ACIDS AND INDEXES:

AGENDA:

- **https://www.mongodb.com/resources/basics/databases/acid-transactions**
- **Indexes?**
- **Demonstrate creation of different types of indexes on collection (unique, sparse, compound and multikey indexes)**
- **Demonstrate optimization of queries using 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:**
    - Ensures an entire transaction either succeeds completely or fails entirely.
    - Imagine a bank transfer. The funds are either deducted from one account and added to another, or neither happens.
2. **Consistency:**
    - Maintains the database's state according to pre-defined rules.
    - A transaction cannot leave the database in an invalid or unexpected state.
3. **Isolation:**
    - Guarantees that concurrent transactions are isolated from each other.
    - Changes made by one transaction are invisible to others until the first transaction is completed.
4. **Durability:**
    - Ensures that once a transaction commits (finishes successfully), the changes are persisted permanently.
    - 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.

Examples of ACID vs. Non-ACID:

- An ACID transaction might involve updating an inventory database. It either successfully reduces the stock for a purchased item or fails entirely, preventing partial updates.
- A non-ACID system might allow partial updates, leading to inconsistencies. Imagine a purchase failing after deducting stock but before updating the customer's order status.

WHAT IS INDEXES?

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 arrays.

• **Optimizing Queries:** Proper indexing can significantly improve query performance. By creating indexes on fields used in query filters and sorting, MongoDB can quickly locate relevant documents.

CONCEPTS UNDER ACIDS:

1.ATOMICITY:

Delving deeper into Atomicity:

Atomicity in ACID transactions ensures that a transaction, which is a series of database operations treated as a single unit, is indivisible. It guarantees all-or-nothing behavior. Here's a breakdown:

- **Single Unit:** A transaction is viewed as one atomic unit. The database considers it a single operation for success or failure.
- **All-or-Nothing:** Either all the operations within the transaction are completed successfully, updating the database as intended, or none of them are applied. This prevents partially completed transactions that could leave the database in an inconsistent state.

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.

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.

Underlying Mechanisms:

Databases implement atomicity using techniques like transaction logs and rollback mechanisms. These logs track the changes made within a transaction, allowing the database to undo any modifications if the transaction fails.

Example (without Atomicity):

- Imagine a shopping cart transaction without atomicity.
- Items are added to the cart (decreased inventory).
- But the payment fails due to insufficient funds.
- Without atomicity, the inventory might remain decreased even though the purchase wasn't completed, leading to inconsistencies.

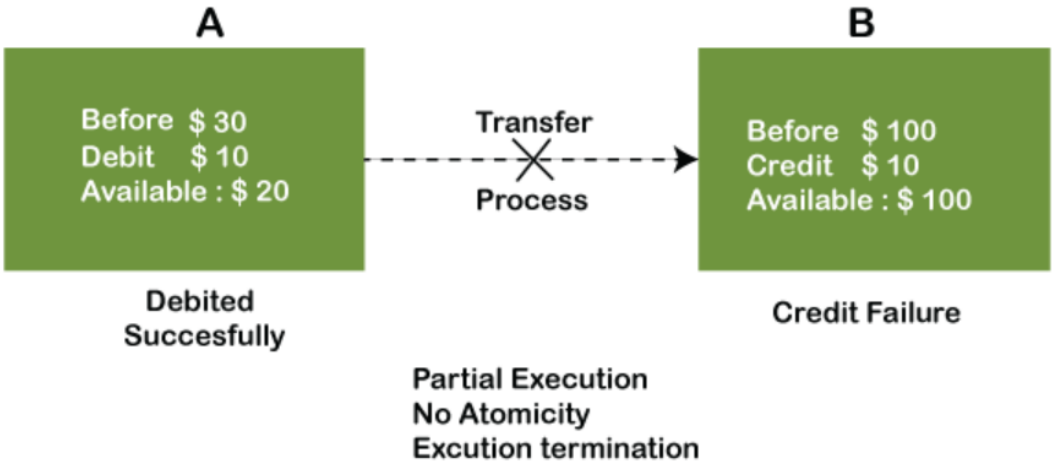
ACID and Real-World Applications:

Atomicity is crucial in various scenarios:

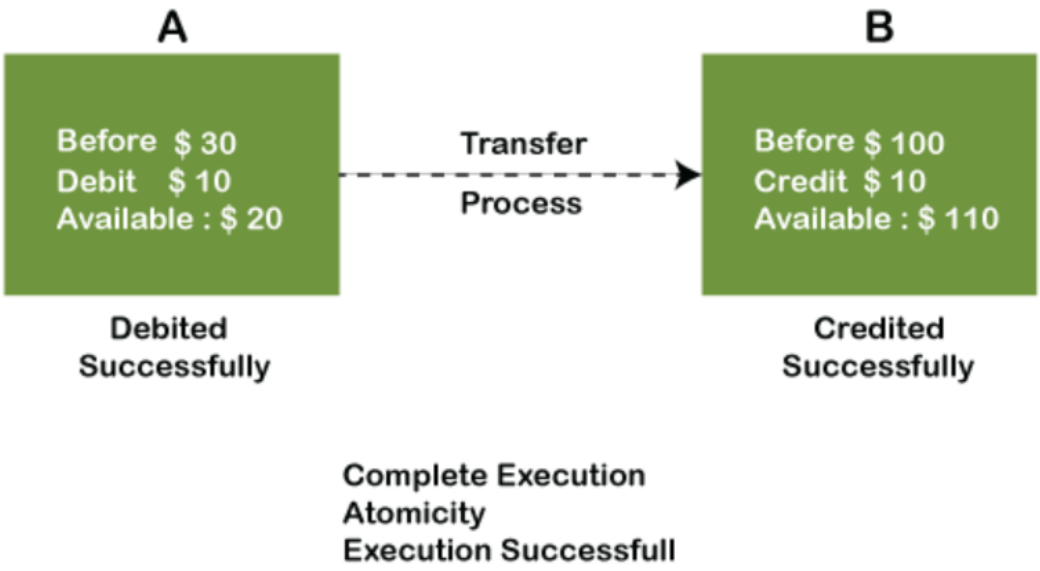
- **Financial transactions:** Guaranteeing successful transfers or rollbacks.
- **Inventory management:** Maintaining accurate stock levels after successful purchases.
- **Data synchronization:** Ensuring consistent updates across multiple databases.

By upholding atomicity, databases ensure reliable data manipulation and a predictable environment for critical operations.

PARTIAL EXECUTION WITH NO ATOMICITY:



COMPLETE EXECUTION WITH ATOMICITY:



2.CONSISTENCY:

Consistency in ACID transactions guarantees that a database adheres to pre-defined rules and maintains a valid state after each transaction. It ensures the data remains consistent and reflects a logical, expected state.

Here's a breakdown of the concept:

- **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.

Imagine a recipe database:

- Consistency prevents adding ingredients to a non-existent dish (data integrity).
- It might also enforce constraints like quantities or ingredient types (predefined rules).

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 (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.

ACID and Real-World Applications:

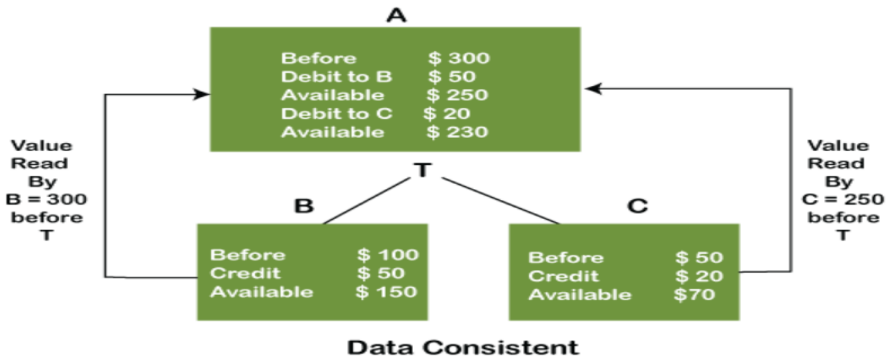
Consistency is essential in various scenarios:

- **Financial databases:** Maintaining accurate account balances and transaction histories.
- **Inventory management:** Ensuring stock levels reflect actual inventory after sales.
- **Healthcare records:** Guaranteeing accurate and consistent patient information.

By upholding consistency, ACID transactions ensure reliable data manipulation and a predictable environment for critical applications.

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

EVENTUAL CONSISTENCY:

Eventual consistency is a data consistency model commonly used in distributed systems, particularly NoSQL databases. It prioritizes availability and scalability over immediate data synchronization across all nodes in the system.

Here's the gist:

- **Updates are replicated across multiple servers (nodes) for redundancy and fault tolerance.**
- **However, there might be a slight delay in propagating updates to all nodes.**
- **The system eventually converges, meaning all nodes will eventually reflect the same data.**

Imagine a geographically distributed database system:

- A user updates their profile on a server in London.
- This update needs to be replicated to servers in New York, Tokyo, etc.
- Eventual consistency guarantees that eventually, all servers will have the same updated profile information, but there might be a temporary lag.

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.

Trade-offs to consider:

- **Stale reads:** You might occasionally read outdated data if a node hasn't received the latest update yet.
- **Complex logic:** Applications need to be designed to handle potential inconsistencies during the convergence period.

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.

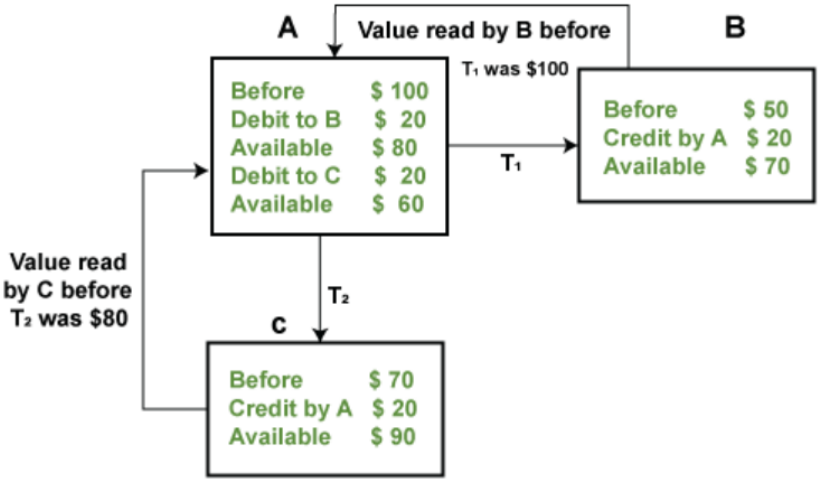
Eventual consistency vs. Strong consistency:

- **Strong consistency:** Guarantees that all reads reflect the latest updates across all nodes at any given time. This offers stricter consistency but can impact performance and scalability.

The choice between eventual and strong consistency depends on the specific needs of your application. If high availability and scalability are crucial, and slight data delays are acceptable, eventual consistency can be a good option. However, for applications where data accuracy is paramount, strong consistency might be more suitable.

3.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

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.

Here's a breakdown:

- **Concurrent Transactions:** Multiple transactions can access and modify the database at the same time.
- **Isolation Guarantees:** Isolation ensures that each transaction appears to be executed in isolation from others.
    - Changes made by one transaction are invisible to other transactions until the first transaction is completed (committed).

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.
- **Concurrency Control:** Enables multiple users to access and modify the database concurrently without compromising data integrity.

Isolation Levels: Databases offer different isolation levels, defining the degree of interaction allowed between transactions. These range from Read Uncommitted (least restrictive) to Serializable (strictest).

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:

- **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).

Drawbacks:

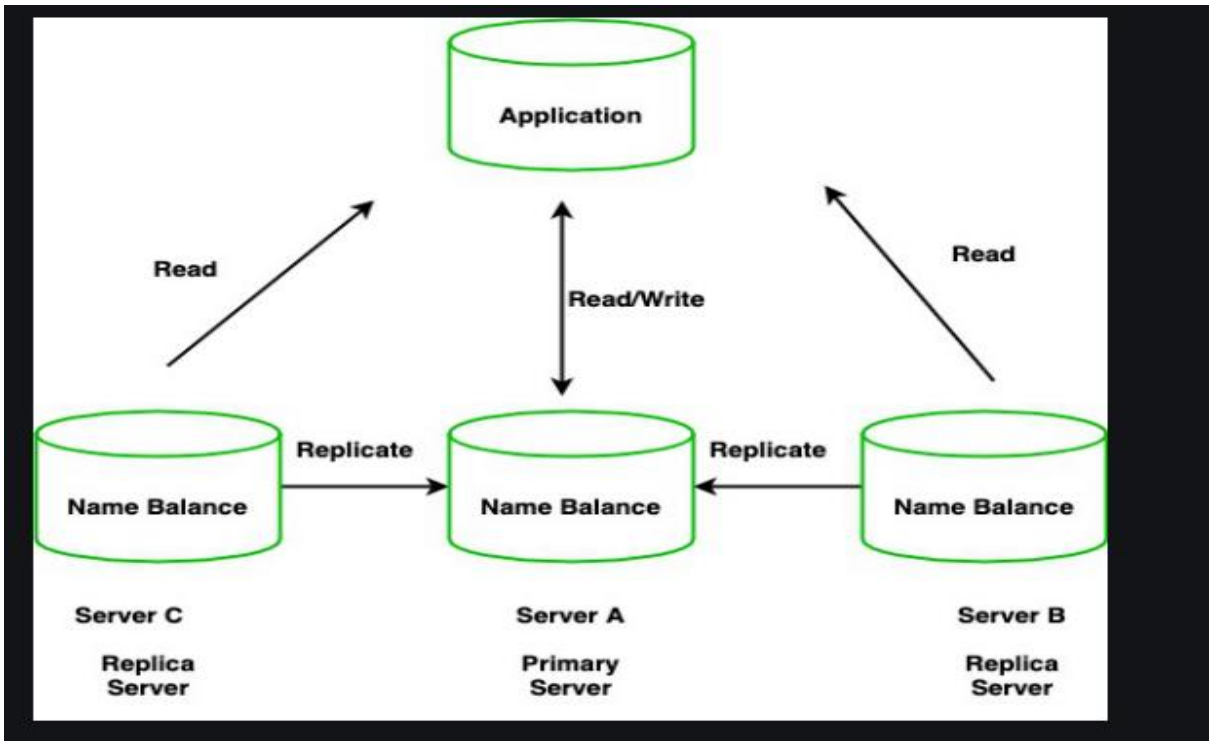
- **Fragile King:** If something happens to the king, there's a delay in choosing a new leader (manual failover).
- **Limited Growth:** The king can only handle so many updates himself (limited write scalability).

Modern Alternative: Replica Sets

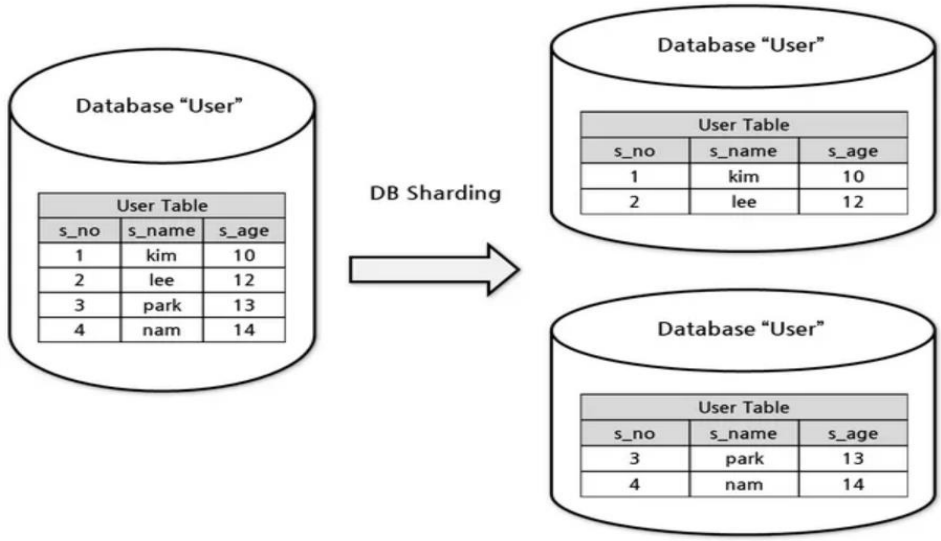
Think of a council of advisors, all with copies of the decrees and the ability to make decisions (write data).

- **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).

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



SHARDING:



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.

Here's a breakdown of the concept:

- **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.

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.
- **High Availability:** If one shard becomes unavailable, other shards remain operational, minimizing downtime and improving data availability.

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.

Real-World Applications:

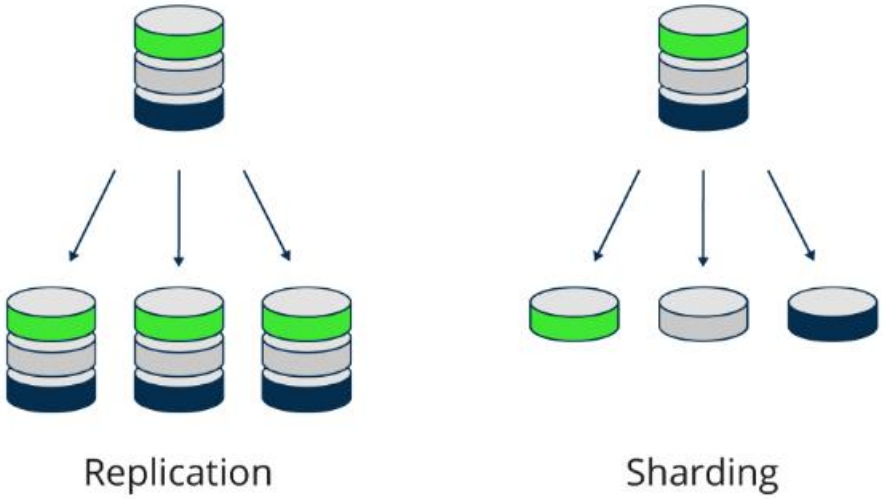
Sharding is used in various scenarios where large datasets need to be managed efficiently:

- **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.
- **Content Management Systems:** Sharding can be used to manage vast amounts of website content and user data efficiently.

Remember: Sharding introduces additional complexity in managing a database, but for very large datasets, the benefits of scalability and performance often outweigh the overhead. It's

crucial to carefully evaluate your specific needs and data volume before implementing sharding in your system.

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.

Choosing the Right Technique:

- **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:

Analogy:

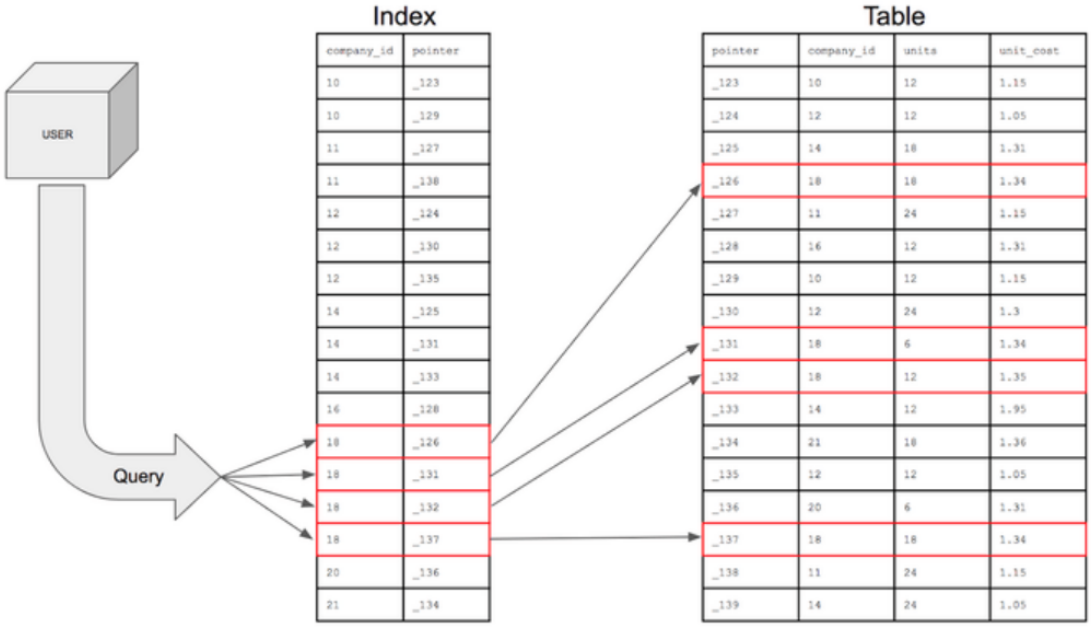
Imagine a massive library with countless books. Here's how sharding with replication combines the best of both worlds:

- **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:

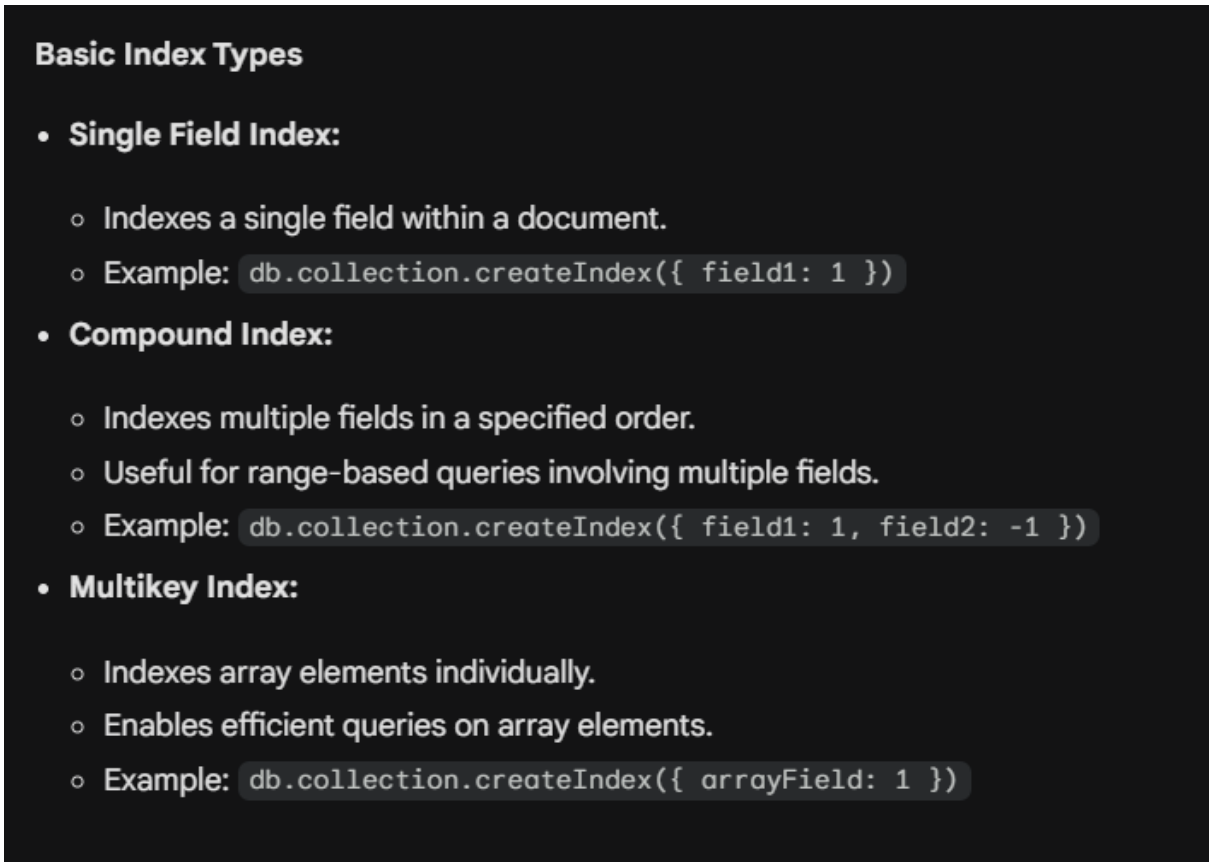
- **Definition:** A special type of index that includes all the columns needed to satisfy a specific query without requiring access to the actual data table.
- **Benefits:** Can significantly improve query performance if the index covers all the columns used in the WHERE clause, SELECT clause, and any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

Choosing the Right Index:

The choice of index type depends on your specific needs and query patterns. Consider the following factors:

- **Query types:** Frequently used filters, sorting criteria, and join conditions.
- **Data distribution:** How data is distributed within the indexed column(s).
- **Storage space:** Trade-offs between performance gains and storage overhead.

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



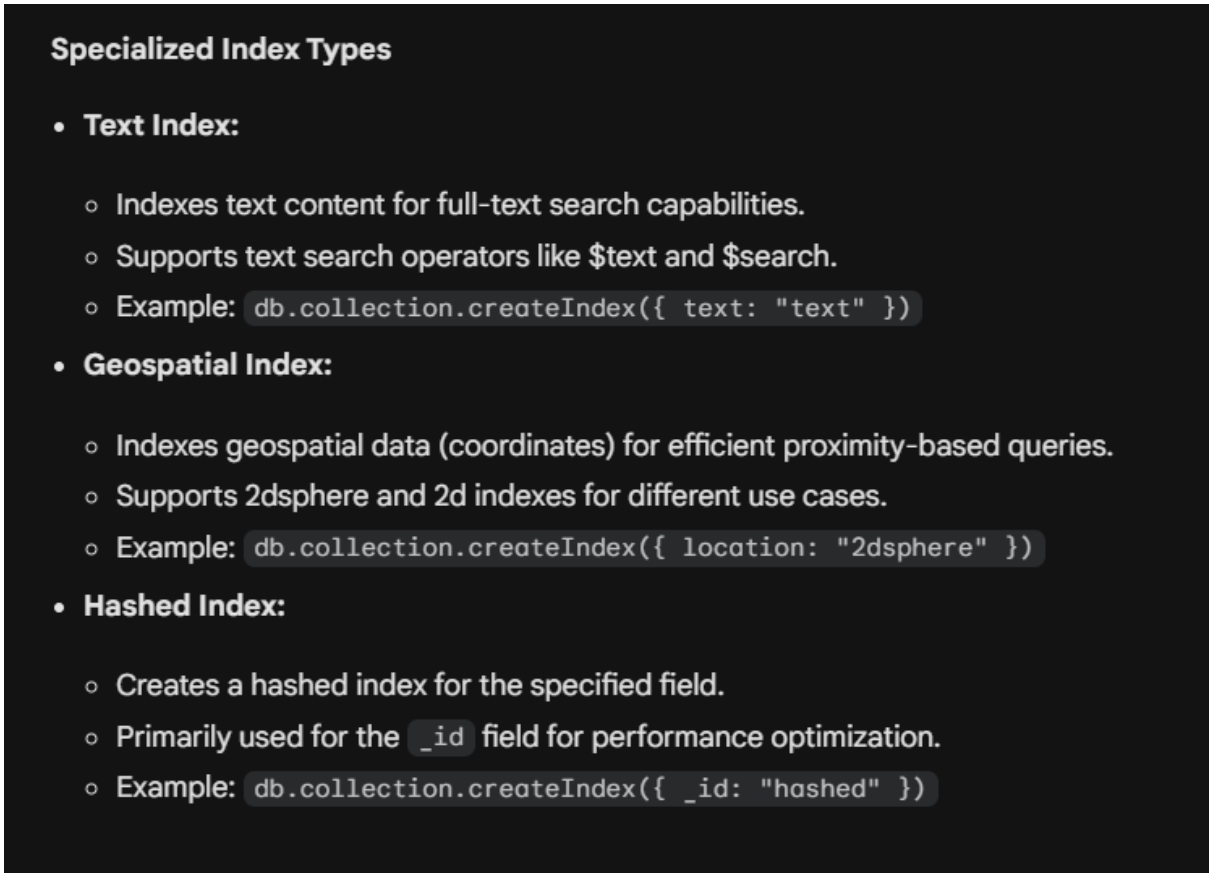
SPECILISED INDEX TYPE:

(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).

Choosing the Right Index:

- **Query Patterns:** Consider frequently used filters, sorting criteria, and joins.
- **Data Distribution:** How data is spread out within the indexed columns.
- **Storage Space:** Balance performance gains with storage needs.



ADDITIONAL CONSIDERATION:

Based on the information we have so far, "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:
  - **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. In an inventory system, it might be used for:

- **Obsolete Parts:** Automatically removing records for spare parts that are no longer used or have been replaced by newer versions.

Possible Interpretation:

The term "SPARES UNIQUE AND TTL INDEX" could potentially indicate a special type of index within an inventory management system. This index might be:

- Created on the **Part Number** column in the "SPARES" table.
- Ensures that each spare part has a unique identifier (enforced by the UNIQUE constraint).
- Potentially combined with a Time-To-Live (TTL) mechanism on the same index. This means rows with outdated or unused parts (identified by the unique part number) might be automatically removed after a certain timeframe.

Important Note:

This is just a possible interpretation based on the limited information. The actual functionality of such an index would depend on the specific software or database system being used. It's best to consult the system's documentation or contact the software vendor for a definitive explanation.

If you can share more context about where you encountered this term, I might be able to provide a more specific interpretation.

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