Class 8

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.

Here's a breakdown of each ACID property:

1. **Atomicity:**
    a. Ensures an entire transaction either succeeds completely or fails entirely.
    b. Imagine a bank transfer. The funds are either deducted from one account and added to another, or neither happens.
2. **Consistency:**
    a. Maintains the database's state according to pre-defined rules.
    b. A transaction cannot leave the database in an invalid or unexpected state.
3. **Isolation:**
    a. Guarantees that concurrent transactions are isolated from each other. o Changes made by one transaction are invisible to others until the first transaction is completed.
4. **Durability:**
    a. Ensures that once a transaction commits (finishes successfully), the changes are persisted permanently.
    b. Even if a system failure occurs, the committed data remains intact.

Importance of ACID:

ACID properties are crucial for maintaining data integrity and preventing inconsistencies in multi-user database environments. They guarantee reliable data handling and predictable outcomes for database operations.

INDEXES:

Special data structures in MongoDB that speed up queries by organizing data efficiently. They're similar to indexes in books, pointing to specific locations for faster retrieval.

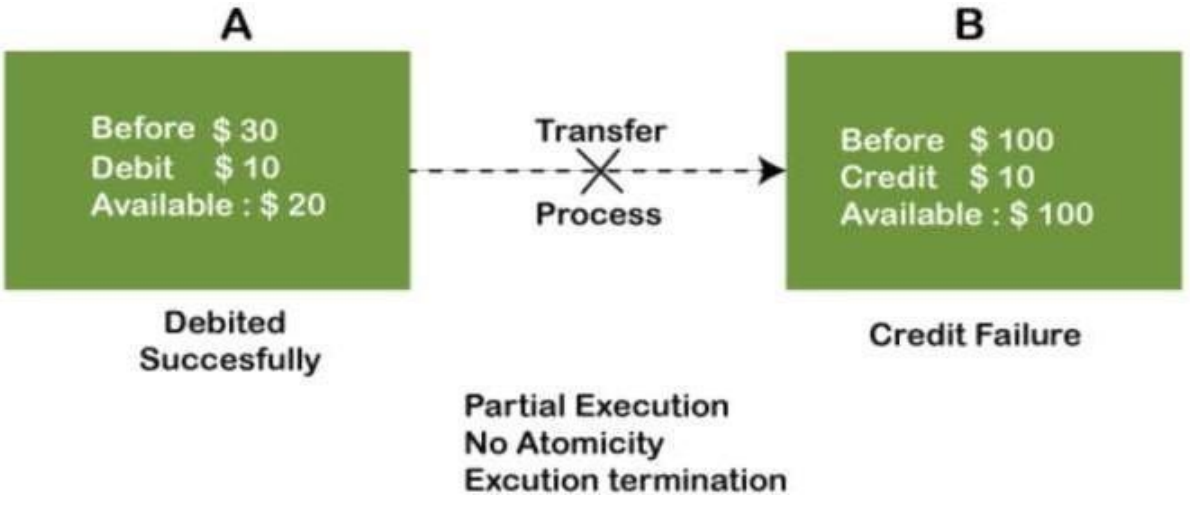
• Index Types:

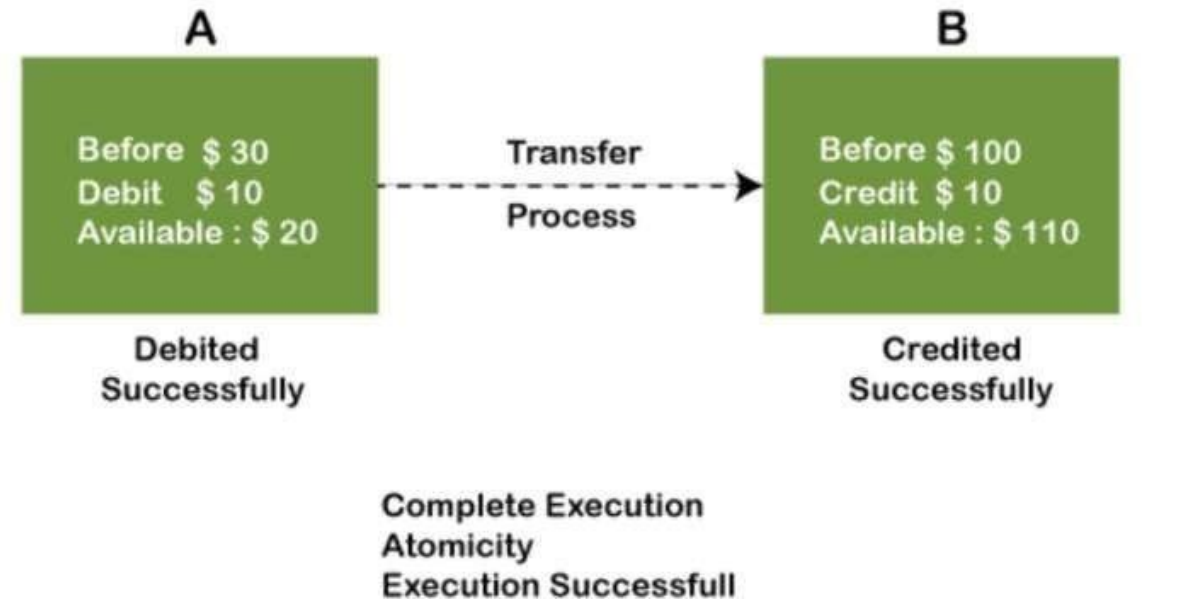
- **Unique:** Enforces uniqueness for a specific field or combination of fields, ensuring no duplicate values.
- **Sparse**: Only indexes documents with the specified field and a value (useful for optional fields).
- **Compound:** Indexes multiple fields together, optimizing queries that involve those fields in combination.
- **Multikey:** Indexes multiple fields within an array, allowing efficient searches within array.

Benefits of Atomicity:

- Data Consistency: Prevents partial updates that could lead to inconsistencies.
- Reliability: Ensures predictable outcomes for transactions, protecting against unexpected errors.
- Data Integrity: Maintains the validity and accuracy of data within the database.

PARTIALEXECUTIONWITHNOATOMICITY:





CONSISTENCY:

In MongoDB, consistency refers to the level of agreement between data stored in the database and reflects how up-to-date and accurate the data is across all nodes in a distributed system. MongoDB offers different levels of consistency based on the chosen configuration.

You can choose between strong consistency and eventual consistency. Strong consistency ensures that all reads reflect the latest write operation, providing the most up-to-date data but potentially impacting performance. On the other hand, eventual

consistency allows for faster reads by not guaranteeing immediate updates across all nodes, leading to a temporary inconsistency that resolves over time.

- **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 predefined constraints and rules. These rules could involve data types, relationships between tables, or specific values allowed for certain fields.

Benefits of Consistency:

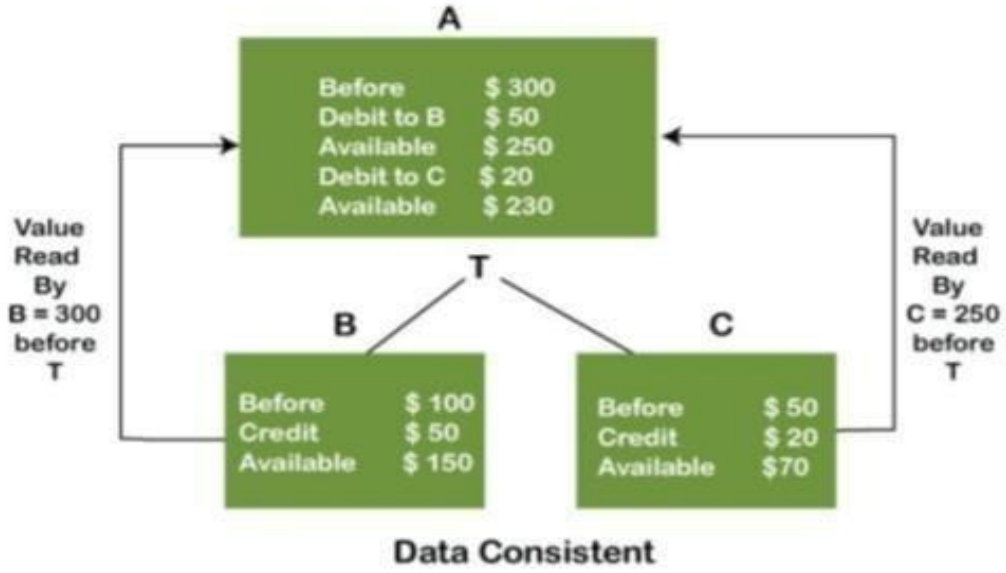
- **Data Accuracy:** Ensures data within the database remains accurate and reflects the real world.
- **Valid State:** Maintains the database in a valid and usable state, preventing inconsistencies that could lead to errors.
- **Predictable Outcomes**: Guarantees that transactions produce expected results, upholding data integrity.

Mechanisms for Consistency:

Databases enforce consistency through various mechanisms:

- **Constraints:** Data types, primary/foreign keys, and other constraints define valid data states.
- **Triggers:** Automated procedures executed before or after specific operations to enforce rules.
- **Cascading Updates/Deletes**: Ensuring related data is updated or deleted consistently when changes are made.

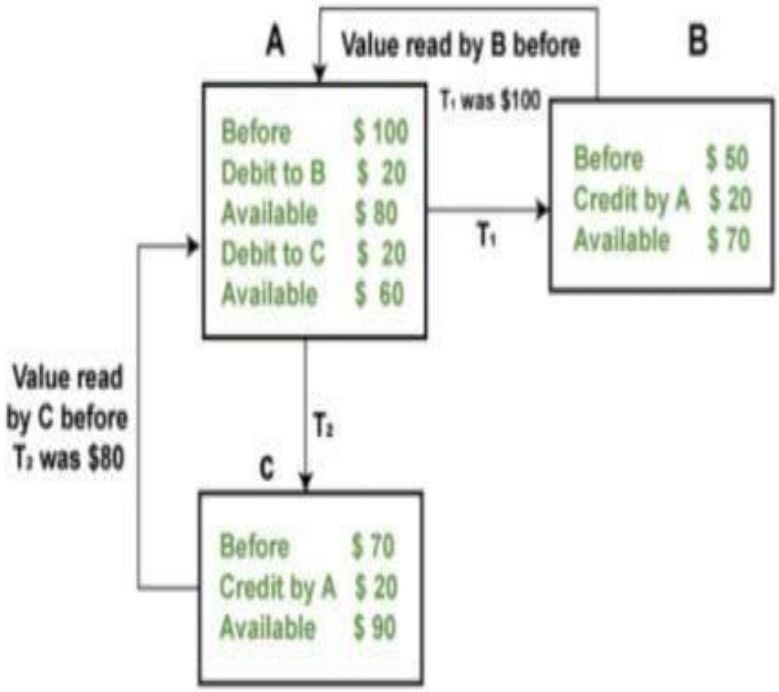
EXAMPLE:



ISOLATION:

Isolation in ACID transactions ensures that concurrent transactions are treated as if they were executed one at a time, even if they happen simultaneously. It prevents data inconsistencies and unexpected results.

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

Benefits of Isolation:

*Data Integrity

*Predictable Outcomes

*Concurrency Control

*Concurrency control

*Prevents data corruption

*Enhanced reliability

REPLECATION(MASTER –SLAVE):

Imagine a kingdom with data as its treasure.

- Master: The king, the single source of truth for all updates (decrees). Everyone looks to the king for the latest information.
- Slaves: Loyal messengers who receive updates from the king and spread them throughout the kingdom (replicate the data). They can also answer questions from the people (handle read requests).

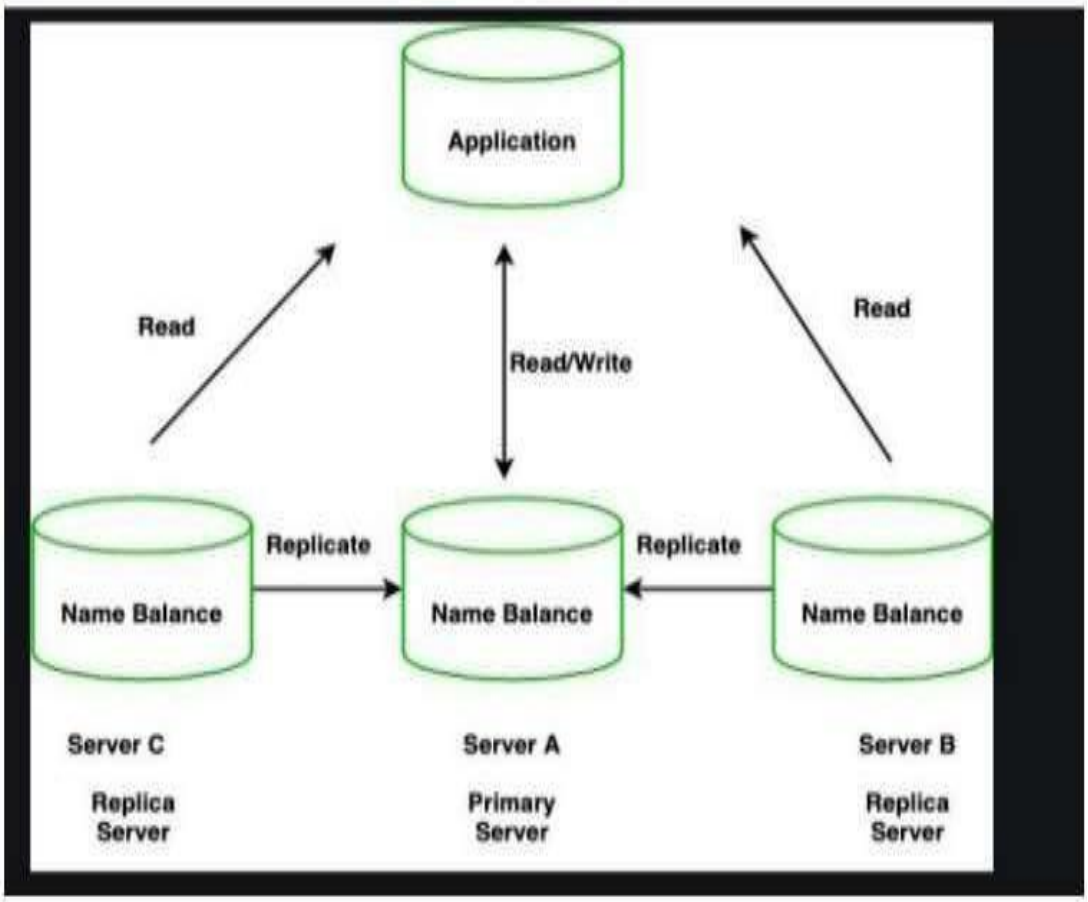
Benefits:

- *High availability
- *Data durability
- *Scalability
- *Disaster recovery
- *Load balancing

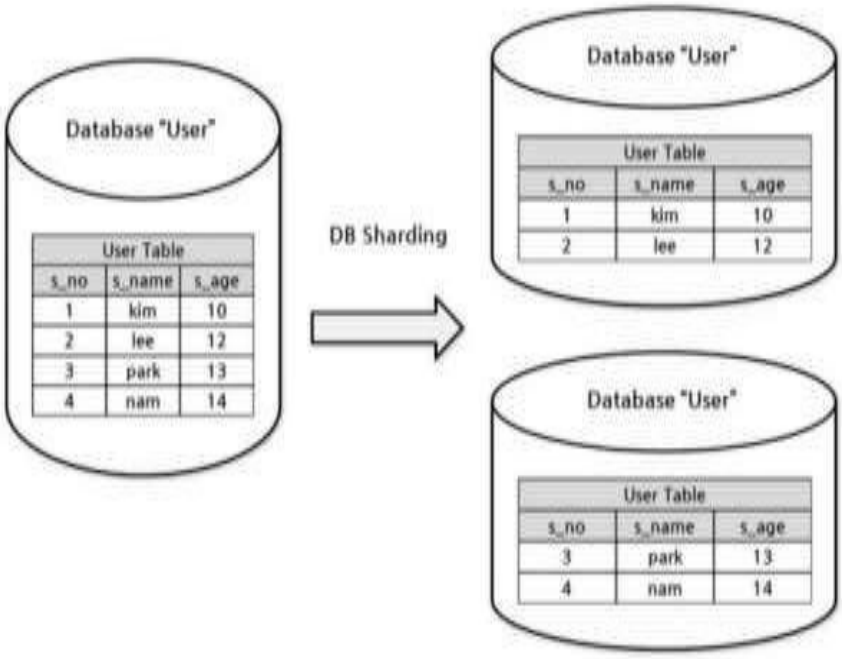
Modern Alternative:

Replica Sets Think of a council of advisors, all with copies of the decrees and the ability to make decisions (write data).

- **Automatic Backup:** If one advisor is unavailable, another can step up to lead (automatic failover).
- **Stronger Kingdom**: The council can handle more updates together (improved write scalability).


Application
Read
Read/Write
Read
Replicate
Replicate
Name Balance
Name Balance
Name Balance
Server C
Server A
Server B
Replica Server
Primary Server
Replica Server

SHARDING:


Database "User"
User Table
s_no | s_name | s_age
1 | kim | 10
2 | lee | 12
3 | park | 13
4 | nam | 14
DB Sharding
Database "User"
User Table
s_no | s_name | s_age
1 | kim | 10
2 | lee | 12
Database "User"
User Table
s_no | s_name | s_age
3 | park | 13
4 | nam | 14

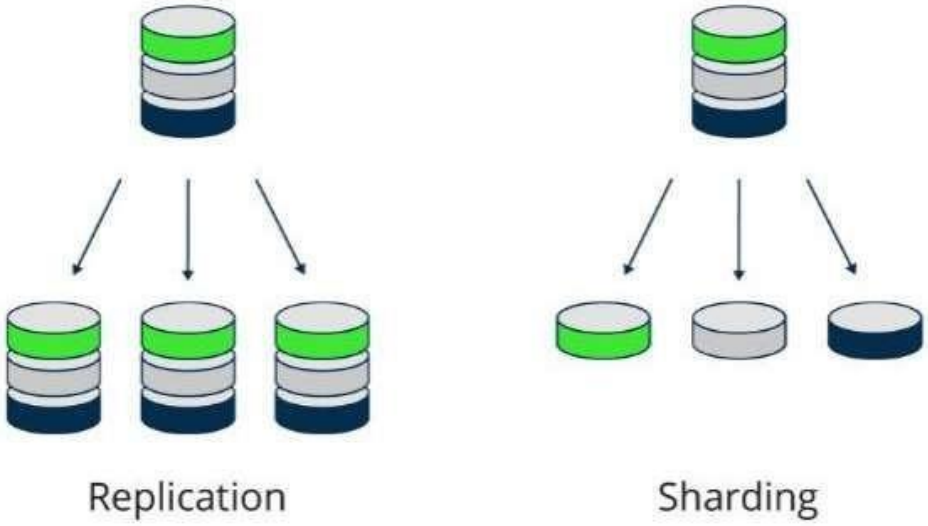
Sharding is a database optimization technique used to distribute a large dataset across multiple servers (shards) for improved scalability and performance. Imagine having a massive library with countless books. Sharding helps organize and access these books efficiently.

- **Large Datasets**: Sharding is particularly beneficial for managing very large datasets that would be cumbersome for a single server to handle.
- **Horizontal Partitioning**: Sharding involves dividing the data horizontally across multiple servers. This means each shard stores a subset of the entire dataset based on a predefined criteria (shard key).
- **Shard Key:** This is a critical element used to determine which shard a particular data record belongs to. Common shard keys include user ID, date range, or geographic location.

Implementation Considerations:

- **Shard Key Selection:** Choosing the right shard key is crucial. It should evenly distribute data across shards and facilitate efficient querying based on frequently used criteria.
- **Query Routing**: When a query needs to access data from multiple shards, the database needs to efficiently route the query to the relevant shards and combine the results.
- **Schema Consistency**: Maintaining consistency across all shards is important. This might involve using techniques like schema replication or distributed transactions.

REPLICATION V/S SHARDING:



ANOLOGY:

- **Sharding**: Imagine a massive library with countless books. Sharding divides the books by genre (shard key) across multiple shelves (shards). This allows faster browsing and easier access to specific genres.
- **Replication:** Imagine having copies of your favorite books at home and your office (replicas). This ensures you can access the book even if one location is unavailable.

REPLICATION:

replication is the process of synchronizing data across multiple nodes to ensure data availability, reliability, and fault tolerance. Replication works by maintaining copies of the data on different nodes, where one node acts as the primary (master) and the others as secondary (slave) nodes.

SHARDING:

Sharding in MongoDB is a method of distributing data across multiple machines to improve scalability and performance. It involves partitioning data into smaller chunks called shards and storing these chunks on different servers or nodes

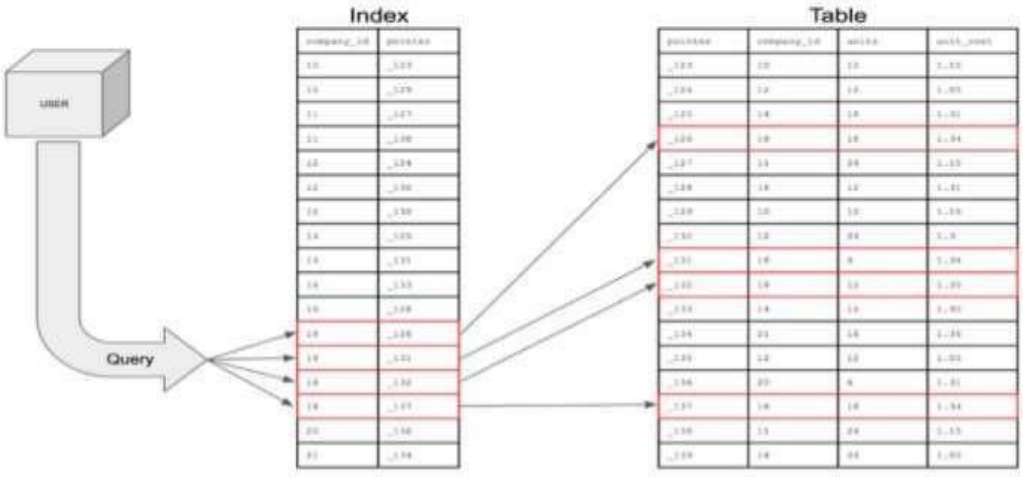
REPLICATION WITH SHARDING:

- **Analogy:** 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**))

CONCEPTS UNDER INDEXES:



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.

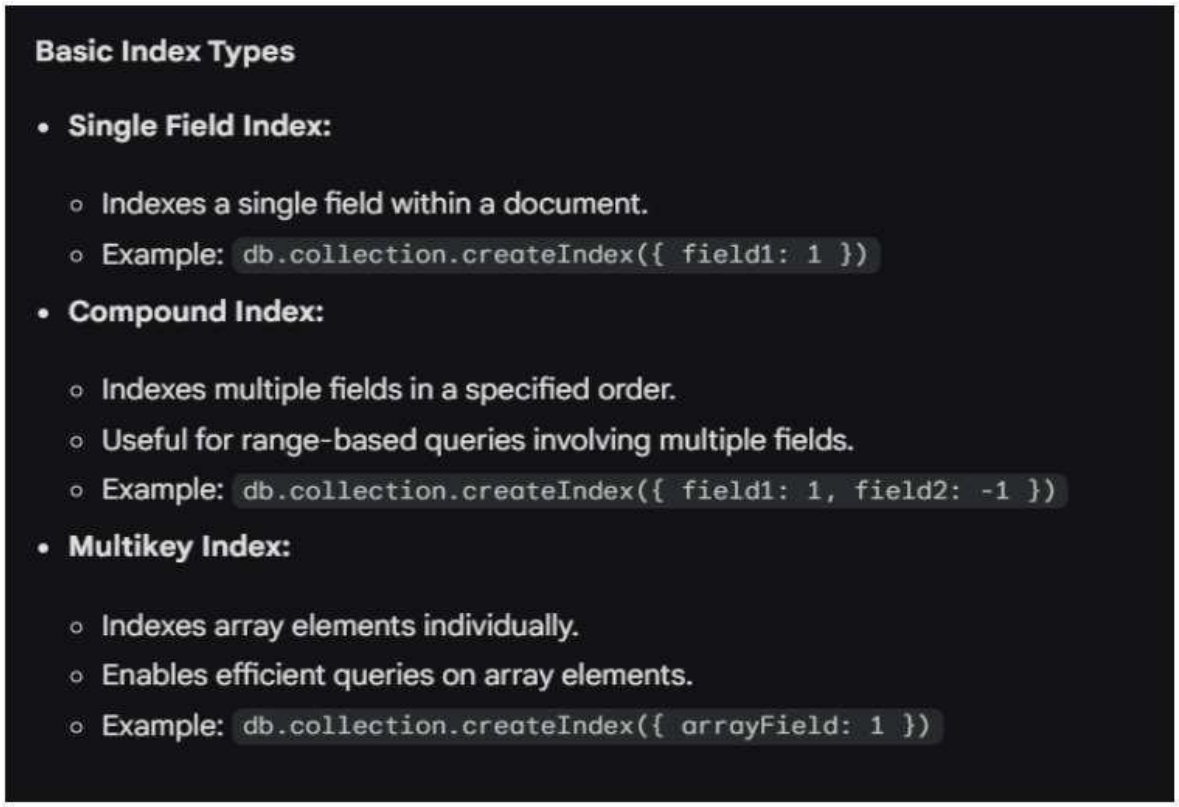
6. Sparse Index:

- **Definition:** An index that only includes rows where the indexed column has a value. This can be useful for columns that allow null values and where a significant portion of the data might be null
- **Benefits:** Saves storage space compared to a regular index, but might not improve query performance as effectively if a large portion of the data has null values in the indexed column.

7. Covering Index:

- **Definition**: A special type of index that includes all the columns needed to satisfy a specific query without requiring access to the actual data table.
- **Benefits**: Can significantly improve query performance if the index covers all the columns used in the WHERE clause, SELECT clause, and any JOIN conditions.

By understanding the different types of indexes and choosing them strategically, you can significantly optimize the performance of your database queries.

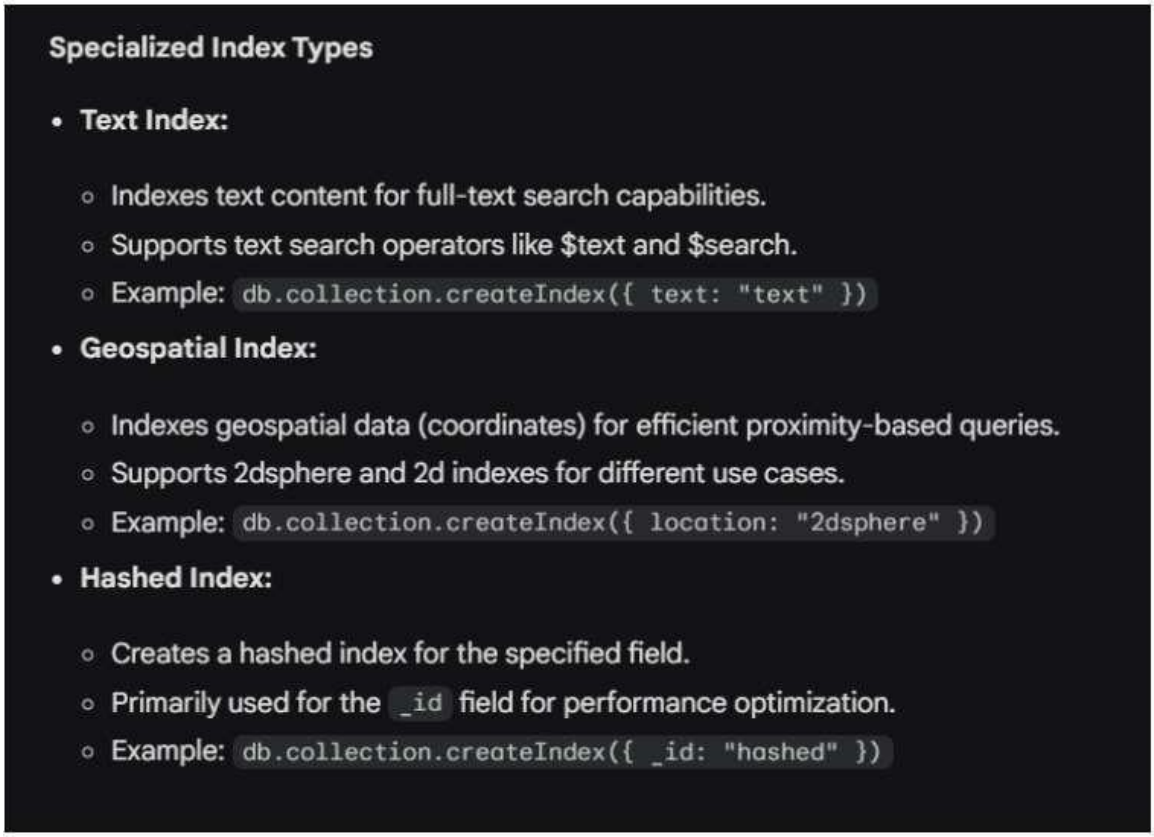


SPECILISED INDEX TYPE: (Specialized):

Full-Text Search: Efficiently search text data within columns (e.g., product descriptions).

Geospatial: Optimized for queries involving geographic data (e.g., finding nearby restaurants).

Hash: Speeds up equality checks on specific columns (e.g., user login credentials).



ADDITIONAL CONSIDERATION:

Based on the information we have so far, "SPARES UNIQUE AND TTL INDEX" isn't a standard term used in database indexing.

1. **SPARES INDEX:**
   - This could refer to a table or collection containing information about spare parts in an inventory management system.

2. **UNIQUE INDEX:**

   • 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:

3. **TTL INDEX (Time To Live):**

   • This refers to a mechanism that automatically removes data after a specified period. In an inventory system, it might be used for