

## ACID & Indexes

### What is Acid:

In MongoDB, ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure reliable transaction processing within the database. Here's a breakdown of each property in the context of MongoDB:

- **Atomicity:**

**Single-Document Atomicity:** MongoDB operations are atomic at the single-document level. This means that any update or insert operation involving a single document is guaranteed to be atomic. Starting from MongoDB 4.0, multi-document transactions are also supported, allowing multiple operations across multiple documents to be treated as a single atomic unit.

- **Consistency:**

**Enforcing Rules:** MongoDB ensures that the database remains in a consistent state before and after a transaction. All constraints, such as unique indexes, are maintained throughout the operation. Any transaction must bring the database from one valid state to another, preserving database invariants.

- **Isolation:**

**Transaction Isolation:** During the execution of a transaction, the operations are isolated from other operations. This ensures that intermediate states are not visible to other transactions. MongoDB uses snapshot isolation to provide this guarantee, meaning that a transaction operates on a snapshot of the database at a specific point in time.

- **Durability:**

**Commit Durability:** Once a transaction is committed, MongoDB guarantees that the changes will be permanently saved to disk. This means that even in the event of a crash or power failure, the committed data will not be lost. MongoDB ensures durability through journaling, which logs changes before they are written to the database files.

It's important to note that traditionally, MongoDB wasn't known for ACID compliance on a multi-document level. However, since MongoDB 4.0, it offers multi-document ACID transactions, allowing developers to perform operations across multiple documents atomically and consistently.

**Importance of acids:** ACID transactions ensure data remains consistent in a database. In data models where related data is split between multiple records or documents, multi-record or multi-document ACID transactions can be critical to an application's success. The importance of ACID properties. MongoDB is crucial for ensuring data reliability, consistency, and integrity, especially in applications that require robust transaction management.

## Acid vs Non-acids:

- MongoDB is not entirely non-ACID.
- MongoDB provides ACID guarantees at the document level.

## Key Differences:

- **ACID Transactions:**
  - Supported in MongoDB 4.0 and later.
  - Ensures all operations in a transaction are completed or none are.
  - Provides atomicity across multiple documents.
- **Non-ACID Operations:**
  - Operations are atomic only at the document level.
  - Multi-document operations may lead to inconsistencies if one operation fails and others succeed.
  - Commonly used prior to MongoDB 4.0 for handling individual document updates.

## Key Points to Remember:

- **Document-Level ACID:** MongoDB excels at operations involving a single document.
- **Multi-Document Transactions:** While supported, they might have performance implications and require careful consideration.
- **Data Modeling:** Designing your data model with related information within a single document can often eliminate the need for multi-document transactions.

## Indexes:

Indexes in MongoDB are special data structures that store a small portion of the data set in an easy-to-traverse form. They support efficient execution of queries by enabling the database to quickly locate and access the required data. Without indexes, MongoDB must perform a collection scan, i.e., scan every document in a collection, to select those documents that match the query statement.

## Types of Indexes in MongoDB

1. **Single Field Indexes:**
  - Created on a single field of a document.
  - Example: Creating an index on the name field.

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

## 2.Compound Indexes:

- Created on multiple fields of a document.

- Example: Creating an index on lastName and firstName fields.

```
db.collection.createIndex({ lastName: 1, firstName: 1 });
```

### 3.Multikey Indexes:

- Created on fields that contain arrays.
- Example: Creating an index on the tags array field.

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

### 4.Text Indexes:

- Created on string fields for text search.
- Example: Creating a text index on the description field.

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

### 5.Geospatial Indexes:

- Created on fields that store geographical data.
- Example: Creating a 2dsphere index for GeoJSON data.

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

### 6.Hashed Indexes:

- Created on fields for hashed sharding.
- Example: Creating a hashed index on the userId field.

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

### Benefits of Indexes

- **Improved Query Performance:**
  - Indexes speed up query processing by allowing MongoDB to quickly locate documents without scanning the entire collection.
- **Efficient Sorting:**
  - Indexes can help in sorting query results efficiently.

- **Unique Constraints:**
  - Unique indexes enforce uniqueness of the indexed fields, ensuring data integrity.

### Example Use Cases

#### 1. Single Field Index:

```
db.users.createIndex({ age: 1 });
```

#### 2. Compound Index:

```
db.users.createIndex({ lastName: 1, firstName: 1 });
```

#### 3. Text Index:

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

#### 4. Geospatial Index:

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

### Managing Indexes

- **Viewing Indexes:**

```
db.collection.getIndexes();
```

- **Dropping Indexes:**

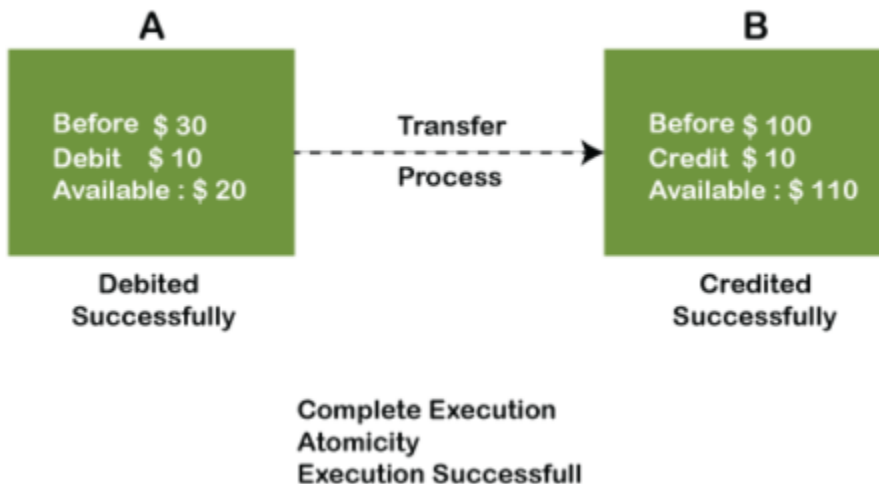
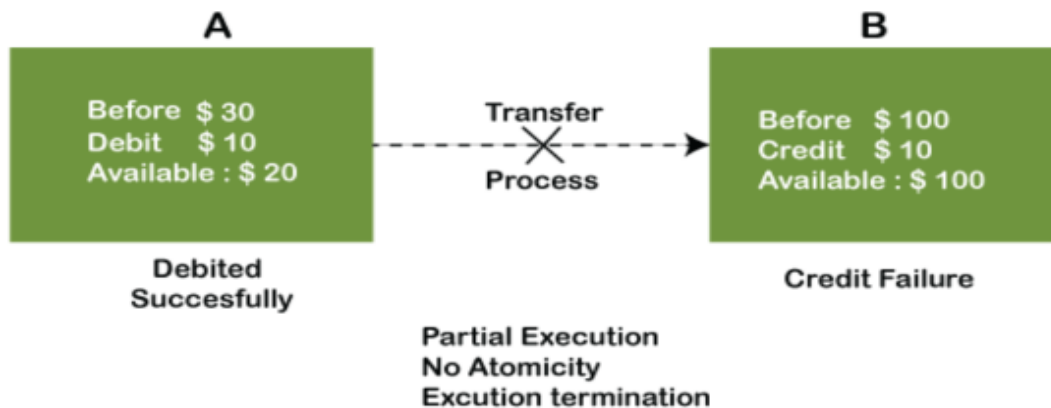
```
db.collection.dropIndex({ name: 1 });
```

Indexes are a powerful tool in MongoDB that help optimize query performance and ensure efficient data retrieval. However, they also consume additional disk space and can impact write performance, so it's essential to use them judiciously based on the specific needs of your application.

### Concepts under acids:

## Atomicity:

- **Definition:** Atomicity ensures that a series of operations within a transaction are all completed successfully or none of them are.
- **MongoDB Implementation:**
  - **Single Document Operations:** Atomicity is inherently supported for single document operations in MongoDB. Any modifications (inserts, updates, deletes) to a single document are atomic.
  - **Multi-Document Transactions:** Starting from MongoDB 4.0, MongoDB supports multi-document transactions in replica sets, and from MongoDB 4.2, this support extends to sharded clusters. This allows a group of operations to be treated as a single unit of work, ensuring all-or-nothing execution.



Example:

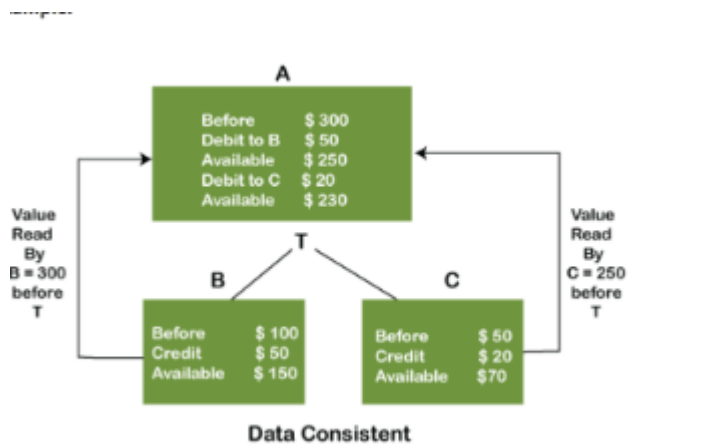
```
const session = client.startSession();
session.startTransaction();

try {
  await collection.insertOne({ _id: 1, name: 'Alice' }, { session });
  await collection.updateOne({ _id: 1 }, { $set: { age: 30 } }, { session });

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

## Consistency

- **Definition:** Consistency ensures that a transaction brings the database from one valid state to another, maintaining database invariants.
- **MongoDB Implementation:**
  - MongoDB ensures consistency within a single document by default. Multi-document transactions also ensure consistency by enforcing all constraints and checks during the transaction.




**Example:** Ensuring that an account balance does not go negative

```
const session = client.startSession();
session.startTransaction();

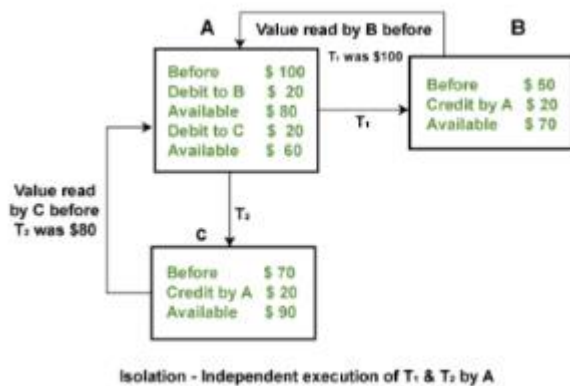
try {
  const account = await accountsCollection.findOne({ accountId: '123' }, { session })
  if (account.balance >= withdrawalAmount) {
    await accountsCollection.updateOne(
      { accountId: '123' },
      { $inc: { balance: -withdrawalAmount } },
      { session }
    );
  } else {
    throw new Error('Insufficient balance');
  }

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



### Isolation:

- **Definition:** Isolation ensures that concurrently executing transactions do not affect each other and the intermediate state of a transaction is invisible to other transactions.
- **MongoDB Implementation:**
  - MongoDB's multi-document transactions provide read and write isolation for the duration of the transaction.
  - The default isolation level in MongoDB transactions is "read committed," which means a transaction only sees data that has been committed by other transactions.



**Example:** Ensuring isolation between transactions.

```
const session = client.startSession();
session.startTransaction();

try {
  const user1 = await usersCollection.findOne({ userId: 'u1' }, { session });
  const user2 = await usersCollection.findOne({ userId: 'u2' }, { session });

  // Perform some operations on user1 and user2...

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

## Durability

- **Definition:** Durability ensures that once a transaction has been committed, it will remain so, even in the event of a system failure.
- **MongoDB Implementation:**
  - MongoDB writes data to the journal before acknowledging the transaction to the client, ensuring that the transaction can be recovered in case of a crash.



- In a replica set, MongoDB ensures durability by replicating the data across multiple nodes.

**Example:** Committing a transaction and ensuring durability.

```
const session = client.startSession();
session.startTransaction();

try {
  await collection.insertOne({ _id: 1, name: 'Alice' }, { session });
  await session.commitTransaction();
} catch (error) {
  await session.abortTransaction();
  throw error;
} finally {
  session.endSession();
}
```

### Replication:

Replication in MongoDB is a process that allows data to be duplicated across multiple servers to ensure high availability, redundancy, and disaster recovery. By using replication, MongoDB can provide data redundancy and increase data availability with automatic failover and recovery.

### Key Concepts of Replication in MongoDB

#### 1. Replica Set:

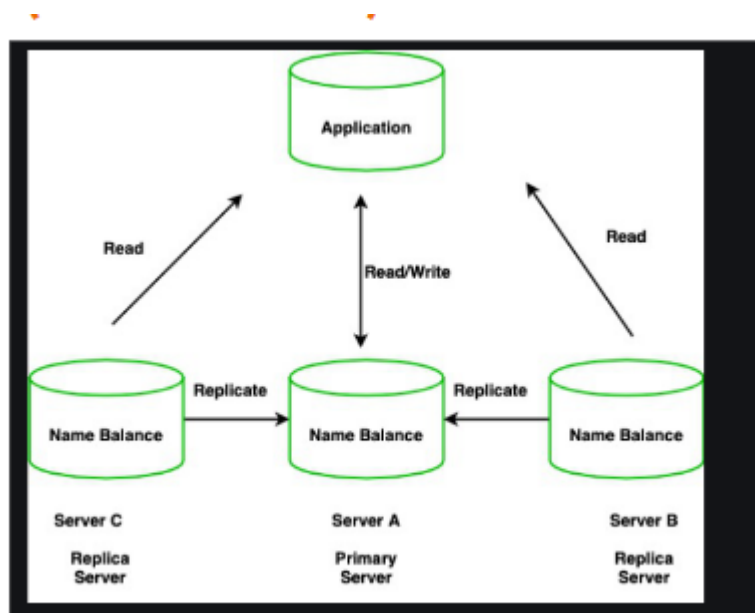
- A replica set in MongoDB is a group of mongod instances that maintain the same data set. Replica sets provide redundancy and high availability.
- A replica set consists of:
  - **Primary Node:** The primary node receives all write operations. It is the only node that can accept write operations, and it applies all changes to its data set and records the operations in its oplog (operations log).
  - **Secondary Nodes:** Secondary nodes replicate the data from the primary node. They apply operations from the primary's oplog to their data sets to keep in sync with the primary. Secondary nodes can also serve read operations if they are configured to do so.
  - **Arbiter:** An arbiter participates in elections for primary but does not hold data and cannot become a primary. It helps to break ties in the election process.

#### 2. Oplog (Operations Log):

- The oplog is a special capped collection that keeps a rolling record of all operations that modify the data stored in the primary node. Secondary nodes replicate data by replaying the operations in the primary's oplog.

## Benefits of Replication

- **High Availability:** By having multiple copies of data on different servers, MongoDB ensures that the system remains operational even if some servers fail.
- **Data Redundancy:** Replication provides multiple copies of data across different nodes, which protects against data loss.
- **Automatic Failover:** If the primary node goes down, an election is held to determine which secondary node will become the new primary, ensuring continued availability.
- **Read Scaling:** Secondary nodes can serve read operations if they are configured to allow read preferences, thereby distributing the read load.



## Example of Setting Up a Replica Set

1. **Start MongoDB Instances:** Start multiple MongoDB instances on different ports.

```
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:** Connect to one of the instances using the MongoDB shell and initiate the replica set.

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

### Check Replica Set Status:

Verify the replica set status to ensure that all nodes are properly configured.

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

### Configuring Read Preferences:

We can configure read preferences to allow read operations from secondary nodes. This can be useful for distributing the read load.

```
db.getMongo().setReadPref("secondary");

// Read from the nearest node
db.getMongo().setReadPref("nearest");
```

### Example of Automatic Failover

If the primary node goes down, MongoDB will automatically elect a new primary from the secondary nodes.

```
// Step down the current primary to test failover
rs.stepDown();

// Check the status to see the new primary
rs.status();
```

### Key Commands for Managing Replica Sets

- Adding Members:

```
rs.add("localhost:27020");
```

- Removing Members:

```
rs.remove("localhost:27019");
```

- Reconfiguring the Replica Set:

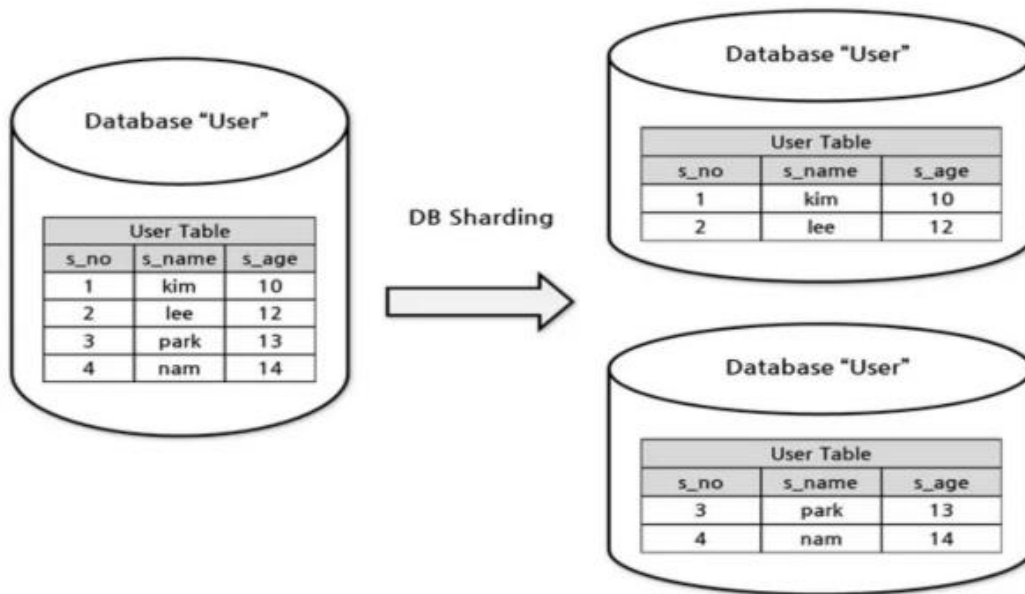
```
cfg = rs.conf();  
cfg.members[0].priority = 2;  
rs.reconfig(cfg);
```

### Key Takeaways

- **Atomicity:** Ensures all operations within a transaction are completed or none are.
  - **Consistency:** Ensures database constraints and invariants are maintained.
  - **Isolation:** Ensures transactions are executed independently.
  - **Durability:** Ensures committed transactions persist despite failures.
- 
- **Document-Level ACID:** MongoDB has always provided strong ACID guarantees at the document level.
  - **Multi-Document ACID:** While newer versions support multi-document transactions, it's important to understand the performance implications and use cases where it's necessary.
  - **Data Modeling:** Effective data modeling can often minimize the need for multi-document transactions.

MongoDB's support for multi-document ACID transactions allows for complex operations across multiple documents while maintaining data integrity and consistency, making it suitable for applications that require reliable and robust transaction management.

### Sharding:



Activate Windows

Sharding in MongoDB is a method for distributing data across multiple servers. This allows MongoDB to handle large datasets and high-throughput operations by splitting data into smaller, more manageable pieces called shards. Each shard is a subset of the data and can be stored on different physical servers. Here's a more detailed look at sharding in MongoDB:

## Key Concepts

- Shard:**
  - A shard is a single instance of MongoDB that holds a portion of the dataset.
  - It can be a replica set to ensure high availability and data redundancy.
- Shard Key:**
  - A shard key is a field or fields that determine how data is distributed across the shards.
  - Choosing an appropriate shard key is critical for balanced distribution and efficient queries.
- Config Servers:**
  - Config servers store metadata and configuration settings for the cluster.
  - They manage the sharded cluster's metadata, including the location of chunks and the state of each shard.
- Mongos:**
  - Mongos acts as a query router, directing client requests to the appropriate shard(s) based on the shard key.
  - It provides a single interface for client applications to interact with the sharded cluster.
- Chunks:**
  - Data is divided into chunks based on the shard key, and each chunk is assigned to a shard.
  - The size of a chunk is configurable, typically ranging from 64MB to 256MB.

## Sharding Process

1. **Enable Sharding:**
  - Sharding must be enabled on the database level before collections within the database can be sharded.
  - `sh.enableSharding("myDatabase")`
2. **Create a Shard Key:**
  - A shard key must be chosen and indexed on the collection to be sharded.
  - `db.myCollection.createIndex({ shardKey: 1 })`
  - `sh.shardCollection("myDatabase.myCollection", { shardKey: 1 })`
3. **Distribute Data:**
  - MongoDB automatically distributes data across shards based on the shard key.
  - If necessary, data is rebalanced to ensure even distribution.
4. **Scaling:**
  - As data grows, new shards can be added to the cluster.
  - MongoDB handles data migration and rebalancing automatically.

## Advantages of Sharding

- **Horizontal Scalability:** Distributes data across multiple servers, increasing storage capacity and computational power.
- **High Availability:** Replica sets within shards ensure data redundancy and failover capabilities.
- **Improved Performance:** Distributes read and write operations across multiple servers, reducing load and improving query performance.

## Challenges and Best Practices

- **Choosing the Right Shard Key:** A poorly chosen shard key can lead to uneven data distribution and performance bottlenecks.
- **Managing Shard Balancing:** Continuous monitoring is required to ensure data is evenly distributed, avoiding hotspots.
- **Handling Failures:** Config servers and replica sets must be properly managed to handle failures and ensure data integrity.

## Example Configuration

1. **Enable Sharding on Database:**

```
sh.enableSharding("myDatabase")
```

2. **Create an Index on the Shard Key:**

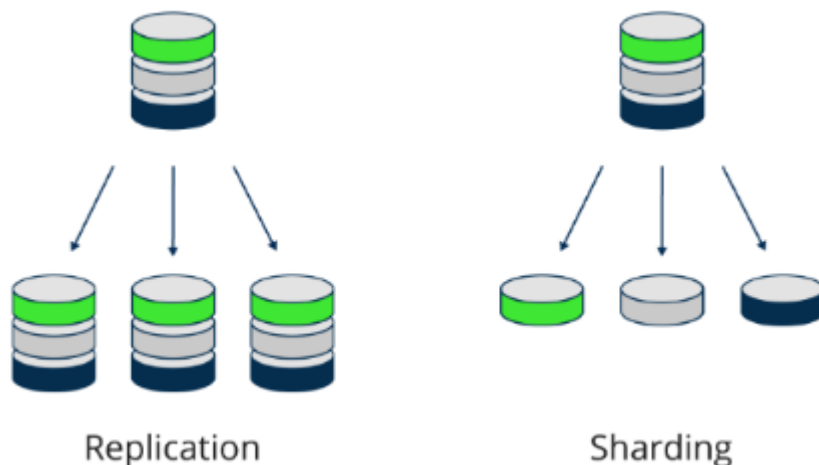
```
db.myCollection.createIndex({ userId: 1 })
```

### 3. Shard the Collection:

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

Sharding is a powerful feature in MongoDB, but it requires careful planning and management to ensure optimal performance and reliability.

#### Replication VS Sharding



Replication and sharding are two fundamental strategies in MongoDB used to handle data distribution and ensure high availability and scalability. They serve different purposes and are often used together in a production environment. Here's a comparison of replication and sharding:

#### Replication

##### Purpose:

- Ensures high availability and data redundancy.
- Provides fault tolerance by duplicating data across multiple servers.

##### Components:

- **Replica Set:** A group of MongoDB instances that maintain the same data set.
  - **Primary:** Receives all write operations.
  - **Secondary:** Copies data from the primary. Can serve read operations if configured.
  - **Arbiter:** Participates in elections but does not hold data. Used to break ties.

##### Mechanism:

- Write operations are sent to the primary, and the primary applies them to its data set.
- Secondary members replicate the data from the primary asynchronously.

- In case of primary failure, an election is held to choose a new primary from the secondaries.

### Use Cases:

- High availability.
- Data redundancy and backup.
- Disaster recovery.

### Advantages:

- Automatic failover in case of primary failure.
- Increased read performance (if reads are distributed to secondaries).
- Data redundancy ensures data safety.

### Challenges:

- Increased storage requirements as data is duplicated.
- Potential lag between primary and secondaries in data replication.

## Sharding

### Purpose:

- Provides horizontal scalability by distributing data across multiple servers.
- Manages large data sets by splitting them into smaller, manageable pieces.

### Components:

- **Shards:** Each shard holds a subset of the data.
- **Shard Key:** Determines how data is distributed across shards.
- **Config Servers:** Store metadata and configuration settings for the cluster.
- **Mongos:** Acts as a query router, directing operations to the appropriate shard.

### Mechanism:

- Data is partitioned into chunks based on the shard key.
- Each chunk is assigned to a shard.
- MongoDB automatically balances the data and distributes load across shards.

### Use Cases:

- Handling large volumes of data.
- Applications requiring high throughput for read and write operations.
- Distributing data geographically.

### Advantages:



- Allows for horizontal scaling by adding more shards as data grows.
- Distributes read and write load across multiple servers.
- Can manage large datasets that cannot fit on a single server.

### Challenges:

- Choosing an appropriate shard key is critical and can be complex.
- Requires careful management to ensure balanced data distribution.
- Increased complexity in setup and maintenance.

### When to Use Replication vs. Sharding

#### Replication:

- When the primary goal is high availability and disaster recovery.
- When you need to increase read performance by distributing reads to secondaries.
- Suitable for smaller datasets that fit on a single server but need redundancy.

#### Sharding:

- When the dataset is too large to fit on a single server.
- When you need to scale out horizontally to handle high-throughput operations.
- Suitable for applications with large amounts of data and high performance requirements.

### Combined Use

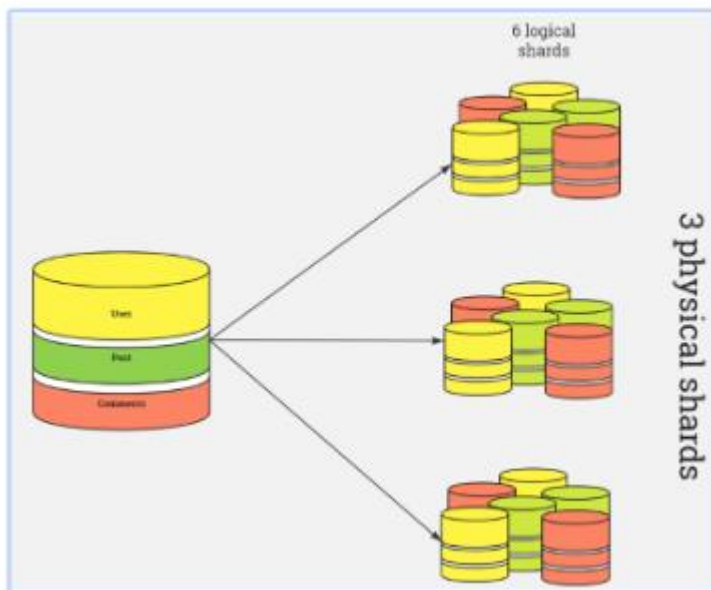
In many cases, replication and sharding are used together to leverage the benefits of both:

- **Sharded Cluster with Replica Sets:** Each shard in a sharded cluster is a replica set, combining the benefits of horizontal scaling and high availability.

## Key Differences

Feature	Replication	Sharding
Purpose	High availability, data redundancy	Horizontal scaling, performance
Data distribution	Copies entire dataset	Splits data into chunks
Number of copies	Multiple copies of the same data	Multiple copies of different data subsets
Scaling method	Vertical scaling (adding more resources to a server)	Horizontal scaling (adding more servers)

## Replication + Sharding



## Combining Replication and Sharding

For optimal performance and reliability, it's common to use both replication and sharding together. Each shard is typically a replica set, providing high availability and data redundancy within each shard.

In MongoDB, combining sharding and replication provides both horizontal scalability and high availability. Each shard in a sharded cluster is typically a replica set, ensuring data redundancy and fault tolerance. Here's a detailed overview of how sharding and replication work together in MongoDB:

## Sharded Cluster with Replica Sets

### Components

1. **Shards:**
  - Each shard is a replica set that contains a subset of the data.
  - Provides high availability and redundancy within each shard.
2. **Replica Sets:**
  - Consist of multiple MongoDB instances: one primary and multiple secondaries.
  - Ensure data is replicated and available in case of failure.
3. **Config Servers:**
  - Store metadata and configuration settings for the sharded cluster.
  - Manage information about chunk distribution and shard state.
4. **Mongos:**
  - Acts as a query router, directing client requests to the appropriate shard(s) based on the shard key.
  - Provides a single entry point for client applications.
5. **Chunks:**
  - Data is divided into chunks based on the shard key.
  - Each chunk is assigned to a shard, and MongoDB automatically balances these chunks across the shards.

### Setting Up a Sharded Cluster with Replica Sets

1. **Configure Replica Sets:**
  - Set up multiple MongoDB instances for each shard, configured as a replica set.
  - Example configuration for a replica set:

```
rs.initiate(  
  {  
    _id: "shard1",  
    members: [  
      { _id: 0, host: "shard1a:27017" },  
      { _id: 1, host: "shard1b:27017" },  
      { _id: 2, host: "shard1c:27017", arbiterOnly: true }  
    ]  
  }  
)
```

2. **Start Config Servers:**
  - Config servers store the metadata for the sharded cluster.
  - At least three config servers are recommended for redundancy.

### 3. Start Mongos Instances:

- Start one or more Mongos instances to route client requests.
- Connect Mongos to the config servers.

### 4. Enable Sharding on the Database:

```
sh.enableSharding("myDatabase")
```

### 5. Create a Shard Key and Shard the Collection:

- Create an index on the shard key:

```
db.myCollection.createIndex({ userId: 1 })
```

- Shard the collection:

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

### 6. Add Shards to the Cluster:

- Add the replica sets as shards:

```
sh.addShard("shard1/shard1a:27017,shard1b:27017,shard1c:27017")
sh.addShard("shard2/shard2a:27017,shard2b:27017,shard2c:27017")
```

### *Advantages of Combining Sharding and Replication*

- **Scalability:** Distributes data across multiple servers, allowing horizontal scaling to manage large datasets.
- **High Availability:** Replica sets within each shard ensure data redundancy and fault tolerance.
- **Load Balancing:** Distributes read and write operations across multiple shards, improving performance.
- **Automatic Failover:** Replica sets provide automatic failover in case of node failures within a shard.

### *Example Workflow*

#### 1. Data Ingestion:

- Data is ingested through Mongos, which determines the appropriate shard based on the shard key.

- Writes are directed to the primary node of the appropriate replica set.
2. **Read Operations:**
- Reads can be distributed across the primary and secondary nodes of the replica sets, depending on the read preference.
  - Mongos routes the read queries to the appropriate shard(s).
3. **Balancing and Failover:**
- MongoDB automatically balances chunks across shards to ensure even data distribution.
  - In case of a primary node failure within a shard, a secondary node is promoted to primary, ensuring continuous availability.

By combining sharding and replication, MongoDB provides a robust and scalable solution for managing large, high-throughput applications with the assurance of high availability and data redundancy.

## In summary:

- **Replication** is about data redundancy and availability within a single dataset.
- **Sharding** is about distributing data across multiple machines to handle large datasets and high throughput.

By understanding the differences between replication and sharding, you can effectively design and scale your MongoDB deployments to meet the specific needs of your application.

## Indexes:



## Types of indexes:

## Basic Index Types

- **Single Field Index:**

- Indexes a single field within a document.
- Example: `db.collection.createIndex({ field1: 1 })`

- **Compound Index:**

- Indexes multiple fields in a specified order.
- Useful for range-based queries involving multiple fields.
- Example: `db.collection.createIndex({ field1: 1, field2: -1 })`

- **Multikey Index:**

- Indexes array elements individually.
- Enables efficient queries on array elements.
- Example: `db.collection.createIndex({ arrayField: 1 })`

---

## Specialized Index Types

- **Text Index:**

- Indexes text content for full-text search capabilities.
- Supports text search operators like `$text` and `$search`.
- Example: `db.collection.createIndex({ text: "text" })`

- **Geospatial Index:**

- Indexes geospatial data (coordinates) for efficient proximity-based queries.
- Supports 2dsphere and 2d indexes for different use cases.
- Example: `db.collection.createIndex({ location: "2dsphere" })`

- **Hashed Index:**

- Creates a hashed index for the specified field.
- Primarily used for the `_id` field for performance optimization.
- Example: `db.collection.createIndex({ _id: "hashed" })`

### Additional Considerations

- **Sparse Indexes:**

- Only index documents where the indexed field exists.
- Can improve performance for sparse datasets.

- **Unique Indexes:**

- Ensure that the indexed field has unique values across all documents.

- **TTL Indexes:**

- Automatically expire documents after a specified time.

**Choosing the right index type** depends on your specific data structure, query patterns, and performance requirements. Careful index design can significantly improve query performance, but excessive indexing can impact write performance.

**Would you like to delve deeper into a specific index type or discuss index creation strategies for a particular use case?**

**Important note:**

<https://g.co/gemini/share/7e3cd8a53fcf>