Module-3

Q. 05	a	What is NOSQL? Explain CAP Theorem.
	b	Explain NOSQL Data Architecture Patterns.
OR		
Q. 06	a	Explain Shared Nothing Architecture for Big Data tasks.
	b	Explain MONGO DATABASE.

1)With Neat Diagram explain the following for shared-Nothing Architecture for Big data Tasks

(i) Single Server model
(ii) Sharding very large databases
(iii) Master Slave distribution model.
(iv) Peer-to-Peer distribution model.

1. **Single Server Model**:
    - **Description**: In this model, all the data and processing are handled by a single server. There is no distribution of data or tasks across multiple systems. This makes it the simplest form of architecture.
    - **Advantages**:
        - Easy to implement and maintain.
        - Minimal configuration is needed since everything resides on one server.
    - **Disadvantages**:
        - Limited scalability: It cannot handle large-scale data or increased traffic.
        - Vulnerable to failures: If the server goes down, the entire system is affected.
    - **Use Case**:
        - Suitable for small applications or startups with limited datasets and low traffic, like basic websites or small analytics applications.

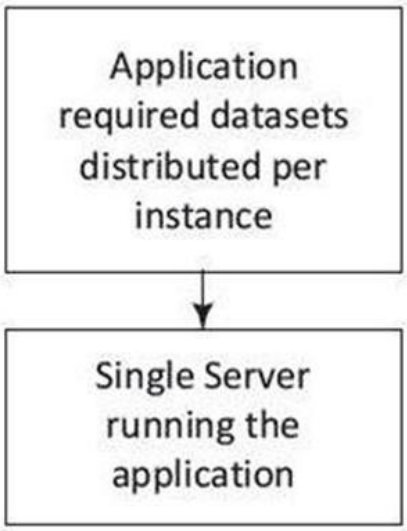
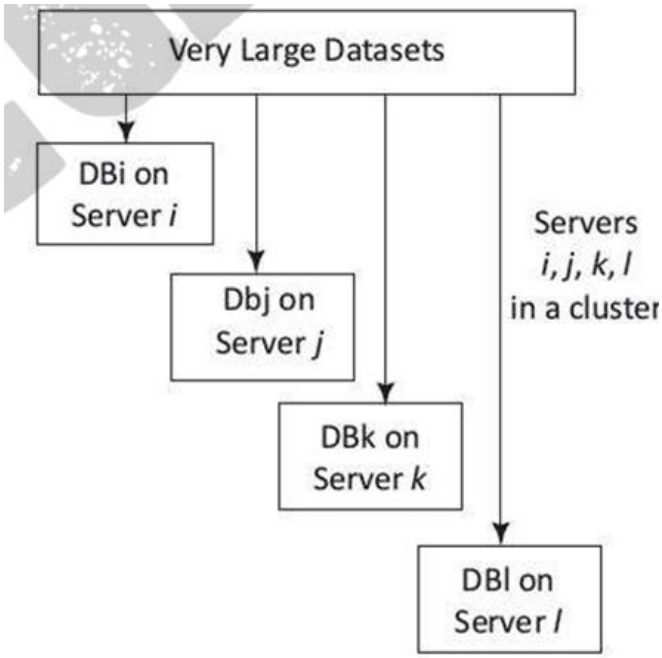


Figure: single server model

2. **Sharding Very Large Databases**:
    - **Description**: Sharding is a technique where large datasets are divided into smaller, independent chunks called "shards." Each shard is stored and processed on a separate server. This ensures that no single server handles the entire dataset.
    - **Advantages**:
        - Improves scalability by distributing data across multiple servers.
        - Increases performance by enabling parallel processing.
        - Reduces the risk of overloading a single server.
    - **Disadvantages**:
        - Requires careful planning of how data is partitioned to avoid uneven distribution (data skew).
        - Complex to manage and query, especially across multiple shards.
    - **Use Case**:
        - Used in systems with massive datasets like e-commerce platforms (Amazon), social media applications (Facebook), or search engines (Google).



3. **Master-Slave Distribution Model**:
    - **Description**: This model consists of a central "master" node that coordinates all tasks and distributes them to "slave" nodes for execution. The master node controls data consistency, task assignments, and synchronization.
    - **Advantages**:
        - Simplifies coordination as the master node manages all communication and task assignments.
        - Works well for read-heavy operations since slave nodes can handle read requests.
    - **Disadvantages**:
        - The master node can become a bottleneck under heavy load.
        - Single point of failure: If the master fails, the system may collapse unless there's a backup master.
    - **Use Case**:
        - Ideal for applications requiring centralized control, such as batch processing systems like Hadoop MapReduce.

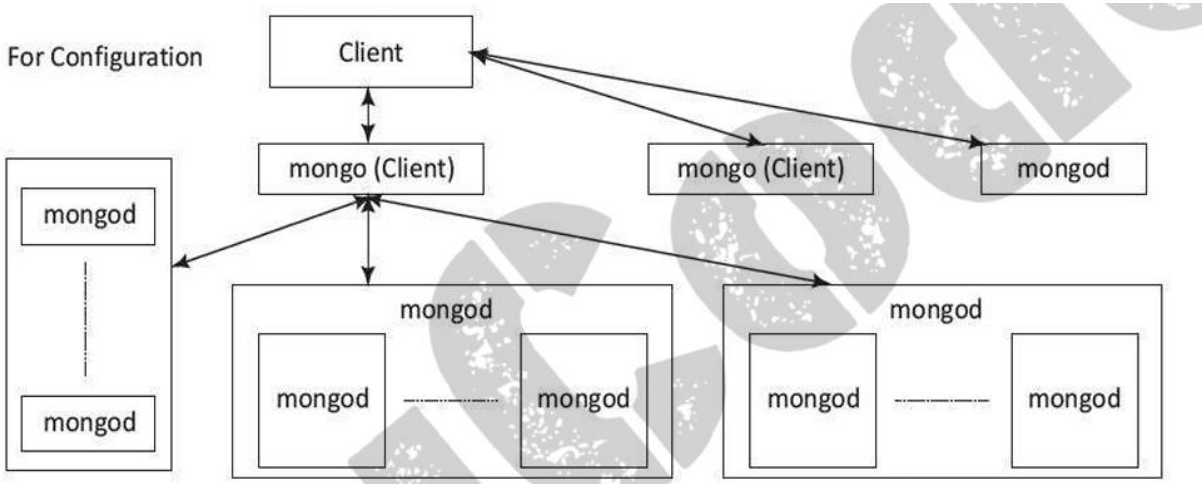


Figure: Master-slave distribution model. Mongo is a client and mongod is the server

4. **Peer-to-Peer Distribution Model**:

   - **Description**: In this decentralized model, all nodes (peers) are equal and have the same responsibilities. Each node can perform tasks independently and share data or tasks with other nodes as needed.

   - **Advantages**:

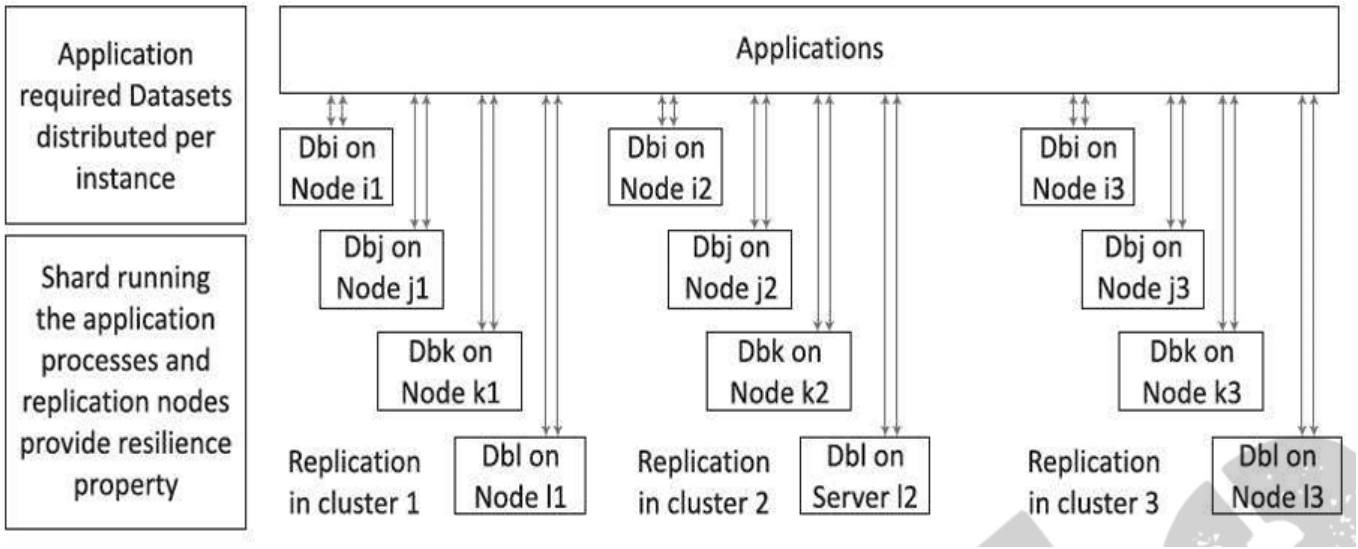
     - Highly scalable since new nodes can be added seamlessly.
     - Fault-tolerant: The failure of one node doesn't affect the entire system as tasks are redistributed.

   - **Disadvantages**:

     - Complex to manage due to the need for advanced algorithms for coordination and synchronization.
     - May result in slower performance for certain types of tasks.

   - **Use Case**:

     - Suitable for systems where decentralization is critical, such as distributed file systems (BitTorrent), blockchain networks (Bitcoin), or content delivery networks (CDNs).



2) Discuss Nosql data Architecture pattern with example?Or Define key-value store with example and what are the advantages of key-value store?

NoSQL databases provide flexible schema models to store large-scale, semi-structured, and unstructured data. Common NoSQL data architecture patterns include Key-Value Store, Document Store, Column-Family Store, and Graph Database. Below, we discuss the Key-Value Store, its advantages, and an example.

Key-Value Store

Definition:

A Key-Value Store is the simplest NoSQL data architecture pattern. It stores data as a collection of key-value pairs, where:

- The key is a unique identifier for the data.
- The value can be any type of data, such as strings, numbers, or even complex objects like JSON, XML, or binary files.

This model is highly scalable and is used for applications requiring fast lookups based on keys.

Example:

Imagine a Key-Value Store for a shopping cart application:

Key	Value
cart:user1	{"item": "Laptop", "qty": 1}
cart:user2	{"item": "Headphones", "qty": 2}

user:preferences	{"theme": "dark", "currency": "USD"}

Explanation:

- **cart:user1** stores the shopping cart details for user1.
- **user:preferences** holds the preferences for a user.

Operations:

1. Get(key): Retrieve a value using its key.
   Example: Get("cart:user1") → {"item": "Laptop", "qty": 1}
2. Put(key, value): Insert or update a key-value pair.
   Example: Put("cart:user3", {"item": "Mouse", "qty": 1})
3. Delete(key): Remove a key-value pair.
   Example: Delete("cart:user2")

Advantages of Key-Value Store

1. Scalability: Can handle large amounts of data efficiently by distributing it across multiple nodes.
2. Flexibility: Values can be of any type, allowing diverse data formats like JSON, XML, and binary objects.
3. High Performance: Fast lookups and data retrieval as it is based on direct key access.
4. Simplicity: Easy to implement and understand due to its straightforward data model.
5. Reliability: Ensures fault tolerance through replication and partitioning mechanisms.
6. Portability: Can be easily integrated into a wide range of systems and applications.
7. Low Operational Costs: Simple architecture reduces maintenance overhead.
8. Ease of Updates: Updates are performed efficiently since they involve only the value associated with the key.

Limitations of Key-Value Store

1. Lack of Querying Capabilities: Does not support complex queries or conditions like SQL (WHERE clause).

2. No Indexing on Values: Values cannot be directly searched or indexed.
3. Key Dependency: The entire data retrieval depends on having the correct key.
4. Data Duplication: Managing unique keys and avoiding redundancy can become challenging as data grows.

Typical Use Cases for Key-Value Stores

1. Session Management: Storing session data for web applications.
2. Caching: Providing fast access to frequently used data.
3. Shopping Carts: Managing shopping cart details for e-commerce applications.
4. Configuration Management: Storing application settings and configurations.

3)Discuss the characteristics of Nosql data store along with the features in NoSQL transactions, and solutions or Explain about Nosql data store and its characteristics

NoSQL (Not Only SQL) is a type of database that provides a flexible approach to storing and managing large-scale, semi-structured, and unstructured data. Unlike traditional relational databases (RDBMS) that use tables, rows, and fixed schemas, NoSQL databases allow for schema flexibility, scalability, and high performance. NoSQL databases are optimized for big data applications and distributed architectures.

Characteristics of NoSQL Data Stores

1. Schema Flexibility:
    - NoSQL databases are schema-less, meaning the data structure can vary across records.
    - This is useful for managing semi-structured and unstructured data, such as JSON, XML, and multimedia files.
2. Scalability:
    - Designed for horizontal scaling, allowing the database to distribute data across multiple servers (nodes) to handle increasing workloads.
    - This is ideal for big data applications that process terabytes or petabytes of data.
3. High Performance:

- NoSQL databases offer low-latency data access, making them suitable for real-time applications like IoT, e-commerce, and social media.

4. Distributed Architecture:
    - Data is distributed across clusters of nodes to ensure availability and reliability.
    - Fault tolerance is achieved through data replication.
5. Support for Unstructured and Semi-Structured Data:
    - Handles diverse data formats, including key-value pairs, documents, graphs, and columns.
6. Eventual Consistency:
    - Provides eventual consistency rather than strict consistency, ensuring availability and partition tolerance (as per the CAP theorem).
7. No Joins:
    - Does not support complex join operations, leading to faster query execution.
8. Variety of Data Models:
    - Supports various data models like Key-Value Store, Document Store, Column-Family Store, and Graph Database.

Features in NoSQL Transactions

NoSQL databases often relax some of the strict ACID (Atomicity, Consistency, Isolation, Durability) properties of transactions to provide higher scalability and performance. Instead, they follow BASE properties:

1. Basic Availability (BA):
    - Guarantees availability even during partial system failures.
2. Soft State (SS):
    - The system state may change over time, even without input, due to eventual consistency.

3. Eventual Consistency (EC):

- Ensures that data will become consistent eventually but does not guarantee immediate consistency after a write operation.

CAP Theorem:

NoSQL databases are designed around the CAP theorem, which states that a distributed system can provide only two of the following three guarantees:

1. Consistency: All nodes have the same data at the same time.
2. Availability: Every request gets a response, even in the event of failures.
3. Partition Tolerance: The system continues to function despite network partitions.

Most NoSQL systems prioritize availability and partition tolerance, with eventual consistency.

NoSQL Solutions for Big Data

1. High Scalability:
   - NoSQL databases are designed to scale horizontally by adding more nodes to the cluster, making them suitable for handling big data workloads.
2. Replication:
   - Multiple copies of data are stored across nodes to ensure fault tolerance and availability.
3. Sharding:
   - Large datasets are divided into smaller subsets (shards) distributed across different nodes for improved performance.
4. Schema-Less Design:
   - Data can be stored without a predefined schema, enabling flexibility and accommodating changes in data structures over time.
5. Integrated Caching:
   - Built-in caching mechanisms increase query performance by reducing data retrieval times.
6. Open Source and Cost-Effective:
   - Many NoSQL databases, such as MongoDB and Cassandra, are open-source, reducing costs for enterprises.

Examples of NoSQL Databases and Their Use Cases

1. Key-Value Store:
    - Example: Redis, DynamoDB
    - Use Case: Caching, session management.
2. Document Store:
    - Example: MongoDB, CouchDB
    - Use Case: Content management systems, e-commerce applications.
3. Column-Family Store:
    - Example: Cassandra, HBase
    - Use Case: Time-series data, log analysis.
4. Graph Database:
    - Example: Neo4j, ArangoDB
    - Use Case: Social networks, fraud detection.

4) Describe Graph database characteristic, typical used and examples

Graph Database

A Graph Database is a type of NoSQL database designed to store, manage, and query data in the form of nodes (entities) and edges (relationships). It excels in scenarios where relationships between data are as critical as the data itself, such as social networks, recommendation systems, and fraud detection.

Characteristics of Graph Databases

1. Node-Edge Relationship:
    - Nodes represent entities or objects (e.g., people, products).
    - Edges represent relationships between nodes (e.g., "friend of," "bought").
    - Each node and edge can store properties (key-value pairs) for metadata.
2. Schema Flexibility:
    - Graph databases are schema-less, allowing for easy modification of the data structure as the application evolves.
3. High Connectivity:

- Optimized for queries involving relationships, making it efficient for highly connected datasets.

4. Efficient Traversal:
    - Designed to traverse relationships efficiently, even in large datasets, with minimal performance degradation.
5. Specialized Query Languages:
    - Uses graph-specific query languages like Cypher (Neo4j), Gremlin, and SPARQL for querying and traversing relationships.
6. ACID Compliance:
    - Many graph databases ensure atomicity, consistency, isolation, and durability (ACID) for reliable transaction management.
7. Real-Time Insights:
    - Offers real-time data exploration and insights, making it suitable for dynamic and interactive applications.

Typical Uses of Graph Databases

1. Social Networks:
    - Representing users and their connections (friends, followers).
    - Use Case: Building platforms like Facebook or LinkedIn.
2. Recommendation Engines:
    - Analyzing user behavior to suggest products, movies, or friends.
    - Use Case: E-commerce platforms and OTT services (Netflix, Amazon).
3. Fraud Detection:
    - Identifying suspicious patterns and anomalous connections in transactions or networks.
    - Use Case: Banking and financial sectors.
4. Knowledge Graphs:
    - Structuring and connecting data from various sources to create an interconnected knowledge base.

- Use Case: Google Knowledge Graph.

5. Supply Chain and Logistics:
    - Mapping and optimizing relationships between suppliers, warehouses, and transportation.
    - Use Case: Tracking product flow in logistics networks.
6. Biological Networks:
    - Representing relationships in genetic data, proteins, or metabolic pathways.
    - Use Case: Drug discovery and bioinformatics.
7. IT Operations:
    - Mapping dependencies between servers, applications, and networks.
    - Use Case: Monitoring and troubleshooting IT infrastructure.

Examples of Graph Databases

1. Neo4j:
    - Description: The most widely used graph database, supports the Cypher query language.
    - Use Case: Social networks, fraud detection, and recommendation engines.
2. ArangoDB:
    - Description: A multi-model database supporting graph, document, and key-value storage.
    - Use Case: Knowledge graphs and IT operations.
3. Amazon Neptune:
    - Description: A fully managed graph database service in AWS that supports multiple query languages.
    - Use Case: Recommendation systems and fraud detection.

5)Describe the working principle of CAP thereom

Working Principle of CAP Theorem

The CAP Theorem, also known as Brewer's Theorem, is a principle that describes the limitations of distributed data systems. It states that in a distributed system, it is impossible to guarantee all three of the following properties simultaneously:

1. Consistency (C)
    - Ensures that all nodes in a distributed system see the same data at the same time.
    - Any read operation retrieves the most recent write.
    - Example: A bank transaction where all account balances reflect updates immediately.
2. Availability (A)
    - Ensures that every request receives a response, even in the event of partial system failures.
    - The system remains operational, though the data might not be consistent across all nodes.
    - Example: A web service that always returns some data (even if stale) during outages.
3. Partition Tolerance (P)
    - Ensures the system continues to operate even when there is a network partition (i.e., communication failure between nodes).
    - Example: A global e-commerce platform remains functional across data centers despite network failures.

The Principle

The CAP Theorem states that a distributed database can only provide two out of the three properties simultaneously. Here's why:

1. Consistency + Availability (CA):
    - When a system is consistent and available, it cannot tolerate network partitions.
    - If a partition occurs, the system might deny requests to maintain consistency, sacrificing availability.
    - Example: Traditional RDBMS systems with strong ACID properties (in non-partitioned setups).
2. Consistency + Partition Tolerance (CP):
    - When a system is consistent and partition-tolerant, it sacrifices availability during a partition.
    - The system may delay responses until the partition resolves to ensure consistent data.
    - Example: Distributed databases like HBase and MongoDB in strict consistency modes.

3. Availability + Partition Tolerance (AP):
    - When a system is available and partition-tolerant, it sacrifices consistency.
    - The system responds to requests but may return outdated or inconsistent data during a partition.
    - Example: DynamoDB and Cassandra, which prioritize availability and partition tolerance.

Explanation Through an Example

Consider a distributed banking system with multiple nodes managing account balances.

1. Scenario 1: If a partition occurs between Node A and Node B:
    - For CA: The system might stop processing requests to maintain consistent account balances across nodes, sacrificing availability.
    - For CP: The system may delay responses until the partition is resolved, ensuring consistency at the cost of availability.
    - For AP: The system may allow updates independently on both nodes, ensuring availability but risking inconsistent balances.

6)Explain four different ways of handling big data problems

 Evenly Distributing Data on Clusters Using Hash Rings:

- This approach involves distributing datasets across nodes in a cluster using consistent hashing algorithms.
- A hash ring is used to map datasets to specific locations, ensuring balanced data distribution.
- This method helps avoid overloading any single node and ensures efficient resource utilization. For example, data from a collection can be evenly assigned to processors using only the hashed collection ID.

 Using Replication for Horizontal Distribution:

- Replication involves creating real-time backup copies of data across multiple nodes in a cluster.
- It ensures fault tolerance, as even if one node fails, the replicated data on other nodes can be accessed.

- Horizontal scaling is achieved by allowing multiple nodes to handle client read requests simultaneously. This is particularly useful for high-availability systems.

 Moving Queries to Data Instead of Moving Data to Queries:

- In large distributed systems, instead of transferring massive datasets to a central location for querying, the query logic is moved closer to where the data resides.
- This reduces network bandwidth usage and improves processing speed.
- For example, cloud-based systems like Hadoop distribute query processing across nodes to ensure data remains localized.

 Distributing Queries Across Multiple Nodes:

- Client queries are analyzed and split across multiple data nodes or their replicas.
- Query execution is parallelized, allowing multiple nodes to process parts of the query simultaneously.
- This enhances performance, as query evaluation is separated from execution, ensuring faster and more efficient data retrieval.

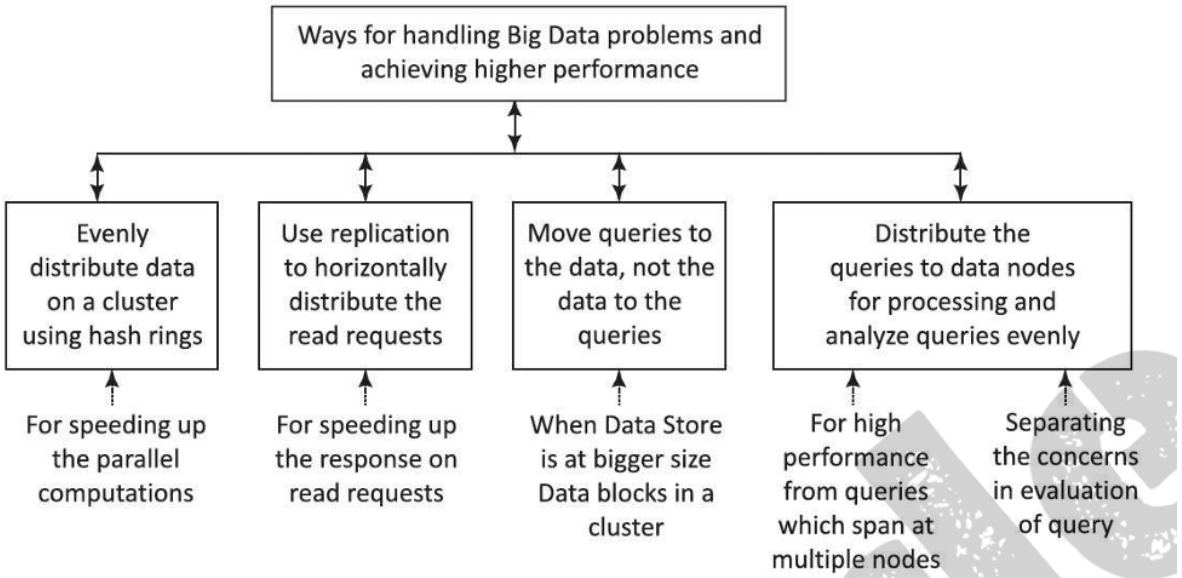


Figure: Four ways for handling big data problems

7)Explain different components of Cassandra

1. Cluster

- A cluster is a collection of nodes or servers grouped together to work as a distributed system.
- Data in Cassandra is partitioned and distributed across all nodes in a cluster.

- Clusters ensure fault tolerance, scalability, and load balancing.

---

2. Keyspace

- A keyspace is the highest-level namespace in Cassandra, similar to a database in traditional RDBMS.
- It contains all the tables, column families, and indexes.
- Keyspace Properties:
    - Replication Factor: Number of data copies stored across nodes.
    - Replica Placement Strategy: Determines how replicas are distributed across nodes (e.g., SimpleStrategy or NetworkTopologyStrategy).
    - Durable Writes: Ensures data is persisted even in the event of a crash.

3. Node

- A node is the basic storage unit where data is stored.
- Each node is part of a cluster and is identified by a token that determines the range of data it holds.
- Nodes communicate with each other to share workload and ensure data consistency.

---

4. Table (Column Family)

- A table is a collection of rows (like in RDBMS) but structured in a key-value pair format.
- Tables in Cassandra do not enforce a fixed schema, offering flexibility in adding columns.
- Each table has a primary key that determines how data is distributed across the cluster.

---

5. Partitioner

- The partitioner is responsible for distributing data across the nodes in a cluster.
- It determines which node will store a specific row of data based on its partition key.
- Common partitioners include:
    - Murmur3Partitioner (default): Provides even data distribution.
    - RandomPartitioner.

---

6. Commit Log

- A commit log is a write-ahead log used for data durability.
- All data changes are written to the commit log before being written to the table.
- In case of a system crash, the commit log can be replayed to restore data.

---

7. SSTable (Sorted String Table)

- SSTables are immutable data files stored on disk.
- They are created after data is written to the table and flushed from memory (Memtable).
- SSTables are organized in sorted order, making reads efficient.

8. Memtable

- A Memtable is an in-memory structure where data is temporarily stored before being flushed to disk as an SSTable.
- Memtables improve write performance as they avoid frequent disk writes.

---

9. Replication

- Cassandra ensures high availability by replicating data across multiple nodes.
- The replication factor determines how many copies of the data are stored.
- Replication strategies include:
    - SimpleStrategy: Best for single data center.
    - NetworkTopologyStrategy: Designed for multiple data centers.

---

10. Consistency Level

- The consistency level defines how many nodes must acknowledge a read or write operation for it to be considered successful.
- Examples:

- ONE: One node acknowledges.
- QUORUM: Majority of nodes acknowledge.
- ALL: All replicas must acknowledge.

---

11. CQL (Cassandra Query Language)

- CQL is the language used to interact with Cassandra databases.
- It is similar to SQL but tailored for Cassandra's distributed, NoSQL architecture.

---

12. Gossip Protocol

- Cassandra uses a peer-to-peer gossip protocol for communication between nodes.
- This ensures that every node knows the state of other nodes in the cluster, enabling efficient coordination.

8)Explain different data types built into Cassandra

1. Basic Data Types

Basic data types are used for storing simple, scalar values.

- ASCII:
  Used for storing ASCII character strings. It accepts only standard ASCII characters.
  Example: 'Hello'.
- BIGINT:
  A 64-bit signed integer, typically used for storing large numbers like IDs or timestamps.
  Example: 123456789.
- BLOB:
  Stores arbitrary binary data in hexadecimal format. This is useful for storing images or files.
  Example: 0x48656c6c6f.
- BOOLEAN:
  Represents logical values, either true or false.
  Example: true.

- COUNTER:
  A special type for storing values that can only be incremented or decremented. Commonly used for tracking counts.
  Example: A page view counter.
- DECIMAL:
  Stores high-precision decimal numbers, often used in financial data.
  Example: 12345.678.
- DOUBLE:
  A double-precision floating-point number for scientific calculations or large-scale data.
  Example: 3.14159265359.
- FLOAT:
  A single-precision floating-point number, often used for less precise calculations.
  Example: 3.14.
- INET:
  Used to store IP addresses (both IPv4 and IPv6).
  Example: 192.168.1.1.
- INT:
  A 32-bit signed integer, useful for numeric values like ages or counts.
  Example: 42.
- TEXT:
  A Unicode string, similar to VARCHAR, used to store textual data.
  Example: 'Cassandra is great!'.
- TIMESTAMP:
  Stores date and time as a 64-bit integer representing milliseconds since January 1, 1970 (Unix epoch).
  Example: 2024-12-05 15:30:00.
- UUID:
  Stores a universally unique identifier. Useful for generating unique keys.
  Example: 550e8400-e29b-41d4-a716-446655440000.
- VARINT:
  Stores arbitrary-precision integers, meaning it can handle numbers of any size.
  Example: 12345678901234567890.

2. Collection Data Types

Cassandra supports complex data structures for managing collections of data within a single column.

- List:
  An ordered collection of values, where duplicates are allowed.
  Example: ['item1', 'item2', 'item3'].
- Set:
  An unordered collection of unique values.
  Example: {'item1', 'item2', 'item3'}.
- Map:
  A collection of key-value pairs, where keys must be unique.
  Example: { 'key1': 'value1', 'key2': 'value2' }.

9)Compare contrast between RDBMS and MongoDB Database

Aspect	**RDBMS**	**MongoDB**
Data Model	Relational: Data is stored in structured tables with rows and columns.	Document-Oriented: Data is stored in flexible, JSON-like documents.
Schema	Fixed schema: Requires predefined schema before storing data.	Schema-less: Documents can have different structures, offering flexibility.
Query Language	SQL (Structured Query Language) for data manipulation.	MongoDB Query Language (MQL) or JSON-based querying.
Joins	Supports JOIN operations to combine data from multiple tables.	Does not support joins directly but uses **embedded documents** and **reference documents**.
Transactions	ACID (Atomicity, Consistency, Isolation, Durability) is fully supported.	Supports **BASE** (Basically Available, Soft state, Eventually consistent) properties.
Scalability	Vertical scaling (adding more resources to a single server).	Horizontal scaling (adding more nodes to the cluster).
Replication	Typically manual, through configurations like master-slave setups.	Built-in replication using replica sets for high availability and fault tolerance.

Performance	Slower for unstructured or semi-structured data.	Faster for large-scale, unstructured, or semi-structured data.
Data Relationships	Enforces strong relationships through primary and foreign keys.	Relationships are loosely defined using references or embedded documents.
Data Storage	Tabular structure with rows and columns.	BSON (Binary JSON)-based documents.
Flexibility	Rigid: Schema changes are challenging and require migration.	Flexible: Easy to modify structure without affecting existing data.
Indexes	Supports indexes for fast querying.	Supports advanced indexing, including geospatial and text indexing.
Use Cases	Best for structured data and applications requiring strong consistency (e.g., banking).	Ideal for dynamic, semi-structured data like content management, IoT applications, etc.

MongoDB data types:

Double	**Represents a float value.**
String	**UTF-8 format string.**
Object	**Represents an embedded document.**
Array	**Sets or lists of values.**
Binary data	**String of arbitrary bytes to store images, binaries.**

Object id	Objectlds (MongoDB document identifier, equivalent to a primary key) are: small, likely unique, fast to generate, and ordered. The value consists of 12- bytes, where the first four bytes are for timestamp that reflects the instance when ObjectId creates.
Boolean	Represents logical true or false value.
Date	BSON Date is a 64-bit integer that represents the number of milliseconds since the Unix epoch (Jan 1, 1970).
Null	Represents a null value. A value which is missing or unknown is Null.
Regular Expression	RegExp maps directly to a JavaScript RegExp
32—bit integer	Numbers without decimal points save and return as 32-bit integers.
Timestamp	A special timestamp type for internal MongoDB use and is not associated with the regular date type. Timestamp values are a 64-bit value, where first 32 bits are time, t (seconds since the Unix epoch), and next 32 bits are an incrementing ordinal for operations within a given second.

MongoDB querying commands

Command	Functionality
Mongo	Starts MongoDB; (*mongo is MongoDB client). The default database in MongoDB is test.
db.help ()	Runs help. This displays the list of all the commands.
db.stats ()	Gets statistics about MongoDB server.

Use <database name>	Creates database
Db	Outputs the names of existing database, if created earlier
Dbs	Gets list of all the databases
db.dropDatabase ()	Drops a database
db.database name.insert ()	Creates a collection using insert ()
db.<database name>.find ()	Views all documents in a collection
db.<database name>.update ()	Updates a document
db.<database name>.remove ()	Deletes a document

MongoDB is an open-source, NoSQL database management system developed by MongoDB Inc. Initially introduced as a Platform as a Service (PAAS) by 10gen (now MongoDB Inc.), it was later released as an open-source database server in 2009. MongoDB is designed to handle large volumes of data with scalability and flexibility. Below are the core details and features:

Key Characteristics of MongoDB:

1. **Non-relational**: Unlike traditional relational databases, MongoDB does not use tables with fixed schemas. It is based on a document model.
2. **NoSQL**: MongoDB belongs to the NoSQL family, meaning it does not rely on traditional SQL queries.
3. **Distributed**: MongoDB supports horizontal scaling by distributing data across multiple servers.
4. **Open Source**: MongoDB is freely available and can be modified.
5. **Document-based**: Data is stored as documents (JSON-like format), making it more flexible than relational databases.
6. **Cross-platform**: MongoDB is platform-independent and can run on different operating systems.
7. **Scalable**: It can handle large amounts of data and scale easily by adding more servers.
8. **Flexible data model**: MongoDB's schema-less structure allows for easy modification of data structures over time.
9. **Indexed**: MongoDB supports indexing to make queries efficient.
10. **Multi-master**: MongoDB supports replication with multiple master nodes, ensuring high availability.
11. **Fault-tolerant**: It has built-in mechanisms to recover from server failures.

Features of MongoDB:

1. **Database (DB)**: MongoDB stores data in databases, each of which is a physical container for collections. Each database gets its own set of files on the server's file system, and multiple databases can run on a single server. The default database in MongoDB is named test. The server process is called mongod, while the client program is mongo.
2. **Collection**: A collection is a group of MongoDB documents, similar to a table in relational databases. Collections are schema-less, meaning they can store documents with different fields. Each collection is designed for a specific purpose and exists within a single database.
3. **Document**: Documents are the fundamental unit of data storage in MongoDB, similar to rows in relational tables. MongoDB stores documents in BSON (Binary JSON) format, which is a binary representation of JSON. The structure of a document is well-defined, and it can store data in the form of key-value pairs.
4. **Flexible Data Storage**: MongoDB uses BSON for storing data, making it more flexible than traditional relational databases. Data can vary between documents in the same collection, and the schema can evolve over time.
5. **Querying**: MongoDB allows complex querying and real-time data aggregation. It supports a rich query language, similar to SQL, for accessing data. MongoDB also allows for dynamic queries, which can be adjusted based on the data in the documents.
6. **Indexing**: MongoDB supports indexing to improve query performance. Indexes can be created on fields to speed up search operations.
7. **Aggregation**: MongoDB provides real-time aggregation capabilities to process data in real time. The aggregation framework allows you to perform operations like filtering, sorting, and grouping on your data.
8. **No Joins**: MongoDB does not require complex joins as relational databases do. Instead, data can be embedded directly in documents, reducing the need for multiple table lookups.

9. **Distributed System**: MongoDB is a distributed database system, which means that it can run across multiple servers. This helps ensure high availability and horizontal scalability by automatically distributing data across multiple nodes (servers).

Summary of MongoDB's Key Features:

- **Database and Collection Management**: MongoDB uses databases and collections to store documents. Each collection holds multiple documents and can have different fields.
- **BSON Format**: Documents are stored in BSON format, which is a binary version of JSON, making it highly efficient.
- **Scalability and Distribution**: MongoDB can scale horizontally across multiple servers, ensuring that it can handle large volumes of data efficiently.
- **Dynamic Schema**: MongoDB collections do not require a fixed schema, allowing for flexibility in data storage.
- **Rich Querying and Aggregation**: MongoDB provides powerful querying, indexing, and aggregation features to analyze and retrieve data efficiently.