


# Chapter 4

👤 Created by	 chess pog
🕒 Created time	@April 2, 2024 3:39 AM
🏷️ Tags	



Differentiate between structured, semi-structured and unstructured data and discuss the Taxonomy of NoSQL. (8)

## 1. Structured Data:

- **Definition:** Structured data is highly organized and formatted in a predefined manner. It typically fits neatly into tables or databases, with rows and columns that define the attributes and relationships within the data.
- **Characteristics:**
  - Organized: Data is organized into a well-defined structure with fixed fields.
  - Easily searchable: Because of its organized nature, structured data is easily searchable and retrievable.
  - Supports relational databases: Often used in relational databases where data is organized into tables and can be queried using SQL.
- **Examples:** Data in relational databases, spreadsheets, CSV files, etc.
- **Use Cases:** Financial records, inventory databases, CRM systems, etc.

## 2. Semi-Structured Data:

- **Definition:** Semi-structured data does not conform to the strict structure of traditional relational databases but has some organizational properties. It contains tags, markers, or keys that separate semantic elements and groups of data.
- **Characteristics:**
  - Flexible: Semi-structured data allows for flexibility in terms of adding or changing fields without needing a predefined schema.
  - Contains some structure: While not as rigidly organized as structured data, it still retains some level of structure through tags or keys.
  - Suitable for hierarchical representation: Supports hierarchical structures like JSON, XML, etc.
- **Examples:** JSON files, XML files, log files, NoSQL databases, etc.
- **Use Cases:** Web data (e.g., HTML pages), machine-generated data (e.g., logs), configuration files, etc.

## 3. Unstructured Data:

- **Definition:** Unstructured data lacks a predefined data model or organization. It does not conform to a fixed schema, and the information within it is not organized in a pre-defined manner.
- **Characteristics:**
  - No predefined structure: Unstructured data lacks any formal structure or organization.
  - Varied formats: It can be in the form of text, images, audio, video, etc., with no inherent organization.
  - Requires advanced processing techniques: Analyzing and extracting insights from unstructured data requires advanced techniques such as natural language processing (NLP), computer vision, etc.
- **Examples:** Text documents, social media posts, emails, images, videos, audio recordings, etc.

- **Use Cases:** Sentiment analysis, image recognition, speech recognition, content categorization, etc.

In database management, scaling refers to the capability of a database system to handle increased workload or data volume. There are primarily two types of scaling in databases: vertical scaling and horizontal scaling.

### 1. Vertical Scaling (or Scaling Up):

- Vertical scaling involves increasing the capacity of a single server, typically by adding more CPU, memory, or storage resources to it.
- This approach is relatively straightforward and can be effective for smaller-scale applications.
- However, there's a limit to how much a single server can be scaled vertically, and it can become expensive and difficult to maintain as the system grows.

### 2. Horizontal Scaling (or Scaling Out):

- Horizontal scaling involves distributing the workload across multiple servers, typically in a clustered or distributed computing environment.
- Instead of adding more resources to a single server, new servers are added to the system to share the load.
- This approach offers better scalability since additional servers can be added as needed to handle increasing demands.
- Horizontal scaling often requires changes to the application architecture to support distributed computing, such as partitioning data or implementing load balancing.
- Cloud computing platforms often facilitate horizontal scaling by providing auto-scaling features that automatically add or remove resources based on demand.



Define CAP theorem by highlighting its use case. When do we require columnar databases? Explain with examples (5+5)

## CAP Theorem

The CAP theorem, also known as Brewer's theorem, is a fundamental principle in distributed systems theory that states that it is impossible for a distributed data system to simultaneously provide all three of the following guarantees:

- **Consistency:** All nodes in the distributed system have the same data at the same time. If a write is successful, all subsequent reads will reflect that write. This ensures that the data is always in a valid state.
- **Availability:** Every request made to the system receives a response, even if some nodes are failing. The system remains responsive to client requests even under failures.
- **Partition Tolerance:** The system continues to function even if communication between nodes is unreliable, meaning some messages between nodes may be lost or delayed due to network issues.

According to the CAP theorem, a distributed system can prioritize any two out of these three properties, but it cannot simultaneously achieve all three. This means that in the presence of a network partition (P), a distributed system must choose between consistency (C) and availability (A).

In real-world scenarios, different systems may make different trade-offs based on their specific requirements and priorities. For example, some systems may prioritize consistency and partition tolerance, sacrificing availability, while others may prioritize availability and partition tolerance, sacrificing consistency.

So, CAP theorem can be summarized as: For any distributed database, one of the following can hold:

1. If a database guarantees availability and partition tolerance, it must forfeit consistency. Egg: Cassandra, CouchDB and so on.
2. If a database guarantees consistency and partition tolerance, it must forfeit availability. Egg: HBase, Mongo DB and so on.
3. If a database guarantees availability and consistency, there is no possibility of network partition. Egg: RDBMS like MySQL, Postgres and so on

## Use case of CAP Theorem

For example, let's say you're designing a system for a banking application where account balances need to be consistent across all data centers. In this

case, you would prioritize consistency over availability. Even if some data centers become temporarily unreachable due to network partitions, you want to ensure that all users see the same account balances to avoid discrepancies or potential financial losses.

On the other hand, consider a social media platform where users are posting updates and sharing content in real-time. In this scenario, you might prioritize availability over consistency. Even if network partitions occur, you want to ensure that users can still access and interact with the platform without experiencing downtime or disruptions. Consistency can be eventually achieved through mechanisms like eventual consistency or conflict resolution.

## **Absolute consistency is generally sacrificed**

In distributed systems, absolute consistency (where all nodes see the same data at the same time) is often sacrificed in favor of availability and partition tolerance. Achieving absolute consistency in a distributed system would require synchronous communication between all nodes, which can be impractical and often leads to increased latency and decreased availability.

Instead, many distributed systems opt for a form of eventual consistency, where it's acknowledged that there may be a short period of time where different nodes have slightly different views of the data. Over time, these inconsistencies are resolved, and the system converges to a consistent state. This approach allows for better availability and partition tolerance, as nodes can continue to operate independently even in the presence of network partitions or node failures.

So, while consistency is important, it's often sacrificed to ensure that the system remains available and functional even in challenging network conditions.

### **Eventual Consistency:**

Eventual consistency is a consistency model employed in distributed systems where data may not be immediately consistent across all nodes but will eventually converge to a consistent state given enough time and assuming no further updates. In other words, after a certain period of time with no new updates and with the resolution of any network partitions, all replicas of the data will eventually agree on the same value.

Eventual consistency allows for high availability and fault tolerance, as it permits each replica to operate independently without needing constant coordination with other replicas. However, it also means that there may be temporary inconsistencies between replicas until convergence is achieved.

This model is often used in systems where immediate consistency is not critical, such as social media feeds or search engine indexing, where users can tolerate some delay in seeing the most up-to-date information.

### **Tunable Consistency:**

Tunable consistency refers to the ability to adjust the level of consistency according to the specific requirements of an application. It allows developers to choose a consistency level that best suits their application's needs, balancing consistency, availability, and partition tolerance.

In tunable consistency models, developers can typically choose from a range of consistency levels, each offering a different trade-off between consistency and availability. These levels often include options like strong consistency, eventual consistency, and various levels of consistency in between.

For example, in a tunable consistency system, developers might choose strong consistency when performing critical transactions that require immediate consistency guarantees, while opting for eventual consistency for less critical data where availability and fault tolerance are more important.

Tunable consistency allows developers to tailor the consistency guarantees of their system to match the specific requirements of their application, providing flexibility and the ability to optimize performance based on the application's needs.

## **Properties of Distributed System**

1. **Scalability:** Distributed systems should be able to scale horizontally to handle increasing amounts of data and traffic by adding more machines to the system.
2. **Fault Tolerance:** Distributed systems should be resilient to failures, meaning that if a node or component fails, the system should continue to operate without losing data or functionality.
3. **Consistency:** Distributed systems should maintain consistency across all nodes, ensuring that all nodes have the same view of the data at any given

time.

4. **Partition Tolerance:** Distributed systems should continue to operate even if network partitions occur, meaning that nodes can communicate and coordinate effectively despite network failures or delays.
5. **High Availability:** Distributed systems should be available and responsive even in the face of failures, ensuring that users can access the system and its services whenever needed.
6. **Decentralization:** Distributed systems often distribute control and decision-making across multiple nodes, reducing bottlenecks and single points of failure.
7. **Replication:** Data is often replicated across multiple nodes in a distributed system to improve fault tolerance, availability, and performance.
8. **Data Distribution:** Distributed systems should efficiently distribute data across nodes to ensure load balancing and optimal resource utilization.
9. **Eventual Consistency:** In some distributed systems, eventual consistency is acceptable, meaning that while data might be temporarily inconsistent across nodes, it will eventually converge to a consistent state.
10. **Concurrency Control:** Distributed systems should provide mechanisms for managing concurrent access to shared resources, such as data, to prevent conflicts and ensure data integrity.

## NoSQL

NoSQL, which stands for "Not Only SQL," is a term used to describe a broad category of database management systems that diverge from the traditional relational database management systems (RDBMS). NoSQL databases are designed to handle large volumes of unstructured or semi-structured data, which may not fit well into the rigid tabular format of relational databases. Here are the main properties of NoSQL databases:

1. **Schema-less or Flexible Schema:** Unlike relational databases, which require a predefined schema for data storage, NoSQL databases typically allow for schema flexibility. This means that you can store data without having to define a rigid schema in advance, allowing for easier handling of unstructured or semi-structured data.
2. **Horizontal Scalability:** NoSQL databases are designed to scale out horizontally by adding more machines or nodes to the database cluster.

This allows them to handle large volumes of data and high traffic loads more easily than traditional relational databases, which often scale vertically by adding more powerful hardware.

3. **High Availability:** Many NoSQL databases are designed with built-in replication and distribution mechanisms that ensure high availability and fault tolerance. This means that even if some nodes in the database cluster fail, the database can continue to operate without downtime or data loss.
4. **Distributed Architecture:** NoSQL databases are typically designed with distributed architectures that allow them to distribute data across multiple nodes or servers in a cluster. This enables them to achieve high levels of scalability and performance by leveraging the resources of multiple machines.
5. **Optimized for Specific Use Cases:** NoSQL databases are often optimized for specific use cases or data models, such as document-oriented, key-value, column-family, or graph databases. This allows them to provide better performance and scalability for certain types of applications and workloads.
6. **Support for Unstructured and Semi-Structured Data:** NoSQL databases are well-suited for storing and querying unstructured or semi-structured data, such as JSON documents, XML data, binary data, and other non-tabular data formats. This makes them ideal for handling diverse data types commonly found in modern web applications, IoT devices, and big data environments.



Differentiate between a RDBMS and a NoSQL Databases. (3)

## RDMS VS NoSQL

### 1. Data Model:

- **RDBMS:** Relational databases follow a tabular data model, where data is organized into tables with rows and columns. They enforce a strict schema, requiring predefined structures for data storage.
- **NoSQL:** NoSQL databases support various data models, including document-oriented (e.g., MongoDB), key-value (e.g., Redis), column-



family (e.g., Cassandra), and graph (e.g., Neo4j). They offer more flexibility in data modeling, allowing for schema-less or flexible schemas.

## **2. Scalability:**

- **RDBMS:** Relational databases traditionally scale vertically by adding more powerful hardware. Horizontal scaling can be challenging due to the complexity of maintaining data consistency across distributed systems.
- **NoSQL:** NoSQL databases are designed for horizontal scalability, allowing them to distribute data across multiple nodes in a cluster. They can handle large volumes of data and high traffic loads more easily by adding more machines to the cluster.

## **3. ACID Compliance:**

- **RDBMS:** Relational databases typically adhere to ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring data integrity and transactional consistency.
- **NoSQL:** NoSQL databases may sacrifice strict ACID compliance for performance and scalability. While some NoSQL databases offer ACID transactions (especially in single-node setups), others prioritize eventual consistency or provide weaker consistency models (e.g., eventual consistency or causal consistency).

## **4. Query Language:**

- **RDBMS:** Relational databases use SQL (Structured Query Language) for data definition, manipulation, and querying. SQL provides powerful capabilities for complex joins, aggregations, and transactions.
- **NoSQL:** NoSQL databases may have their own query languages or APIs tailored to their specific data models. For example, document-oriented NoSQL databases often use JSON-based query languages, while key-value stores may offer simple get/put operations.

## **5. Use Cases:**

- **RDBMS:** Relational databases excel in scenarios requiring complex queries, transactions, and strict data consistency, such as traditional business applications, financial systems, and applications with well-defined schemas.

- **NoSQL:** NoSQL databases are well-suited for handling unstructured or semi-structured data, high-volume data ingestion, real-time analytics, content management systems, and applications requiring flexible schemas or horizontal scalability.

## 6. Examples:

- **RDBMS:** Examples of RDBMS include MySQL, PostgreSQL, Oracle Database, SQL Server, and SQLite.
- **NoSQL:** Examples of NoSQL databases include MongoDB, Cassandra, Redis, Couchbase, Amazon DynamoDB, and Neo4j.

## NoSQL Taxonomy

NoSQL Taxonomy refers to the categorization or classification of NoSQL (Not Only SQL) databases based on their data models, architecture, and characteristics.

### 1. Document Store:

Document-oriented databases store data in a semi-structured format, typically using JSON (JavaScript Object Notation) or BSON (Binary JSON) documents. Each document can have its own unique structure, and data within documents can be queried using keys or attributes. Document stores are flexible and schema-less, allowing for easy modification and adaptation to evolving data schemas. This flexibility makes them suitable for a wide range of use cases, including content management systems, e-commerce applications, and real-time analytics.

#### Examples:

- **MongoDB:** One of the most popular document databases, MongoDB is known for its scalability, flexibility, and rich query capabilities. It stores data in BSON documents and supports features such as indexing, replication, and sharding.

### 2. Graph Database:

Graph databases are designed to represent and store relationships between data entities as nodes and edges in a graph structure. Nodes represent entities (such as people, products, or locations), and edges represent relationships between nodes (such as friendships, purchases, or connections). Graph databases excel at traversing and querying complex networks of relationships, making them ideal for applications involving

social networks, recommendation engines, fraud detection, and network analysis.

**Examples:**

- **Amazon Neptune:** Amazon Neptune is a fully managed graph database service provided by AWS (Amazon Web Services). It supports both property graphs and RDF (Resource Description Framework) graphs and offers high availability, durability, and security..

**3. Key-Value Store:**

Key-value databases store data as a collection of key-value pairs, where each key is unique and maps to a corresponding value. They are optimized for simple data retrieval and storage operations, making them efficient for use cases such as caching, session management, and user preferences. Key-value stores are highly scalable and can handle large volumes of data with low latency.

**Examples:**

- **Redis:** Redis is an in-memory key-value store known for its high performance, rich data types, and versatile use cases. It supports features such as persistence, replication, clustering, and pub/sub messaging.

**4. Columnar Database:**

Columnar databases are optimized for analytical queries that involve aggregating data across large datasets. They organize data by column rather than by row, which offers several advantages for certain use cases:

1. **Compression:** Since columns typically contain similar types of data, they can be compressed more effectively than rows, resulting in reduced storage requirements and improved query performance.
2. **Query Performance:** Columnar databases excel at analytical queries that involve scanning large portions of data, such as aggregations, filtering, and data mining operations. By storing related data together, columnar databases can quickly retrieve only the columns needed for a particular query, minimizing disk I/O and speeding up query execution.
3. **Data Warehousing:** Columnar databases are commonly used in data warehousing scenarios where the focus is on running complex

analytical queries against large datasets. Examples include business intelligence applications, data analytics platforms, and reporting tools.

**Examples:**

- Apache Cassandra: While primarily known as a column-family database, Cassandra also supports a wide columnar data model. It is designed for high availability, linear scalability, and eventual consistency. Cassandra is commonly used for time-series data, IoT applications, and real-time analytics.
- Apache HBase: Built on top of Apache Hadoop, HBase is a distributed columnar database that provides random, real-time read/write access to large datasets. It is designed for scalability, fault tolerance, and consistent performance, making it ideal for use cases such as online gaming, social media analytics, and ad targeting.

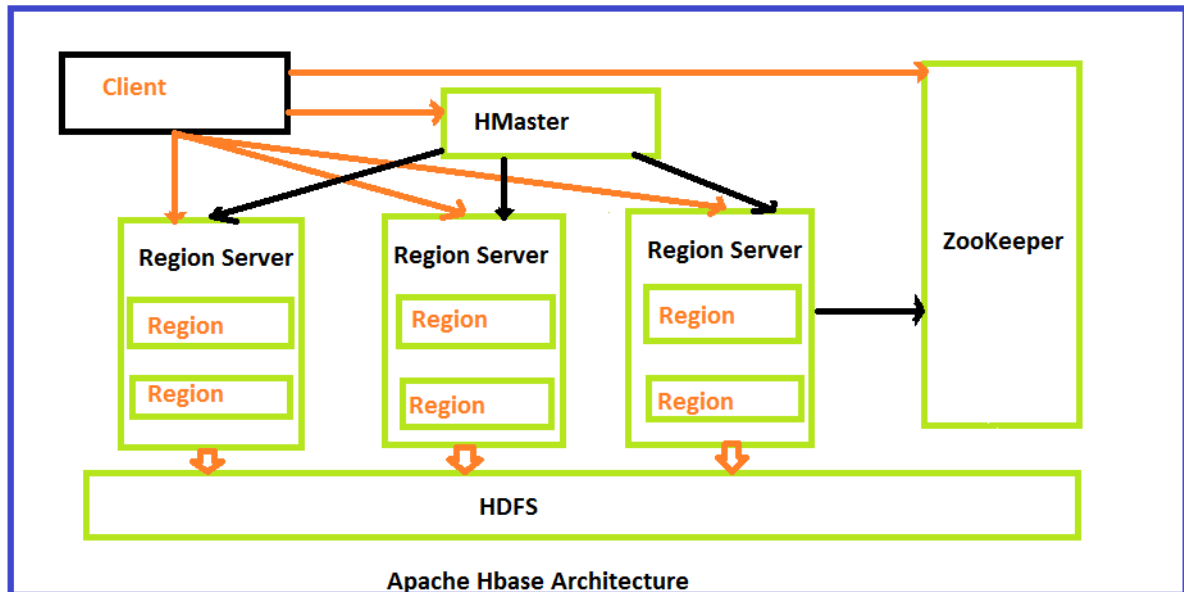


Explain Hbase components in cluster architecture and define the role of zookeeper in Hbase. Write Hbase commands to store and select data in Hbase database . (10)



List down the categories of NoSQL databases.(taxonomy) Explain architecture of Hbase (6+6)

## Hbase



HBase is a distributed, scalable, and column-oriented NoSQL database built on top of the Hadoop Distributed File System (HDFS). Its architecture is designed for handling large volumes of data in a distributed environment. Let's break down the components you mentioned:

### 1. HMaster:

- HMaster is a special daemon responsible for coordinating and managing the cluster. It oversees the assignment of regions to RegionServers, monitors their health, and handles administrative tasks such as schema changes and balancing regions across the cluster.
- HMaster keeps track of which regions are assigned to which RegionServers and manages the metadata about the HBase cluster.

### 2. HRegionServer:

- HRegionServer is a process running on each node in the Hadoop cluster. It manages one or more regions (data partitions) of the HBase tables.
- HRegionServer handles data read and write requests from clients. It also handles operations like compactions, splits, and flushing data to disk.
- Each HRegionServer communicates with the HMaster to report its status and receive instructions.

### 3. HRegions:

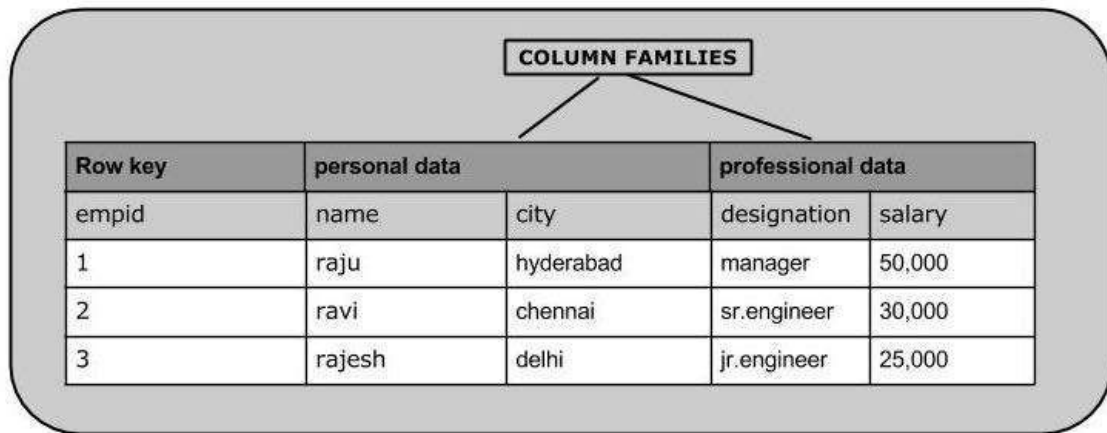
- HRegion is a contiguous portion of a table's data, and each table is divided into multiple regions. These regions are distributed across the cluster.
- HRegions contain the actual data stored in HBase tables. They are stored in HDFS files called HFiles, which are immutable and append-only.
- When a region grows too large, it can be split into two or more smaller regions to improve performance and scalability. This process is called region splitting.

#### **4. Zookeeper:**

- Zookeeper is a centralized service for maintaining configuration information, providing distributed synchronization, and group services.
- HBase uses Zookeeper for coordinating distributed operations, such as leader election, cluster membership management, and distributed locking.
- Zookeeper helps in managing the state of the HBase cluster and ensuring consistency and reliability in the presence of failures.

#### **5. HDFS (Hadoop Distributed File System):**

- HDFS is the underlying distributed file system used by HBase for storing its data. It is a scalable and fault-tolerant file system designed for storing large datasets.
- HBase stores its data in HDFS in the form of HFiles, which are stored in blocks across the HDFS cluster.
- HDFS provides the necessary storage infrastructure for HBase, ensuring data durability, fault tolerance, and scalability.



ZooKeeper in HBase serves multiple purposes:

1. **Leader Election:** ZooKeeper helps in electing a leader among the HMaster candidates. The HMaster leader coordinates activities such as assigning regions to RegionServers and handling administrative tasks.
2. **Cluster Coordination:** ZooKeeper maintains metadata about the cluster, such as the location of regions, live RegionServers, and the current HMaster.
3. **Configuration Management:** HBase uses ZooKeeper to store and distribute configuration information across the cluster. This allows all nodes in the cluster to have access to consistent configuration settings.
4. **Lock Management:** ZooKeeper provides distributed locks that HBase uses to coordinate actions that need to be performed by only one node at a time, ensuring consistency and avoiding conflicts.

Now, here are some basic HBase commands to store and select data in the HBase database:

#### 1. Creating a Table:

```
create 'my_table', 'cf1', 'cf2'
```

#### 1. Inserting Data:

```
put 'my_table', 'row1', 'cf1:col1', 'value1'
```

```
put 'my_table', 'row1', 'cf1:col2', 'value2'
```

### 1. Retrieving Data:

```
get 'my_table', 'row1'
```

### 1. Scanning Data:

```
scan 'my_table'
```

These commands allow you to create a table, insert data into it, retrieve specific rows, and scan the entire table for data. You can further refine the scans and queries using filters and qualifiers based on your requirements.



Why hbase is called column-oriented NoSQL database built on top of HDFS? What are the commands to STORE, SELECT, MODIFY, and DELETE records from a table of Hbase. (10)

HBase is called a column-oriented NoSQL database because it organizes data in tables with rows and columns, much like traditional relational databases. However, unlike relational databases where data is stored in rows and accessed by row keys, HBase stores data in columns and accesses data based on column families and column qualifiers. This column-oriented storage model allows for efficient read and write operations, especially when dealing with large amounts of data.

HBase is built on top of Hadoop Distributed File System (HDFS), which provides scalability, fault-tolerance, and reliability for storing large datasets across a cluster of commodity hardware.

Here are some basic commands to perform CRUD operations (Create, Read, Update, Delete) in HBase:



### 1. Create Table:

```
shellCopy code
create 'tableName', 'columnFamily1', 'columnFamily2', ...
```

Example:

```
shellCopy code
create 'employee', 'personal', 'professional'
```

### 1. Insert/Store Record:

```
shellCopy code
put 'tableName', 'rowKey', 'columnFamily:columnQualifier',
'value'
```

Example:

```
shellCopy code
put 'employee', '1001', 'personal:name', 'John'
put 'employee', '1001', 'personal:age', '30'
put 'employee', '1001', 'professional:title', 'Software Eng
ineer'
```

### 1. Select/Read Record:

```
shellCopy code
get 'tableName', 'rowKey'
```

Example:

```
shellCopy code
get 'employee', '1001'
```

### 1. Modify Record:

```
shellCopy code
put 'tableName', 'rowKey', 'columnFamily:columnQualifier',
'newValue'
```

Example:

```
shellCopy code
put 'employee', '1001', 'professional:title', 'Senior Software Engineer'
```

### 1. Delete Record:

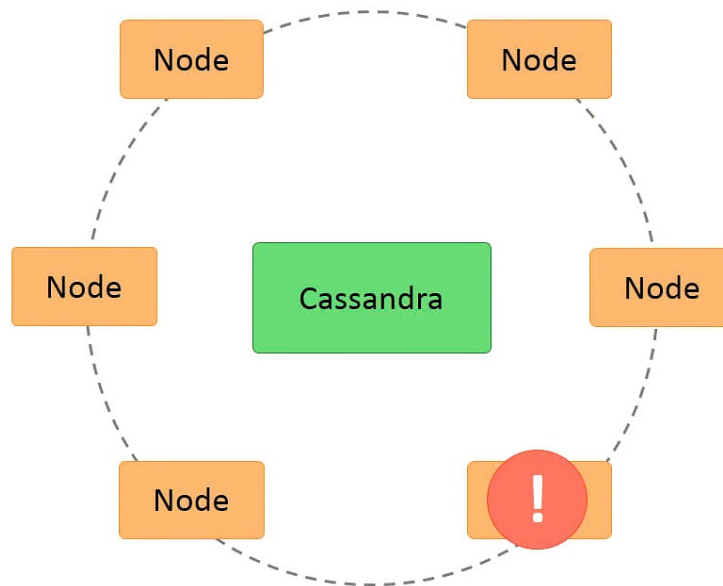
```
shellCopy code
delete 'tableName', 'rowKey', 'columnFamily:columnQualifier'
```

Example:

```
shellCopy code
delete 'employee', '1001', 'personal:age'
```

These are some of the basic commands in HBase for CRUD operations. There are more advanced operations and configurations available depending on the specific requirements and use cases.

## Cassandra



Cassandra is a distributed, decentralized, highly available, and fault-tolerant NoSQL database system. Its architecture is designed to handle massive amounts of data across multiple nodes while providing high performance and scalability. Here's an overview of the architecture of Cassandra:

### 1. **Node:**

- The basic building block of a Cassandra cluster is a node. Each node is an individual instance of the Cassandra database running on a physical or virtual machine.
- Nodes can be added or removed dynamically to scale the cluster up or down based on the workload.

### 2. **Data Distribution:**

- Cassandra uses a distributed hash table (DHT) to distribute data across multiple nodes in the cluster.
- Data is partitioned and stored in partitions called "token ranges". Each node is responsible for storing data within a specific token range.
- Cassandra uses consistent hashing to determine which node is responsible for storing each piece of data.

### 3. **Replication:**

- Cassandra replicates data across multiple nodes to ensure fault tolerance and high availability.

- Replication strategy and replication factor determine how many copies of each piece of data are stored and on which nodes.
- Different data centers can also be used for replication to provide geographic redundancy and disaster recovery.

#### **4. Gossip Protocol:**

- Cassandra uses a gossip protocol for communication and discovery between nodes.
- Nodes periodically exchange information about their state, including their status, uptime, and data they are responsible for.
- Gossip ensures that each node has an up-to-date view of the cluster topology and can adjust its behavior accordingly.

#### **5. Data Model:**

- Cassandra has a flexible schema that allows for dynamic addition and modification of columns.
- Data is organized into tables, similar to relational databases, with rows identified by a unique primary key.
- Each table can have multiple columns, and each column can have multiple values (similar to a wide-row/column-family model).

#### **6. Read and Write Operations:**

- Cassandra supports fast read and write operations by minimizing disk seeks and utilizing in-memory data structures.
- Write operations are typically asynchronous and durable, meaning data is first written to a commit log and then to an in-memory data structure called a memtable before being flushed to disk.
- Read operations can be served from memtables or disk based on the data availability and consistency level requirements.

#### **7. Fault Tolerance and High Availability:**

- Cassandra is designed to be fault-tolerant, with no single point of failure.
- Data is replicated across multiple nodes, so if one node fails, data can be retrieved from replicas on other nodes.

- Cassandra uses strategies like hinted handoff, read repair, and anti-entropy repair to maintain consistency and recover from failures.



Explain eventual consistency and tunable consistency in context of Cassandra. (5)

### **Eventual Consistency:**

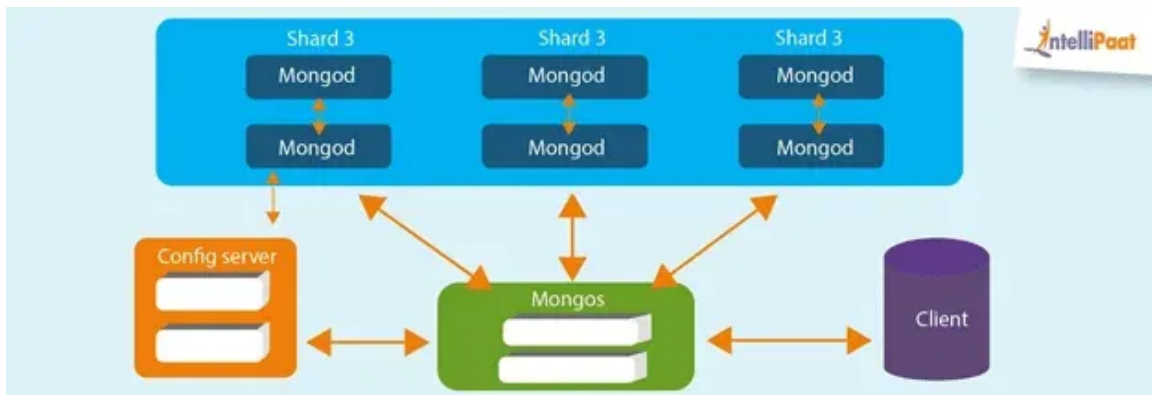
Eventual consistency is a consistency model where, after a certain period of time with no updates, all replicas of the data will converge to the same value. In Cassandra, eventual consistency is achieved through the use of a distributed replication strategy, typically the "last write wins" approach. When a write operation occurs, it is asynchronously propagated to multiple replicas across different nodes in the cluster. However, due to network latency and other factors, these replicas may not immediately receive the update. As a result, during this period of inconsistency, clients may read different versions of the data from different replicas. Eventually, all replicas will converge to the same value, ensuring eventual consistency.

### **Tunable Consistency:**

Tunable consistency in Cassandra refers to the ability to adjust the level of consistency for read and write operations based on specific requirements of the application. Cassandra provides tunable consistency through its consistency level settings, which allow developers to specify how many replicas must respond to a read or write operation for it to be considered successful. Consistency levels range from "ALL" (requiring all replicas to respond) to "ONE" (requiring only one replica to respond), providing a spectrum of consistency options. By adjusting the consistency level, developers can balance between consistency, availability, and partition tolerance according to the needs of their application. This flexibility allows developers to achieve the desired level of consistency while optimizing performance and fault tolerance.

## **MongoDB**

MongoDB is a popular NoSQL database that stores data in a flexible, JSON-like format called BSON (Binary JSON). It follows a distributed architecture that provides high scalability, availability, and performance. Here's an overview of the architecture of MongoDB:



1. **Shards:** Shards are individual MongoDB instances that store a portion of the data in the database. Each shard contains a subset of the total data set. When data is distributed across multiple shards, it allows for horizontal scalability, meaning you can increase the capacity of your database by adding more shards.
2. **Mongos (MongoDB Router):** Mongos acts as a router or proxy for client requests. It's essentially the interface between the application and the sharded cluster. When a client wants to read or write data, it sends the request to Mongos, which then routes the request to the appropriate shard or shards. Mongos is responsible for understanding the sharding configuration and directing queries accordingly.
3. **Config Servers:** Config servers store metadata and configuration settings for the sharded cluster. This includes information about which data is stored on which shards, as well as chunk ranges (chunks are portions of data distributed across shards). Config servers maintain a map of the cluster's data distribution, which is crucial for Mongos to route requests correctly.
4. **Client:** The client could be any application or process that interacts with the MongoDB database. Clients send read and write operations to the Mongos instances, which then route those operations to the appropriate shards.

In summary, the sharded architecture of MongoDB allows for horizontal scalability by distributing data across multiple shards. Mongos acts as the interface between clients and the cluster, while config servers store metadata about the cluster's configuration. This setup enables efficient querying and scaling of MongoDB databases to handle large volumes of data.



How can you model RDMS table in Mongo DB? Give an example.  
(4+6)

To model a relational database management system (RDBMS) table in MongoDB, you typically need to denormalize the data and structure it in a way that fits MongoDB's document-oriented model. Here's an example of how you can model an RDBMS table into a MongoDB collection:

Let's say you have an RDBMS table named `employees` with the following columns:

1. `id` (Primary Key)
2. `name`
3. `age`
4. `department`

You can model this table in MongoDB as a collection named `employees` with documents representing individual employees. Each document would contain fields corresponding to the columns of the RDBMS table.

Here's how you might structure the data in MongoDB:

```
[
  {
    "_id": 1,
    "name": "John Doe",
    "age": 30,
    "department": "Engineering"
  },
  {
    "_id": 2,
    "name": "Jane Smith",
    "age": 35,
    "department": "Marketing"
  },
  {
    "_id": 3,
```

```
    "name": "Bob Johnson",
    "age": 40,
    "department": "Sales"
  }
]
```

In this MongoDB collection:

- Each document represents an employee.
- The `_id` field serves as the unique identifier for each document (similar to the primary key in RDBMS).
- The `name`, `age`, and `department` fields correspond to the columns in the RDBMS table.

This denormalized structure allows you to efficiently query and manipulate the data in MongoDB. However, it's important to note that the specific structure of your MongoDB documents will depend on your application's requirements and use cases. You may need to further denormalize or nest data within documents based on how you plan to query and update the data.



Explain the term NO-SQL. Justify for distributed scenario normalization contradicts the data availability. (3+7)

In databases, normalization is the process of organizing data in a database efficiently. This involves reducing data redundancy and improving data integrity. Normalization typically involves dividing large tables into smaller ones and defining relationships between them.

In a distributed scenario, data is spread across multiple nodes or servers to improve scalability, fault tolerance, and performance. However, achieving full normalization in such a distributed environment can lead to challenges, particularly concerning data availability. Here's why:



1. **Data Distribution:** When data is distributed across multiple nodes, enforcing strict normalization rules can lead to frequent cross-node joins, which can be resource-intensive and impact performance. In a distributed system, joins involving data from different nodes can incur network latency and overhead.
2. **Transaction Coordination:** Normalization often requires maintaining strong consistency, meaning that all distributed copies of related data must be updated atomically to reflect changes. Achieving this level of consistency across distributed nodes can be complex and may require coordination mechanisms like distributed transactions, which can introduce latency and decrease availability.
3. **Partitioning:** NoSQL databases often use partitioning to distribute data across nodes. However, fully normalized data may not partition well, as related data that needs to be accessed together may be spread across different partitions. This can lead to increased network communication and reduced performance.
4. **Data Redundancy:** Normalization aims to minimize data redundancy by breaking down data into smaller tables. However, in a distributed environment, some degree of redundancy may be necessary to ensure data availability and fault tolerance. Redundant copies of data can be stored across multiple nodes to ensure that data remains accessible even if some nodes fail.



Hbase, Cassandra and MongoDB are called column oriented NoSQL database? How row-oriented database differ from column-oriented database? Explain with suitable examples. (10)

HBase, Cassandra, and MongoDB are actually not all considered column-oriented databases. HBase and Cassandra are column-family stores, which are a type of NoSQL database that organizes data into columns grouped together within column families. MongoDB, on the other hand, is a document-oriented database, which stores data in flexible, JSON-like documents.

Row-oriented and column-oriented databases are two different ways of organizing and storing data, each with its own advantages and disadvantages.

### 1. Row-Oriented Database:

- In a row-oriented database, data is stored and retrieved by rows. Each row represents a record or entity, and all the columns related to that entity are stored together.
- Row-oriented databases are typically good for transactional workloads or when data is accessed and updated row by row.
- Example: MySQL, PostgreSQL. Consider a table storing customer information. Each row represents a single customer, with columns for attributes such as name, age, address, etc. When you query for a specific customer, the database retrieves the entire row of data.

### 2. Column-Oriented Database:

- In a column-oriented database, data is stored and retrieved by columns. Each column is stored together, independent of other columns, and all the values for a specific column are stored contiguously.
- Column-oriented databases are advantageous for analytical workloads or when you need to aggregate data across multiple rows and columns.
- Example: Apache HBase, Apache Cassandra. Consider a table storing sales data. Each column represents a different attribute such as product ID, customer ID, sales amount, etc. When you need to calculate total sales for a specific product across all customers, a column-oriented database can efficiently retrieve and aggregate just the sales amount column for that product.

### 1. Data Storage Format:

- **Row-oriented databases:** In a row-oriented database, data is stored in rows. This means that all the values for a single record are stored together.
- **Column-oriented databases:** In a column-oriented database, data is stored in columns. This means that all the values for a single attribute (or column) are stored together.

### 2. Access Patterns:

- **Row-oriented databases:** These are optimized for transactional workloads and operations that involve retrieving entire records. In OLTP (Online Transaction Processing) scenarios, where individual records are frequently accessed and modified, row-oriented databases tend to perform well.
- **Column-oriented databases:** These are optimized for analytical or reporting workloads, where queries typically involve aggregating data from a few columns across a large number of rows. In OLAP (Online Analytical Processing) scenarios, column-oriented databases are often faster because they only need to read the specific columns involved in a query.

### 3. Performance:

- **Row-oriented databases:** These are generally more suitable for transactional operations with a smaller number of records being retrieved or modified at a time.
- **Column-oriented databases:** These are more efficient for analytical operations that involve scanning and processing large amounts of data across a few columns.

### 4. Compression:

- **Row-oriented databases:** Compression is typically less effective in row-oriented databases because similar data isn't grouped together.
- **Column-oriented databases:** Compression can be highly effective in column-oriented databases because similar data is stored together, making it easier to identify and eliminate redundancy.

### 5. Aggregation and Analytics:

- **Row-oriented databases:** These may suffer performance issues when performing aggregate queries over large datasets because they have to read and process entire rows, even if only a few columns are needed.
- **Column-oriented databases:** These excel at aggregate queries because they only need to access the columns relevant to the query, ignoring irrelevant data.

### 6. Indexing:

- **Row-oriented databases:** Traditional indexing techniques like B-trees are commonly used to optimize row-oriented databases for efficient

row retrieval.

- **Column-oriented databases:** Indexing strategies can differ from row-oriented databases due to the nature of data storage. Techniques like bitmap indexes are more commonly used to speed up query performance.