ACID AND INDEXES

ACIDS:

The term ACID stands for Atomicity, Consistency, Isolation, and Durability. MongoDB ACID transactions are fundamental for ensuring data integrity in database transactions. ACID properties play a crucial role in maintaining the reliability and consistency of data operations.

ATOMICITY: Atomicity in database operations ensures that all changes within a single operation are either fully applied or not applied at all, thus maintaining data integrity. In MongoDB, atomicity is achieved through mechanisms like transactions and single-document operations.

CONSISTENCY: Maintaining data integrity. Each transaction transitions the database from one valid state to another, adhering to any defined data constraints. This ensures data always reflects a consistent and logical structure.

ISOLATION: Current transaction protection. Multiple transactions happening at the same time won't interfere with each other's data. MongoDB achieves this using mechanisms like read-after-write locks, ensuring each transaction operates on a consistent snapshot of the data.

DURABILITY: Committed changes survive. Once a transaction commits successfully, those changes are persisted to the database and won't be lost even in case of system failures like crashes. This guarantees data integrity and prevents data loss.

FEW CONCEPTS:

1. ATOMICITY:

MongoDB, atomicity refers to the indivisibility of write operations, specifically at the document level. It guarantees all-or-nothing behavior for updates on a single document:

All or Nothing: When performing CRUD (Create, Read, Update, Delete) operations on a single document, either all the changes succeed or none of them do. This ensures data consistency within a document.

Limited Scope: Atomicity applies only to individual documents. If your operation involves multiple documents, they might be updated independently, and some might succeed while others fail.

Transactions for Complex Updates: For operations requiring changes across multiple documents while ensuring atomicity, you'll need to use MongoDB transactions. These are different from single-document atomic operations and have their own functionalities.

A) PARTIAL EXECUTION WITH NO ATOMICITY:

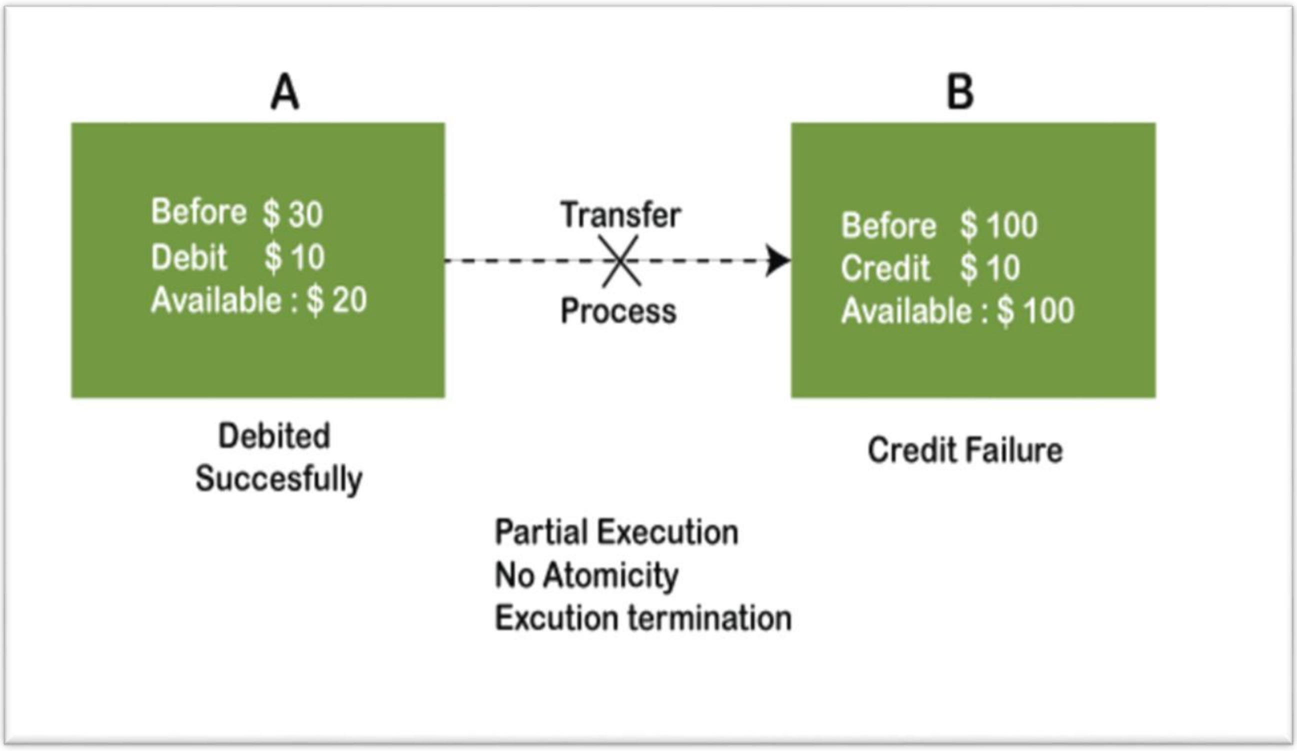
MongoDB, partial execution with no atomicity refers to a situation where updates to multiple documents are attempted, but due to the lack of atomicity guarantees across those documents, some updates might succeed while others fail.

Here's a breakdown:

→ Atomicity Refresher: Remember, atomicity ensures all-or-nothing execution. In a database context, it means either all operations within a transaction are completed successfully, or none are applied. This prevents inconsistencies.

→ No Atomicity for Multiple Documents: MongoDB offers atomicity at the document level. This means individual CRUD operations on a single document are guaranteed to succeed or fail entirely.

However, it doesn't guarantee atomicity for updates involving multiple documents.



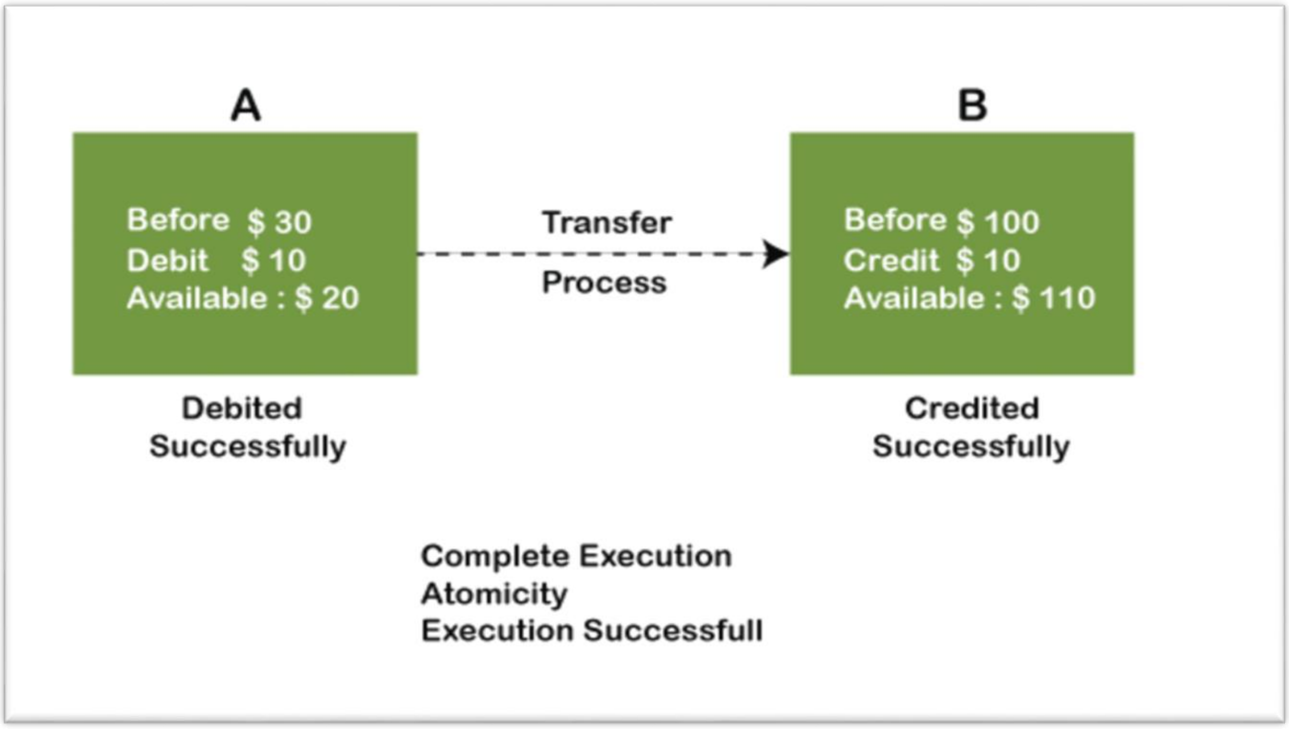
B) COMPLETE EXECUTION WITH ATOMICITY:

In MongoDB, complete execution with atomicity refers to updates happening exactly as planned, all at once, or not at all. This ensures data consistency, especially when modifying multiple documents as a single logical unit.

Here's a breakdown:

→ Atomicity Refresher: Atomicity guarantees all-or-nothing behavior for write operations. In MongoDB, this applies to individual documents. Either all changes within a single document update (CRUD) succeed or none do.

→ Complete Execution: When complete execution with atomicity occurs, all intended updates are applied successfully. This is crucial for maintaining data integrity, especially when modifications involve multiple documents.



2. CONSISTENCY:

Consistency in MongoDB refers to how data across multiple copies (replicas) in a cluster remains synchronized after a write operation. Unlike traditional relational databases that guarantee strict consistency (all replicas reflect the latest write immediately), MongoDB offers a balance between consistency, availability, and partition tolerance.

Here's a breakdown of MongoDB's approach to consistency:

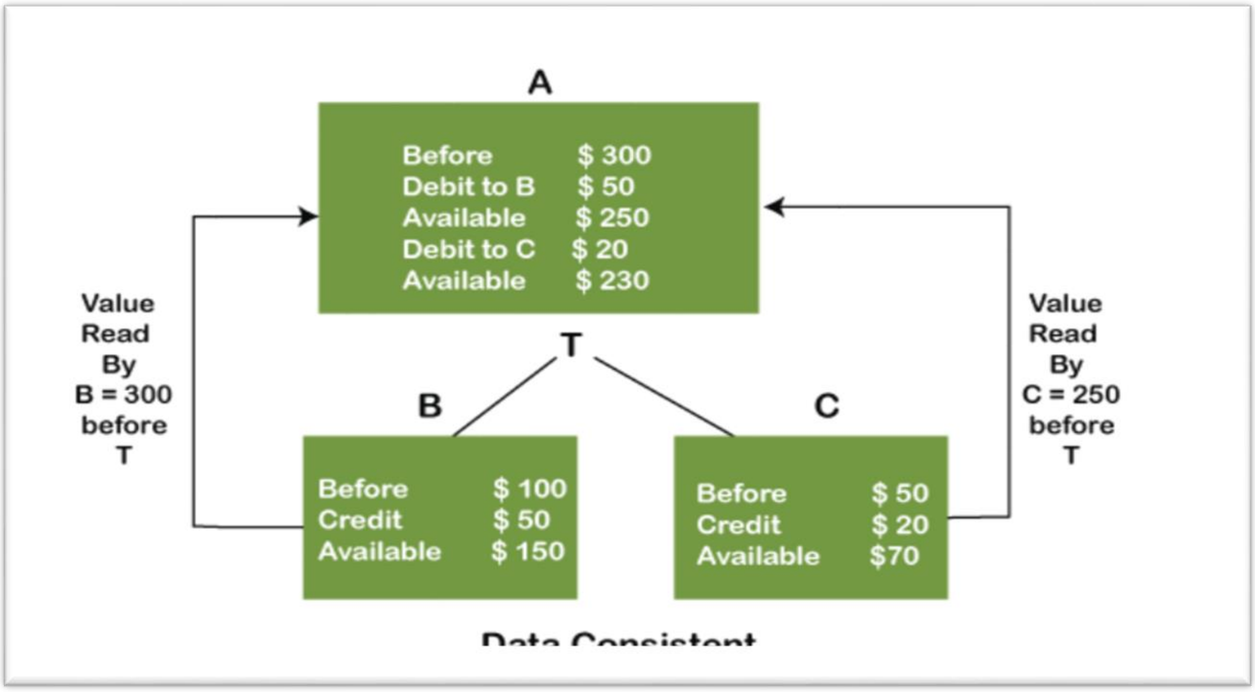
Eventual Consistency: This is the core principle. Writes are initially applied to a primary replica and then asynchronously replicated to secondary replicas. Eventually, all replicas will have the same data, but there might be a slight lag between them.

Immediate Consistency: MongoDB prioritizes availability. Reads directed to the primary replica typically reflect the latest writes.

Read Preferences: You can configure how consistent your reads are by specifying read preferences. Options like "majority" ensure reading from a majority of replicas, offering stronger consistency guarantees but potentially impacting performance.

Causal Consistency: Introduced in MongoDB 3.6, causal consistency ensures that causally related operations within a client session are reflected in the expected order across reads. This requires specific session management by your application.

EXAMPLE:



➢ Eventual consistency:

Eventual consistency in MongoDB ensures that all replicas (copies) of the database in a cluster will eventually reflect the same data after a write, but there might be a temporary lag between them.

This means:

Writes are first applied to the primary replica.

The primary then asynchronously replicates the changes to secondary replicas.

This asynchronous process can introduce a brief delay before the update is reflected on all replicas.

Benefits of Eventual Consistency:

Enhanced Availability: Even during network partitions or replica failures, the primary remains available for reads and writes, ensuring system uptime.

Reduced Write Latency: By not waiting for all replicas to synchronize, MongoDB minimizes write latency, leading to faster write performance for your application.

Scalability Made Easy: Adding new replicas doesn't affect write operations on the primary, simplifying horizontal scaling of your MongoDB deployment.

3. ISOLATION:

There are two main contexts for isolation in MongoDB:

1) <u>Read Isolation:</u>

This refers to the level of consistency between reads and writes happening concurrently in the database. MongoDB offers different read concern levels that determine what data a read operation sees:

Read Uncommitted (default): Reads data that may not reflect the latest changes (not a point-in-time snapshot). This is the least isolated level.

Read Committed: Reads data committed before the read operation started. Provides some isolation but may miss recent writes.

Snapshot: Reads a consistent view of the data as of a specific point in time. Offers strong isolation but can impact performance.

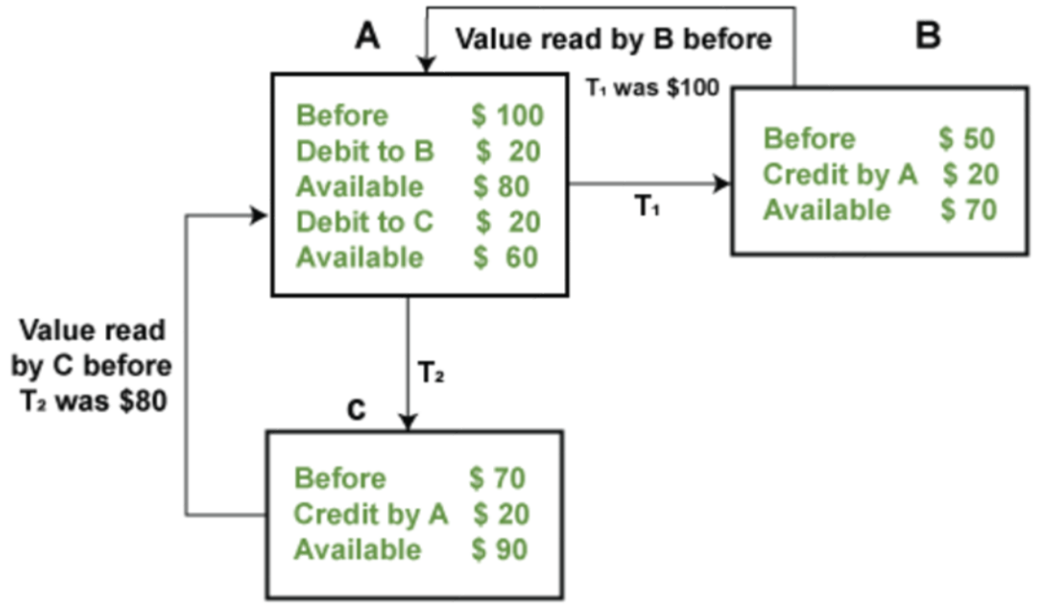
<u>2)Workload Isolation:</u>

This focuses on separating workloads within a MongoDB deployment for better performance and manageability. MongoDB provides features like:

- Shard Zones: Distributes data across shards based on a defined range, isolating workloads by data location.

- Tiered Hardware: Assigns workloads to specific hardware based on their resource requirements.
- Replica Set Member Control: Directs reads and writes to specific members within a replica set for functional separation.

The choice of isolation level depends on your specific needs. For reads where the latest data isn't critical, prioritizing performance with read uncommitted might be suitable. For data consistency and integrity, read committed or snapshot isolation offer stronger guarantees. Workload isolation helps manage complex deployments by separating different application workloads or geographically distributed data.



Isolation - Independent execution of T_1 & T_2 by A

➢ REPLICATION:

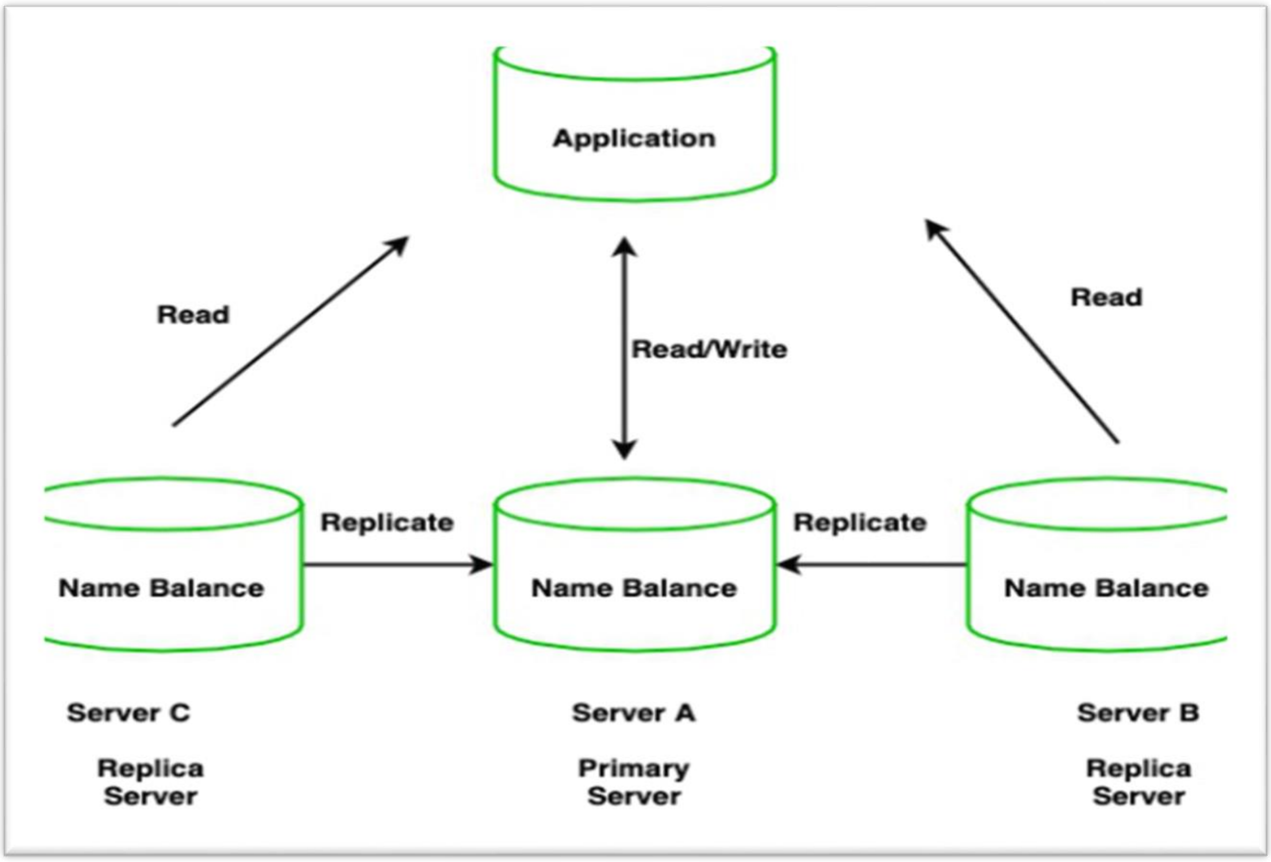
Replication in MongoDB refers to the process of creating and maintaining multiple copies of your data across different servers. This is achieved through a group of MongoDB instances called a replica set. Replica sets provide several key benefits:

- Redundancy and High Availability: With multiple copies of your data, you're protected from hardware failures or outages on a single server. If the primary server fails, one of the secondary servers can automatically take over, minimizing downtime and ensuring continued data access.
- Scalability: Replica sets can be scaled by adding more secondary servers. This can improve read performance by distributing read queries across multiple servers.
- Disaster Recovery: Geographically dispersed replica sets can be used for disaster recovery purposes. In case of a major outage at one location, the replica set at the other location can be used to restore data and operations.

Here's a breakdown of how replication works in a MongoDB replica set:

- Replica Set Members: A replica set consists of multiple members, typically at least three for fault tolerance. There are three main member roles:
- Primary: The primary node is the sole member that accepts write operations. It replicates these operations to the secondary nodes.
- Secondary: Secondary nodes maintain a copy of the data from the primary and apply the received updates. They can also be used for read operations (depending on configuration).
- Arbiter (Optional): An arbiter node doesn't store data but helps resolve conflicts during elections for a new primary if the existing primary becomes unavailable.

- Replication Process: The primary node keeps a record of all writes in an operation log (oplog). Secondary nodes continuously replicate the oplog from the primary and apply the operations to their own data. This asynchronous replication allows the secondary to stay up-to-date without impacting the performance of the primary.
- Failover: If the primary fails, a replica set election is held among the remaining members to choose a new primary. This ensures minimal downtime and continued data availability.



➢ SHARDING:

In MongoDB, sharding addresses the challenge of managing extremely large datasets or high-throughput operations on a single server. It essentially involves splitting your data across multiple machines for better scalability and performance.

Sharding Process:

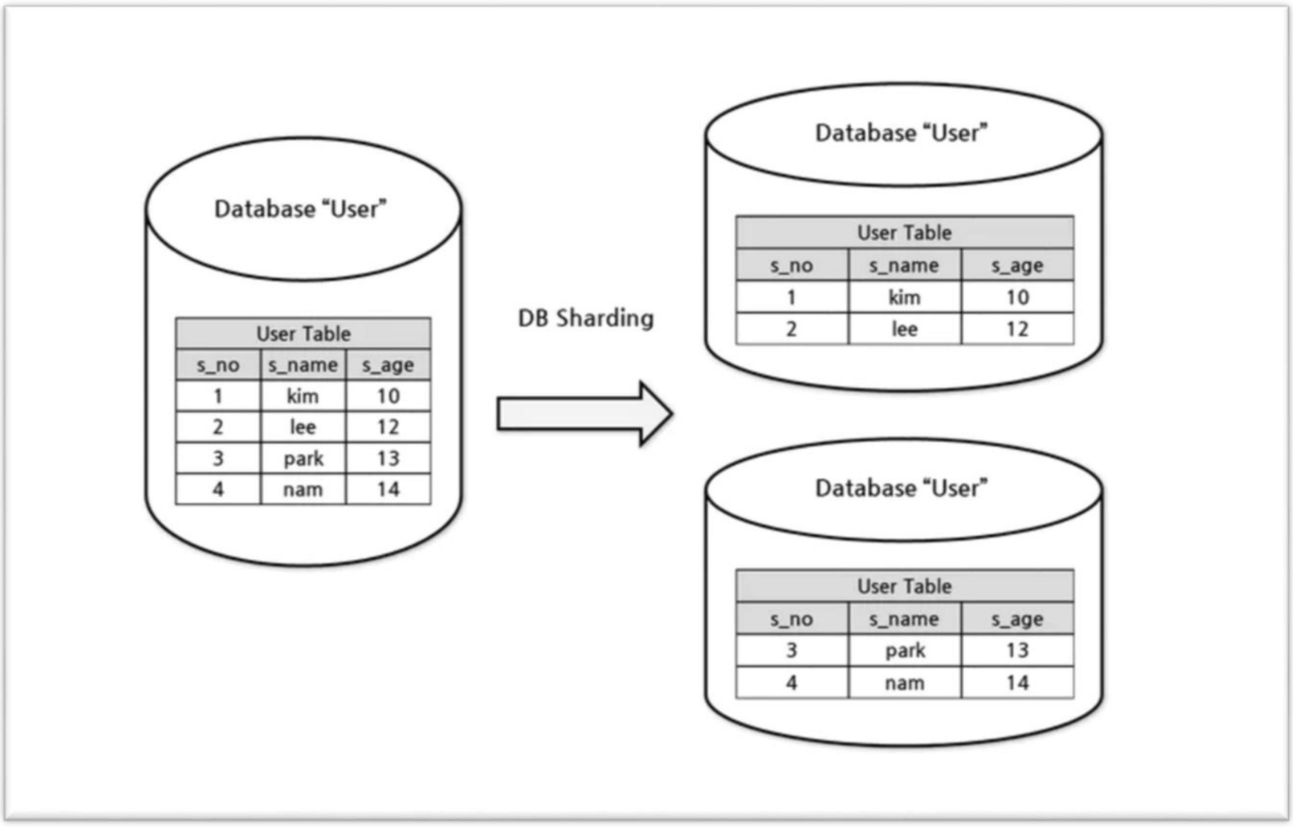
- Shard Key Selection: You define a shard key, which is a field or set of fields in your sharded collection's documents. This key determines how documents are distributed across shards. Ideally, the shard key should be a frequently accessed field that ensures even distribution of data and efficient query routing.
- Data Splitting: Based on the shard key, MongoDB partitions the collection's data into chunks. These chunks are then assigned to specific shards.
- Client Interaction: Client applications connect to the mongos instance and send queries. Mongos intercepts these queries, uses the config server metadata to identify the relevant shards, and routes the queries to those shards.
- Result Aggregation: Each shard executes the query on its portion of the data and returns the results to the mongos. Mongos then aggregates the results from all shards and presents a unified response to the client.

Benefits of Sharding:

- Horizontal Scalability: By adding more shards (i.e., more servers), you can easily scale your database to handle growing data volumes and increasing workloads.

- Improved Read Performance: By distributing read queries across multiple shards, sharding can significantly improve read performance for large datasets.

- Efficient Write Handling: Writes are typically directed to the shard containing the primary copy of the document based on the shard key. This avoids overloading a single server with write operations.



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

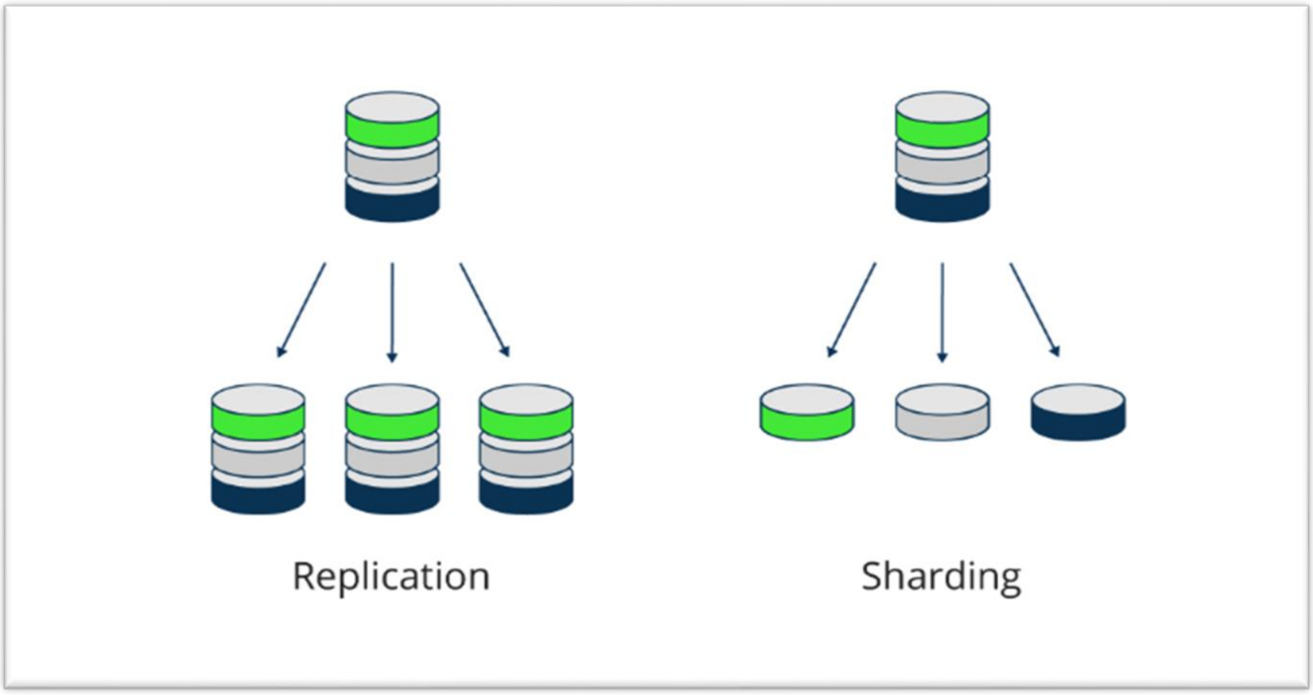
➢ REPLICATION VS SHARDING:

Replication:

- Focuses on data availability and fault tolerance.
- Creates duplicate copies of your entire dataset across multiple servers in a replica set.
- One server is designated as the primary where writes happen.
- Other servers are secondaries, constantly syncing data from the primary for read operations and failover.
- If the primary fails, a secondary can be promoted, minimizing downtime.
- Analogy: Imagine a library with multiple copies of the same book (data) on different shelves (servers). The librarian (primary) keeps track of who borrows which book (writes). Backup librarians (secondaries) have copies for reference (reads) and can step in if the main librarian is unavailable.

Sharding:

- Addresses horizontal scaling for massive datasets.
- Partitions the data into smaller chunks called shards, distributed across multiple servers (sharded cluster).
- Requires a shard key to determine which shard stores a specific document.
- Improves read and write performance for large datasets.
- Analogy: Imagine a giant library with different sections (shards) on separate floors (servers). Each section has specific categories (defined by shard key). Patrons can quickly find books (data) in their designated sections, reducing overall search time.



➢ REPLICATION + SHARDING:

Combining replication and sharding is a powerful approach to achieve both high availability and scalability in MongoDB deployments. Here's a breakdown of how they work together:

Replication for Availability:

Creates copies of your data across multiple servers (replica set).

One server acts as the primary, receiving writes and replicating them to secondaries.

If the primary fails, a secondary can be promoted, minimizing downtime.

Sharding for Scalability:

Partitions large datasets into smaller chunks (shards) based on a shard key.

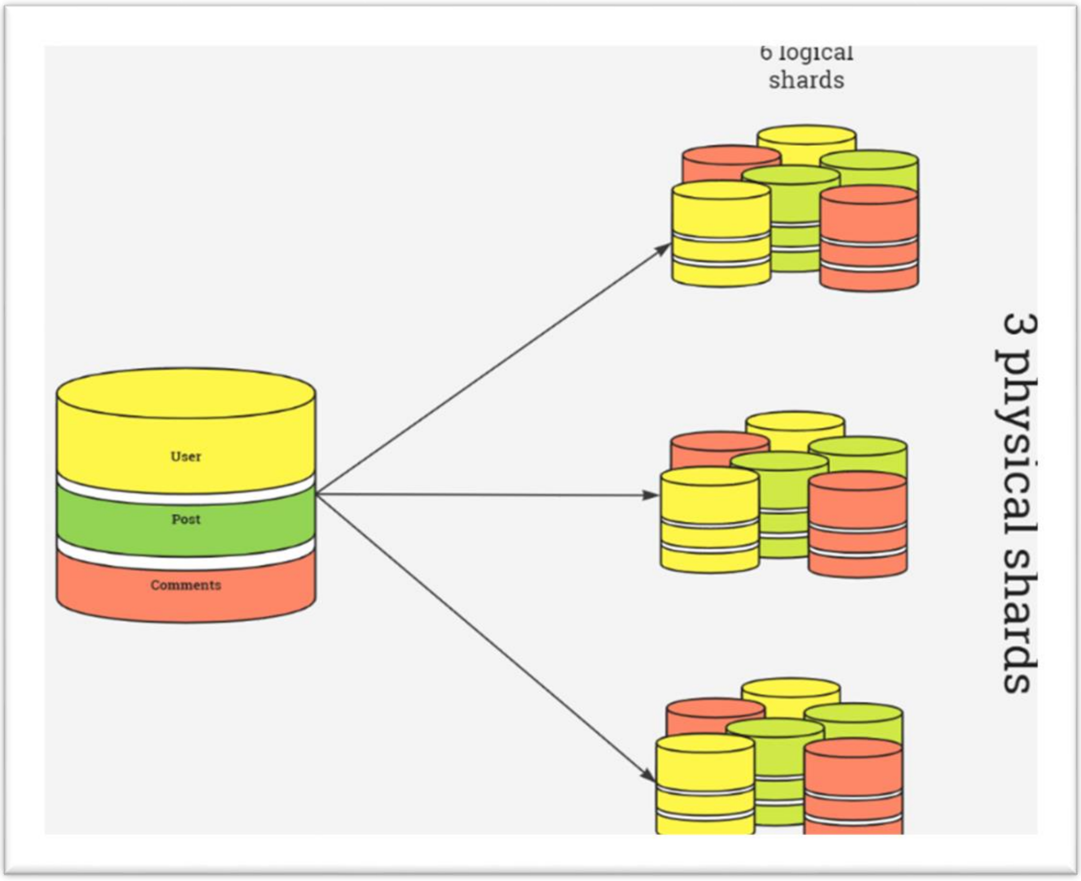
Spreads shards across multiple replica sets for parallel processing.

Enables horizontal scaling by adding more servers to handle increased data volume or queries.

Benefits of Combining Replication and Sharding:

- High Availability: Provides redundancy with both replica sets and sharded clusters.
- Scalability: Handles massive datasets and high-throughput operations efficiently.
- Flexibility: Scales reads and writes independently by adding replica sets or shards.
- Complexity: Managing sharded clusters can be more involved than replica sets.

- Shard Key Selection: Choosing the right shard key is crucial for even distribution and avoiding hotspots.
- Operational Overhead: Requires additional planning and maintenance for sharded clusters.



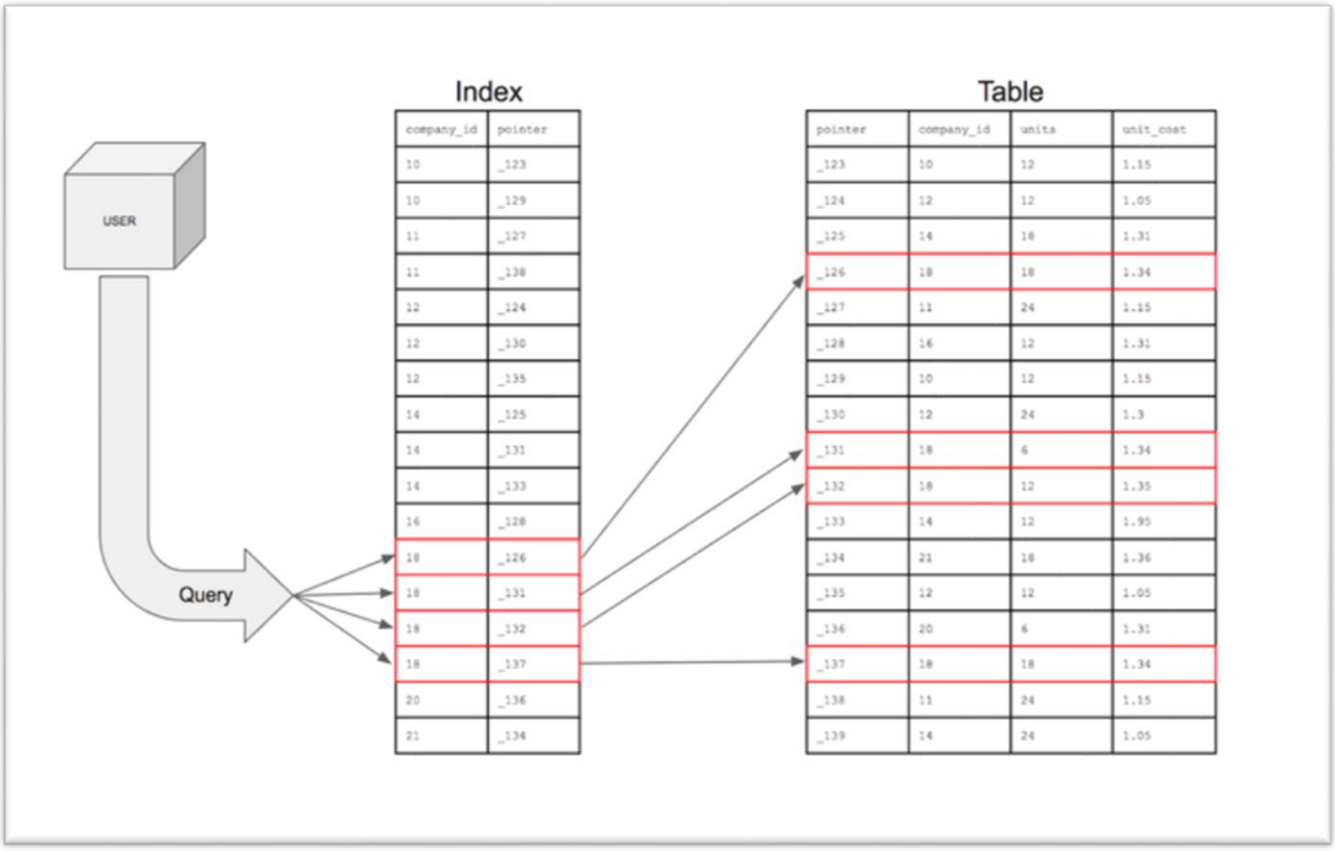
INDEXES:

Indexes support efficient execution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to return query results. If an appropriate index exists for a query, MongoDB uses the index to limit the number of documents it must scan.

Although indexes improve query performance, adding an index has negative performance impact for write operations.

Benefits of Using Indexes

- Faster Queries: Queries that utilize indexed fields experience a significant performance boost, leading to quicker retrieval of data.
- Improved Sorting: If your queries involve sorting results, indexes on the sort field can expedite the process by utilizing the inherent ordering within the index.



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:

BASIC INDEX TYPES -

- ***SINGLE FIELD INDEXES:*** Single field indexes are indexes that improve performance for queries that specify ascending or descending sort order on a single field of a document.

- ***COMPOUND INDEXES:*** Compound indexes are indexes that improve performance for queries that specify ascending or descending sort order for multiple fields of a document. You must specify the direction (ascending or descending) for each field in the index.

- ***MULTIKEY INDEXES:*** Multikey indexes are indexes that improve the performance of queries on fields that contain array values. Multikey indexes behave differently from non-multikey indexes in terms of query coverage, index bound computation, and sort behavior.

SPECIALISED INDEX TYPES -

- ***CLUSTERED INDEXES:*** Clustered indexes are indexes that improve the performance of insert, update, and delete operations on clustered collections. Clustered collections store documents ordered by the clustered index key value.

- ***GEOSPATIAL INDEXES:*** Geospatial Indexes supports queries of geospatial coordinate data using 2d sphere indexes. With a 2dsphere index, you can query the geospatial data for inclusion, intersection, and proximity.

- ***HASHED INDEXES:*** Hashed indexes maintain entries with hashes of the values of the indexed field. Hashed indexes support sharding using hashed shard keys. Hashed based sharding uses a hashed index of a field as the shard key to partition data across your sharded cluster.

ADDITIONAL CONSIDERATIONS -

- ***UNIQUE INDEXES:*** Unique indexes ensure that the indexed fields do not store duplicate values. By default, MongoDB creates a unique index on the _id field during the creation of a collection. To create a unique index, specify the field or combination of fields that you want to prevent duplication on and set the unique option to true.

- ***SPARSE INDEXES:*** Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value. The index skips over any document that is missing the indexed field.

- ***TTL INDEXES:*** In MongoDB, TTL indexes (Time to Live) are special single-field indexes that automatically remove documents from a collection after a specific timeframe. This is particularly useful for data that only needs to be stored for a limited duration, such as:

- Machine-generated event data
- Shopping carts
- Logs
- Session information

Benefits of TTL Indexes

- Automated data expiration: TTL indexes eliminate the need for you to write custom code to manually delete expired documents, streamlining data management.
- Efficiency: They provide an efficient way to remove data as they leverage background threads within MongoDB to handle deletions based on the TTL value.