**1)What is the MongoDb?**

MongoDB is a popular, open-source, NoSQL database that stores data in JSON-like documents called BSON (Binary JSON) instead of traditional tables. This flexible, document-oriented approach allows for storing both structured, semi-structured, and unstructured data. MongoDB is widely used in web and mobile applications for its simplicity, ease of use, and scalability.

**2) MongoDB is schema less ?**

MongoDB is **schema-less** by default, meaning it doesn't require a predefined schema for the data stored in collections. This is one of the key features that sets it apart from traditional relational databases.

However, MongoDB does support **schema definitions** through tools like **Mongoose** (for Node.js), which allows you to define schemas to enforce a certain structure on your documents (data).

**3) What is the Mongoose?**

Mongoose is a powerful tool that acts as an **Object Document Mapper (ODM)** for MongoDB in Node.js. It provides several key features to help you work with MongoDB more effectively, including **schema definitions**, **validation**, and **database connectivity**.

It creates the schema and enforce the validation and types and db connectivity.

4)**Difference between MongoDB and MySQL/PostgreSQL?**

|  |  |  |
| --- | --- | --- |
| **Feature** | **MongoDB** | **MySQL/PostgreSQL** |
| Data Model | Document-based | Table-based (Relational) |
| Schema | Dynamic | Fixed/Defined Schema |
| Query Language | JSON-like queries | SQL |
| Joins | Manual (lookup) | Native joins |

5) **What is indexing in MongoDB?**

Indexing improves the performance of queries. Without indexes, MongoDB must perform a collection scan. Example:

6) **What is sharding?**

Sharding is horizontal scaling where data is distributed across multiple servers (shards). It's used for high-volume databases.

**7) Replication in MongoDB**

**Replication** in MongoDB is the process of **duplicating** data across multiple servers to ensure **high availability**, **data redundancy**, and **fault tolerance**.

* **Replica Set**: A **replica set** is a group of MongoDB servers that maintain the same data. It consists of one **primary** node and multiple **secondary** nodes.
* **Primary Node**: The **primary node** handles all **write operations** (inserts, updates). It is the main server where data is modified.
* **Secondary Nodes**: **Secondary nodes** replicate the data from the primary. They only handle **read operations** by default, though they can be configured to handle writes in special cases (e.g., during failover).
* **Replication Process**: When a write operation happens on the **primary**, it is recorded in an **operation log** (oplog) and replicated to the **secondary nodes**. The secondary nodes apply these changes in the same order.
* **Automatic Failover**: If the **primary** node goes down, one of the **secondary nodes** is automatically **elected** to become the new **primary**. This ensures that the database remains available.

**Why is Replication Important?**

* **High Availability**: Ensures the database remains available even if one or more nodes fail.
* **Data Redundancy**: Prevents data loss as copies are maintained across multiple nodes.
* **Scalability**: You can direct read requests to secondary nodes, improving read scalability.

**8) What is ACID ? How it is archieved in MongoDB?**

**ACID** stands for four key properties that are essential for reliable database transactions:

1. **Atomicity**:
   * A transaction is treated as a single, indivisible unit. Either all operations within the transaction are completed successfully, or none are. If one operation fails, the entire transaction is rolled back, and no changes are applied to the database.
2. **Consistency**:
   * A transaction must transition the database from one valid state to another. The data must be consistent both before and after the transaction. Any rules or constraints (such as data types, validations) must not be violated during the transaction.
3. **Isolation**:
   * Each transaction is isolated from other transactions. Even if multiple transactions are executed concurrently, each one should appear to execute independently without interfering with others. Uncommitted changes in one transaction should not be visible to others.
4. **Durability**:
   * Once a transaction is committed, its changes are permanent. The database guarantees that even in the event of a system crash or failure, the committed data will not be lost.

**9) MongoDB Transactions ?**

**How to Use Transactions in MongoDB**

MongoDB provides **multi-document transactions** starting from **version 4.0**, which allow you to perform multiple operations on different documents and collections while ensuring **ACID** (Atomicity, Consistency, Isolation, Durability) properties.

Here's a step-by-step guide on how to use **transactions in MongoDB**.

**Pre-requisites**

* **MongoDB 4.0+** (since multi-document transactions are supported from version 4.0 onwards).
* **Replica Set**: Transactions are only supported in a **replica set** configuration, not in standalone servers.

**Steps to Use Transactions**

1. **Create a Session**:
   * MongoDB transactions are bound to a **session**. You need to start a session before beginning a transaction.
2. **Start a Transaction**:
   * Once the session is created, you can start a transaction.
3. **Perform Operations**:
   * Perform the operations you want to include in the transaction (inserts, updates, deletes) within the transaction context.
4. **Commit the Transaction**:
   * If all operations in the transaction succeed, you call commitTransaction() to commit the changes.
5. **Abort the Transaction**:
   * If an error occurs during the transaction, you can use abortTransaction() to roll back all changes made during the transaction.
6. **End the Session**:
   * After the transaction is completed (whether committed or aborted), always call endSession() to clean up the session.

**Example of Using Transactions**

**Scenario:**

You have two collections: **users** and **orders**, and you want to:

1. Deduct money from the **users** balance.
2. Insert a new order in the **orders** collection.

Both actions should be part of a single transaction.

const { MongoClient } = require('mongodb');

async function runTransaction() {

const client = new MongoClient('mongodb://localhost:27017');

try {

await client.connect();

const session = client.startSession(); // Create a session

// Start a transaction

session.startTransaction();

const usersCollection = client.db('shop').collection('users');

const ordersCollection = client.db('shop').collection('orders');

// Operation 1: Deduct balance from user

await usersCollection.updateOne(

{ \_id: 1 },

{ $inc: { balance: -50 } },

{ session } // Specify the session for the operation

);

// Operation 2: Insert a new order

await ordersCollection.insertOne(

{ userId: 1, totalAmount: 50, status: 'pending' },

{ session } // Specify the session for the operation

);

// Commit the transaction if everything is successful

await session.commitTransaction();

console.log("Transaction committed successfully.");

} catch (error) {

// If an error occurs, abort the transaction

await session.abortTransaction();

console.error("Transaction aborted due to an error:", error);

} finally {

// End the session after the transaction is complete

session.endSession();

await client.close();

}

}

// Run the function

runTransaction().catch(console.error);

**Optimizing Database Performance:**

**10. How would you optimize database queries for better performance?**

* **Use Indexes**: Indexing frequently queried fields (like user\_id, product\_id) helps speed up lookups.
* **Optimize Queries**: Avoid SELECT \* and fetch only necessary columns.
* **Use Caching**: Cache frequently accessed data to reduce repeated database calls.
* **Query Limiting**: Use LIMIT or pagination for large result sets.
* **Denormalization**: In some cases, especially in NoSQL (MongoDB), denormalization (duplicating some data) may improve query performance, though it comes at the cost of data redundancy.

What is data modeling?

**11) What is the difference between a collection and a document in MongoDB?**

* **Collection**: A collection is a group of MongoDB documents. It’s similar to a table in relational databases but doesn't require a fixed schema.
* **Document**: A document is a single record in a collection. It's a set of key-value pairs, where the key is a string and the value can be a variety of data types (string, number, array, or even another document). It’s stored in BSON format (binary JSON).

**12) What is meant by 'denormalization' in MongoDB schema design?**

* **Denormalization** refers to the process of storing related data within a single document, rather than separating it into multiple collections (tables) and using references or joins. In MongoDB, this is often used to avoid the performance overhead of joins in relational databases, enabling faster reads.
* For example, you might store user details along with the user’s orders in the same document, rather than storing users in one collection and orders in another.

**13) When would you choose embedding documents over referencing them?**

* **Use embedding when**:
  + The embedded data is frequently accessed together with the parent document (e.g., a product with its reviews) or frequently read document
  + The embedded data doesn’t grow too large or require updates independently (e.g., a user’s address list).
  + It is not suitable if document have frequent udpates.
  + You want to optimize read performance by retrieving all the data in a single query.
* **Example**: Embedding order items within an order document if the items don’t need to be independently queried or updated.

**14) What is meant by ‘normalized’ data models? When should you use them?**

* **Normalized data model**: In a normalized model, data is split across multiple collections (tables) to reduce redundancy and improve consistency. Each collection stores a specific type of entity (e.g., a user collection, an orders collection).
* **When to use**:
  + When data integrity and consistency are critical.
  + When updates to data occur frequently, and denormalization could lead to anomalies (e.g., multiple updates to the same field across multiple documents).
  + When data doesn’t have to be retrieved in its entirety all at once (e.g., retrieving orders separately from customers).

**15)** **What is a reference in MongoDB, and how do you implement it?**

* **Reference**: A reference in MongoDB is a way to link documents across collections by storing the \_id of one document in another document, similar to a foreign key in relational databases.
* **Implementation**: You store the \_id of the referenced document in the referencing document. Then, when needed, you can perform a join-like operation using $lookup in an aggregation pipeline.
  + **Example**: In a posts collection, you can store a user\_id reference to the user document in a users collection.

**16)** **What are embedded documents, and when should you use them in MongoDB?**

* **Embedded documents** are documents within other documents in MongoDB. They allow you to store related data inside the parent document, which is useful for representing hierarchical data.
* **Use embedded documents**:
  + When the data is inherently part of the parent document (e.g., a list of items in an order).
  + When you want to improve read performance by reducing the need for multiple queries or joins.
  + When the embedded data is small, doesn’t require independent queries, and doesn't change frequently.

**17) What are some advantages and disadvantages of using embedded documents?**

* **Advantages**:
  + **Faster reads**: Since all related data is stored together, fetching a document with embedded data in a single query is faster than performing multiple joins.
  + **Simpler design**: Embedding can simplify the data model by keeping related data together.
  + **Reduced joins**: Embedded documents avoid the need for joins, which can improve performance in many use cases.
* **Disadvantages**:
  + **Data duplication**: If the same embedded document is used in multiple parent documents, it can lead to data duplication.
  + **Limited scalability**: Large embedded documents can lead to performance issues since MongoDB has a maximum document size limit (16MB).
  + **Difficult to update**: If embedded data grows and needs to be updated independently, managing changes can be more difficult.

**18) How do you design a schema for a blogging platform with users and posts?**

**Indexes:**

**19) What is an index in MongoDB, and why is it important?**

Index in MongoDB is a data structure that improves the speed of data retrieval operations on a collection, like searching and sorting, at the cost of additional space and time spent maintaining the index during write operations.

Importance:

* Faster Queries: Without indexes, MongoDB performs a full collection scan, which is inefficient, especially with large datasets. Indexes allow MongoDB to quickly locate the data it needs.
* Improves Read Performance: Indexes can significantly speed up queries, especially when searching for documents that match specific fields (e.g., find queries, sorting, etc.).
* Optimizes Sorting: Sorting queries are faster if the field being sorted is indexed.
* Reduces Query Time: Particularly for fields that are frequently queried, having an index reduces the time taken to fetch results.
  1. **How do you create an index in MongoDB?**
* You can create an index using the createIndex() method.
* Example to create an index on a field (name):
* javascript
* Copy
* db.collection.createIndex({ name: 1 }) // 1 for ascending order, -1 for descendingx
* To create an index on multiple fields (compound index):
* javascript
* Copy
* db.collection.createIndex({ name: 1, age: -1 }) // Compound index on name (ascending) and age (desce

**21) How does MongoDB handle atomic operations?**

* MongoDB supports atomic operations at the document level. This means that if you update a document, the entire update is performed as a single unit: either all changes are applied, or none are.
* For example, using the update() method on a single document is an atomic operation. If multiple fields are updated, MongoDB ensures that either all the fields are updated together, or none are.
* Atomicity in embedded documents: MongoDB ensures atomicity even when you update an embedded document within a larger document.

**22) What is a multi-document transaction in MongoDB? How does it work?**

Multi-document transactions in MongoDB allow you to execute multiple operations on multiple documents or collections in a way that guarantees atomicity, consistency, isolation, and durability (ACID properties).

How it works:

* You start a transaction with startSession().
* Perform multiple read and write operations within the transaction.
* Commit or abort the transaction based on the outcome of the operations.

**23) What is Projection in MongoDB?**

Projection in MongoDB refers to specifying which fields you want to include or exclude from the documents returned by a query. Projections allow you to limit the data you retrieve, making your queries more efficient and reducing the data sent over the network.

Projection in MongoDB is a powerful tool for optimizing **query performance** and **network usage**.

db.collection.find(query, projection)

24)**How do you use projection in the aggregation framework?**

In MongoDB aggregation, you use the $project stage to reshape the data and specify which fields should be included or excluded.

js

Copy

db.orders.aggregate([

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

{ $project: { orderId: 1, customer: 1, \_id: 0 } }

])

This pipeline filters orders with status: "shipped" and returns only the orderId and customer fields, excluding the \_id field.