ACID AND INDEXES

Introduction

MongoDB is a widely used NoSQL database that supports a range of features for managing data. Two critical aspects of MongoDB are its support for ACID transactions and its indexing capabilities. This report provides an overview of ACID transactions and indexes in MongoDB, along with examples demonstrating their usage in MongoDB query language.

ACID Transactions in MongoDB

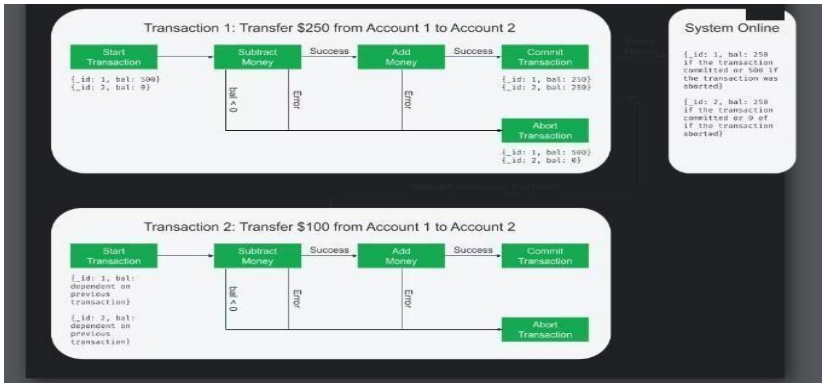
ACID stands for Atomicity, Consistency, Isolation, and Durability. MongoDB supports multi-document ACID transactions, which allow for operations involving multiple documents to be executed with the assurance that the transactions meet the ACID properties.

Key Properties of ACID Transactions

1. **Atomicity**: Ensures that a series of operations within a transaction are treated as a single unit. If one operation fails, the entire transaction is rolled back.
2. **Consistency**: Guarantees that the database transitions from one valid state to another, maintaining all defined rules and constraints.
3. **Isolation**: Ensures that transactions are isolated from one another, preventing operations in one transaction from affecting operations in another.
4. **Durability**: Guarantees that once a transaction has been committed, it will remain committed, even in the case of a system crash.

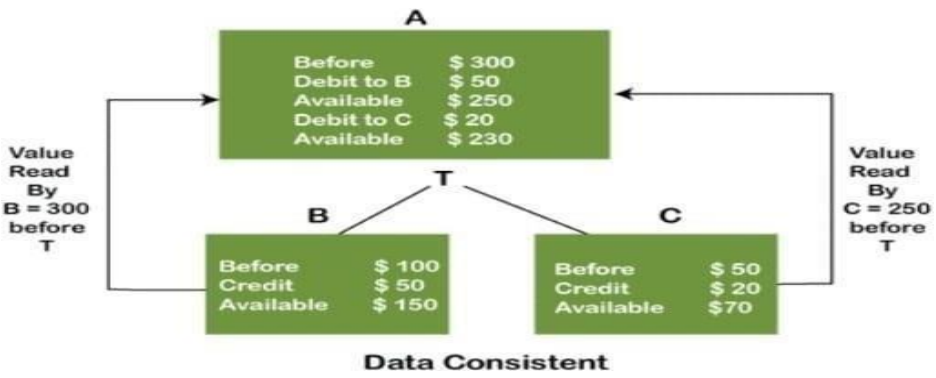
1. Atomicity

Atomicity ensures that all operations within a transaction are executed as a single unit. If one operation fails, the entire transaction is rolled back



2. Consistency

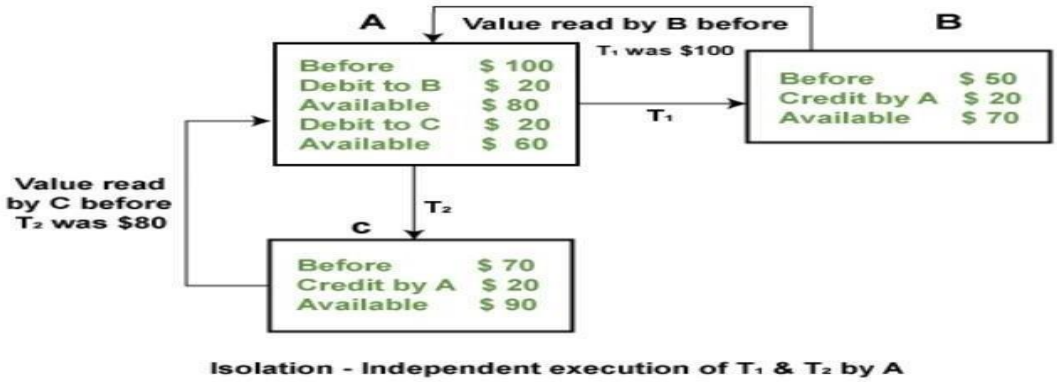
Consistency ensures that the database transitions from one valid state to another and adheres to all constraints and rules.



The transaction ensures that the stock level does not go negative, maintaining data consistency.

3. Isolation

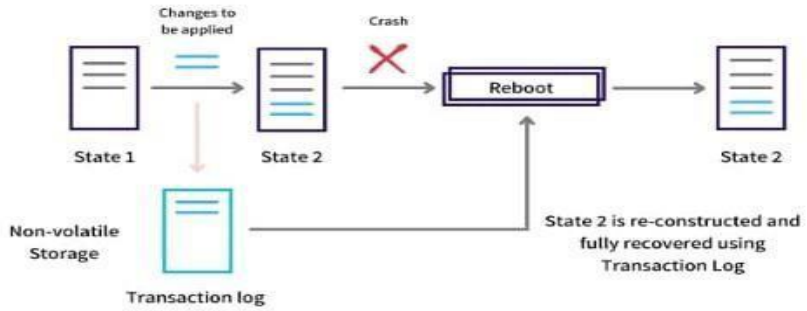
Isolation ensures that operations in one transaction are not visible to other transactions until the transaction is committed.



Isolation - Independent execution of T_1 & T_2 by A

4. Durability

Durability ensures that once a transaction has been committed, it will remain committed even in the event of a system crash.



Durability

Why are ACID transactions important?

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.

In MongoDB, ACID transactions work within the context of replication and sharding, ensuring that multi-document operations maintain consistency, atomicity, isolation, and durability. Here's how these concepts interplay:

1. Replication

Replication in MongoDB involves copying data from a primary node to one or more secondary nodes. This process ensures data availability and redundancy. MongoDB achieves replication using replica sets, which are groups of MongoDB servers that maintain the same data set.

How ACID Transactions Work with Replication

1. **Atomicity and Consistency**: Transactions in MongoDB are atomic and consistent across the replica set. When a transaction is committed on the primary node, the changes are recorded in the primary's oplog (operation log). This log is used to replicate changes to secondary nodes.
2. **Isolation**: MongoDB provides snapshot isolation for transactions. Each transaction operates on a snapshot of the database at the start of the transaction, ensuring that operations in one transaction do not interfere with those in other concurrent transactions. This isolation level is maintained across the replica set.
3. **Durability**: Once a transaction is committed, it is written to the primary's data files and replicated to secondaries. MongoDB ensures that committed transactions are durable by using the write concern and acknowledgment mechanisms to ensure that data is safely stored and replicated.
4. **Replication Lag**: It's important to note that while the primary node commits transactions immediately, secondary nodes might experience replication lag. Therefore, the committed data might not be immediately visible on secondaries.

2. Sharding

Sharding is MongoDB's method for distributing data across multiple servers to handle large datasets and high throughput. Each shard is a replica set, and the entire sharded cluster is managed by a configuration server and a router (mongos).

How ACID Transactions Work with Sharding

1. **Atomicity and Consistency**: In a sharded cluster, MongoDB supports multi-document transactions that span multiple shards. This ensures that operations affecting documents across different shards are atomic and consistent. Transactions are coordinated across shards to ensure that all operations are committed or rolled back together.
2. **Isolation**: Transactions in sharded clusters provide snapshot isolation. The transaction sees a consistent snapshot of the data as it was at the start of the transaction, regardless of changes made by other transactions.
3. **Durability**: Similar to replication, once a transaction is committed, the changes are written to the primary nodes of each shard and are replicated according to the replica set's replication rules. MongoDB ensures that these changes are durable by acknowledging the transaction's commit once it is confirmed across the primary shards.
4. **Transaction Coordination**: MongoDB uses a distributed transaction coordinator to manage transactions that span multiple shards. This coordinator ensures that all shards involved in the transaction commit or abort the changes together.

INDEXES

Indexes in MongoDB are essential for optimizing query performance and ensuring efficient data retrieval. They allow MongoDB to quickly locate documents within a collection without scanning every document. Here's a comprehensive overview of indexes in MongoDB:

1. Single-Field Index

- **Description**: An index on a single field in a document. It is useful for optimizing queries that search or sort based on that field.

2. Compound Index

- **Description**: An index on multiple fields. It is used to optimize queries that involve multiple fields and can improve performance for complex queries.

3. Text Index

- **Description**: An index that supports text search operations on string content. It allows for full-text search within documents.

4. Geospatial Index

- **Description**: An index for querying geospatial data. It supports location-based queries, such as finding nearby points.

5. Hashed Index

- **Description**: An index that hashes the value of a field to ensure an even distribution of documents across shards. It is commonly used in sharded clusters.

6. Unique Index

- **Description**: Ensures that all values in the indexed field are unique. It is useful for enforcing constraints such as ensuring no duplicate values.

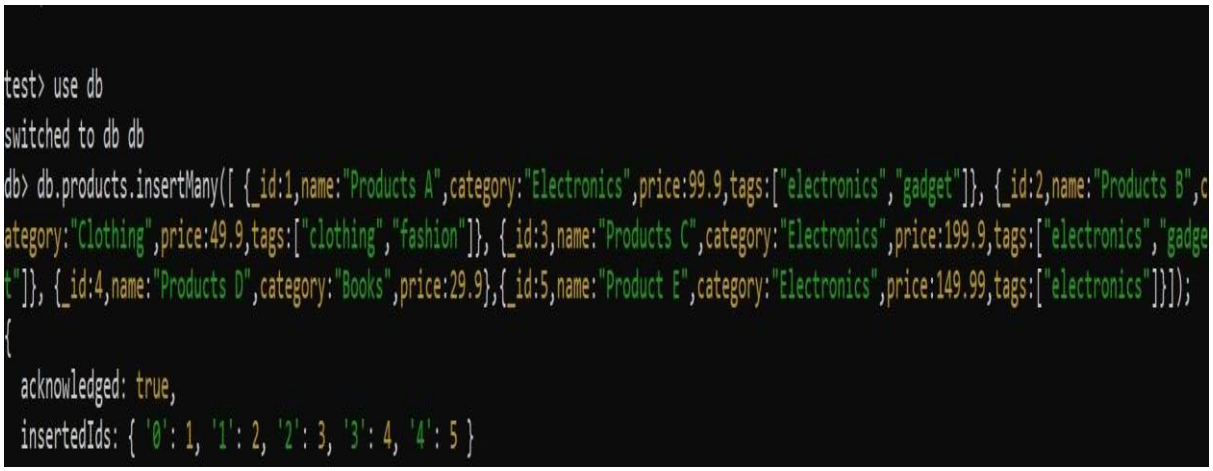
7. Partial Index

- **Description**: An index that only includes documents that match a specified filter expression. It can improve performance and reduce index size by indexing only relevant documents

CREATING DIFFERENT TYPES OF INDEXES IN MONGODB:

Let’s define a sample collection ,To insert many of the _id’s we use a command

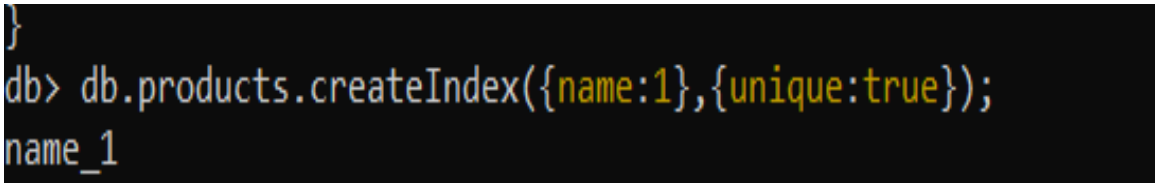
db.products.insertMany([{ _id: 1, name: "Product A", category: "Electronics", price: 99.99, tags: ["electronics", "gadget"] },{ _id: 2, name: "Product B", category: "Clothing", price: 49.99, tags: ["clothing", "fashion"] },{ _id: 3, name: "Product C", category: "Electronics", price: 199.99, tags: ["electronics", "gadget"] },{ _id: 4, name: "Product D", category: "Books", price: 29.99 }, { _id: 5, name: "Product E", category: "Electronics", price: 149.99, tags: ["electronics"] }]);



1. **Unique index:**

Ensures that each value in the indexed field is unique across all documents.

db.products.createIndex({ name: 1 }, { unique: true });



2. **Sparse index:**

Indexes only documents where the specified field exists.

db.products.createIndex({ tags: 1 }, { sparse: true });

```
db> db.products.createIndex({tags:1},{sparse:true});
tags_1
```

3 Compound index:

Indexes multiple fields in a specified order.

db.products.createIndex({ category: 1, price: -1 });

```
db> db.products.createIndex({category:1,price:-1});
category_1_price_-1
```

To verify if the indexes have been created we use a command:

db.products.getIndexes();