ACID AND INDEXES

ACID in MongoDB:

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that ensure database transactions are processed reliably .MongoDB supports a subset of ACID properties, which are:

- **Atomicity:** Similar to relational databases, MongoDB guarantees atomicity for single-document updates. This means either the entire update to a single document is successful, or none of the changes are applied.
- **Consistency:** MongoDB enforces data integrity through validation rules and document schema. However, unlike fully ACID-compliant systems, MongoDB achieves consistency at the document level, not necessarily across multiple documents within a single transaction.
- **Isolation:** MongoDB provides some isolation mechanisms to prevent concurrent transactions from interfering with each other's data. However, the isolation level can be weaker compared to relational databases.
- **Durability:** MongoDB ensures durability by writing data to the journal before updating the actual data files. This guarantees that even in case of a system crash, committed transactions persist on disk.

Indexing in MongoDB:

Indexing in MongoDB is a powerful feature that allows you to efficiently query and manage your data. Here's a detailed overview of how indexing works in MongoDB, the different types of indexes, including.

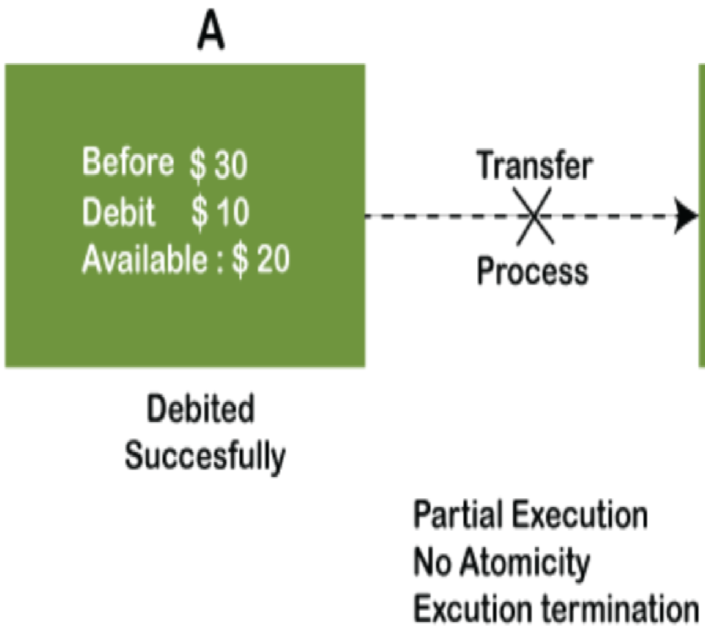
Types of Indexes in MongoDB:

1. **Single Field Index:**
   - The most basic type of index.
   - Created on a single field of a document.
2. **Compound Index:**
   - Index on multiple fields.
   - Useful for queries that involve multiple fields.
3. **Multikey Index:**
   - Used to index array fields.
   - MongoDB creates an index for each element of the array.
4. **Text Index:**
   - Supports text search queries on string content.
   - Allows searching for words within string fields.
5. **Hashed Index:**
   - Indexes the hash of the field value.
   - Supports hash-based sharding.
6. **Geospatial Index:**
   - Supports queries of geospatial data.

- Types include 2d indexes and 2dsphere indexes.

ATOMICITY:

In MongoDB, atomicity applies specifically to operations on single documents. This means that when you perform a CRUD (Create, Read, Update, Delete) operation on a single document, either the entire operation succeeds or none of it does. It's like an all-or-nothing proposition for that particular document.



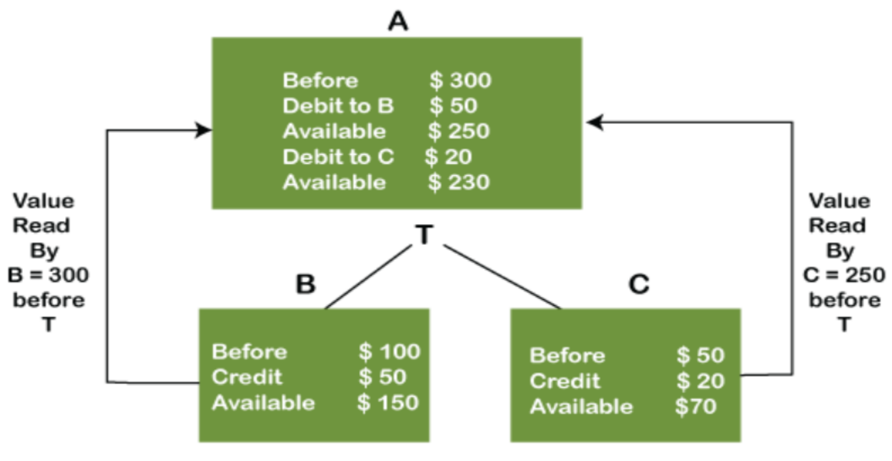
Benefits of Atomicity in MongoDB:

- **Ensures Data Consistency:** The primary benefit of atomicity is that it prevents documents from ending up in an inconsistent state. Updates to a single document are treated as a whole, meaning either all changes are applied successfully or none are.
- **Prevents Data Corruption:** By guaranteeing all or nothing updates, atomicity safeguards against data corruption that might occur if only a portion of an intended change is applied.
- **Improves Data Integrity:** Since atomicity ensures complete or no updates, it strengthens the overall integrity of your data.
- **Provides a Reliable Update Mechanism:** Atomicity simplifies data manipulation by providing a reliable all-or-nothing approach..
- **Reduces Need for Complex Error Handling:** With atomicity, the need for intricate error handling code to address partial updates is lessened. If an error occurs during an operation, the entire update is rolled back, leaving the document unchanged.

CONSISTENCY:

The word **consistency** means that the value should remain preserved always. In DBMS, the integrity of the data should be maintained, which means if a change in the database is made, it should remain preserved always. In the case of transactions, the integrity of the data is very essential so that the database remains consistent before and after the transaction. The data should always be correct.

Example:



Data Consistent

Types of Consistency in MongoDB:

1. Eventual Consistency (Default):

- This is the default consistency model in MongoDB. Data updates are replicated asynchronously across the cluster. While eventually all replicas will have the same data, there might be a brief lag between when a write operation is committed on the primary node and when it's reflected on all secondary nodes.

2. Strong Consistency (Read After Write):

- This consistency model ensures that after a write operation is committed on the primary node, any subsequent read operation (even on a secondary) will always return the latest data.
- This is achieved by forcing the read operation to wait until confirmation is received from a configurable number of secondary nodes acknowledging the write.

3. Causal Consistency (Client Sessions):

- This consistency model focuses on ensuring a specific order for causally related operations within a client session.

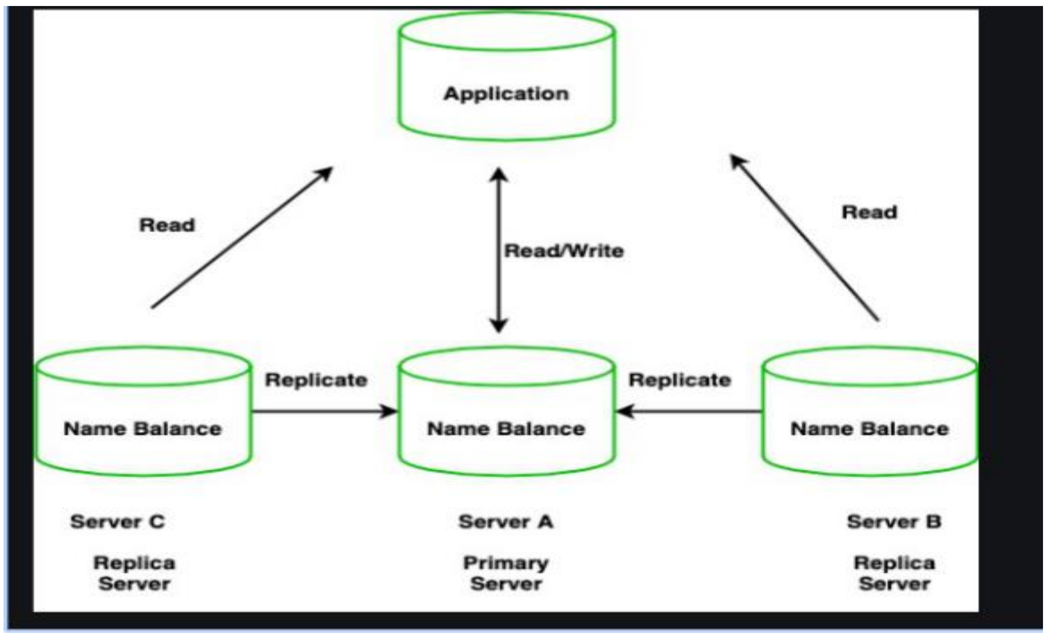
- This requires enabling causally consistent client sessions and using specific read and write concerns within the session.

REPLICATION:

Replication in MongoDB is a process that involves creating multiple copies of data across different servers to ensure high availability, data redundancy, and fault tolerance. This allows your MongoDB database to remain operational even in the event of hardware failures, server crashes, or maintenance tasks.

Benefits of Replication:

- **High Availability:** Automatic failover to a secondary node ensures minimal downtime.
- **Data Redundancy:** Multiple copies of data protect against data loss.
- **Horizontal Scaling:** Enables distributing read operations across multiple nodes.
- **Disaster Recovery:** Provides data backup and recovery options.

- 

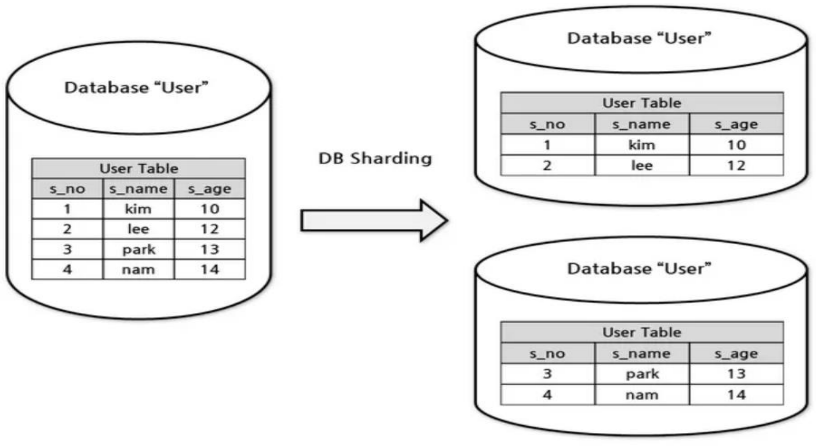
SHARDING:

Sharding in MongoDB is a method for distributing data across multiple servers, allowing for horizontal scaling of a database. This is essential for handling large datasets and high-throughput applications, as it provides both load balancing and improved performance.

Types of Sharding in MongoDB:

1. **Range-Based Sharding**

2. **Hash-Based Sharding**
3. **Zone-Based Sharding (or Zone Sharding)**



User Table		
s_no	s_name	s_age
1	kim	10
2	lee	12
3	park	13
4	nam	14

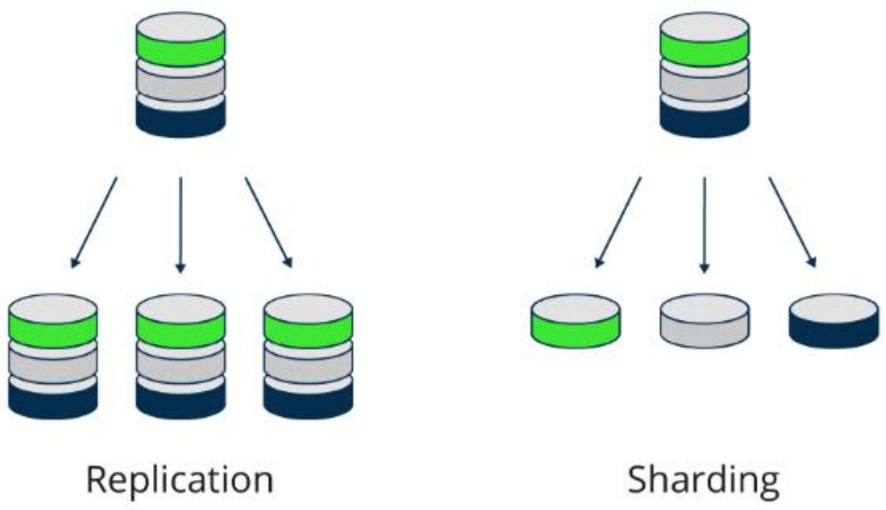
User Table		
s_no	s_name	s_age
1	kim	10
2	lee	12

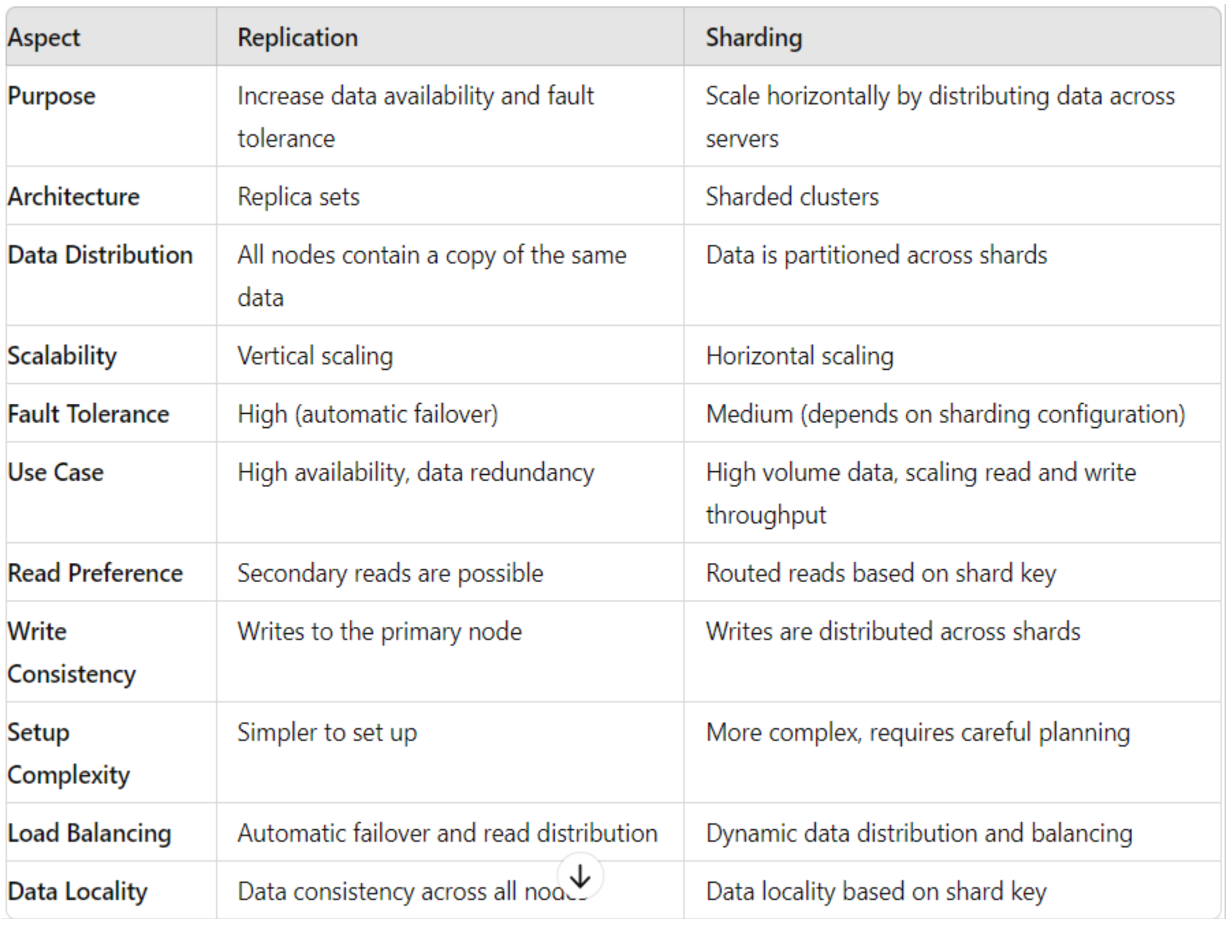
User Table		
s_no	s_name	s_age
3	park	13
4	nam	14

Benefits of Sharding in MongoDB:

- **Horizontal Scalability:** Distribute data across multiple servers to handle increased load and large datasets.
- **Increased Throughput:** Parallelize operations across multiple shards for better read and write performance.
- **Improved Fault Tolerance:** Data replication across shards ensures that the cluster remains operational even if some nodes fail.
- **Load Balancing:** Automatically balances data across shards to ensure even distribution and prevent bottlenecks.

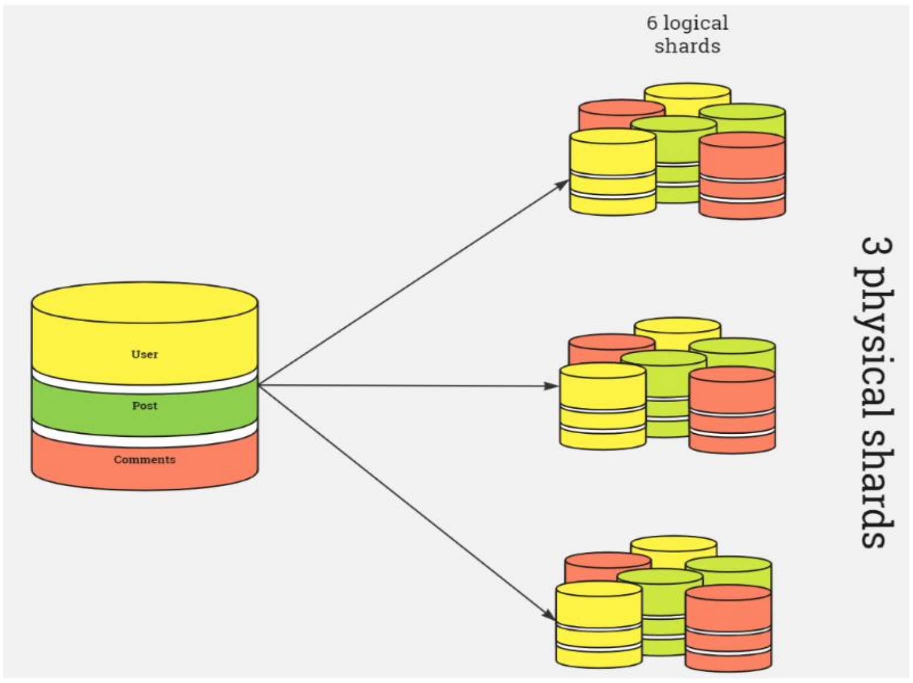
Replication vs Sharding:





Aspect	Replication	Sharding
Purpose	Increase data availability and fault tolerance	Scale horizontally by distributing data across servers
Architecture	Replica sets	Sharded clusters
Data Distribution	All nodes contain a copy of the same data	Data is partitioned across shards
Scalability	Vertical scaling	Horizontal scaling
Fault Tolerance	High (automatic failover)	Medium (depends on sharding configuration)
Use Case	High availability, data redundancy	High volume data, scaling read and write throughput
Read Preference	Secondary reads are possible	Routed reads based on shard key
Write Consistency	Writes to the primary node	Writes are distributed across shards
Setup Complexity	Simpler to set up	More complex, requires careful planning
Load Balancing	Automatic failover and read distribution	Dynamic data distribution and balancing
Data Locality	Data consistency across all no	Data locality based on shard key

REPLICATION + SHARDING:



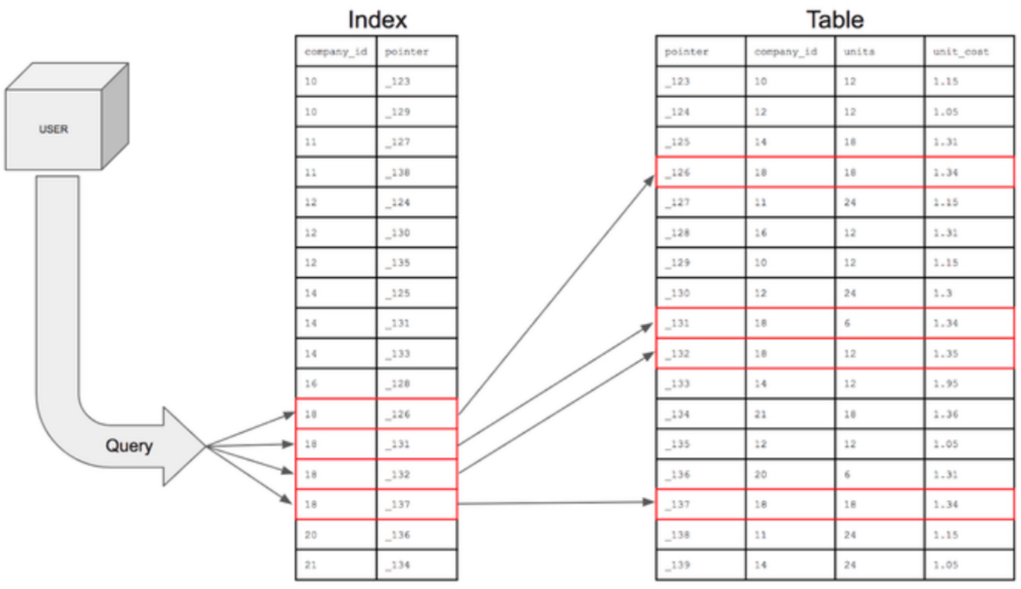
How it Works:

Combining replication and sharding in MongoDB allows you to achieve both high availability and scalability for your database. Replication provides data redundancy and fault tolerance, while sharding distributes data across multiple servers to handle large datasets and high transaction volumes.

- **Replica Set**: A group of MongoDB servers that maintain the same data set, providing redundancy and high availability.
- **Shard**: A subset of the entire database, each shard contains a portion of the data and can be replicated using replica sets.
- **Config Server**: Stores metadata and configuration settings for the sharded cluster, such as the mapping of data chunks to shards.
- **Query Router (mongos)**: Acts as an interface between the application and the sharded cluster, routing client requests to the appropriate shards based on the shard key.

INDEXES:

Indexes are special data structures that store a small portion of the collection's data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index, and a reference to the actual document in the collect.



Index

company_id	pointer
10	_123
10	_129
11	_127
11	_138
12	_124
12	_130
12	_135
14	_125
14	_131
14	_133
16	_128
18	_126
18	_131
18	_132
18	_137
20	_136
21	_134

Table

pointer	company_id	units	unit_cost
_123	10	12	1.15
_124	12	12	1.05
_125	14	18	1.31
_126	18	18	1.34
_127	11	24	1.15
_128	16	12	1.31
_129	10	12	1.15
_130	12	24	1.3
_131	18	6	1.34
_132	18	12	1.35
_133	14	12	1.95
_134	21	18	1.36
_135	12	12	1.05
_136	20	6	1.31
_137	18	18	1.34
_138	11	24	1.15
_139	14	24	1.05

Types of Indexes in MongoDB:

1. **Single Field Index:**
   - The most basic type of index.
   - Created on a single field of a document.
2. **Compound Index:**
   - Index on multiple fields.
   - Useful for queries that involve multiple fields.
3. **Multikey Index:**
   - Used to index array fields.
   - MongoDB creates an index for each element of the array.
4. **Text Index:**
   - Supports text search queries on string content.
   - Allows searching for words within string fields.
5. **Hashed Index:**
   - Indexes the hash of the field value.
   - Supports hash-based sharding.
6. **Geospatial Index:**
   - Supports queries of geospatial data.
   - Types include 2d indexes and 2dsphere indexes.

Benefits of Indexes in MongoDB

1. Improved Query Performance

- **Faster Data Retrieval**: Indexes allow MongoDB to quickly locate and access documents that match query criteria, reducing the need for full collection scans.

2. Reduced I/O Operations

- **Minimized Disk Reads**: By using indexes, MongoDB reduces the number of disk reads required to satisfy a query, as it doesn't need to scan every document in the collection.

3. Enhanced Aggregation Performance

- **Optimized Aggregation Stages**: Indexes can speed up aggregation operations, particularly those that involve `$match` and `$sort` stages.

5. Improved Application Performance

- **Faster Query Response Times**: Applications that rely on MongoDB for data retrieval benefit from faster query response times due to the efficient use of indexes.

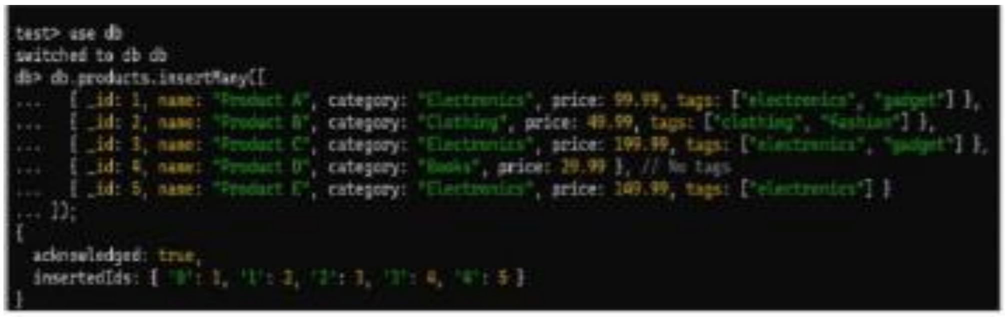
6. Efficient Data Management

- **Data Consistency**: Unique indexes ensure that certain fields have unique values across documents, helping maintain data consistency and integrity.

7. Reduced Load on Database

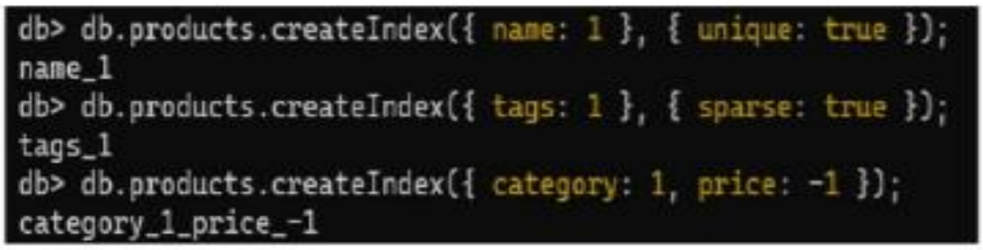
- **Less Server Load**: By optimizing query execution, indexes reduce the computational load on the database server, allowing it to handle more queries simultaneously.

Inserting documents into collection:



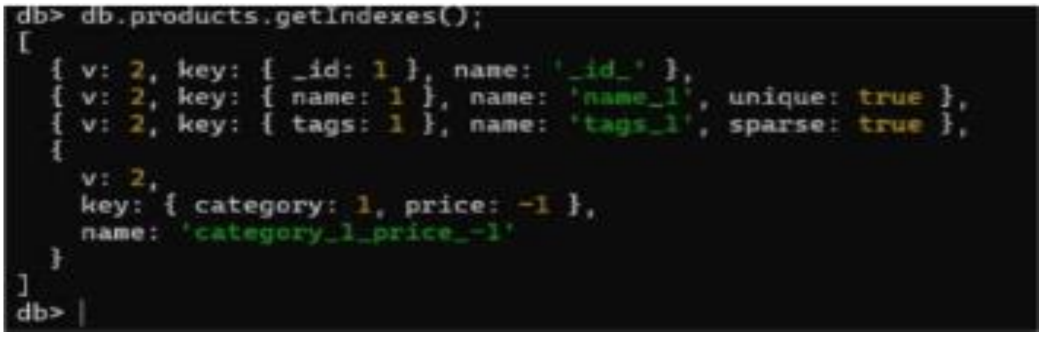
```
test> use db
switched to db db
db> db.products.insertMany([
...   { _id: 1, name: "Product A", category: "Electronics", price: 99.99, tags: ["electronics", "gadget"] },
...   { _id: 2, name: "Product B", category: "Clothing", price: 49.99, tags: ["clothing", "fashion"] },
...   { _id: 3, name: "Product C", category: "Electronics", price: 199.99, tags: ["electronics", "gadget"] },
...   { _id: 4, name: "Product D", category: "Books", price: 29.99 }, // No tags
...   { _id: 5, name: "Product E", category: "Electronics", price: 149.99, tags: ["electronics"] }
... ]);
{
  acknowledged: true,
  insertedIds: { '0': 1, '1': 2, '2': 3, '3': 4, '4': 5 }
}
```

Creating Different Types of Indexes:



```
db> db.products.createIndex({ name: 1 }, { unique: true });
name_1
db> db.products.createIndex({ tags: 1 }, { sparse: true });
tags_1
db> db.products.createIndex({ category: 1, price: -1 });
category_1_price_-1
```

To get the indexes we use the below code:



```
db> db.products.getIndexes();
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { name: 1 }, name: 'name_1', unique: true },
  { v: 2, key: { tags: 1 }, name: 'tags_1', sparse: true },
  {
    v: 2,
    key: { category: 1, price: -1 },
    name: 'category_1_price_-1'
  }
]
db> |
```