: { key: "masterKey" } }
  }
});
```

📌 4. Backup & Restore Strategies

📌 Backup Strategies

- **Mongodump**:
  - Creates a binary export of the database.
  - Example:

```bash
mongodump --uri="mongodb://localhost:27017" --out=/backup
```

- **File System Snapshots**:
  - Use file system tools (e.g., LVM, EBS snapshots) to create backups.
- **MongoDB Atlas**:
  - Provides automated backups for cloud-hosted MongoDB instances.

📌 Restore Strategies

- **Mongorestore**:
  - Restores data from a `mongodump` backup.
  - Example:

```bash
mongorestore --uri="mongodb://localhost:27017" /backup
```

- **File System Snapshots**:
  - Restore data from file system snapshots.
- **Point-in-Time Recovery**:
  - Use MongoDB's **oplog** to restore data to a specific point in time.

📌 5. Additional Security Features

📌 Auditing

- Tracks database activities for compliance and security monitoring.
- Example: Enable auditing in the configuration file.

```yaml
auditLog:
  destination: file
  format: JSON
  path: /var/log/mongodb/audit.log
```

📌 **Network Security**

- Restrict access to MongoDB instances using firewalls and VPNs.
- Use **bindIp** to limit MongoDB to specific IP addresses.

```yaml
net:
  bindIp: 127.0.0.1,192.168.1.100
```

📌 **Database Auditing**

- Logs all database operations for compliance and security analysis.
- Example: Enable auditing in the configuration file.

```yaml
auditLog:
  destination: file
  format: JSON
  path: /var/log/mongodb/audit.log
```

Replication in MongoDB ....

📌 1. What is Replication?

Replication is the process of **copying data from one MongoDB server (Primary) to multiple servers (Secondaries)** in real time.

📌 Why is Replication Important?

✔ **High Availability** → If the primary server fails, a secondary takes over.
✔ **Disaster Recovery** → Protects against data loss.
✔ **Scalability** → Distributes read operations across multiple secondaries.
✔ **Backup & Maintenance** → Allows backups without affecting the primary database.

How Does Replication Work?

1️⃣**A Primary node** handles all **write operations**.
2️⃣**Secondary nodes** replicate the data from the primary asynchronously.
3️⃣If the **primary node fails**, a secondary **automatically becomes the new primary** (via **election**).

📌 2. Replica Set Architecture

Replica set is a group of MongoDB instances that maintain the same data set. It consists of:

Primary Node

- The primary node is the only member of the replica set that can accept write operations.
- All changes to the data are recorded in the oplog (operation log).

Secondary Nodes

- Secondary nodes replicate data from the primary node.
- They can serve read operations (if configured) but cannot accept writes.
- Secondary nodes can be configured for specific purposes (e.g., reporting, backups).

Arbiter Node

- An arbiter is a lightweight member that participates in elections but does not store data.
- Used to break ties in elections when there is an even number of nodes.

Oplog

- The oplog (operation log) is a capped collection that records all write operations.
- Secondary nodes use the oplog to replicate changes from the primary node.

📌 3. Primary & Secondary Nodes

Primary Node

- Responsibilities:
  - Accepts all write operations.
  - Records changes in the oplog.
- Failover:
  - If the primary node fails, an election is triggered to select a new primary.

Secondary Nodes

- Responsibilities:
  - Replicate data from the primary node.
  - Can serve read operations (if configured).
- Types of Secondary Nodes:
  - **Hidden**: Not visible to applications (used for backups or reporting).
  - **Delayed**: Replicates data with a delay (used for disaster recovery).
  - **Arbiter**: Participates in elections but does not store data.

Election Process

- When the primary node fails, the replica set automatically elects a new primary.
- The election process ensures that only one primary node exists at a time.
- Factors influencing elections:
  - Priority (configured for each node).
  - Node availability and health.

📌 4. Read and Write Concern

📌 Write Concern

- Controls the level of acknowledgment requested from MongoDB for write operations.
- Options:
  - `w: 1` : Acknowledges writes to the primary node (default).
  - `w: "majority"` : Acknowledges writes to a majority of nodes.
  - `w: 2` : Acknowledges writes to at least two nodes.
- Example:

javascript Copy
```javascript
db.orders.insert(
  { orderId: 101, product: "Laptop", price: 1200 },
  { writeConcern: { w: "majority", j: true } }
);
```

📌 Read Concern

- Controls the consistency and isolation properties of read operations.
- Options:
  - `local` : Reads the most recent data on the primary or secondary node.
  - `majority` : Reads data that has been acknowledged by a majority of nodes.
  - `linearizable` : Ensures linearizable reads (strongest consistency).
- Example:

javascript Copy
```javascript
db.orders.find({ orderId: 101 }).readConcern("majority");
```

📌 Read Preference

- Controls how read operations are distributed across replica set members.
- Options:
  - `primary` : Reads from the primary node (default).
  - `secondary` : Reads from secondary nodes.
  - `nearest` : Reads from the node with the lowest latency.
- Example:

javascript Copy
```javascript
db.orders.find({ orderId: 101 }).readPref("secondary");
```

📌 5. Example: Setting Up a Replica Set

1. **Start MongoDB Instances**:
   - Start three MongoDB instances with the `--replSet` option.

bash Copy

```bash
mongod --replSet rs0 --port 27017 --dbpath /data/db1
mongod --replSet rs0 --port 27018 --dbpath /data/db2
mongod --replSet rs0 --port 27019 --dbpath /data/db3
```

2. **Initialize the Replica Set**:
   - Connect to one of the instances and initialize the replica set.

javascript Copy

```javascript
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "localhost:27017" },
    { _id: 1, host: "localhost:27018" },
    { _id: 2, host: "localhost:27019" }
  ]
});
```

3. **Verify the Replica Set**:
   - Check the status of the replica set.

javascript Copy

```javascript
rs.status();
```

Sharding in MongoDB ……………………………..……….

📌 1. What is Sharding?

- **Definition**:
  - Sharding is a database architecture pattern that splits data across multiple servers (shards) to distribute the load.
- **Components of a Sharded Cluster**:
  - **Shards**: Individual MongoDB instances that store a subset of the data.
  - **Config Servers**: Store metadata and configuration settings for the cluster.
  - **Query Routers (Mongos)**: Route client requests to the appropriate shard.

📌 2. Why is Sharding Required?

- **Scalability**:
  - Sharding allows MongoDB to handle datasets that are too large for a single server.
- **Performance**:
  - Distributes read and write operations across multiple servers, improving throughput.
- **Storage**:
  - Enables storage of large datasets by splitting them across multiple machines.
- **Use Cases**:
  - Large-scale applications (e.g., social networks, e-commerce platforms).
  - High-throughput systems (e.g., real-time analytics, IoT data processing).

📌 3. Shard Keys & Best Practices

Shard Key

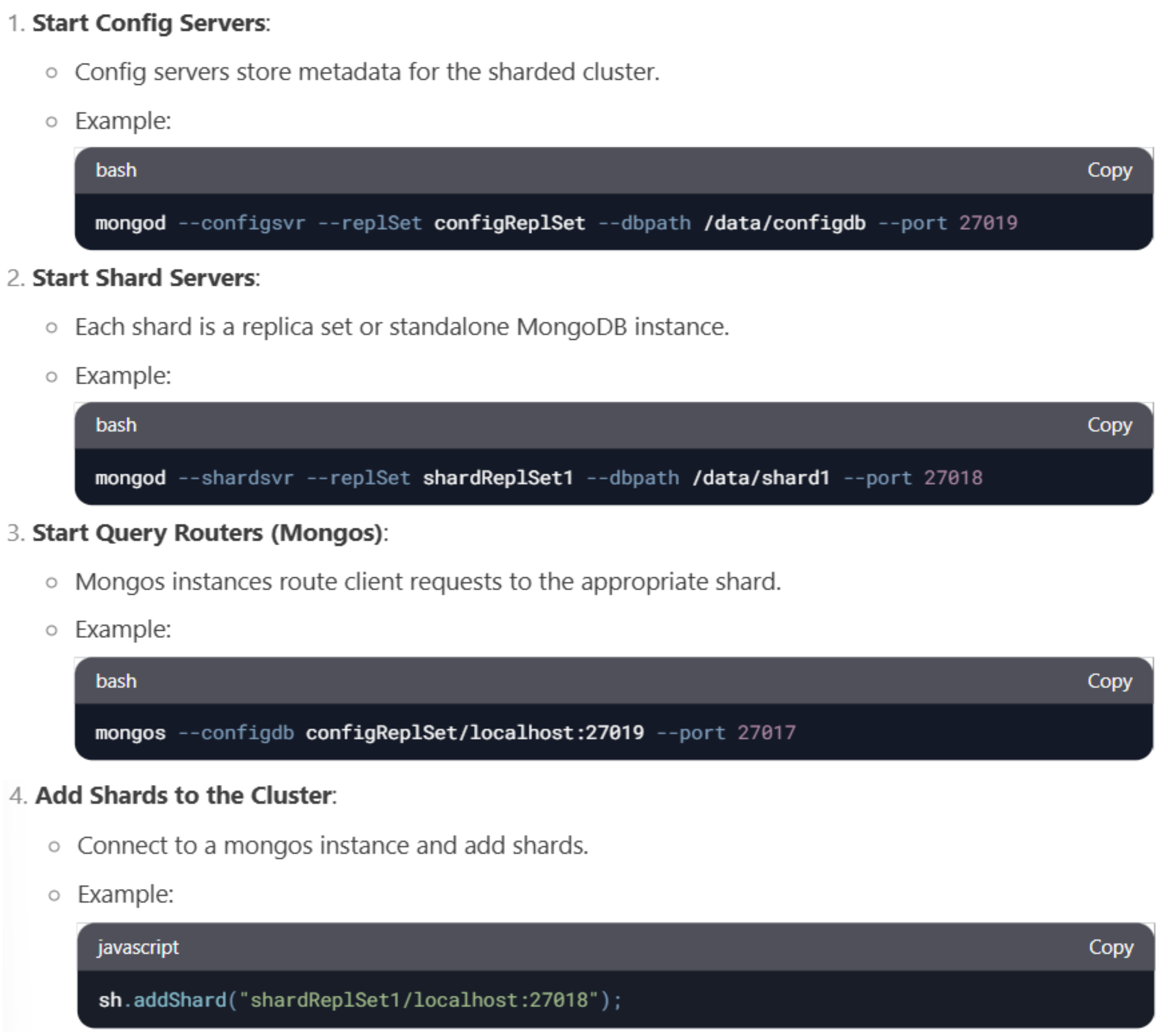
- A shard key is a field or set of fields used to distribute data across shards.
- MongoDB uses the shard key to partition data into **chunks**, which are then distributed across shards.

Choosing a Shard Key

- **Cardinality**:
  - Choose a shard key with high cardinality (many unique values) to ensure even distribution.
- **Write Distribution**:
  - Avoid shard keys that cause all writes to go to a single shard (e.g., monotonically increasing keys like timestamps).
- **Query Isolation**:
  - Choose a shard key that aligns with common query patterns to minimize cross-shard queries.

📌 4. Configuring Sharded Clusters

Steps to Set Up a Sharded Cluster



1. **Start Config Servers**:
   - Config servers store metadata for the sharded cluster.
   - Example:

```bash
mongod --configsvr --replSet configReplSet --dbpath /data/configdb --port 27019
```

2. **Start Shard Servers**:
   - Each shard is a replica set or standalone MongoDB instance.
   - Example:

```bash
mongod --shardsvr --replSet shardReplSet1 --dbpath /data/shard1 --port 27018
```

3. **Start Query Routers (Mongos)**:
   - Mongos instances route client requests to the appropriate shard.
   - Example:

```bash
mongos --configdb configReplSet/localhost:27019 --port 27017
```

4. **Add Shards to the Cluster**:
   - Connect to a mongos instance and add shards.
   - Example:

```javascript
sh.addShard("shardReplSet1/localhost:27018");
```

5. **Enable Sharding for a Database**:
   - Enable sharding for a specific database.
   - Example:

     ```javascript
     sh.enableSharding("myDatabase");
     ```
6. **Shard a Collection**:
   - Choose a shard key and shard the collection.
   - Example:

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

📌 5. Example: Sharding a Collection

1. **Enable Sharding**:

   ```javascript
   sh.enableSharding("ecommerce");
   ```
2. **Shard the Orders Collection**:

   ```javascript
   sh.shardCollection("ecommerce.orders", { customerId: 1 });
   ```
3. **Verify Sharding**:

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