Class -8 ACIDS AND INDEX

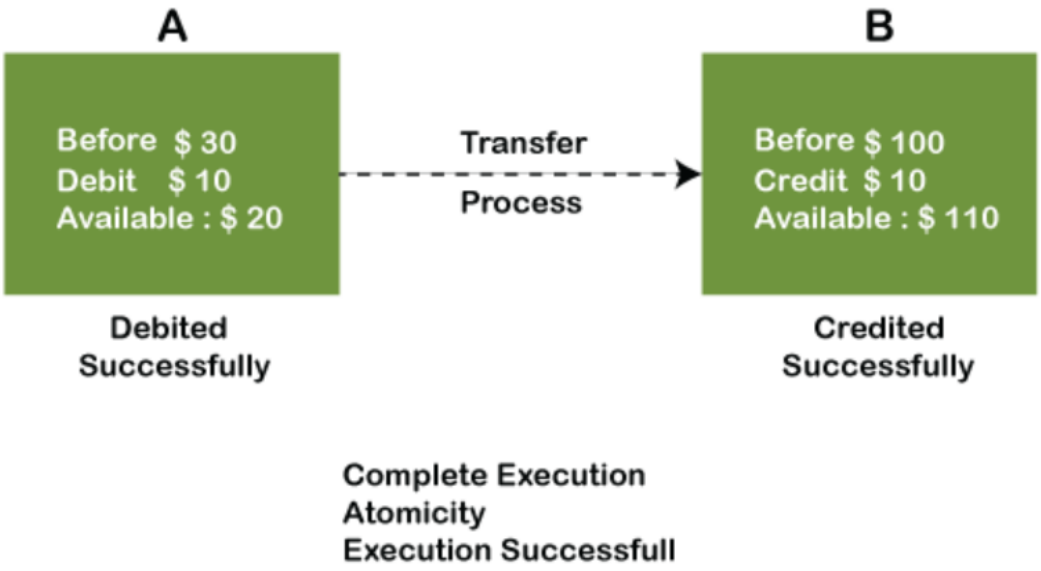
Acids and index:

In MongoDB, **ACID** and **index** are crucial concepts that ensure the reliability and efficiency of database operations. While ACID principles are essential for transaction management, indexes are used to improve query performance.

ACID in MongoDB

Atomicity

- In MongoDB, single-document operations (like inserts, updates, and deletes) are atomic. This means that these operations will complete fully or not at all.
- MongoDB also supports multi-document transactions, providing atomicity across multiple documents, collections, and even databases.



Consistency:

- MongoDB ensures consistency through its document validation rules and schema design. It maintains the integrity of data by enforcing constraints such as unique indexes.
- Transactions in MongoDB ensure that operations result in a consistent state, maintaining data integrity according to defined rules.

Isolation

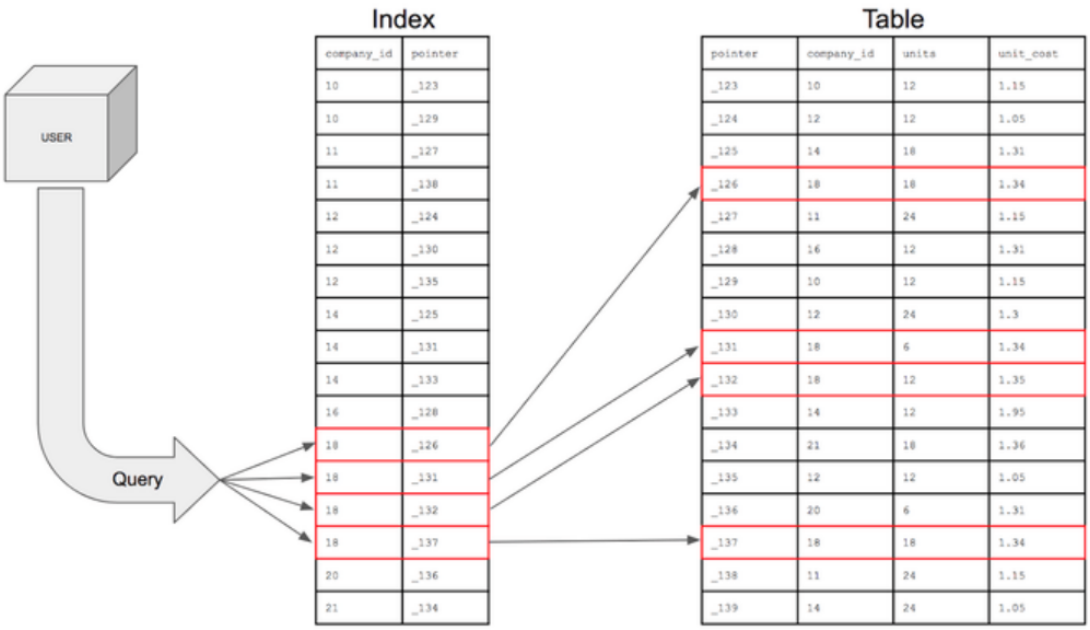
- MongoDB uses isolation to ensure that operations within a transaction do not interfere with each other. In a multi-document transaction, MongoDB provides snapshot isolation, which means that all read operations within the transaction see a consistent snapshot of the data as it was at the start of the transaction.
- Changes made in a transaction are not visible outside the transaction until it is committed.

Durability

- MongoDB ensures durability by writing data to the journal before committing a transaction. This ensures that even if there is a failure, the committed transactions will not be lost.

Indexes in MongoDB

Indexes in MongoDB are special data structures that store a small portion of the collection's data set in an easy-to-traverse form. They improve the efficiency of query operations but require additional space and maintenance.



Types of Indexes

1. **Single Field Index**: Created on a single field of a document.
    - Example: To create an index on the `name` field

```
db.collection.createIndex({ name: 1 });
```

2.Compound Index: Created on multiple fields to support queries that match on multiple fields.

- Example: To create an index on the `name` and `age` fields:

```
db.collection.createIndex({ name: 1, age: -1 });
```

3.Multikey Index: Created on array fields, allowing queries to search for array elements.

- Example: To create an index on an array field `tags`:

```
db.collection.createIndex({ tags: 1 });
```

4.Text Index: Supports text search queries on string content.

- Example: To create a text index on the `description` field:

```
db.collection.createIndex({ description: "text" });
```

5.Geospatial Indexes: Supports queries for geospatial data.

- **2dsphere Index**: For spherical (Earth-like) geometry queries.
  - Example: To create a 2dsphere index on the `location` field:

```
db.collection.createIndex({location:"2dsphere" });
```

- **2d Index**: For flat geometry queries.
  - Example: To create a 2d index on the `location` field:

```
db.collection.createIndex({ location: "2d" });
```

6.Wildcard Index: Indexes all fields in the document or a subset of fields.

- Example: To create a wildcard index:

```
db.collection.createIndex({ "$**": 1 });
```

7.Hashed Index: Indexes the hash of the value of a field and is used for equality searches.

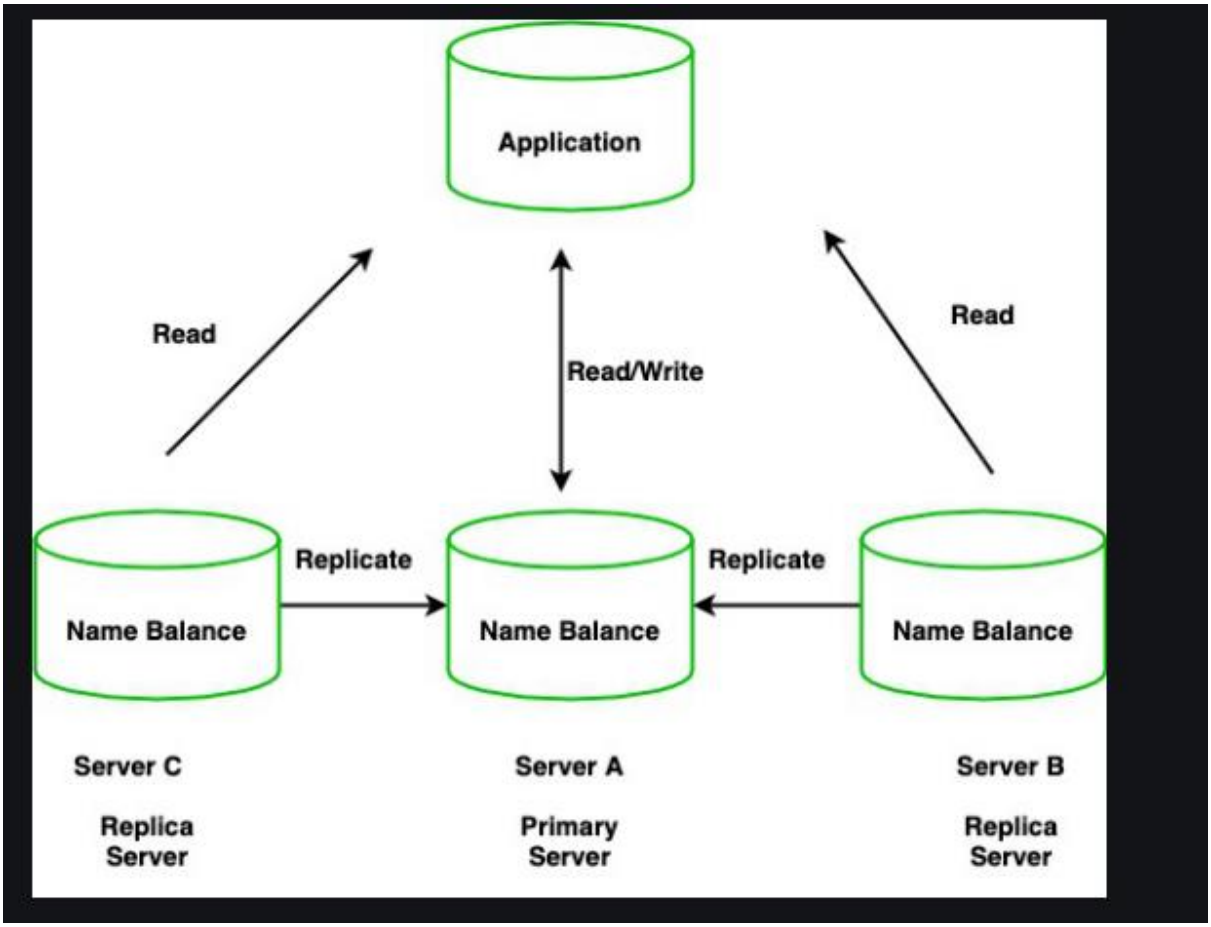
- Example: To create a hashed index on the `user_id` field:

```
db.collection.createIndex({ user_id: "hashed" });
```

Replication:

Replication in MongoDB is the process of synchronizing data across multiple servers. It provides redundancy and increases data availability by allowing the

system to continue functioning even if one or more servers fail. Replication is achieved using **replica sets**.



Benefits of Replication:

1. **High Availability**: Ensures that the database is available even if the primary node fails.
2. **Redundancy**: Maintains multiple copies of data to prevent data loss.
3. **Read Scaling**: Secondary nodes can be configured to handle read operations, thereby distributing the read load.

Example of Setting Up a Replica Set

1. **Start MongoDB Instances**:

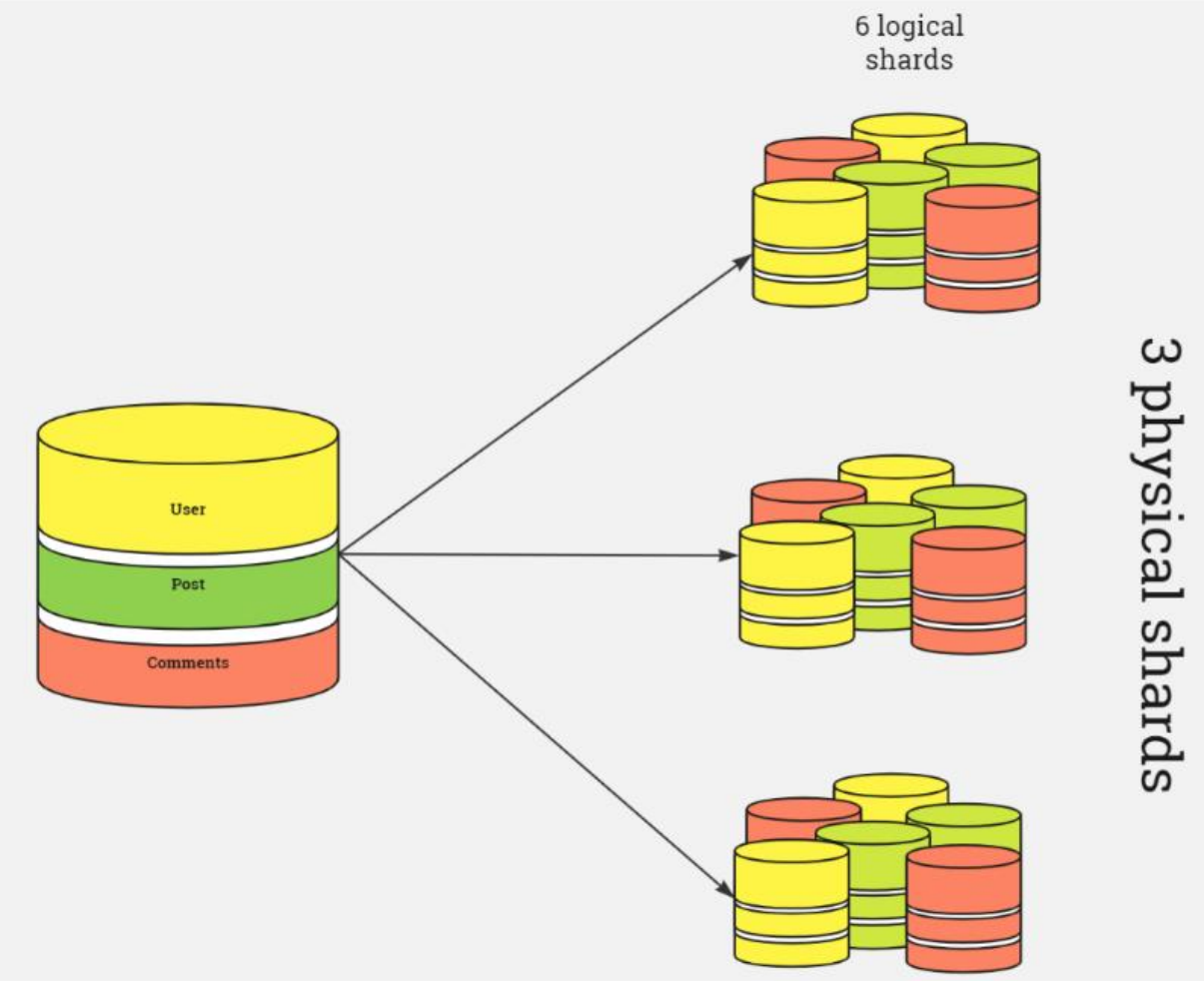
```
mongod --replSet "rs0" --port 27017 --dbpath /data/db1
mongod --replSet "rs0" --port 27018 --dbpath /data/db2
mongod --replSet "rs0" --port 27019 --dbpath /data/db3
```

2. **Initiate the Replica Set**:

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

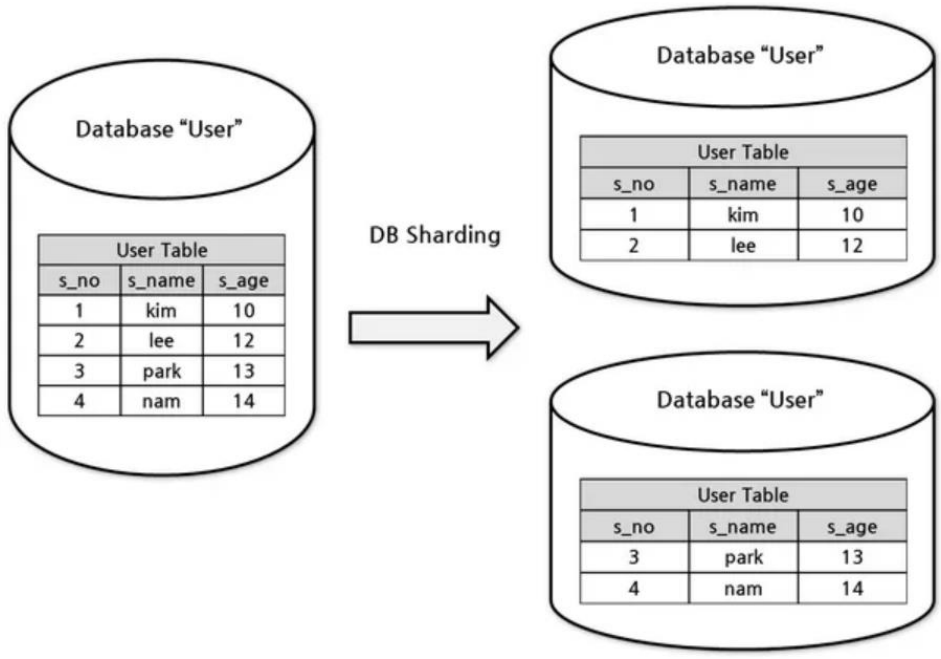
3. **Check the Status**:

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



Sharding:

Sharding in MongoDB is the process of distributing data across multiple servers or clusters. It allows MongoDB to handle large data sets and high-throughput operations by dividing the data into smaller, more manageable pieces called **shards**.



Components of Sharding:

1. **Shards**: These are the individual database instances that store the actual data. Each shard contains a subset of the sharded data.
2. **Config Servers**: These servers store the metadata and configuration settings for the cluster.
3. **Query Routers (mongos)**: These servers handle client requests, routing queries to the appropriate shards.

Benefits of Sharding:

1. **Horizontal Scalability**: Enables scaling out by adding more servers to handle increased load.
2. **Load Balancing**: Distributes data and operations across multiple servers, preventing any single server from becoming a bottleneck.

3. **Large Data Set Handling**: Allows MongoDB to manage very large datasets by partitioning the data.

Difference between Replication and sharding:

replication

Purpose

- **High Availability**: Ensures that the database is always available, even in the event of hardware failures.
- **Data Redundancy**: Keeps multiple copies of data to prevent data loss.

Key Concepts

- **Replica Set**: A group of MongoDB servers that maintain the same data set.
    - **Primary Node**: Receives all write operations.
    - **Secondary Nodes**: Replicate data from the primary and can serve read operations.
    - **Arbiter**: Participates in elections for primary but does not hold data.

Implementation

- **Data Replication**: All data is copied from the primary node to secondary nodes.
- **Automatic Failover**: If the primary node fails, an election is held to select a new primary from the secondary nodes.
- **Read Scaling**: Reads can be distributed across the primary and secondary nodes, improving read performance.

Use Cases

- **Disaster Recovery**: Protects against data loss by maintaining copies of data.
- **Read Scalability**: Distributes read operations across multiple nodes to balance the load.

Example

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

Sharding

Purpose

- **Horizontal Scalability**: Allows the database to scale out by distributing data across multiple servers or clusters.
- **Load Balancing**: Distributes read and write operations across multiple shards to prevent any single server from becoming a bottleneck.

Key Concepts

- **Shard**: An individual MongoDB instance that holds a subset of the data.
- **Config Server**: Stores metadata and configuration settings for the cluster.

- **Query Router (mongos)**: Routes client requests to the appropriate shard(s).

Implementation

- **Data Partitioning**: Data is split into smaller chunks and distributed across multiple shards based on the shard key.
- **Automatic Balancing**: MongoDB automatically moves data between shards to ensure an even distribution.
- **Scalability**: Both read and write operations can be distributed across multiple shards, improving overall performance.

Use Cases

- **Large Data Sets**: Manages very large datasets that exceed the capacity of a single server.
- **High Throughput**: Distributes read and write operations to handle high transaction volumes.

Example

```
sh.addShard("shardReplSet1/localhost:27022,localhost:27023,localhost:27024");
sh.enableSharding("myDatabase");
sh.shardCollection("myDatabase.myCollection", { shardKeyField: 1 });
```

Key Differences

Purpose

- **Replication**: Provides data redundancy and high availability.
- **Sharding**: Provides horizontal scalability and load balancing.

Data Distribution

- **Replication**: Copies the entire dataset to multiple nodes.
- **Sharding**: Distributes different portions of the dataset across multiple shards.

Scaling

- **Replication**: Improves read scalability by distributing read operations across primary and secondary nodes.
- **Sharding**: Improves both read and write scalability by distributing data and operations across multiple shards.

Failover

- **Replication**: Automatic failover with election of a new primary if the current primary fails.
- **Sharding**: Does not inherently provide failover; relies on replication within shards for redundancy and availability.

Complexity

- **Replication**: Simpler to set up and manage compared to sharding.
- **Sharding**: More complex to set up and manage due to the need for proper shard key selection, data distribution, and balancing.