**UNIT 2: NO SQL Databases**

# SYLLABUS:

NoSQL – CAP Theorem – Sharding - Document based – MongoDB Operation: Insert, Update, Delete, Query, Indexing, Application, Replication, Sharding, Deployment – Using MongoDB with PHP / JAVA – Advanced MongoDB Features – Cassandra: Data Model, Key Space, Table Operations, CRUD Operations, CQL Types – HIVE: Data types, Database Operations, Partitioning – HiveQL – OrientDB Graph database – OrientDB Features

# CONTEXT:

1. NoSQL:

a. Introduction to NoSQL databases and their characteristics.

b. Comparison between NoSQL and relational databases.

c. Different types of NoSQL databases: key-value stores, document-based databases, columnar databases, and graph databases.

d. Use cases and advantages of NoSQL databases.

e. CAP theorem and its relevance to NoSQL databases.

f. ACID vs. BASE consistency models in NoSQL databases.

2. CAP Theorem:

a. Explanation of the CAP theorem and its three components: Consistency, Availability, and Partition Tolerance.

b. Detailed discussion of the trade-offs involved in achieving CAP properties.

c. Examples of popular databases and their approach to CAP trade-offs.

d. CAP theorem in the context of NoSQL databases.

3. Sharding:

a. Introduction to database sharding and its role in scalability.

b. Techniques for sharding: range-based partitioning, hash-based partitioning, and consistent hashing.

c. Sharding strategies and considerations for distributing data across shards.

d. Benefits and challenges of sharding in distributed databases.

e. Tools and technologies for implementing sharding in different databases.

4. MongoDB:

a. Overview of MongoDB as a document-based NoSQL database.

b. MongoDB data model: collections, documents, and fields.

c. CRUD operations in MongoDB: Insert, Update, Delete, and Query.

d. Indexing in MongoDB and its impact on query performance.

e. Application integration with MongoDB using PHP and Java.

f. Replication in MongoDB: replica sets, primary-secondary architecture, and data redundancy.

g. Sharding in MongoDB: concepts, sharding key selection, and data distribution.

h. Deployment options for MongoDB: standalone, replica sets, and sharded clusters.

i. Advanced features in MongoDB: aggregation framework, transactions, geospatial indexing, and full-text search.

5. Cassandra:

a. Introduction to Cassandra as a distributed NoSQL database.

b. Cassandra data model: keyspaces, tables, rows, and columns.

c. CRUD operations in Cassandra: Create, Read, Update, and Delete.

d. Data modelling in Cassandra: denormalization, partition keys, clustering columns, and secondary indexes.

e. CQL (Cassandra Query Language) types: basic types, collections, and user-defined types.

f. Replication in Cassandra: replication factor, consistency levels, and fault tolerance.

g. Scaling and sharding in Cassandra: adding nodes, virtual nodes, and data distribution.

h. Advanced features of Cassandra: lightweight transactions, materialized views, and compaction strategies.

6. Hive:

a. Introduction to Hive as a data warehousing and analytics platform built on top of Hadoop.

b. Hive data types: primitive types, complex types, and type conversion.

c. Database operations in Hive: creating databases, tables, and partitions.

d. Data loading techniques in Hive: INSERT, LOAD DATA, and external tables.

e. HiveQL: syntax, data manipulation statements, and data query operations.

f. Partitioning in Hive: benefits, partition types, and partition pruning.

g. Performance optimization techniques in Hive: indexing, query optimization, and statistics.

7. OrientDB:

a. Introduction to OrientDB as a graph database management system.

b. Features of OrientDB: ACID transactions, schema-less design, and hybrid document-graph model.

c. Graph database concepts: vertices, edges, properties, and graph traversal.

d. OrientDB graph operations: creating vertices and edges, updating properties, and traversing the graph.

e. Indexing in OrientDB: automatic indexing, manual indexing, and full-text search.

f. OrientDB as a multi-model database: support for SQL, graph, and document querying.

# 2.1. NoSQL-Overview:

## 2.1.1Understanding NoSQL Databases and Their Characteristics

NoSQL (Not only SQL) databases are a class of database management systems that provide a flexible and scalable alternative to traditional relational databases. Unlike relational databases, which are based on the SQL (Structured Query Language) model, NoSQL databases use various data models to store and retrieve data. This chapter provides an overview of NoSQL databases, their characteristics, and how they differ from relational databases.

### 2.1.1.1 Characteristics of NoSQL Databases

NoSQL databases possess the following key characteristics:

1. Scalability: NoSQL databases are designed to handle large volumes of data and high traffic loads. They can scale horizontally by adding more servers, enabling efficient data distribution and improved performance.
2. Flexibility: NoSQL databases allow for dynamic and schema-less data models, enabling easy adaptation to evolving data structures without rigid schemas or predefined relationships.
3. High Availability: NoSQL databases prioritize availability by replicating data across multiple nodes, ensuring uninterrupted access to data even in the presence of failures.
4. Performance: NoSQL databases optimize for read and write operations, providing high-speed data access by leveraging techniques such as in-memory caching and parallel processing.
5. Distributed Architecture: NoSQL databases are built with distributed architectures that enable data to be distributed across multiple servers or clusters, supporting fault tolerance and load balancing.
6. Complex-free working: Unlike SQL databases, NoSQL databases are not complicated. They store data in an unstructured or a semi-structured form that requires no relational or tabular arrangement. Perhaps they are easier to use and can be accomplished by all.
7. Durable: If durability is not one of its most striking features, then what is? NoSQL databases are highly durable as they can accommodate data ranging from heterogeneous to homogeneous. Not only can they accommodate structured data, but they can also incorporate unstructured data that requires no query language. Undoubtedly, these databases are durable and efficient.

## 2.1.2: Comparison between NoSQL and Relational Databases

Relational databases have been the traditional choice for structured data storage and management. However, NoSQL databases offer several advantages over relational databases, making them suitable for specific use cases. This section compares the two types of databases and highlights their differences.

### 2.1.2.1 Key Differences between NoSQL and Relational Databases

Certainly! Here are some of the key differences between NoSQL and Relational databases:

|  |  |  |
| --- | --- | --- |
| Property | NoSQL | SQL / Relational Databases |
| 1. Data Model | NoSQL databases use flexible and schema-less data models. They can handle unstructured, semi-structured, and structured data. Common NoSQL data models include key-value, document, column-family, and graph. | Relational databases follow a structured data model based on tables, rows, and columns. They enforce a fixed schema where data must conform to predefined table structures. |
| 2. Scalability | NoSQL databases are designed to scale horizontally, meaning they can handle large amounts of data and high traffic by distributing the workload across multiple servers or clusters. | Relational databases traditionally scale vertically, where they are limited by the capacity of a single server. However, some relational databases also offer mechanisms for horizontal scalability. |
| 3. Data Relationships | NoSQL databases can handle relationships between data but do not provide built-in support for complex joins across multiple collections or tables. Instead, denormalization and embedding techniques are often used to optimize data retrieval. | Relational databases excel in handling complex relationships between tables. They offer robust support for joins, foreign keys, and enforcing referential integrity. |
| 4. Schema Flexibility | NoSQL databases allow flexible schemas, meaning each record or document can have a different structure. They accommodate evolving data requirements and enable easy schema changes without affecting existing data. | Relational databases have rigid schemas, and any changes to the schema require careful planning and potentially altering existing data, which can be time-consuming and disruptive. |
| 5. ACID Transactions | NoSQL databases, particularly in the AP category, often sacrifice full ACID (Atomicity, Consistency, Isolation, Durability) transaction support to achieve scalability and performance. They may offer limited transactional guarantees or eventual consistency instead. | Relational databases prioritize ACID transactions, providing strong consistency and isolation guarantees. They ensure that transactions either complete in full or roll back entirely. |
| 6. Query Language | NoSQL databases often have their own query languages or interfaces specific to their data models. Examples include MongoDB's query language, Cassandra Query Language (CQL), or graph traversal languages like Gremlin. | Relational databases commonly use SQL (Structured Query Language) as the standard language for querying and manipulating data. SQL offers a rich set of operations for data retrieval, filtering, sorting, and aggregating. |
| 7. Vertical Partitioning | NoSQL databases can easily handle vertical partitioning, which involves splitting data vertically based on specific attributes or columns. It allows efficient storage and retrieval of frequently accessed attributes. | Relational databases typically store entire rows in a table, making vertical partitioning less straightforward. Data separation often requires creating separate tables and managing relationships between them. |
| 8. Use Cases | NoSQL databases are well-suited for handling large volumes of unstructured or semi-structured data, real-time analytics, high-traffic web applications, and scenarios that demand horizontal scalability and agility in schema design. | Relational databases excel in use cases that require complex querying, strong consistency, ACID transactions, and maintaining data integrity. They are commonly used for financial systems, e-commerce platforms, and applications with strict data relationships. |

## 2.1.3: Types of NoSQL Databases

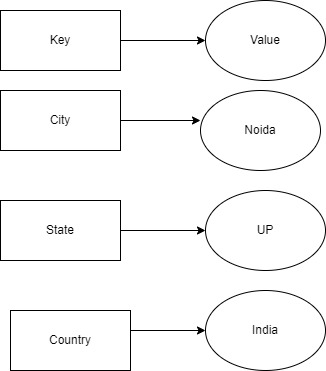
NoSQL databases can be classified into several types based on their data models and storage mechanisms. This section explores the most common types of NoSQL databases.

* Key-Value Stores: Key-value stores store data as a collection of key-value pairs, where each key is unique and associated with a value. They offer high-speed read and write operations, making them suitable for caching, session management, and simple data retrieval.
* Document-Based Databases: Document-based databases store data in flexible, self-describing documents, typically in formats such as JSON or XML. They allow for hierarchical data structures and are well-suited for content management systems, real-time analytics, and unstructured data storage.
* Columnar Databases: Columnar databases organize data by column rather than by row, providing efficient data compression and improved query performance for analytical workloads. They are commonly used in data warehousing, time-series analysis, and business intelligence applications.
* Graph Databases: Graph databases store data in nodes and edges, representing entities and relationships between them. They excel at handling highly interconnected data, making them ideal for social networks, recommendation systems, and network analysis.

### Key-Value Stores

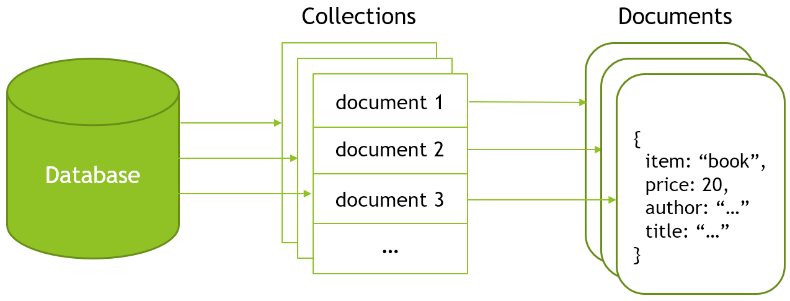
Key-value stores are simple yet powerful NoSQL databases that store data as a collection of key-value pairs. Each key is unique and associated with a corresponding value, which can be any type of data, such as strings, numbers, or even complex objects.

Key-value stores offer fast and efficient data retrieval by directly accessing values using their unique keys. They are highly scalable and can handle large volumes of data with ease. Key-value stores are commonly used for caching, session management, and storing user preferences.



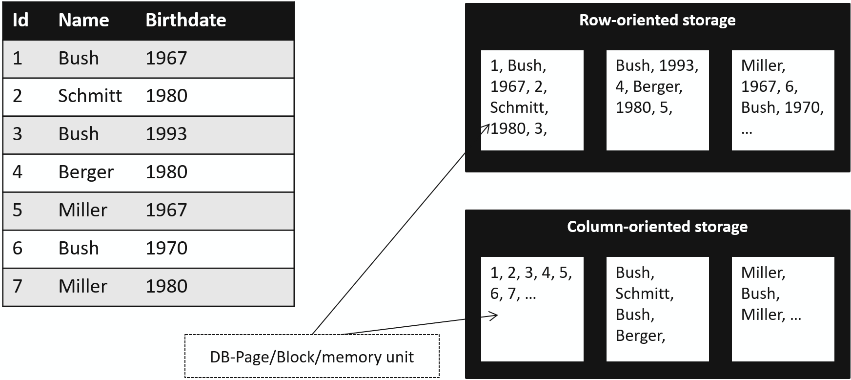
### Document-Based Databases

Document-based databases store data in flexible, self-describing documents, typically in formats like JSON (JavaScript Object Notation) or XML (eXtensible Markup Language). Each document contains key-value pairs, and documents can be nested to represent complex data structures.Document-based databases provide schema flexibility, allowing for dynamic and evolving data models. They are well-suited for content management systems, real-time analytics, and applications with semi-structured or unstructured data.



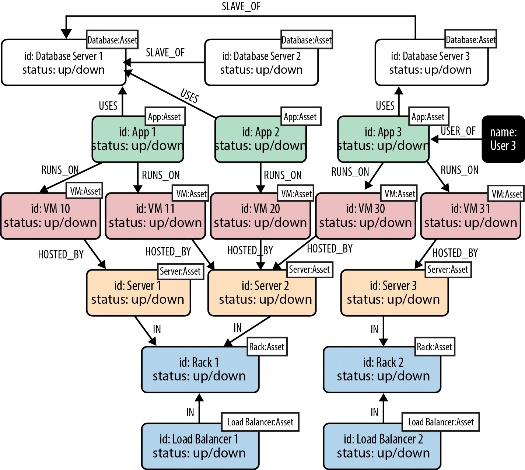
### Columnar Databases

Columnar databases organize data based on columns rather than rows. Instead of storing complete records together, they store data vertically, column by column. This storage format offers significant advantages for analytical workloads and data warehousing. Columnar databases provide high compression rates, efficient data retrieval, and query performance for analytical queries that involve aggregations and reporting. They are commonly used in business intelligence applications, time-series analysis, and data analytics.



### Graph Databases

Graph databases model data as nodes and edges, representing entities and relationships between them. Nodes represent entities (e.g., people, products), and edges represent the connections or relationships between entities. Graph databases excel at handling highly interconnected data and complex relationships. They are well-suited for use cases such as social networks, recommendation systems, fraud detection, and network analysis.



## 2.1.4: Use Cases and Advantages of NoSQL Databases

NoSQL databases offer distinct advantages and excel in various use cases where traditional relational databases may fall short. This section explores some common use cases and benefits of using NoSQL databases.

### 2.1.4.1 Use Cases for NoSQL Databases

Big Data: NoSQL databases handle the storage and processing of large-scale data sets, such as web analytics, sensor data, and log files.

Real-Time Applications: NoSQL databases provide low-latency data access, making them suitable for applications that require real-time updates and responsiveness, such as gaming, chat applications, and financial systems.

Content Management Systems: Document-based databases excel in managing content-heavy applications like content management systems, blogs, and e-commerce platforms.

Scalable Web Applications: NoSQL databases enable the horizontal scaling of web applications, supporting high user loads and accommodating rapidly growing data.

### 2.1.4.2 Advantages of NoSQL Databases

Scalability: NoSQL databases offer seamless horizontal scalability, enabling efficient handling of large data volumes and increased traffic.

Flexibility: NoSQL databases allow for flexible schema-less data models, adapting to changing requirements and evolving data structures.

High Performance: NoSQL databases optimize read and write operations, providing high-speed data access and improved application performance.

Cost-Effectiveness: NoSQL databases can be more cost-effective than relational databases, as they can run on commodity hardware and require less administration.

## 2.1.5: CAP Theorem and Its Relevance to NoSQL Databases

### 2.1.5.1 Understanding the CAP Theorem

The CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide consistency, availability, and partition tolerance. These three properties form the pillars of distributed systems, and the CAP theorem helps us understand the trade-offs that need to be made in such systems.

- Consistency (C): Consistency refers to ensuring that all nodes in a distributed system have the same view of the data at any given point in time. In other words, all reads and writes are guaranteed to see the latest, most up-to-date data.

- Availability (A): Availability ensures that the system remains responsive and accessible, even in the face of failures or network partitions. It means that every request made to the system receives a timely response, either with the requested data or an error message.

- Partition Tolerance (P): Partition tolerance deals with the system's ability to function and tolerate network partitions or communication failures between different nodes in a distributed system. Network partitions occur when nodes are unable to communicate with each other, leading to the system being split into multiple isolated parts.

### 2.1.5.2 Relevance of CAP Theorem to NoSQL Databases

NoSQL databases are built with a focus on scalability and high availability, which makes them prone to CAP theorem implications. Given the distributed nature of NoSQL databases, they often prioritize either availability or consistency in the face of network partitions.

- AP Systems: Some NoSQL databases prioritize Availability and Partition tolerance over strong Consistency. In AP systems, the database continues to provide service even in the presence of network partitions. This approach sacrifices strong consistency, allowing for eventual consistency or conflicting views of data across different nodes. Examples of AP NoSQL databases include Cassandra and RIAK.

- CP Systems: Other NoSQL databases prioritize Consistency and Partition tolerance, ensuring that data remains consistent across all nodes, even in the presence of network partitions. In CP systems, availability may be compromised during network partitions to maintain strong consistency. Examples of CP NoSQL databases include MongoDB and Redis.

## 2.1.6: ACID vs. BASE Consistency Models in NoSQL Databases

### 2.1.6.1. ACID Consistency Model

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that ensure reliable transaction processing in relational databases. The ACID consistency model guarantees that transactions are executed in an "all-or-nothing" manner, maintaining data integrity and consistency.

- Atomicity: Atomicity guarantees that a transaction is treated as a single unit of work, ensuring that either all operations within the transaction are executed successfully or none of them are.

- Consistency: Consistency ensures that the database remains in a valid state before and after the execution of a transaction. It enforces integrity constraints and rules defined by the database schema.

- Isolation: Isolation ensures that concurrent transactions do not interfere with each other, maintaining data integrity and preventing conflicts.

- Durability: Durability guarantees that once a transaction is committed, its effects are permanent and will survive any subsequent failures, such as power outages or system crashes.

### 2.1.6.2. BASE Consistency Model

BASE (Basically Available, Soft state, eventually consistent) is an alternative consistency model commonly associated with NoSQL databases. Unlike the strict consistency model of ACID, BASE allows for more relaxed consistency guarantees in order to achieve high availability and scalability.

- Basically Available: BASE systems prioritize high availability, ensuring that the system remains accessible and responsive even in the presence of failures or network partitions.

- Soft state: soft state implies that the system's state can change over time, allowing for eventual consistency. It means that during concurrent updates, different nodes may have different views of the data for a certain period.

- Eventually Consistent: Eventually consistency implies that given enough time and the absence of further updates, all replicas in a distributed system will eventually converge to the same consistent state. It does not guarantee immediate consistency across all nodes.

NoSQL databases often adopt the BASE consistency model to handle the challenges of scalability, high availability, and distributed data storage, while sacrificing the strict consistency guarantees provided by ACID.

# 2.2. CAP Theorem:

## 2.2.1. Explanation of the CAP theorem and its three components: Consistency, Availability, and Partition Tolerance.

The CAP theorem, proposed by computer scientist Eric Brewer, states that it is impossible for a distributed data system to simultaneously provide all three of the following properties: Consistency, Availability, and Partition Tolerance.

1. Consistency: Consistency refers to the requirement that all nodes in a distributed system see the same data at the same time. In other words, any read operation on the system will always return the most recent write or an error. Achieving consistency ensures that data remains coherent and correct across all nodes. In a consistent system, updates or modifications made to the data are immediately visible to all nodes in the system.

2. Availability: Availability refers to the ability of a distributed system to continue functioning and providing responses to user requests, despite individual node failures or network partitions. An available system ensures that every request eventually receives a response, even if it is a slightly stale or outdated version of the data. It focuses on providing uninterrupted access to data and services, even in the presence of failures or disruptions.

3. Partition Tolerance: Partition Tolerance refers to the system's ability to maintain its operations and availability even when network partitions occur. A network partition happens when communication failures or network issues isolate certain nodes or groups of nodes from each other. Network partitions can lead to message delays, loss, or inconsistency in data propagation. A partition-tolerant system can continue to function and serve requests, ensuring that the system remains operational despite these partitions.

According to the CAP theorem, in the face of a network partition, a distributed system must choose between consistency and availability, as it is impossible to guarantee both simultaneously. The theorem does not imply that one needs to sacrifice partition tolerance. Instead, it suggests that during a partition, the system can either maintain consistency and sacrifice availability (CP), or prioritize availability and sacrifice strong consistency (AP).

It is important to note that the CAP theorem assumes that network partitions are inevitable in distributed systems, and it forces system designers to make trade-offs based on the desired system behaviour and the specific requirements of their applications. The choice between CP and AP depends on the nature of the application and the importance of consistency and availability in different scenarios.

In practice, various databases and distributed systems adopt different strategies to handle the trade-offs imposed by the CAP theorem. Some systems prioritize consistency over availability, while others prioritize availability over strong consistency. NoSQL databases often choose AP, focusing on availability and partition tolerance while allowing for flexible consistency models.

Overall, the CAP theorem serves as a guide for understanding the challenges and trade-offs involved in designing and implementing distributed systems, enabling architects and developers to make informed decisions based on the specific needs of their applications.

## 2.2.2. Detailed discussion of the trade-offs involved in achieving CAP properties.

The trade-offs involved in achieving CAP properties are as follows:

- Consistency vs. Availability: The trade-off between consistency and availability is the core decision in the CAP theorem. In scenarios where consistency is prioritized, the system ensures that all nodes have the most up-to-date data, but it may lead to unavailability during network partitions or node failures. On the other hand, prioritizing availability allows the system to remain accessible and provide responses even during network partitions, but it may sacrifice immediate consistency, leading to eventual consistency or stale data.

- Consistency vs. Partition Tolerance: In the face of a network partition, maintaining both consistency and availability becomes challenging. If a system aims for strong consistency, it may choose to block or pause operations until the partition is resolved to maintain data coherence. This can lead to unavailability during partitions, violating the partition tolerance requirement. Alternatively, a system can prioritize partition tolerance by allowing independent operation of nodes during a partition, which may result in temporary inconsistency.

- Availability vs. Partition Tolerance: During network partitions, ensuring availability becomes difficult. To provide availability, the system may allow each node to operate independently, even if it leads to inconsistencies across nodes. This approach prioritizes partition tolerance at the expense of availability during partitions. However, when the partition is resolved, the system can converge and provide consistent data again.

## 2.2.3. Examples of popular databases and their approach to CAP trade-offs.

Different databases adopt varying strategies to handle the trade-offs imposed by the CAP theorem. Here are a few examples:

- Relational databases (e.g., MySQL, PostgreSQL): Relational databases traditionally prioritize consistency over availability and partition tolerance. They aim to provide strong consistency guarantees and typically block or reject operations during network partitions to maintain data integrity. These databases follow the CP model.

- Key-value stores (e.g., Redis): Key-value stores often prioritize partition tolerance and availability over immediate consistency. They aim to provide high availability and fast response times, even if it means eventual consistency or temporary data divergence during partitions. These databases follow the AP model.

- Distributed file systems (e.g., Hadoop HDFS): Distributed file systems often prioritize partition tolerance and availability as well. They are designed to handle large-scale data storage and processing, and they focus on maintaining system availability even during network partitions. Consistency may be sacrificed temporarily during partitions, but the system eventually converges to a consistent state when the partition is resolved.

## 2.2.4. CAP theorem in the context of NoSQL databases.

NoSQL databases, which emerged to handle the challenges of big data and scalability, often emphasize availability and partition tolerance over immediate consistency. They aim to provide high scalability, fault tolerance, and low-latency access to data. Many NoSQL databases follow the AP model, sacrificing strict consistency for the sake of availability and partition tolerance.

NoSQL databases come in various forms, including document databases (e.g., MongoDB), column-family stores (e.g., Cassandra), graph databases (e.g., Neo4j), and more. Each type of NoSQL database has its own approach to handling consistency and availability trade-offs, with some providing tuneable consistency levels to allow developers to choose the appropriate trade-off for their specific use cases.

In summary, the CAP theorem is a fundamental principle in distributed systems that highlights the trade-offs between Consistency, Availability, and Partition Tolerance. Different types of databases and distributed systems make different choices regarding these trade-offs, depending on their design goals and the requirements of the applications they serve. NoSQL databases often prioritize availability and partition tolerance while allowing for flexible consistency models.

# 2.3. Shading

Database sharding is a technique used to horizontally partition a database into multiple smaller databases called shards. Each shard contains a subset of the data, and collectively, the shards store the entire dataset. Sharding is primarily employed to improve scalability in large-scale distributed databases. By distributing data across multiple shards, the overall system can handle a higher volume of data and requests, leading to improved performance and throughput.

## 2.3.1. Techniques for sharding: range-based partitioning, hash-based partitioning, and consistent hashing:

There are several techniques for implementing sharding:

- Range-based partitioning: In this approach, data is divided based on a specified range of values. For example, a range-based partitioning scheme may divide data based on customer IDs, where each shard contains data for a specific range of customer IDs.

- Hash-based partitioning: This technique involves applying a hash function to a data attribute, such as a customer ID or a key, to determine which shard the data should be stored in. The hash function evenly distributes data across the available shards.

- Consistent hashing: Consistent hashing combines elements of both range-based and hash-based partitioning. It uses a hash function to assign data to shards but also maintains a consistent mapping between shards and data items, even when the number of shards changes. This helps minimize the need for data redistribution when adding or removing shards from the system.

## 2.3.2. Sharding strategies and considerations for distributing data across shards:

When distributing data across shards, there are several strategies and considerations to keep in mind:

- Key selection: Choosing an appropriate key for sharding is crucial. It should distribute the data evenly across shards to avoid hotspots where a single shard receives a disproportionately high load.

- Data distribution: The distribution of data across shards should be balanced to ensure that each shard has a similar amount of data. This helps maintain uniform performance across the system.

- Query routing: An effective strategy for routing queries to the appropriate shard is necessary. This can be done by including shard information in the query or using a centralized metadata service to map data items to their corresponding shards.

- Data integrity and consistency: Maintaining data integrity and consistency across shards can be challenging. Techniques such as distributed transactions, two-phase commit protocols, or eventual consistency models may be employed to address these challenges.

## 2.3.4. Benefits and challenges of sharding in distributed databases:

Sharding offers several benefits and also presents some challenges:

### - Benefits:

- Improved scalability: Sharding allows databases to handle large amounts of data and increased workload by distributing it across multiple shards.

- Enhanced performance: With data distributed across shards, read and write operations can be parallelized, leading to improved query response times and throughput.

- Fault isolation: Sharding enables isolating failures to specific shards, reducing the impact on the overall system.

### - Challenges:

- Data distribution complexity: Determining how to divide the data and distribute it across shards can be a complex task, especially as the system grows or changes.

- Data integrity and consistency: Maintaining consistency across shards and handling distributed transactions can be challenging.

- Shard management: Adding or removing shards from the system requires careful planning and coordination to ensure data redistribution and rebalancing.

## 2.3.5. Tools and technologies for implementing sharding in different databases:

There are various tools and technologies available for implementing sharding in different databases:

- MySQL Cluster: Provides built-in sharding capabilities through its NDB storage engine, allowing for horizontal scaling.

- Apache Cassandra: A distributed NoSQL database that natively supports sharding and provides tuneable consistency levels.

- MongoDB: Offers sharding as a built-in feature, allowing for horizontal scaling across multiple shards.

- Amazon Aurora: A cloud-based relational database service that supports automatic scaling and sharding for high-performance applications.

- Google Cloud Spanner: A globally distributed and strongly consistent database that automatically shards data to ensure scalability and high availability.

These are just a few examples, and there are many other databases and frameworks available that provide sharding capabilities based on specific requirements and use cases.

# 2.4. MongoDB

~~a. Overview of MongoDB as a document-based NoSQL database.~~

~~b. MongoDB data model: collections, documents, and fields.~~

~~c. CRUD operations in MongoDB: Insert, Update, Delete, and Query.~~

~~d. Indexing in MongoDB and its impact on query performance.~~

e. Application integration with MongoDB using PHP and Java.

f. Replication in MongoDB: replica sets, primary-secondary architecture, and data redundancy.

g. Sharding in MongoDB: concepts, sharding key selection, and data distribution.

h. Deployment options for MongoDB: standalone, replica sets, and sharded clusters.

i. Advanced features in MongoDB: aggregation framework, transactions, geospatial indexing, and full-text search.

Like any other database management language, MongoDB is based on a NoSQL database that is used for storing data in a key-value pair. Its working is based on the concept of document and collection. It is also an open-source, a document-oriented, cross-platform database system that is written using C++. In this chapter, you will learn more about MongoDB and its importance.

Mongo DB can be defined as a document-oriented database system that uses the concept of NoSQL. It also provides high availability, high performance, along with automatic scaling. This open-source product was developed by the company - 10gen in October 2007, and the company also maintains it. MongoDB exists under the General Public License (GPL) as a free database management tool as well as available under Commercial license as of the manufacturer. MongoDB was also intended to function with commodity servers. Companies of different sizes all over the world across all industries are using MongoDB as their database.

**Terminologies:**

Here are some key terminologies that you must know to get into the in-depth of MongoDB:

**Database**: In MongoDB, a database can be defined as a physical container for collections of data. Here, on the file system, every database has its collection of files residing. Usually, a MongoDB server contains numerous databases.

### Popular Organizations That Use MongoDB

* Adobe
* McAfee
* LinkedIn
* FourSquare
* MetLife
* eBay
* SAP

**Collections**: Collections can be defined as a cluster of MongoDB documents that exist within a single database. You can relate this to that of a table in a relational database management system. MongoDB collections do not implement the concept of schema. Documents that have collection usually contain different fields. Typically, all the documents residing within a collection are meant for a comparable or related purpose.

**Documents**: A document can be defined as a collection of key-value pairs that contain dynamic schema. Dynamic schema is something that documents of the equal collection do not require for having the same collection of fields or construction, and a common field is capable of holding various types of data.

**Fields**: Fields are the individual key-value pairs within a document. Each field has a unique key, which acts as an identifier for that particular piece of data within the document. Fields can hold a wide range of values, including scalar values (such as strings or numbers), arrays, or nested documents.

Here is a table showing the relation between the terminologies used in RDBMS and MongoDB:

|  |  |
| --- | --- |
| RDBMS | MongoDB |
| Database | Database |
| Table | Collection |
| Tuple or Row | Document |
| Column | Field |
| Table Join | Embedded Documents |
| Primary Key | Primary key / Default key |
| Mysqld / Oracle | mongod |

**Where Is MongoDB Used?**

Beginners need to know the purpose and requirement of why to use MongoDB or what is the need of it in contrast to SQL and other database systems. In simple words, it can be said that every modern-day application involves the concept of big data, analyzing different forms of data, fast features improvement in handling data, deployment flexibility, which old database systems are not competent enough to handle. Hence, MongoDB is the next choice.

**Why Use MongoDB?**

Some basic requirements are supported by this NoSQL database, which is lacking in other database systems. These collective reasons make MongoDB popular among other database systems:

* Document-Oriented data storage, i.e., data, is stored in a JSON style format, which increases the readability of data as well.
* Replication and high availability of data.
* MongoDB provides Auto-sharding.
* Ad hoc queries are supported by MongoDB, which helps in searching by range queries, field, or using regex terms.
* Indexing of values can be used to create and improve the overall search performance in MongoDB. MongoDB allows any field to be indexed within a document.
* MongoDB has a rich collection of queries.
* Updating of data can be done at a faster pace.
* It can be integrated with other popular programming languages also to handle structured as well as unstructured data within various types of applications.

**Advantages of Using MongoDB**

* It is easy to set up, i.e., install the MongoDB.
* Since MongoDB is a schema-less database, so there is no hassle of schema migration.
* Since it is a document-oriented language, document queries are used, which plays a vital role in supporting dynamic queries.
* Easily scalable.
* It is easy to have a performance tuning as compared to other relational databases.
* It helps in providing fast accessing of data because of its nature of implementing the internal memory to store the data.
* MongoDB is also used as a file system that can help in easy management of load balancing.
* MongoDB also supports the searching using the concept of regex (regular expression) as well as fields.
* Users can run MongoDB as a windows service also.
* It does not require any VM to run on different platforms.
* It also supports sharding of data.

## 2.4.1. CRUD operations in MongoDB: Insert, Update, Delete, and Query

MongoDB is a popular NoSQL database known for its flexibility, scalability, and ease of use. In this document, we will delve into the various MongoDB operations, including insert, update, delete, and query. Before that it is necessary to know how to create and drop databases and collections.

### Creating and Deleting Databases:

#### Use() method:

MongoDB use DATABASE\_NAME is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of use DATABASE statement is as follows −

use DATABASE\_NAME

Example

If you want to use a database with name **<mydb>**, then **use DATABASE** statement would be as follows −

>use mydb

switched to db mydb

To check your currently selected database, use the command **db**

>db

mydb

If you want to check your databases list, use the command **show dbs**.

>show dbs

local 0.78125GB

test 0.23012GB

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

>db.movie.insert({"name":"tutorials point"})

>show dbs

local 0.78125GB

mydb 0.23012GB

test 0.23012GB

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

#### The dropDatabase() Method

MongoDB **db.dropDatabase()** command is used to drop a existing database.

**Syntax**

Basic syntax of **dropDatabase()** command is as follows −

db.dropDatabase()

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

**Example**

First, check the list of available databases by using the command, **show dbs**.

>show dbs

local 0.78125GB

mydb 0.23012GB

test 0.23012GB

>

If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows −

>use mydb

switched to db mydb

>db.dropDatabase()

>{ "dropped" : "mydb", "ok" : 1 }

>

Now check list of databases.

>show dbs

local 0.78125GB

test 0.23012GB

>

#### The createCollection() Method

MongoDB **db.createCollection(name, options)** is used to create collection.

**Syntax**

Basic syntax of **createCollection()** command is as follows −

db.createCollection(name, options)

In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| Name | String | Name of the collection to be created |
| Options | Document | (Optional) Specify options about memory size and indexing |

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use −

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| capped | Boolean | (Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also. |
| autoIndexId | Boolean | (Optional) If true, automatically create index on \_id field.s Default value is false. |
| size | number | (Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also. |
| max | number | (Optional) Specifies the maximum number of documents allowed in the capped collection. |

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

**Examples**

Basic syntax of **createCollection()** method without options is as follows −

>use test

switched to db test

>db.createCollection("mycollection")

{ "ok" : 1 }

>

You can check the created collection by using the command **show collections**.

>show collections

mycollection

system.indexes

The following example shows the syntax of **createCollection()** method with few important options −

> db.createCollection("mycol", { capped : true, autoIndexID : true, size : 6142800, max : 10000 } ){

"ok" : 0,

"errmsg" : "BSON field 'create.autoIndexID' is an unknown field.",

"code" : 40415,

"codeName" : "Location40415"

}

>

In MongoDB, you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

>db.tutorialspoint.insert({"name" : "tutorialspoint"}),

WriteResult({ "nInserted" : 1 })

>show collections

mycol

mycollection

system.indexes

tutorialspoint

>

#### The drop() Method

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

**Syntax**

Basic syntax of **drop()** command is as follows −

db.COLLECTION\_NAME.drop()

**Example**

First, check the available collections into your database **mydb**.

>use mydb

switched to db mydb

>show collections

mycol

mycollection

system.indexes

tutorialspoint

>

Now drop the collection with the name **mycollection**.

>db.mycollection.drop()

true

>

Again check the list of collections into database.

>show collections

mycol

system.indexes

tutorialspoint

>

drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

#### Datatypes:

MongoDB supports many datatypes. Some of them are −

* **String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
* **Integer** − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
* **Boolean** − This type is used to store a boolean (true/ false) value.
* **Double** − This type is used to store floating point values.
* **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
* **Arrays** − This type is used to store arrays or list or multiple values into one key.
* **Timestamp** − ctimestamp. This can be handy for recording when a document has been modified or added.
* **Object** − This datatype is used for embedded documents.
* **Null** − This type is used to store a Null value.
* **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
* **Date**− This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
* **Object ID** − This datatype is used to store the document’s ID.
* **Binary data** − This datatype is used to store binary data.
* **Code** − This datatype is used to store JavaScript code into the document.
* **Regular expression** − This datatype is used to store regular expression.

### Inserting Document with insert() Method

To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()** method.

**Syntax**

The basic syntax of **insert()** command is as follows −

>db.COLLECTION\_NAME.insert(document)

**Example**

> db.users.insert({

... \_id : ObjectId("507f191e810c19729de860ea"),

... title: "MongoDB Overview",

... description: "MongoDB is no sql database",

... by: "tutorials point",

... url: "http://www.tutorialspoint.com",

... tags: ['mongodb', 'database', 'NoSQL'],

... likes: 100

... })

WriteResult({ "nInserted" : 1 })

>

Here **mycol** is our collection name, as created in the previous chapter. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert a document into it.

In the inserted document, if we don't specify the \_id parameter, then MongoDB assigns a unique ObjectId for this document.

\_id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows −

\_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer)

You can also pass an array of documents into the insert() method as shown below:.

> db.createCollection("post")

> db.post.insert([

{

title: "MongoDB Overview",

description: "MongoDB is no SQL database",

by: "tutorials point",

url: "http://www.tutorialspoint.com",

tags: ["mongodb", "database", "NoSQL"],

likes: 100

},

{

title: "NoSQL Database",

description: "NoSQL database doesn't have tables",

by: "tutorials point",

url: "http://www.tutorialspoint.com",

tags: ["mongodb", "database", "NoSQL"],

likes: 20,

comments: [

{

user:"user1",

message: "My first comment",

dateCreated: new Date(2013,11,10,2,35),

like: 0

}

]

}

])

BulkWriteResult({

"writeErrors" : [ ],

"writeConcernErrors" : [ ],

"nInserted" : 2,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

>

To insert the document you can use **db.post.save(document)** also. If you don't specify **\_id** in the document then **save()** method will work same as **insert()** method. If you specify \_id then it will replace whole data of document containing \_id as specified in save() method.

#### The insertOne() method

If you need to insert only one document into a collection you can use this method.

**Syntax**

The basic syntax of insert() command is as follows −

>db.COLLECTION\_NAME.insertOne(document)

**Example**

Following example creates a new collection named empDetails and inserts a document using the insertOne() method.

> db.createCollection("empDetails")

{ "ok" : 1 }

> db.empDetails.insertOne(

{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Date\_Of\_Birth: "1995-09-26",

e\_mail: "radhika\_sharma.123@gmail.com",

phone: "9848022338"

})

{

"acknowledged" : true,

"insertedId" : ObjectId("5dd62b4070fb13eec3963bea")

}

>

#### The insertMany() method

You can insert multiple documents using the insertMany() method. To this method you need to pass an array of documents.

**Example**

Following example inserts three different documents into the empDetails collection using the insertMany() method.

> db.empDetails.insertMany(

[

{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Date\_Of\_Birth: "1995-09-26",

e\_mail: "radhika\_sharma.123@gmail.com",

phone: "9000012345"

},

{

First\_Name: "Rachel",

Last\_Name: "Christopher",

Date\_Of\_Birth: "1990-02-16",

e\_mail: "Rachel\_Christopher.123@gmail.com",

phone: "9000054321"

},

{

First\_Name: "Fathima",

Last\_Name: "Sheik",

Date\_Of\_Birth: "1990-02-16",

e\_mail: "Fathima\_Sheik.123@gmail.com",

phone: "9000054321"

}

]

)

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("5dd631f270fb13eec3963bed"),

ObjectId("5dd631f270fb13eec3963bee"),

ObjectId("5dd631f270fb13eec3963bef")

]

}

>

MongoDB's **update()** and **save()** methods are used to update document into a collection. The update() method updates the values in the existing document while the save() method replaces the existing document with the document passed in save() method.

### Updating Documents with Update() method

The update() method updates the values in the existing document.

**Syntax**

The basic syntax of **update()** method is as follows −

>db.COLLECTION\_NAME.update(SELECTION\_CRITERIA, UPDATED\_DATA)

**Example**

Consider the mycol collection has the following data.

{ "\_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

>db.mycol.find()

{ "\_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}

{ "\_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

>

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

>db.mycol.update({'title':'MongoDB Overview'},

{$set:{'title':'New MongoDB Tutorial'}},{multi:true})

#### MongoDB Save() Method

The **save()** method replaces the existing document with the new document passed in the save() method.

**Syntax**

The basic syntax of MongoDB save() method is shown below −

>db.COLLECTION\_NAME.save({\_id:ObjectId(),NEW\_DATA})

**Example**

Following example will replace the document with the \_id '5983548781331adf45ec5'.

>db.mycol.save(

{

"\_id" : ObjectId("507f191e810c19729de860ea"),

"title":"Tutorials Point New Topic",

"by":"Tutorials Point"

}

)

WriteResult({

"nMatched" : 0,

"nUpserted" : 1,

"nModified" : 0,

"\_id" : ObjectId("507f191e810c19729de860ea")

})

>db.mycol.find()

{ "\_id" : ObjectId("507f191e810c19729de860e6"), "title":"Tutorials Point New Topic",

"by":"Tutorials Point"}

{ "\_id" : ObjectId("507f191e810c19729de860e6"), "title":"NoSQL Overview"}

{ "\_id" : ObjectId("507f191e810c19729de860e6"), "title":"Tutorials Point Overview"}

>

#### MongoDB findOneAndUpdate() method

The **findOneAndUpdate()** method updates the values in the existing document.

**Syntax**

The basic syntax of **findOneAndUpdate()** method is as follows −

>db.COLLECTION\_NAME.findOneAndUpdate(SELECTIOIN\_CRITERIA, UPDATED\_DATA)

Assume we have created a collection named empDetails and inserted three documents in it as shown below −

> db.empDetails.insertMany(

[

{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Age: "26",

e\_mail: "radhika\_sharma.123@gmail.com",

phone: "9000012345"

},

{

First\_Name: "Rachel",

Last\_Name: "Christopher",

Age: "27",

e\_mail: "Rachel\_Christopher.123@gmail.com",

phone: "9000054321"

},

{

First\_Name: "Fathima",

Last\_Name: "Sheik",

Age: "24",

e\_mail: "Fathima\_Sheik.123@gmail.com",

phone: "9000054321"

}

]

)

Following example updates the age and email values of the document with name 'Radhika'.

> db.empDetails.findOneAndUpdate(

{First\_Name: 'Radhika'},

{ $set: { Age: '30',e\_mail: 'radhika\_newemail@gmail.com'}}

)

{

"\_id" : ObjectId("5dd6636870fb13eec3963bf5"),

"First\_Name" : "Radhika",

"Last\_Name" : "Sharma",

"Age" : "30",

"e\_mail" : "radhika\_newemail@gmail.com",

"phone" : "9000012345"

}

#### MongoDB updateOne() method

This methods updates a single document which matches the given filter.

**Syntax**

The basic syntax of updateOne() method is as follows −

>db.COLLECTION\_NAME.updateOne(<filter>, <update>)

> db.empDetails.updateOne(

{First\_Name: 'Radhika'},

{ $set: { Age: '30',e\_mail: 'radhika\_newemail@gmail.com'}}

)

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }

>

#### MongoDB updateMany() method

The updateMany() method updates all the documents that matches the given filter.

**Syntax**

The basic syntax of updateMany() method is as follows −

>db.COLLECTION\_NAME.update(<filter>, <update>)

**Example**

> db.empDetails.updateMany(

{Age:{ $gt: "25" }},

{ $set: { Age: '00'}}

)

{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }

You can see **the** updated values if you retrieve the contents of the document using the find method as shown below −

> db.empDetails.find()

{ "\_id" : ObjectId("5dd6636870fb13eec3963bf5"), "First\_Name" : "Radhika", "Last\_Name" : "Sharma", "Age" : "00", "e\_mail" : "radhika\_newemail@gmail.com", "phone" : "9000012345" }

{ "\_id" : ObjectId("5dd6636870fb13eec3963bf6"), "First\_Name" : "Rachel", "Last\_Name" : "Christopher", "Age" : "00", "e\_mail" : "Rachel\_Christopher.123@gmail.com", "phone" : "9000054321" }

{ "\_id" : ObjectId("5dd6636870fb13eec3963bf7"), "First\_Name" : "Fathima", "Last\_Name" : "Sheik", "Age" : "24", "e\_mail" : "Fathima\_Sheik.123@gmail.com", "phone" : "9000054321" }

>

### Deleting Documents with The remove() Method

MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

* **deletion criteria** − (Optional) deletion criteria according to documents will be removed.
* **justOne** − (Optional) if set to true or 1, then remove only one document.

**Syntax**

Basic syntax of **remove()** method is as follows −

>db.COLLECTION\_NAME.remove(DELLETION\_CRITTERIA)

**Example**

Consider the mycol collection has the following data.

{\_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},

{\_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},

{\_id : ObjectId("507f191e810c19729de860e3"), title: "Tutorials Point Overview"}

Following example will remove all the documents whose title is 'MongoDB Overview'.

>db.mycol.remove({'title':'MongoDB Overview'})

WriteResult({"nRemoved" : 1})

> db.mycol.find()

{"\_id" : ObjectId("507f191e810c19729de860e2"), "title" : "NoSQL Overview" }

{"\_id" : ObjectId("507f191e810c19729de860e3"), "title" : "Tutorials Point Overview" }

#### Remove Only One

If there are multiple records and you want to delete only the first record, then set **justOne** parameter in **remove()** method.

>db.COLLECTION\_NAME.remove(DELETION\_CRITERIA,1)

#### Remove All Documents

If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. **This is equivalent of SQL's truncate command.**

> db.mycol.remove({})

WriteResult({ "nRemoved" : 2 })

> db.mycol.find()

>

we will learn how to query document from MongoDB collection.

### Querying Document

#### The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.

**Syntax**

The basic syntax of **find()** method is as follows −

>db.COLLECTION\_NAME.find()

**find()** method will display all the documents in a non-structured way.

**Example**

Assume we have created a collection named mycol as −

> use sampleDB

switched to db sampleDB

> db.createCollection("mycol")

{ "ok" : 1 }

>

And inserted 3 documents in it using the insert() method as shown below −

> db.mycol.insert([

{

title: "MongoDB Overview",

description: "MongoDB is no SQL database",

by: "tutorials point",

url: "http://www.tutorialspoint.com",

tags: ["mongodb", "database", "NoSQL"],

likes: 100

},

{

title: "NoSQL Database",

description: "NoSQL database doesn't have tables",

by: "tutorials point",

url: "http://www.tutorialspoint.com",

tags: ["mongodb", "database", "NoSQL"],

likes: 20,

comments: [

{

user:"user1",

message: "My first comment",

dateCreated: new Date(2013,11,10,2,35),

like: 0

}

]

}

])

Following method retrieves all the documents in the collection −

> db.mycol.find()

{ "\_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overview", "description" : "MongoDB is no SQL database", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 100 }

{ "\_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description" : "NoSQL database doesn't have tables", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 20, "comments" : [ { "user" : "user1", "message" : "My first comment", "dateCreated" : ISODate("2013-12-09T21:05:00Z"), "like" : 0 } ] }

>

#### The pretty() Method

To display the results in a formatted way, you can use pretty() method.

**Syntax**

>db.COLLECTION\_NAME.find().pretty()

**Example**

Following example retrieves all the documents from the collection named mycol and arranges them in an easy-to-read format.

> db.mycol.find().pretty()

{

"\_id" : ObjectId("5dd4e2cc0821d3b44607534c"),

"title" : "MongoDB Overview",

"description" : "MongoDB is no SQL database",

"by" : "tutorials point",

"url" : "http://www.tutorialspoint.com",

"tags" : [

"mongodb",

"database",

"NoSQL"

],

"likes" : 100

}

{

"\_id" : ObjectId("5dd4e2cc0821d3b44607534d"),

"title" : "NoSQL Database",

"description" : "NoSQL database doesn't have tables",

"by" : "tutorials point",

"url" : "http://www.tutorialspoint.com",

"tags" : [

"mongodb",

"database",

"NoSQL"

],

"likes" : 20,

"comments" : [

{

"user" : "user1",

"message" : "My first comment",

"dateCreated" : ISODate("2013-12-09T21:05:00Z"),

"like" : 0

}

]

}

#### The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.

**Syntax**

>db.COLLECTIONNAME.findOne()

**Example**

Following example retrieves the document with title MongoDB Overview.

> db.mycol.findOne({title: "MongoDB Overview"})

{

"\_id" : ObjectId("5dd6542170fb13eec3963bf0"),

"title" : "MongoDB Overview",

"description" : "MongoDB is no SQL database",

"by" : "tutorials point",

"url" : "http://www.tutorialspoint.com",

"tags" : [

"mongodb",

"database",

"NoSQL"

],

"likes" : 100

}

#### RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Syntax** | **Example** | **RDBMS Equivalent** |
| Equality | {<key>:{$eg;<value>}} | db.mycol.find({"by":"tutorials point"}).pretty() | where by = 'tutorials point' |
| Less Than | {<key>:{$lt:<value>}} | db.mycol.find({"likes":{$lt:50}}).pretty() | where likes < 50 |
| Less Than Equals | {<key>:{$lte:<value>}} | db.mycol.find({"likes":{$lte:50}}).pretty() | where likes <= 50 |
| Greater Than | {<key>:{$gt:<value>}} | db.mycol.find({"likes":{$gt:50}}).pretty() | where likes > 50 |
| Greater Than Equals | {<key>:{$gte:<value>}} | db.mycol.find({"likes":{$gte:50}}).pretty() | where likes >= 50 |
| Not Equals | {<key>:{$ne:<value>}} | db.mycol.find({"likes":{$ne:50}}).pretty() | where likes != 50 |
| Values in an array | {<key>:{$in:[<value1>, <value2>,……<valueN>]}} | db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty() | Where name matches any of the value in :["Raj", "Ram", "Raghu"] |
| Values not in an array | {<key>:{$nin:<value>}} | db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty() | Where name values is not in the array :["Ramu", "Raghav"] or, doesn’t exist at all |

##### AND in MongoDB

**Syntax**

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND −

>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })

**Example**

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

> db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()

{

"\_id" : ObjectId("5dd4e2cc0821d3b44607534c"),

"title" : "MongoDB Overview",

"description" : "MongoDB is no SQL database",

"by" : "tutorials point",

"url" : "http://www.tutorialspoint.com",

"tags" : [

"mongodb",

"database",

"NoSQL"

],

"likes" : 100

}

>

For the above given example, equivalent where clause will be ' where by = 'tutorials point' AND title = 'MongoDB Overview' '. You can pass any number of key, value pairs in find clause.

##### OR in MongoDB

**Syntax**

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** −

>db.mycol.find(

{

$or: [

{key1: value1}, {key2:value2}

]

}

).pretty()

**Example**

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()

{

"\_id": ObjectId(7df78ad8902c),

"title": "MongoDB Overview",

"description": "MongoDB is no sql database",

"by": "tutorials point",

"url": "http://www.tutorialspoint.com",

"tags": ["mongodb", "database", "NoSQL"],

"likes": "100"

}

>

##### Using AND and OR Together

**Example**

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is 'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'

>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},

{"title": "MongoDB Overview"}]}).pretty()

{

"\_id": ObjectId(7df78ad8902c),

"title": "MongoDB Overview",

"description": "MongoDB is no sql database",

"by": "tutorials point",

"url": "http://www.tutorialspoint.com",

"tags": ["mongodb", "database", "NoSQL"],

"likes": "100"

}

>

##### NOR in MongoDB

**Syntax**

To query documents based on the NOT condition, you need to use $not keyword. Following is the basic syntax of NOT −

>db.COLLECTION\_NAME.find(

{

$not: [

{key1: value1}, {key2:value2}

]

}

)

**Example**

Assume we have inserted 3 documents in the collection empDetails as shown below −

db.empDetails.insertMany(

[

{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Age: "26",

e\_mail: "radhika\_sharma.123@gmail.com",

phone: "9000012345"

},

{

First\_Name: "Rachel",

Last\_Name: "Christopher",

Age: "27",

e\_mail: "Rachel\_Christopher.123@gmail.com",

phone: "9000054321"

},

{

First\_Name: "Fathima",

Last\_Name: "Sheik",

Age: "24",

e\_mail: "Fathima\_Sheik.123@gmail.com",

phone: "9000054321"

}

]

)

Following example will retrieve the document(s) whose first name is not "Radhika" and last name is not "Christopher"

> db.empDetails.find(

{

$nor:[

40

{"First\_Name": "Radhika"},

{"Last\_Name": "Christopher"}

]

}

).pretty()

{

"\_id" : ObjectId("5dd631f270fb13eec3963bef"),

"First\_Name" : "Fathima",

"Last\_Name" : "Sheik",

"Age" : "24",

"e\_mail" : "Fathima\_Sheik.123@gmail.com",

"phone" : "9000054321"

}

##### NOT in MongoDB

**Syntax**

To query documents based on the NOT condition, you need to use $not keyword following is the basic syntax of NOT −

>db.COLLECTION\_NAME.find(

{

$NOT: [

{key1: value1}, {key2:value2}

]

}

).pretty()

**Example**

Following example will retrieve the document(s) whose age is not greater than 25

> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )

{

"\_id" : ObjectId("5dd6636870fb13eec3963bf7"),

"First\_Name" : "Fathima",

"Last\_Name" : "Sheik",

"Age" : "24",

"e\_mail" : "Fathima\_Sheik.123@gmail.com",

"phone" : "9000054321"

}

### Indexing Document

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

#### The createIndex() Method

To create an index, you need to use createIndex() method of MongoDB.

**Syntax**

The basic syntax of **createIndex()** method is as follows().

>db.COLLECTION\_NAME.createIndex({KEY:1})

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

**Example**

>db.mycol.createIndex({"title":1})

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"ok" : 1

}

>

In **createIndex()** method you can pass multiple fields, to create index on multiple fields.

>db.mycol.createIndex({"title":1,"description":-1})

>

This method also accepts list of options (which are optional). Following is the list –

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| background | Boolean | Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is false. |
| unique | Boolean | Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index. The default value is false. |
| name | string | The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order. |
| sparse | Boolean | If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is false. |
| expireAfterSeconds | integer | Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection. |
| weights | document | The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score. |
| default\_language | string | For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is English. |
| language\_override | string | For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language. |

#### The dropIndex() method

You can drop a particular index using the dropIndex() method of MongoDB.

**Syntax**

The basic syntax of DropIndex() method is as follows().

>db.COLLECTION\_NAME.dropIndex({KEY:1})

Here, "key" is the name of the file on which you want to remove an existing index. Instead of the index specification document (above syntax), you can also specify the name of the index directly as:

dropIndex("name\_of\_the\_index")

**Example**

> db.mycol.dropIndex({"title":1})

{

"ok" : 0,

"errmsg" : "can't find index with key: { title: 1.0 }",

"code" : 27,

"codeName" : "IndexNotFound"

}

#### The dropIndexes() method

This method deletes multiple (specified) indexes on a collection.

**Syntax**

The basic syntax of DropIndexes() method is as follows() −

>db.COLLECTION\_NAME.dropIndexes()

**Example**

Assume we have created 2 indexes in the named mycol collection as shown below −

> db.mycol.createIndex({"title":1,"description":-1})

Following example removes the above created indexes of mycol −

>db.mycol.dropIndexes({"title":1,"description":-1})

{ "nIndexesWas" : 2, "ok" : 1 }

>

#### The getIndexes() method

This method returns the description of all the indexes int the collection.

**Syntax**

Following is the basic syntax od the getIndexes() method −

db.COLLECTION\_NAME.getIndexes()

**Example**

Assume we have created 2 indexes in the named mycol collection as shown below −

> db.mycol.createIndex({"title":1,"description":-1})

Following example retrieves all the indexes in the collection mycol −

> db.mycol.getIndexes()

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_",

"ns" : "test.mycol"

},

{

"v" : 2,

"key" : {

"title" : 1,

"description" : -1

},

"name" : "title\_1\_description\_-1",

"ns" : "test.mycol"

}

]

>

# Exercise:

1. Explain the concept of NoSQL databases and their advantages over traditional relational databases.
2. Compare and contrast key-value, document-based, columnar, and graph databases.
3. Discuss the use cases where NoSQL databases are more suitable than relational databases.
4. Explain the CAP theorem and its implications for distributed systems.
5. Discuss the trade-offs between consistency, availability, and partition tolerance.
6. Provide examples of real-world scenarios where different components of the CAP theorem are prioritized.
7. Define sharding and explain its role in scaling databases.
8. Discuss the benefits and challenges of sharding in distributed databases.
9. Describe the different sharding techniques and their implementation strategies.
10. Explain the concept of document-based databases and how MongoDB fits into this category.
11. Discuss the CRUD operations (insert, update, delete, query) in MongoDB.
12. Explain the concept of indexing in MongoDB and its impact on query performance.
13. Discuss the concepts of replication and sharding in MongoDB and how they contribute to scalability and fault tolerance.
14. Describe the deployment options for MongoDB and the considerations involved in choosing the appropriate deployment strategy.
15. Provide examples of using MongoDB with PHP/Java and discuss the advantages of these programming languages with MongoDB.
16. Discuss advanced MongoDB features such as aggregation framework, transactions, and geospatial indexing.
17. Describe the data model in Cassandra and how it differs from traditional relational databases.
18. Explain the concepts of keyspaces and tables in Cassandra and their role in data organization.
19. Discuss the CRUD operations (create, read, update, delete) in Cassandra.
20. Explain the CQL (Cassandra Query Language) types and their usage in Cassandra.
21. Discuss the data types available in Hive and their mapping to the corresponding types in relational databases.
22. Explain the database operations in Hive, such as creating tables, altering tables, and dropping tables.
23. Describe the concept of partitioning in Hive and its benefits in query performance.
24. Provide examples of using HiveQL to query data in Hive.
25. Explain the concept of a graph database and how OrientDB fits into this category.
26. Discuss the features of OrientDB, such as schema-less design, ACID transactions, and indexing.
27. Explain the concept of graph traversal and provide examples of using OrientDB's graph traversal capabilities.

# Bibliography:

Textbooks:

1. "MongoDB: The Definitive Guide" by Kristina Chodorow and Michael Dirolf.
2. "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence" by Martin Fowler and Pramod Sadalage.
3. "Cassandra: The Definitive Guide" by Eben Hewitt.
4. "Hadoop: The Definitive Guide" by Tom White.
5. "Learning SQL" by Alan Beaulieu.
6. "Database Systems: The Complete Book" by Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom.
7. "Data Modeling: A Beginner's Guide" by Andy Oppel.

Online Resources:

1. MongoDB official documentation: <https://docs.mongodb.com/>
2. PHP MongoDB Library: <https://docs.mongodb.com/drivers/php/>
3. MongoDB Java Driver documentation: <https://mongodb.github.io/mongo-java-driver/>
4. Apache Cassandra Documentation: <https://cassandra.apache.org/doc/>
5. Hive official documentation: <https://cwiki.apache.org/confluence/display/Hive/Home>
6. OrientDB documentation: <https://orientdb.com/docs/>

# --DNU--

## 1. NoSQL:

a. Introduction to NoSQL databases and their characteristics.

In recent years, the demand for managing and processing massive amounts of data has grown exponentially. Traditional relational databases, while effective for structured data, often struggle to handle the scale and variety of data generated in modern applications. As a result, NoSQL (Not Only SQL) databases have emerged as a popular alternative. NoSQL databases provide flexible, scalable, and high-performance solutions for storing and retrieving data. This article provides an in-depth introduction to NoSQL databases, exploring their characteristics, data models, and key features.

Section 1: Understanding NoSQL Databases (300 words)

1.1 Definition and Origins:

NoSQL databases refer to a diverse set of database technologies designed to overcome limitations of traditional relational databases. They emerged in the late 2000s, driven by the need to handle massive data volumes generated by web-scale applications such as social media, e-commerce, and IoT.

NoSQL is a type of database management system (DBMS) that is designed to handle and store large volumes of unstructured and semi-structured data. Unlike traditional relational databases that use tables with pre-defined schemas to store data, NoSQL databases use flexible data models that can adapt to changes in data structures and are capable of scaling horizontally to handle growing amounts of data.

The term NoSQL originally referred to “non-SQL” or “non-relational” databases, but the term has since evolved to mean “not only SQL,” as NoSQL databases have expanded to include a wide range of different database architectures and data models.

b. Comparison between NoSQL and relational databases.

c. Different types of NoSQL databases: key-value stores, document-based databases, columnar databases, and graph databases.

d. Use cases and advantages of NoSQL databases.

e. CAP theorem and its relevance to NoSQL databases.

f. ACID vs. BASE consistency models in NoSQL databases.

2. CAP Theorem:

a. Explanation of the CAP theorem and its three components: Consistency, Availability, and Partition Tolerance.

b. Detailed discussion of the trade-offs involved in achieving CAP properties.

c. Examples of popular databases and their approach to CAP trade-offs.

d. CAP theorem in the context of NoSQL databases.

3. Sharding:

a. Introduction to database sharding and its role in scalability.

b. Techniques for sharding: range-based partitioning, hash-based partitioning, and consistent hashing.

c. Sharding strategies and considerations for distributing data across shards.

d. Benefits and challenges of sharding in distributed databases.

e. Tools and technologies for implementing sharding in different databases.

4. MongoDB:

a. Overview of MongoDB as a document-based NoSQL database.

b. MongoDB data model: collections, documents, and fields.

c. CRUD operations in MongoDB: Insert, Update, Delete, and Query.

d. Indexing in MongoDB and its impact on query performance.

e. Application integration with MongoDB using PHP and Java.

f. Replication in MongoDB: replica sets, primary-secondary architecture, and data redundancy.

g. Sharding in MongoDB: concepts, sharding key selection, and data distribution.

h. Deployment options for MongoDB: standalone, replica sets, and sharded clusters.

i. Advanced features in MongoDB: aggregation framework, transactions, geospatial indexing, and full-text search.

5. Cassandra:

a. Introduction to Cassandra as a distributed NoSQL database.

b. Cassandra data model: keyspaces, tables, rows, and columns.

c. CRUD operations in Cassandra: Create, Read, Update, and Delete.

d. Data modelling in Cassandra: denormalization, partition keys, clustering columns, and secondary indexes.

e. CQL (Cassandra Query Language) types: basic types, collections, and user-defined types.

f. Replication in Cassandra: replication factor, consistency levels, and fault tolerance.

g. Scaling and sharding in Cassandra: adding nodes, virtual nodes, and data distribution.

h. Advanced features of Cassandra: lightweight transactions, materialized views, and compaction strategies.

6. Hive:

a. Introduction to Hive as a data warehousing and analytics platform built on top of Hadoop.

b. Hive data types: primitive types, complex types, and type conversion.

c. Database operations in Hive: creating databases, tables, and partitions.

d. Data loading techniques in Hive: INSERT, LOAD DATA, and external tables.

e. HiveQL: syntax, data manipulation statements, and data query operations.

f. Partitioning in Hive: benefits, partition types, and partition pruning.

g. Performance optimization techniques in Hive: indexing, query optimization, and statistics.

7. OrientDB:

a. Introduction to OrientDB as a graph database management system.

b. Features of OrientDB: ACID transactions, schema-less design, and hybrid document-graph model.

c. Graph database concepts: vertices, edges, properties, and graph traversal.

d. OrientDB graph operations: creating vertices and edges, updating properties, and traversing the graph.

e. Indexing in OrientDB: automatic indexing, manual indexing, and full-text search.

f. OrientDB as a multi-model database: support for SQL, graph, and document querying.