ACIDS AND INDEXES:

WHAT IS ACID :

ACID, in the context of databases, stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure data integrity and reliability in transactions, which are sets of database operations treated as a single unit.

- **Atomicity**: MongoDB ensures atomic operations at the document level. This means that updates to a single document are fully completed or not applied at all.

- **Consistency**: MongoDB maintains consistency by ensuring that any data written to the database is valid according to all defined rules, such as unique constraints.

- **Isolation**: MongoDB provides isolation for write operations to a single document, ensuring that operations are not affected by other operations that may be occurring simultaneously.

- **Durability**: MongoDB guarantees that once a write operation has been acknowledged, the data will be written to disk and will persist, even in the event of a server failure.

Key Differences

1. **Atomicity**:
   - **ACID**: Ensures that either both updates occur, or neither does.
   - **Non-ACID**: Each update is independent, leading to possible partial updates.
2. **Consistency**:
   - **ACID**: Ensures data remains consistent throughout the transaction.
   - **Non-ACID**: In case of an error, consistency might be compromised.
3. **Isolation**:
   - **ACID**: Ensures operations are isolated from other operations.
   - **Non-ACID**: Other operations might see partial updates.
4. **Durability**:
   - **ACID**: Once committed, the changes are durable and will survive crashes.
   - **Non-ACID**: Each operation might be durable independently, but not the whole sequence.

Atomicity in MongoDB

1. **Single Document Operations**: MongoDB ensures atomicity for all operations on a single document. This means updates, inserts, and deletes on a single document are all-or-nothing operations.
2. **Multi-Document Transactions**: Starting from MongoDB 4.0, you can use transactions to ensure atomicity across multiple documents and collections. This is particularly useful for maintaining consistency in more complex operations that involve multiple documents.
3. **Error Handling**: If an error occurs during a transaction, MongoDB will abort the transaction, and any changes made during the transaction will not be applied. This rollback mechanism ensures that the database remains in a consistent state.

4. **Session-Based**: Multi-document transactions require a session. Operations within the transaction must be associated with the session to ensure they are executed atomically.

Here are some key benefits:

1. Consistency of Data

2. Simplified Error Handling

3. Reliability

4. Ease of Maintenance

5. Improved Transaction Management

Imagine a financial transaction like a funds transfer between accounts. Atomicity ensures:

1. Funds deducted: The source account's balance is decreased.

2. Funds deposited: The destination account's balance is increased by the same amount.

If any of these steps fail due to a network error, power outage, or other reason, the entire transaction is rolled back. The source account retains its original balance, maintaining data integrity.

Example (without Atomicity):

An example without atomicity in MongoDB would illustrate a scenario where an operation involving multiple steps does not guarantee that all steps are completed successfully. This can lead to data inconsistency if one of the steps fails.

Consider a simple e-commerce application where a user places an order. The application needs to perform two operations:

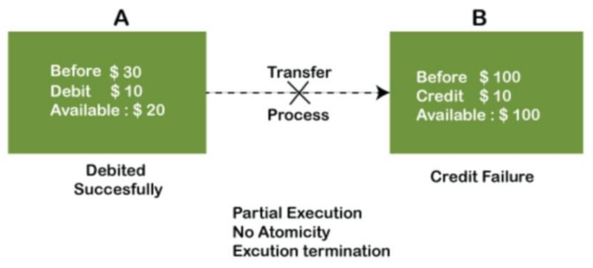
1. Decrement the inventory count of the product.
2. Add the order details to the orders collection.

Without using atomic transactions, if one of these operations fails, the database might end up in an inconsistent state.

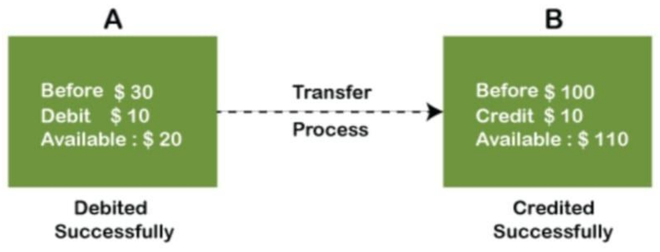
MongoDB, with its support for ACID properties, is widely used in various real-life applications.

1. Banking and Financial Services
2. Healthcare Systems
3. Telecommunications
4. Airline Reservations

PARTIAL EXECUTION WITH NO ATOMICITY:



COMPLETE EXECUTION WITH ATOMICITY:



CONSISTENCY:

• Data Integrity: Consistency safeguards the integrity of data within the database. It prevents transactions from leaving the database in an unexpected, illogical, or invalid state.

• Predefined Rules: The database enforces consistency based on pre-defined constraints and rules. These rules could involve data types, relationships between tables, or specific values allowed for certain fields.

Benefits of Consistency:

• Data Accuracy: Ensures data within the database remains accurate and reflects the real world.

• Valid State: Maintains the database in a valid and usable state, preventing inconsistencies that could lead to errors.

Example (Without Consistency):

Without consistency, imagine updating an order in an e-commerce system:

• Order quantity is increased (successful).

• But stock level isn't updated due to a system error (failure). This inconsistency could lead to overselling and customer dissatisfaction.

EVENTUAL CONSISTENCY: Eventual consistency is a data consistency model commonly used in distributed systems, particularly NoSQL databases.

Use cases for eventual consistency:

• Social media platforms: A user's latest post might not be immediately visible to all followers across the globe.

• E-commerce applications: Inventory levels might not reflect real-time stock availability across all locations.

• Real-time chat applications: There might be a slight delay in seeing newly sent messages for some users.

Key characteristics:

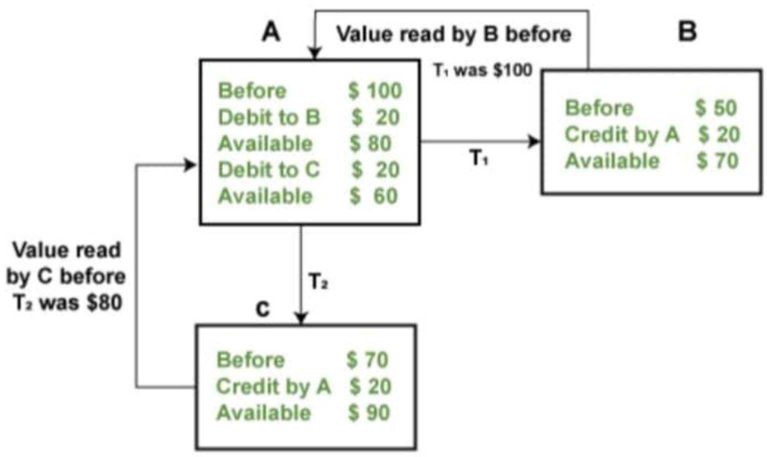
• Availability: High availability is a major benefit. Even if some nodes are unavailable, others can still serve requests.

• Scalability: The system can easily scale by adding more nodes without compromising performance.

• Latency: There can be a delay between updates being made and reflected on all nodes.

ISOLATION:

Example: If two operations are concurrently running on two different accounts, then the value of both accounts should not get affected. The value should remain persistent. As you can see in the below diagram, account A is making T1 and T2 transactions to account B and C, but both are executing independently without affecting each other. It is known as Isolation.



Isolation - Independent execution of T_1 & T_2 by A

Imagine a bank with multiple tellers processing transactions:

• Teller A withdraws money from Account 1.

• Teller B deposits money into Account 1 (same time).

Isolation ensures Teller B sees the updated balance after Teller A's withdrawal is complete, preventing inconsistencies.

Benefits of Isolation:

• Data Integrity: Prevents conflicts and unexpected results that could occur if transactions interfere with each other.

• Predictable Outcomes: Guarantees that transactions produce the intended results, even in a multi-user environment.

Replication (Master - Slave):

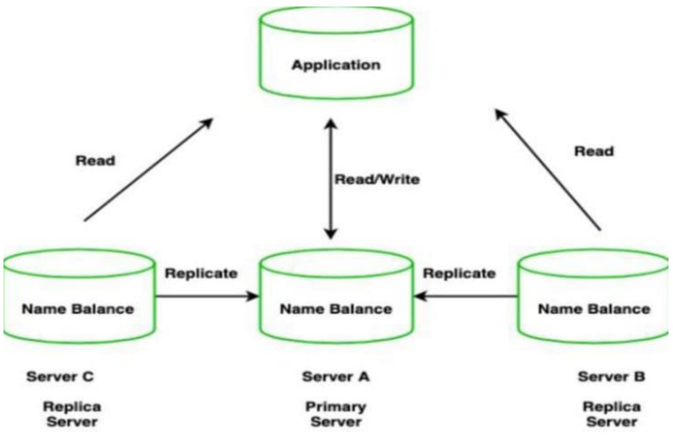
BENEFITS:

• Faster Reads: People can ask the messengers (slaves) for information, reducing the king's workload (improves read performance).

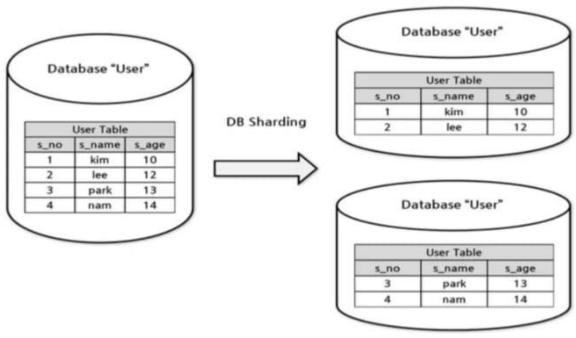
• Backup Plan: If the king falls ill (master fails), the messengers still have copies of the decrees (data backup).

Modern Alternative: Replica Sets

Master-Slave is like the old way, while Replica Sets are the preferred approach for modern data kingdoms (MongoDB deployments).



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

Sharding is a database optimization technique used to distribute a large dataset across multiple servers (shards) for improved scalability and performance.

Benefits of Sharding:

• Scalability: Sharding allows you to add more servers (shards) as your data grows, horizontally scaling your database capacity.

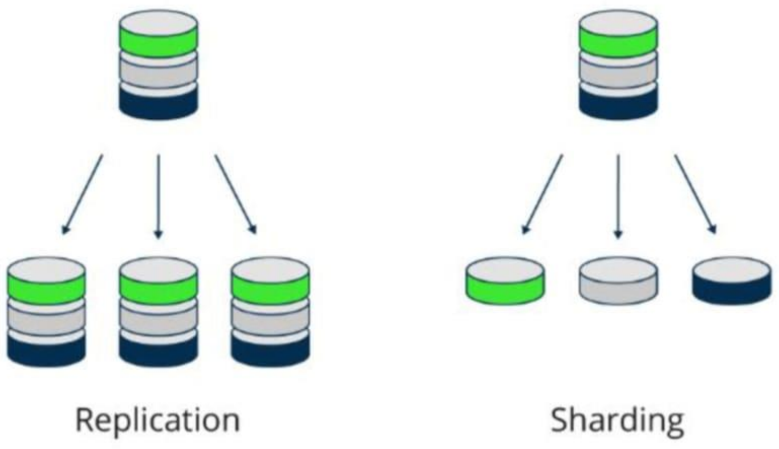
• Improved Performance: Distributing data across multiple servers reduces the load on any single server, leading to faster read and write operations.

Real-World Applications:

• E-commerce Platforms: User data, product information, and order history can be sharded for scalability and faster search experiences.

• Social Media Applications: User profiles, posts, and activity data can be sharded to handle a massive number of users and their interactions.

REPLICATION V/S SHARDING:



• **Sharding:** Consider sharding for very large datasets when horizontal scaling is needed.

• **Replication:** Choose replication for improved read performance, data availability, and disaster recovery, regardless of dataset size.

REPLICATION WITH SHARDING:

Imagine a massive library with countless books. Here's how sharding with replication combines the best of both worlds:

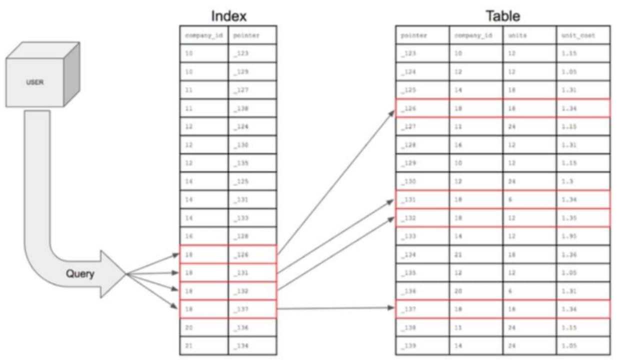
• **Sharding:** Divide the books by genre (shard key) across multiple libraries (shards).

• **Replication:** Within each library (shard), have backup copies of each book (replicas).

This allows for efficient browsing (sharding) and ensures access to books even if one library closes (replication)

Replication and sharding are both powerful techniques for managing databases, but they address different needs. While replication focuses on data redundancy and availability, sharding tackles horizontal scaling for massive datasets. However, in some scenarios, combining these techniques can provide even greater benefits:((Sharded Cluster with Replication)).

<u>CONCEPTS UNDER INDEXES:</u>



TYPES OF INDEXES IN DETAIL:

1.Primary Key Index:

• Definition: A unique identifier for each row in a table. There can only be one primary key per table, and it cannot contain null values.

• Benefits: Enforces data integrity by guaranteeing uniqueness and is automatically used for efficient retrieval based on the primary key.

2. Secondary Index:

• Definition: An index created on a column or set of columns (compound index) other than the primary key. Can have multiple secondary indexes on a table.

• Benefits: Improves query performance when filtering or sorting data based on the indexed columns.

3. Unique Index:

• Definition: Similar to a secondary index, but enforces uniqueness for the indexed column(s). No two rows can have the same value for the indexed column(s).

• Benefits: Ensures data integrity by preventing duplicate values and can also be used for efficient retrieval based on the unique column(s).

4. Clustered Index:

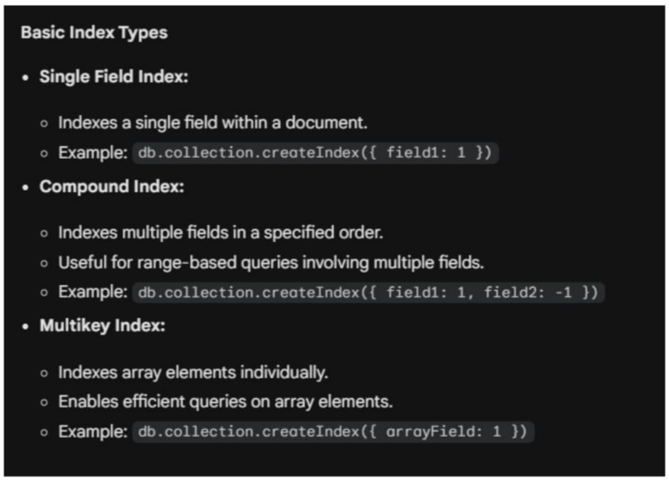
• Definition: A special type of index where the physical order of the data rows in the table is based on the order of the indexed column(s). The data itself is stored in the same order as the index.

• Benefits: Offers the fastest retrieval performance for queries that involve sorting or filtering based on the clustered index column(s). However, only one clustered index can exist per table.

5. Non-Clustered Index:

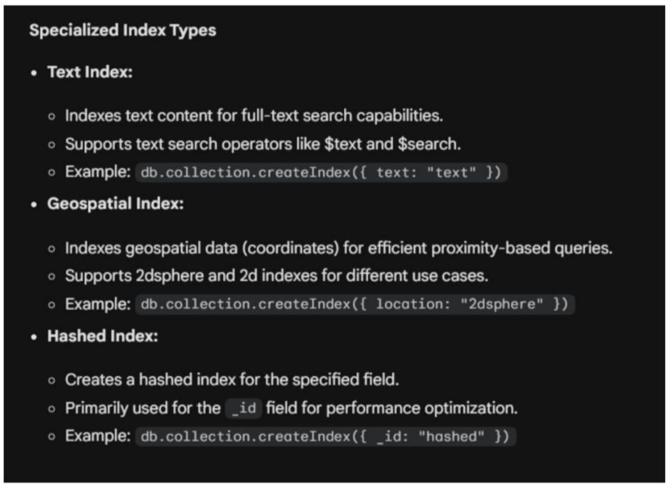
• Definition: The most common type of index. The data rows are not physically stored in the order of the index. The index itself points to the actual data location.

• Benefits: Can be created on multiple columns, and there's no limit on the number of non-clustered indexes per table. Offers good performance for filtering and sorting based on the indexed column(s), but might require an additional step to retrieve the actual data from its physical location.



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 CONSIDERATION:

"SPARES UNIQUE AND TTL INDEX" isn't a standard term used in database indexing. However, let's break down the components and see what it might imply:

1. SPARES:

• This could refer to a table or collection containing information about spare parts in an inventory management system.

2. UNIQUE:

• This suggests an index on a column (or set of columns) that enforces uniqueness. No two rows in the table can have the same value for the indexed column(s). In an inventory context, this could be:

o Part Number: Each spare part likely has a unique identifier to differentiate it from others.

3. TTL (Time To Live):

• This refers to a mechanism that automatically removes data after a specified period.

Refer:

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