MongoDB Cheatsheet

Make a detailed cheatsheet on MongoDB with examples

MongoDB Cheat Sheet

MongoDB is a NoSQL database that stores data in flexible, JSON-like documents. It is highly scalable, available, and supports a rich query language.

---

1. Key Features of MongoDB

- **Document-Oriented**: Stores data as BSON (Binary JSON) documents.
- **Schema-less**: Flexible document structure.
- **Horizontal Scaling**: Uses sharding for large datasets.
- **Rich Query Language**: Supports CRUD operations, indexing, aggregation, and geospatial queries.
- **Cross-Platform**: Available for Windows, macOS, and Linux.

---

2. Basic MongoDB Commands

Connecting to MongoDB

```bash
# Start MongoDB server
mongod
# Start MongoDB client
mongo
```

Create/Use Database

```javascript
use myDatabase
```

Show Databases

```javascript
show dbs
```

Show Collections

```javascript
show collections
```

---

3. CRUD Operations

Insert Data

```javascript
// Insert one document
db.users.insertOne({ name: "John", age: 30, city: "New York" });

// Insert multiple documents
db.users.insertMany([
  { name: "Alice", age: 25, city: "Los Angeles" },
  { name: "Bob", age: 35, city: "Chicago" }
]);
```

Read Data

```javascript
// Find all documents
db.users.find();

// Find with a filter
db.users.find({ age: { $gt: 30 } });

// Find specific fields
db.users.find({ city: "New York" }, { name: 1, _id: 0 });
```

Update Data

```javascript
// Update one document
db.users.updateOne(
  { name: "John" },
  { $set: { age: 31 } }
);

// Update multiple documents
db.users.updateMany(
  { city: "Los Angeles" },
  { $set: { state: "California" } }
);
```

Delete Data

```javascript
// Delete one document
db.users.deleteOne({ name: "Alice" });

// Delete multiple documents
db.users.deleteMany({ city: "Chicago" });
```

---

4. Indexing

Create Index

```javascript
db.users.createIndex({ name: 1 }); // Ascending order
```

View Indexes

```javascript
db.users.getIndexes();
```

Delete Index

```javascript
db.users.dropIndex("name_1");
```

---

5. Aggregation Framework

Basic Aggregation

```javascript
db.sales.aggregate([
  { $match: { status: "A" } },           // Filter
  { $group: { _id: "$item", total: { $sum: "$amount" } } }, // Grouping
  { $sort: { total: -1 } }              // Sorting
]);
```

---

6. Advanced Queries

Regular Expressions

```javascript
db.users.find({ name: { $regex: "^A", $options: "i" } }); // Names starting with "A" (case-insensitive)
```

Array Queries

```javascript
db.users.find({ hobbies: { $in: ["reading"] } }); // Match any value in an array
```

Range Queries

```javascript
db.users.find({ age: { $gte: 25, $lte: 35 } }); // Age between 25 and 35
```

---

7. Transactions

Start a Transaction

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

try {
  session.getDatabase("myDatabase").users.insertOne({ name: "Tom", age: 40 });
  session.getDatabase("myDatabase").orders.insertOne({ orderId: 123, user: "Tom" });

  session.commitTransaction();
} catch (error) {
  session.abortTransaction();
  print("Transaction Aborted: ", error);
}

session.endSession();
```

---

8. Backup and Restore

Backup

```bash
mongodump --db myDatabase --out /path/to/backup
```

Restore

```bash
mongorestore /path/to/backup
```

---

9. Schema Design Tips

- **Embed** when related data is queried together frequently.
- **Reference** when related data is large or reused in multiple collections.
- Use indexes wisely to optimize query performance.

---

10. Example: Building a Blog System

Schema Design

Users Collection:

```json
{
  "_id": ObjectId("..."),
  "name": "John Doe",
  "email": "john@example.com",
  "password": "hashed_password"
}
```

Posts Collection:

```json
{
  "_id": ObjectId("..."),
  "title": "My First Blog",
  "content": "This is my first blog post.",
  "author": ObjectId("..."), // Reference to Users
  "tags": ["intro", "welcome"],
  "createdAt": ISODate("2024-01-01T10:00:00Z")
}
```

Comments Collection:

```json
{
  "_id": ObjectId("..."),
  "postId": ObjectId("..."), // Reference to Posts
  "author": "Jane Doe",
  "content": "Great post!",
  "createdAt": ISODate("2024-01-02T12:00:00Z")
}
```

---

11. Tools and Utilities

- **Compass**: GUI for MongoDB.
- **Mongoose**: ODM library for MongoDB in Node.js.
- **Robo 3T**: Lightweight GUI for managing MongoDB.

MongoDB is versatile and powerful, but proper schema design and indexing are crucial for optimal performance.

compare sql andmongo db. like table vs document which is equivalent to which and show structure

Here's a comparison of SQL (Relational Database) and MongoDB (NoSQL Document Database) concepts, focusing on their equivalents and structure.

Comparison of SQL vs. MongoDB

SQL	MongoDB	Description
Database	**Database**	A container for tables in SQL and collections in MongoDB.
Table	**Collection**	A table in SQL is equivalent to a collection in MongoDB.
Row	**Document**	A row in SQL is equivalent to a document in MongoDB, represented as a JSON-like structure.
Column	**Field**	A column in SQL is equivalent to a field (key) in MongoDB.
Primary Key	**_id Field**	Each document in MongoDB has a unique `` `_id` `` field, similar to a primary key in SQL.
Joins	**Embedding/Referencing**	MongoDB uses embedding or referencing instead of joins to associate data between collections.
Schema	**Dynamic Schema**	SQL has a fixed schema for tables; MongoDB has a flexible schema where documents in a collection can differ.

Structure Example

SQL Table Structure

Table: Users

id	name	age	city
1	John Doe	30	New York
2	Jane Smith	25	Los Angeles

Table: Orders

order_id	user_id	product	price
101	1	Laptop	1200
102	2	Phone	800

MongoDB Collection Structure

Collection: Users

```json
[
  { "_id": 1, "name": "John Doe", "age": 30, "city": "New York" },
  { "_id": 2, "name": "Jane Smith", "age": 25, "city": "Los Angeles" }
]
```

Collection: Orders

```json
[
  { "_id": 101, "user_id": 1, "product": "Laptop", "price": 1200 },
  { "_id": 102, "user_id": 2, "product": "Phone", "price": 800 }
]
```

Embedding vs Referencing

SQL Joins

To get the user and their orders:

```sql
SELECT u.name, o.product, o.price
FROM Users u
JOIN Orders o ON u.id = o.user_id;
```

MongoDB Equivalent

Embedding (Single Query)

```json
[
  {
    "_id": 1,
    "name": "John Doe",
    "age": 30,
    "city": "New York",
    "orders": [
      { "product": "Laptop", "price": 1200 }
    ]
  },
  {
    "_id": 2,
    "name": "Jane Smith",
    "age": 25,
    "city": "Los Angeles",
    "orders": [
      { "product": "Phone", "price": 800 }
    ]
  }
]
```

Referencing (Separate Collections)

- Users:

  ```json
  { "_id": 1, "name": "John Doe", "age": 30, "city": "New York" }
  ```

- Orders:

  ```json
  { "_id": 101, "user_id": 1, "product": "Laptop", "price": 1200 }
  ```

Query for Referencing

```javascript
db.users.aggregate([
  { $lookup: {
      from: "orders",
      localField: "_id",
      foreignField: "user_id",
      as: "orders"
  }}
]);
```

Key Takeaways

1. **SQL is structured and schema-based**, ideal for applications needing strict relationships and ACID compliance.
2. **MongoDB is flexible and schema-less**, suitable for hierarchical or rapidly changing data structures.
3. **Joins in SQL** are replaced by **embedding or `$lookup`** in MongoDB.

give example of how document and collection ook like

Example of a MongoDB Document and Collection

---

1. Document

A **document** in MongoDB is a JSON-like structure that stores data in key-value pairs. Each document is similar to a single row in a relational database table.

Example Document (Single Record):

```
json

{
  "_id": 1,
  "name": "John Doe",
  "age": 30,
  "city": "New York",
  "hobbies": ["reading", "traveling", "coding"]
}
```

---

2. Collection

A **collection** is a group of documents. It is equivalent to a table in a relational database, but unlike a table, it does not enforce a strict schema. Documents within a collection can have different structures.

Example Collection (Multiple Records):

```
json

[
  {
    "_id": 1,
    "name": "John Doe",
    "age": 30,
    "city": "New York",
    "hobbies": ["reading", "traveling", "coding"]
  },
  {
    "_id": 2,
    "name": "Jane Smith",
    "age": 25,
    "city": "Los Angeles",
    "hobbies": ["painting", "music"]
  },
  {
    "_id": 3,
    "name": "Alice Johnson",
    "age": 28,
    "city": "Chicago",
    "skills": {
      "primary": "Data Analysis",
      "secondary": "Machine Learning"
    }
    }
  ]
```

Differences in Structure

- Each document in the collection has a unique `` `_id` `` field (like a primary key in SQL).
- Documents in the same collection can have different fields (`` `hobbies` `` in one document vs. `` `skills` `` in another).

Storage in MongoDB Shell

When you store the above documents in a collection named `` `users` ``:

```
javascript


  db.users.insertMany([
    { "_id": 1, "name": "John Doe", "age": 30, "city": "New York", "hobbies": ["reading", "tra
  veling", "coding"] },
    { "_id": 2, "name": "Jane Smith", "age": 25, "city": "Los Angeles", "hobbies": ["paintin
  g", "music"] },
    { "_id": 3, "name": "Alice Johnson", "age": 28, "city": "Chicago", "skills": { "primary":
  "Data Analysis", "secondary": "Machine Learning" } }
  ]);
```

Querying the Collection

Retrieve all documents from the `` `users` `` collection:

```
javascript


  db.users.find();
```

Output:

```
json


  { "_id": 1, "name": "John Doe", "age": 30, "city": "New York", "hobbies": ["reading", "trave
  ling", "coding"] }
  { "_id": 2, "name": "Jane Smith", "age": 25, "city": "Los Angeles", "hobbies": ["painting",
  "music"] }
  { "_id": 3, "name": "Alice Johnson", "age": 28, "city": "Chicago", "skills": { "primary": "D
  ata Analysis", "secondary": "Machine Learning" } }
```

This flexibility makes MongoDB suitable for dynamic or hierarchical data structures.

similarly explain some important topics that you missed after that exaplain some advanced topics with example.

Here's a continuation of important MongoDB topics, followed by explanations of advanced concepts with examples.

Basic Topics You Need to Know

1. Schema Design

- MongoDB allows a **schema-less design**.
- You can embed related data in a single document or reference other documents in different collections.

Example: Embedding vs. Referencing

- **Embedding** (One-to-Many Relationship in a Single Document):

  ```json
  {
    "_id": 1,
    "name": "John Doe",
    "orders": [
      { "order_id": 101, "product": "Laptop", "price": 1200 },
      { "order_id": 102, "product": "Phone", "price": 800 }
    ]
  }
  ```

- **Referencing** (Separate Collections):
  - `users` Collection:

    ```json
    { "_id": 1, "name": "John Doe" }
    ```

  - `orders` Collection:

    ```json
    { "_id": 101, "user_id": 1, "product": "Laptop", "price": 1200 }
    ```

2. CRUD Operations

Create

```javascript
db.users.insertOne({ name: "John Doe", age: 30, city: "New York" });
```

Read

```javascript
db.users.find({ age: { $gt: 25 } }); // Fetch users older than 25
```

Update

```javascript
db.users.updateOne({ name: "John Doe" }, { $set: { city: "Los Angeles" } });
```

Delete

```javascript
db.users.deleteOne({ name: "John Doe" });
```

---

3. Indexing

Indexes improve query performance by creating a faster lookup mechanism.

Example: Create an Index

```javascript
db.users.createIndex({ name: 1 }); // 1 for ascending, -1 for descending
```

Query Using Index

```javascript
db.users.find({ name: "John Doe" });
```

---

4. Aggregation Framework

Used for data transformation and analysis.

Example: Aggregation Pipeline

```
javascript

db.orders.aggregate([
  { $match: { price: { $gt: 500 } } },
  { $group: { _id: "$user_id", totalSpent: { $sum: "$price" } } }
]);
```

Explanation:

1. `` `$match` ``: Filters orders with price > 500.
2. `` `$group` ``: Groups by `` `user_id` `` and calculates total spent.

Advanced Topics

1. Transactions

MongoDB supports multi-document ACID transactions since version 4.0. Transactions are used to ensure consistency.

Example: Multi-Document Transaction

```
javascript

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

try {
  const users = session.getDatabase("test").users;
  const orders = session.getDatabase("test").orders;

  users.updateOne({ _id: 1 }, { $set: { balance: 500 } });
  orders.insertOne({ user_id: 1, product: "Phone", price: 500 });

  session.commitTransaction();
} catch (error) {
  session.abortTransaction();
} finally {
  session.endSession();
}
```

---

2. Sharding

Sharding distributes data across multiple servers for horizontal scaling.

Key Concepts:

- **Shard Key**: Determines how data is distributed.
- **Chunks**: Data partitions distributed among shards.
- **Config Servers**: Manage metadata and configuration.

Example: Enable Sharding

```javascript
sh.enableSharding("testDB"); // Enable sharding on the database
db.adminCommand({ shardCollection: "testDB.users", key: { _id: "hashed" } });
```

---

3. Replica Sets

A **replica set** ensures high availability by maintaining copies of data on multiple servers.

Key Components:

- **Primary**: Handles all write operations.
- **Secondaries**: Replicate data from the primary.
- **Arbiter**: Helps in election of primary (no data stored).

Example: Configuring a Replica Set

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

---

4. Full-Text Search

MongoDB provides a full-text search to find text within documents.

Example: Create Text Index

```javascript
db.articles.createIndex({ content: "text" });
```

Example: Search Query

```javascript
db.articles.find({ $text: { $search: "MongoDB indexing" } });
```

---

5. Change Streams

Allows real-time tracking of changes in a collection, database, or deployment.

Example: Listen to Changes

```javascript
const changeStream = db.users.watch();
changeStream.on("change", (change) => {
  console.log(change);
});
```

---

6. Data Modeling

- **One-to-One**: Use embedding.
- **One-to-Many**: Use embedding or referencing based on query frequency.
- **Many-to-Many**: Use referencing with intermediate collections.

Example: Many-to-Many with Intermediate Collection

- `students` Collection:

  ```json
  { "_id": 1, "name": "Alice" }
  ```

- `courses` Collection:

  ```json
  { "_id": 101, "title": "Math 101" }
  ```

- `enrollments` Collection:

  ```json
  { "student_id": 1, "course_id": 101, "grade": "A" }
  ```

---

7. Performance Tuning

- **Optimize Indexes**: Create compound indexes for multi-field queries.
- **Query Profiling**: Use the profiler to analyze slow queries.

  ```javascript
  db.setProfilingLevel(2); // Enable profiling
  ```

- **Limit Projections**: Return only needed fields.

```javascript
db.users.find({}, { name: 1, age: 1 });
```

This cheatsheet combines beginner-friendly concepts with advanced techniques, providing a solid understanding of MongoDB for both basic operations and scaling in production environments.

explain advanced topic like sharding and others in detail

Here's a detailed explanation of some **advanced MongoDB topics**, including **Sharding**, **Replica Sets**, **Transactions**, and **Change Streams**, with examples.

---

1. Sharding

Definition

Sharding is a technique for distributing data across multiple servers (shards). It enables **horizontal scaling** for large-scale applications by partitioning the data into smaller, more manageable chunks.

Key Components

- **Shard Key**: A field that determines how data is distributed.
- **Chunks**: Data partitions within each shard.
- **Config Servers**: Store metadata and configuration for the sharded cluster.
- **Query Router (Mongos)**: Routes queries to the appropriate shard(s).

Example of Sharding

1. **Enable Sharding on a Database**

```javascript
sh.enableSharding("testDB");
```

2. **Shard a Collection** Use a shard key (`` `hashed` `` or `` `range` ``):

```javascript
db.adminCommand({
    shardCollection: "testDB.users",
    key: { userId: "hashed" }
});
```

3. **Insert Data** MongoDB automatically distributes the documents across shards based on the shard key.

```javascript
db.users.insertMany([
    { userId: 1, name: "Alice" },
    { userId: 2, name: "Bob" },
    { userId: 3, name: "Charlie" }
]);
```

Sharding Benefits

- **Scalability**: Data is split across multiple servers.
- **High Availability**: With replication in shards, downtime is minimized.
- **Balanced Workloads**: MongoDB automatically balances data across shards.

---

2. Replica Sets

Definition

Replica sets are a group of MongoDB servers that maintain the same dataset. They provide **high availability** and **fault tolerance**.

Key Components

- **Primary**: Handles all write and read operations (default).
- **Secondaries**: Copies data from the primary. Can serve read operations in specific configurations.
- **Arbiter**: Helps elect a new primary in case of failure but does not store data.

Example of a Replica Set

1. **Initialize a Replica Set**

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

2. **Check Status**

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

Failover Example

If the primary goes down, one of the secondaries is elected as the new primary automatically.

Benefits

- **Data Redundancy**: Protects against data loss.
- **Automatic Failover**: Ensures availability during server failure.
- **Scalable Reads**: Secondaries can handle read requests (if configured).

---

3. Transactions

Definition

Transactions in MongoDB allow you to perform multi-document operations with **ACID properties**. This ensures data consistency, even in complex operations.

Example of Transactions

1. **Start a Transaction**

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

2. **Perform Operations**

```javascript
const users = session.getDatabase("test").users;
const orders = session.getDatabase("test").orders;

users.updateOne({ _id: 1 }, { $inc: { balance: -500 } });
orders.insertOne({ user_id: 1, product: "Laptop", price: 500 });
```

3. **Commit or Abort**

```javascript
try {
    session.commitTransaction();
} catch (e) {
    session.abortTransaction();
} finally {
    session.endSession();
}
```

Use Cases

- Financial transactions (e.g., transferring money between accounts).
- Ensuring consistency in multi-document updates.

---

4. Change Streams

Definition

Change Streams allow you to track real-time changes (insert, update, delete) in a MongoDB collection, database, or deployment.

Example of Change Streams

1. **Watch Changes**

```
javascript


  const changeStream = db.users.watch();
  changeStream.on("change", (change) => {
      console.log(change);
  });
```

2. **Example Change Event** When a new document is inserted:

```
json


  {
      "operationType": "insert",
      "fullDocument": { "_id": 1, "name": "Alice" },
      "ns": { "db": "test", "coll": "users" },
      "documentKey": { "_id": 1 }
  }
```

Use Cases

- Real-time analytics.
- Event-driven applications (e.g., triggering notifications).

---

5. Aggregation Framework

The Aggregation Framework allows advanced data transformations, analysis, and summarization.

Example: Aggregation Pipeline

1. **Match and Group**

   ```javascript
   db.orders.aggregate([
       { $match: { status: "delivered" } },
       { $group: { _id: "$user_id", totalSpent: { $sum: "$price" } } }
   ]);
   ```

2. **Explanation**
    - **`$match`**: Filters documents with **`status`** as **`delivered`**.
    - **`$group`**: Groups orders by **`user_id`** and calculates total spending.

---

6. Full-Text Search

MongoDB supports text search on fields with indexed data.

Example

1. **Create a Text Index**

   ```javascript
   db.articles.createIndex({ content: "text" });
   ```

2. **Search for a Phrase**

   ```javascript
   db.articles.find({ $text: { $search: "MongoDB tutorial" } });
   ```

---

7. MongoDB Performance Optimization

- **Indexes**: Speed up read queries.
- **Shard Keys**: Choose efficient keys for balanced data distribution.
- **Query Profiling**: Identify and optimize slow queries.

  ```javascript
  db.setProfilingLevel(2);
  ```

These advanced topics help MongoDB scale, maintain data consistency, and support modern, real-time applications while ensuring high performance. Let me know if you need a deeper dive into any topic!