[Year]

NoSQL Databases: A Comprehensive GUIDE: A Comprehensive Guide

A REFERENCE BOOK FOR ANNA UNIVERSITY-CDE-MCA-DMC6103-UNIT 2

HARYISH ELANGUMARAN

CENTER OF DISTANCE EDUCATION, ANNA UNIVERSITY | Guindy, Chennai

PREFACE

Welcome to the world of NoSQL databases! In this textbook, we delve into the fundamental concepts and advanced features of NoSQL databases, exploring their capabilities and practical applications. From understanding the principles behind NoSQL to hands-on experience with popular NoSQL databases like MongoDB, Cassandra, and OrientDB, this book aims to equip you with the knowledge and skills to harness the power of these flexible and scalable data storage solutions.

Chapter 1 introduces the foundational concept of NoSQL databases. We explore the motivations behind the emergence of NoSQL, the fundamental principles that differentiate it from traditional relational databases, and the implications of the CAP theorem. By understanding the trade-offs between consistency, availability, and partition tolerance, you'll be able to make informed decisions when choosing a NoSQL database for your specific use case.

Chapter 2 delves into sharding, a key technique used in NoSQL databases for horizontal scalability. You'll learn how data is distributed across multiple nodes to handle massive amounts of data and high traffic loads. We'll discuss various sharding strategies and best practices to ensure efficient and balanced data distribution.

Chapter 3 focuses on document-based databases, with a deep dive into MongoDB. You'll learn how MongoDB stores data in flexible, schema-less documents and how to perform essential operations like inserting, updating, deleting, and querying documents. We'll also cover indexing techniques to optimize query performance and explore how to integrate MongoDB with popular programming languages like PHP and Java.

Chapter 4 delves into advanced features of MongoDB, including replication and sharding. You'll understand the importance of data redundancy and high availability through replication, and learn how to configure replica sets for fault tolerance. We'll also explore sharding in MongoDB, enabling horizontal scaling to handle large datasets and high throughput.

Chapter 5 introduces Cassandra, a highly scalable and distributed NoSQL database. We'll cover the data model and key space concepts, and explore various table operations for creating, updating, deleting, and querying data. We'll also delve into CQL types and their usage in Cassandra.

Chapter 6 focuses on Hive, a data warehouse infrastructure built on top of Hadoop. You'll learn about Hive's data types, database operations, and partitioning techniques for improved query performance. We'll dive into HiveQL, the query language used in Hive, and explore its functionalities.

Chapter 7 presents OrientDB, a graph database that leverages the power of graph structures for storing and querying interconnected data. We'll explore OrientDB's features, data model, and the benefits of using a graph database for complex relationships.

Throughout this textbook, we provide comprehensive explanations, practical examples, and code snippets to ensure a thorough understanding of NoSQL databases and their features.

Additionally, we have included a section of reference links and a bibliography for further exploration.

To solidify your understanding, each chapter concludes with exercise questions that test your knowledge and provide opportunities for hands-on practice. By working through these exercises, you'll gain valuable experience in applying the concepts and techniques covered in each chapter.

We hope that this textbook serves as a valuable resource on your journey to mastering NoSQL databases. Whether you are a student, a professional, or an enthusiast, we believe that the knowledge gained from this book will empower you to leverage the capabilities of NoSQL databases and make informed decisions in your data storage and analysis endeavors.

Happy exploring!

Haryish Elangumaran MCA'

Anna University, CDE

NOSQL DATABASES: A COMPREHENSIVE GUIDE

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

SNAPSHOT:

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.

CONTENTS

1. NOSQL-OVERVIEW:

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.

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.

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.

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	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

	(CQL), or graph traversal languages like Gremlin.	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.

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.

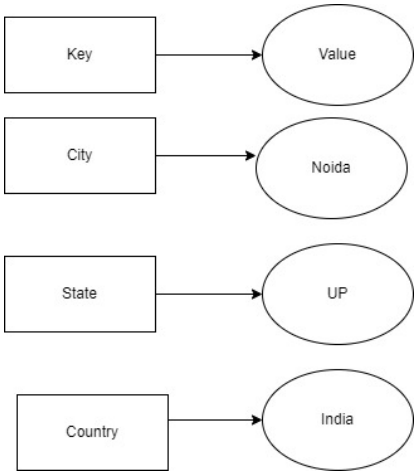
- Key-Value Stores: Key-value stores store data as a collection of key-value pairs, where each key is unique and associated with a value. They offer high-speed read and write operations, making them suitable for caching, session management, and simple data retrieval.
- Document-Based Databases: Document-based 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.

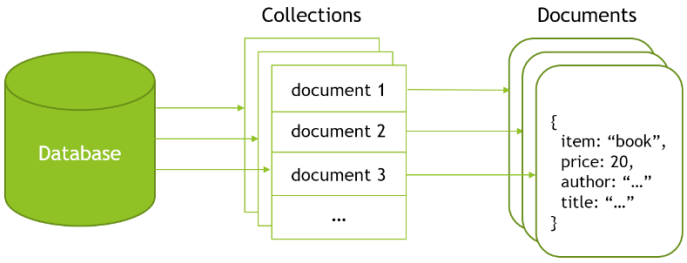
Key-value stores offer fast and efficient data retrieval by directly accessing values using their unique keys. They are highly scalable and can handle large volumes of data with ease. Key-

value stores are commonly used for caching, session management, and storing user preferences.



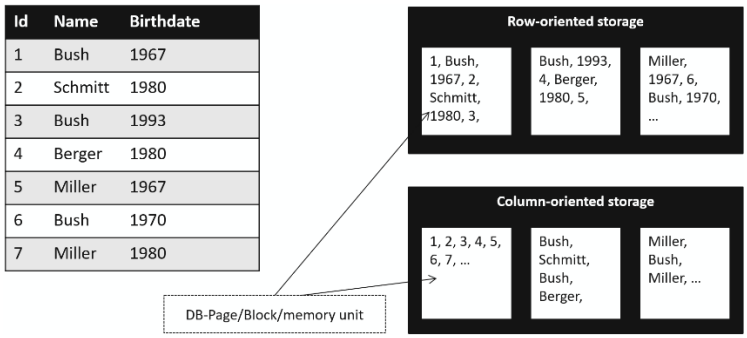
DOCUMENT-BASED DATABASES

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.



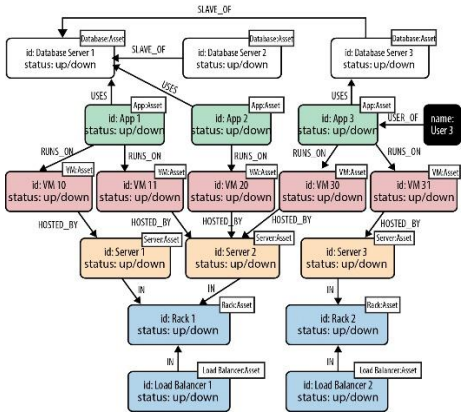
COLUMNAR DATABASES

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.



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.

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.

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.

1.5: CAP THEOREM AND ITS RELEVANCE TO NOSQL DATABASES

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.

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.

1.6: ACID VS. BASE CONSISTENCY MODELS IN NOSQL DATABASES

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.

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. CAP THEOREM:

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. 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.

2.3. 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.

2.4. 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.

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.

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.

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.

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.

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.

4. MONGODB

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.

OVERVIEW:

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.

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.

Popular Organizations That Use MongoDB

- Adobe
- McAfee
- LinkedIn
- FourSquare
- MetLife
- eBay
- SAP

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 can hold 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 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	mongodb

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.

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 based on 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"
```

```
        }
]
>
```

4.2. REPLICATION ON MONGODB

Replication is the process of synchronizing data across multiple servers. Replication provides redundancy and increases data availability with multiple copies of data on different database servers. Replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

WHY REPLICATION?

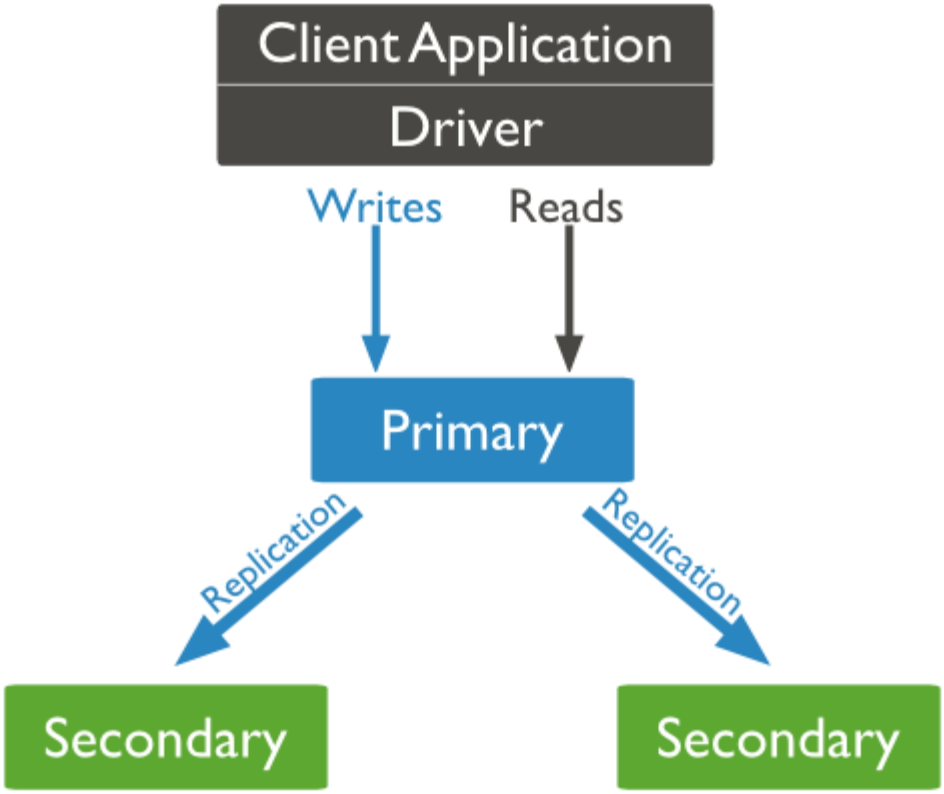
- To keep your data safe
- High (24*7) availability of data
- Disaster recovery
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application

HOW REPLICATION WORKS IN MONGODB

MongoDB achieves replication by the use of replica set. A replica set is a group of mongod instances that host the same data set. In a replica, one node is primary node that receives all write operations. All other instances, such as secondaries, apply operations from the primary so that they have the same data set. Replica set can have only one primary node.

- Replica set is a group of two or more nodes (generally minimum 3 nodes are required).
- In a replica set, one node is primary node and remaining nodes are secondary.
- All data replicates from primary to secondary node.
- At the time of automatic failover or maintenance, election establishes for primary and a new primary node is elected.
- After the recovery of failed node, it again join the replica set and works as a secondary node.

A typical diagram of MongoDB replication is shown in which client application always interact with the primary node and the primary node then replicates the data to the secondary nodes.



MONGODB REPLICATION

- Replica Set Features
- A cluster of N nodes
- Any one node can be primary
- All write operations go to primary
- Automatic failover
- Automatic recovery
- Consensus election of primary

SET UP A REPLICA SET

In this tutorial, we will convert standalone MongoDB instance to a replica set. To convert to replica set, following are the steps −

- Shutdown already running MongoDB server.
- Start the MongoDB server by specifying -- replSet option.
- Following is the basic syntax of --replSet –

```
mongod --port "PORT" --dbpath "YOUR_DB_DATA_PATH" --replSet
"REPLICA_SET_INSTANCE_NAME"

Example

mongod --port 27017 --dbpath "D:\set up\mongodb\data" --replSet rs0
```

- It will start a mongod instance with the name rs0, on port 27017.
- Now start the command prompt and connect to this mongod instance.

- In Mongo client, issue the command rs.initiate() to initiate a new replica set.
- To check the replica set configuration, issue the command rs.conf(). To check the status of replica set issue the command rs.status().

ADD MEMBERS TO REPLICA SET

To add members to replica set, start mongod instances on multiple machines. Now start a mongo client and issue a command rs.add().

Syntax

The basic syntax of rs.add() command is as follows −

```
>rs.add(HOST_NAME:PORT)
```

Example

Suppose your mongod instance name is mongod1.net and it is running on port 27017. To add this instance to replica set, issue the command rs.add() in Mongo client.

```
>rs.add("mongod1.net:27017")
```

You can add mongod instance to replica set only when you are connected to primary node. To check whether you are connected to primary or not, issue the command db.isMaster() in mongo client.

4.3. MONGODB SHARDING

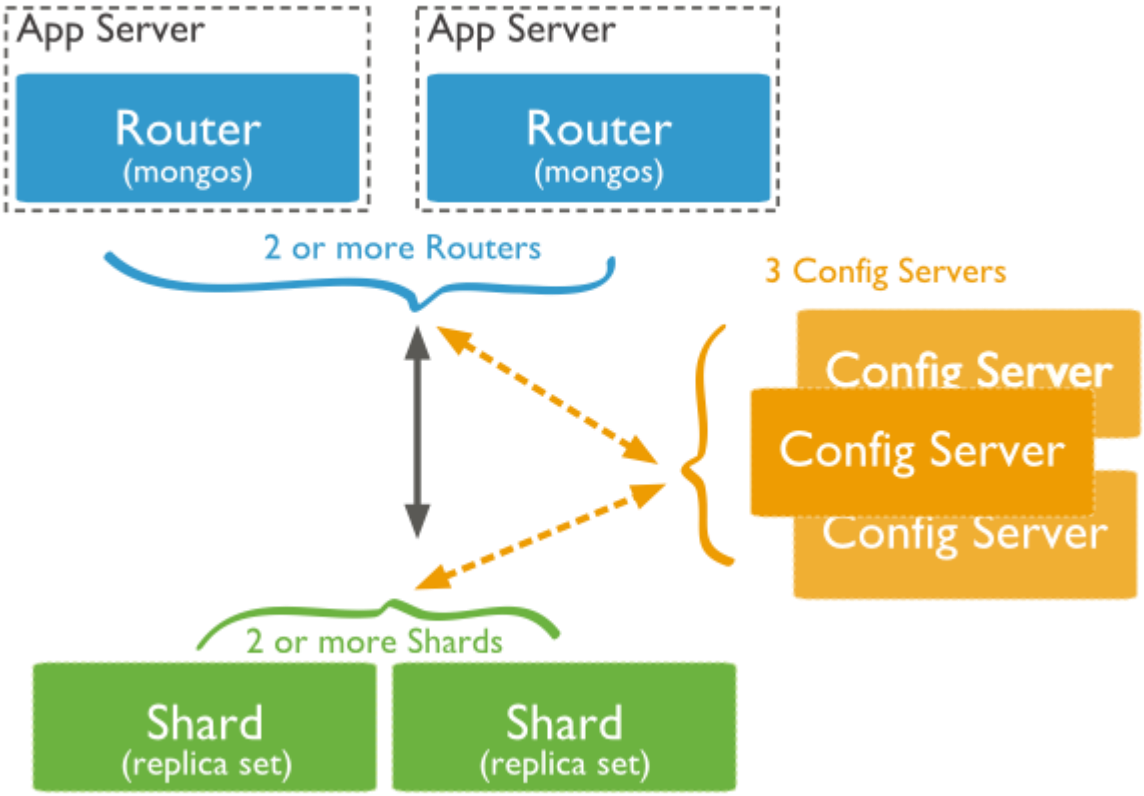
Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

WHY SHARDING?

- In replication, all writes go to master node
- Latency sensitive queries still go to master
- Single replica set has limitation of 12 nodes
- Memory can't be large enough when active dataset is big
- Local disk is not big enough
- Vertical scaling is too expensive

SHARDING IN MONGODB

The following diagram shows the Sharding in MongoDB using sharded cluster.



In the following diagram, there are three main components −

- **Shards** − Shards are used to store data. They provide high availability and data consistency. In production environment, each shard is a separate replica set.
- **Config Servers** − Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. In production environment, sharded clusters have exactly 3 config servers.
- **Query Routers** − Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard. The query router processes and targets the operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Generally, a sharded cluster have many query routers.

4.4. APPLICATION – INTEGRATION OF MONGODB WITH JAVA

Sure! I can explain how to integrate an application with MongoDB using PHP and Java separately, and also provide examples for both scenarios. Let's start with PHP first.

APPLICATION INTEGRATION WITH MONGODB USING PHP

To integrate a PHP application with MongoDB, you need to follow these steps:

1. Install MongoDB PHP driver:

- You can install the MongoDB PHP driver using PECL or Composer. Here, I'll show you the Composer method. Run the following command in your terminal:

```
composer require mongodb/mongodb
```

2. Establish a connection to MongoDB:

- Use the following code snippet to establish a connection to MongoDB:

```
<?php
require 'vendor/autoload.php'; // Include the Composer autoloader

$client = new MongoDB\Client("mongodb://localhost:27017"); // Connect to MongoDB

$database = $client->selectDatabase('mydb'); // Select a database
$collection = $database->selectCollection('mycollection'); // Select a collection
?>
```

3. Perform CRUD operations:

- Once connected, you can perform CRUD (Create, Read, Update, Delete) operations on the MongoDB collections. Here's an example snippet for inserting a document:

```
<?php
$document = [
    'name' => 'John Doe',
    'age' => 30,
    'email' => 'johndoe@example.com'
];

$collection->insertOne($document);
?>
```

That is how you can integrate a PHP application with MongoDB. Now let's move on to Java integration.

APPLICATION INTEGRATION WITH MONGODB USING JAVA (WITH MAVEN):

To integrate a Java application with MongoDB using Maven, you need to follow these steps:

1. Set up Maven:

- Make sure you have Maven installed on your system. You can download it from the official Maven website and follow the installation instructions.

2. Create a Maven project:

- Create a new Maven project using your preferred IDE or by executing the following command in your terminal:

```
mvn archetype:generate -DgroupId=com.example -DartifactId=mongodb-demo -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

3. Add MongoDB Java driver dependency:

- Open the `pom.xml` file in your project directory and add the MongoDB Java driver dependency inside the `<dependencies>` section:

xml

```
<dependencies>
  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>3.12.11</version>
  </dependency>
</dependencies>
```

4. Write Java code:

- Create a Java class in the `src/main/java/com/example` directory with the desired name (e.g., `MongoDBExample.java`). Here is an example snippet for establishing a connection and inserting a document:

java

```
package com.example;

import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;

public class MongoDBExample {
    public static void main(String[] args) {
        try (MongoClient mongoClient =
MongoClients.create("mongodb://localhost:27017")) {
            MongoDatabase database = mongoClient.getDatabase("mydb");
            MongoCollection<Document> collection = database.getCollection("mycollection");

            Document document = new Document("name", "John Doe")
                .append("age", 30)
                .append("email", "johndoe@example.com");

            collection.insertOne(document);
        }
    }
}
```

5. Build and run the application:

- Use the following command in your project directory to build and run the Java application:

```
mvn compile exec:java -Dexec.mainClass="com.example.MongoDBExample"
```

That is how you can integrate a Java application with MongoDB using Maven. Now let's move on to Java integration without Maven.

APPLICATION INTEGRATION WITH MONGODB USING JAVA (WITHOUT MAVEN)

To integrate a Java application with MongoDB without using Maven, you can follow these steps:

1. Download the MongoDB Java driver:

- Visit the MongoDB Java driver download page (https://mongodb.github.io/mongo-java-driver/) and download the JAR file for the desired driver version.

2. Set up your project:

- Create a new directory for your Java project and place the downloaded MongoDB Java driver JAR file inside it.

3. Write Java code:

- Create a Java class in your project directory with the desired name (e.g., `MongoDBExample.java`). Here is an example snippet for establishing a connection and inserting a document:

```
    import com.mongodb.MongoClient;
    import com.mongodb.client.MongoCollection;
    import com.mongodb.client.MongoDatabase;
    import org.bson.Document;

    public class MongoDBExample {
        public static void main(String[] args) {
            try (MongoClient mongoClient = new MongoClient("localhost",
27017)) {
                MongoDatabase database = mongoClient.getDatabase("mydb");
                MongoCollection<Document> collection =
database.getCollection("mycollection");

                Document document = new Document("name", "John Doe")
                    .append("age", 30)
                    .append("email", "johndoe@example.com");

                collection.insertOne(document);
            }
        }
    }
```

4. Compile and run the application:

- Use the following command in your project directory to compile and run the Java application, specifying the classpath for the MongoDB Java driver JAR file:

```
javac -cp mongo-java-driver-x.x.x.jar MongoDBExample.java
java -cp .:mongo-java-driver-x.x.x.jar MongoDBExample
```

Make sure to replace `x.x.x` with the actual version number of the MongoDB Java driver JAR file you downloaded.

That is how you can integrate a Java application with MongoDB both with and without using Maven. Remember to adjust the connection details, such as the MongoDB server URL, database name, and collection name, according to your specific setup.

4.5. MONGODB DEPLOYMENT PROCESS:

When you are preparing a MongoDB deployment, you should try to understand how your application is going to hold up in production. It's a good idea to develop a consistent, repeatable approach to managing your deployment environment so that you can minimize any surprises once you're in production.

The best approach incorporates prototyping your set up, conducting load testing, monitoring key metrics, and using that information to scale your set up. The key part of the approach is to proactively monitor your entire system - this will help you understand how your production system will hold up before deploying, and determine where you will need to add capacity. Having insight into potential spikes in your memory usage, for example, could help put out a write-lock fire before it starts.

To monitor your deployment, MongoDB provides some of the following commands −

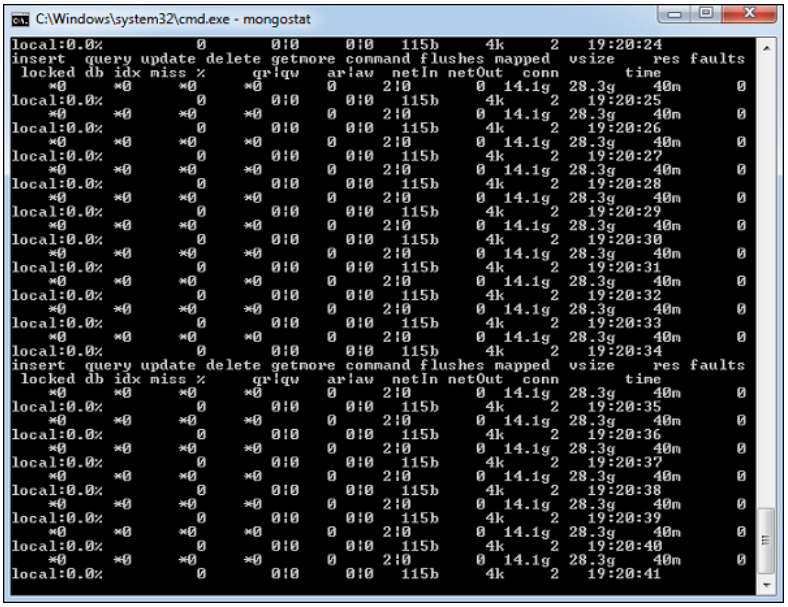
MONGOSTAT

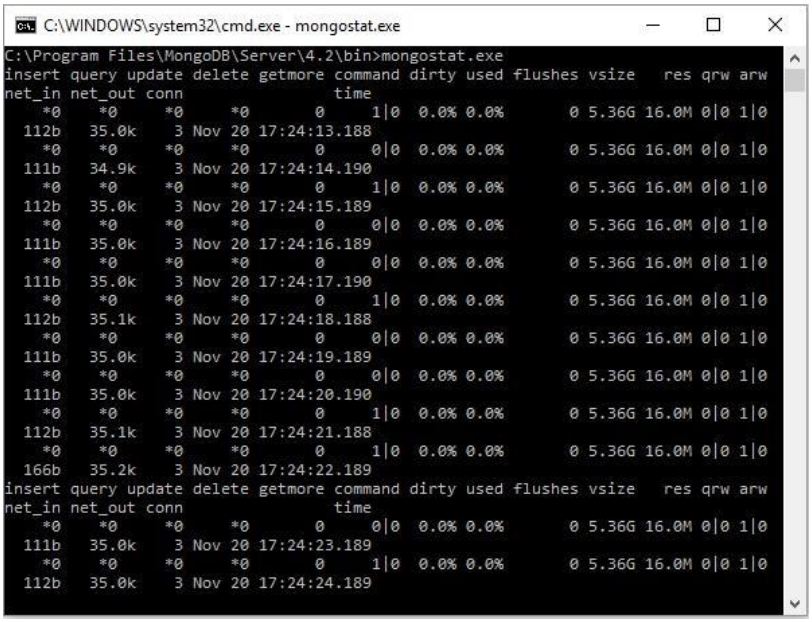
This command checks the status of all running mongod instances and return counters of database operations. These counters include inserts, queries, updates, deletes, and cursors. Command also shows when you are hitting page faults, and showcase your lock percentage. This means that you are running low on memory, hitting write capacity or have some performance issue.

To run the command, start your mongod instance. In another command prompt, go to **bin** directory of your mongodb installation and type **mongostat**.

```
D:\set up\mongodb\bin>mongostat
```

Following is the output of the command −





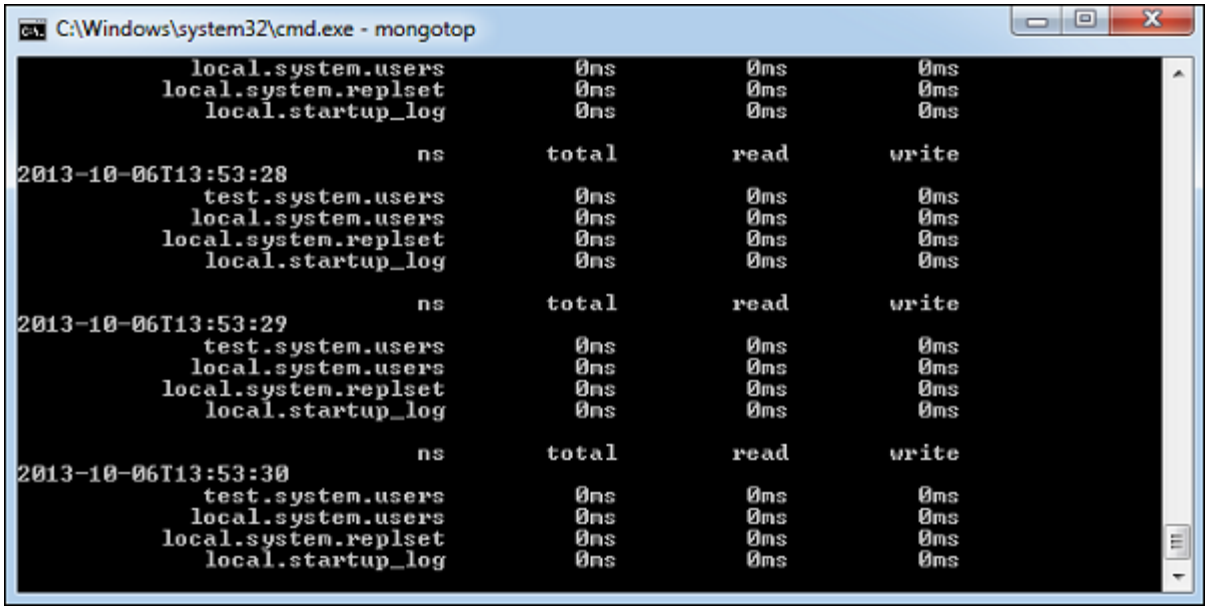
MONGOTOP

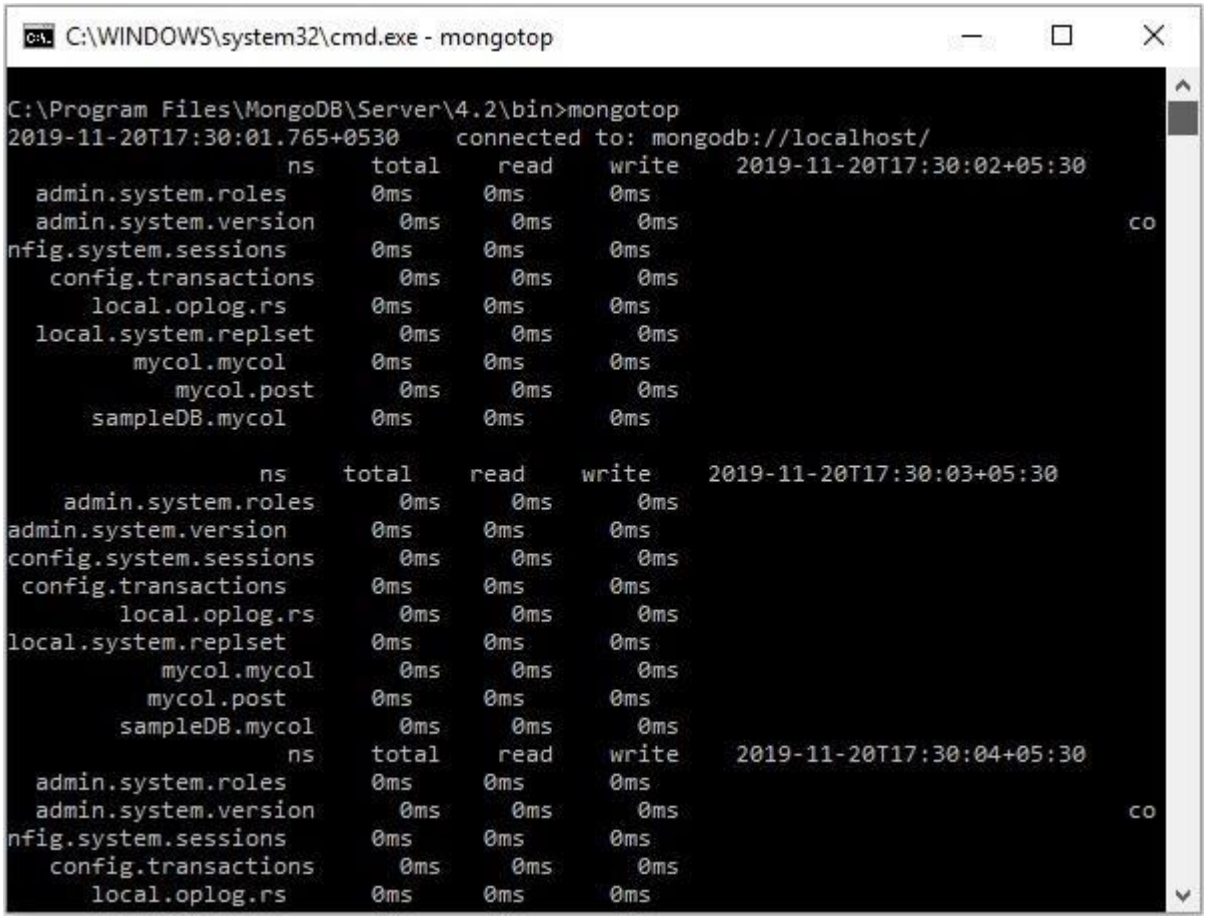
This command tracks and reports the read and write activity of MongoDB instance on a collection basis. By default, **mongotop** returns information in each second, which you can change it accordingly. You should check that this read and write activity matches your application intention, and you're not firing too many writes to the database at a time, reading too frequently from a disk, or are exceeding your working set size.

To run the command, start your mongod instance. In another command prompt, go to **bin** directory of your mongodb installation and type **mongotop**.

```
D:\set up\mongodb\bin>mongotop
```

Following is the output of the command −





To change the **mongotop** command to return information less frequently, specify a specific number after the mongotop command.

```
D:\set up\mongodb\bin>mongotop 30
```

The above example will return values every 30 seconds.

Apart from the MongoDB tools, 10gen provides a free, hosted monitoring service, MongoDB Management Service (MMS), that provides a dashboard and gives you a view of the metrics from your entire cluster.

4.6. ADVANCED MONGODB FEATURES:

1. AGGREGATION FRAMEWORK:

The aggregation framework in MongoDB provides powerful data processing and analysis capabilities. It allows you to perform complex data aggregations, transformations, and computations on your MongoDB collections. Here is an example:

Let us say we have a collection called "orders" with documents representing orders placed by customers. Each document has fields like "customer_id," "order_date," and "total_amount." We want to find the total sales per customer for a given date range. We can use the aggregation framework to achieve this:

```
javascript
db.orders.aggregate([
 {
  $match: {
   order_date: {
    $gte: ISODate("2023-01-01"),
    $lte: ISODate("2023-06-30")
   }
  }
 },
 {
  $group: {
   _id: "$customer_id",
   total_sales: { $sum: "$total_amount" }
  }
 }
])
```

In this example, the aggregation pipeline first matches the orders within the specified date range. Then, it groups the orders by customer ID and calculates the total sales using the `$sum` operator. The result will be a list of customer IDs with their corresponding total sales.

2. TRANSACTIONS:

MongoDB supports multi-document transactions, allowing you to perform multiple operations as a single atomic unit. Transactions ensure that the data remains consistent even when multiple operations are executed simultaneously. Here is an example:

```
javascript
session.startTransaction();

try {
  db.accounts.updateOne({ _id: "A123" }, { $inc: { balance: -100 } });
  db.accounts.updateOne({ _id: "B456" }, { $inc: { balance: 100 } });

  session.commitTransaction();
} catch (error) {
  session.abortTransaction();
} finally {
  session.endSession();
}
```

In this example, we begin a transaction using `session.startTransaction()`. We update two accounts, subtracting $100 from account "A123" and adding $100 to account "B456." If an error occurs during any of the operations, we abort the transaction using `session.abortTransaction()`. Otherwise, we commit the transaction using `session.commitTransaction()` to make the changes persistent.

3. GEOSPATIAL INDEXING:

MongoDB provides geospatial indexing to support efficient querying and analysis of geospatial data. You can store and index locations as points, lines, or polygons and perform various geospatial operations. Here is an example:

Let us assume we have a collection called "places" with documents containing information about different places. Each document has a "location" field representing the coordinates of the place. We want to find places within a certain radius of a given location:

```
javascript
db.places.createIndex({ location: "2dsphere" });

db.places.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [longitude, latitude]
      },
      $maxDistance: 1000
    }
  }
})
```

In this example, we create a 2dsphere index on the "location" field using `db.places.createIndex()`. Then, we use the `$near` operator with `$geometry` to find places near the specified coordinates within a maximum distance of 1000 meters. The result will be a list of places that match the criteria.

4. FULL-TEXT SEARCH:

MongoDB provides full-text search capabilities to perform text-based search queries efficiently. It supports linguistic analysis, relevance ranking, and wildcard searches. Here is an example:

Let us say we have a collection called "articles" with documents representing different articles. Each document has a "title" field and a "content" field. We want to search for articles containing a specific keyword:

```
javascript
db.articles.createIndex({ title: "text", content: "text" });
```

```
db.articles.find({ $text: { $search: "mongodb" } })
```

In this example, we create a text index on both the "title" and "content" fields using `db.articles.createIndex()`. Then, we use the `$text` operator with `$search` to find articles that contain the keyword "mongodb." The result will be a list of articles that match the search criteria.

```
These are just examples to demonstrate the advanced features in MongoDB.
The actual implementation and usage may vary based on your specific
requirements and data model.
```

5. CASSANDRA

Apache Cassandra is an open source, distributed and decentralized/distributed storage system (database), for managing very large amounts of structured data spread out across the world. It provides highly available service with no single point of failure.

Listed below are some of the notable points of Apache Cassandra −

- It is scalable, fault-tolerant, and consistent.
- It is a column-oriented database.
- Its distribution design is based on Amazon's Dynamo and its data model on Google's Bigtable.
- Created at Facebook, it differs sharply from relational database management systems.
- Cassandra implements a Dynamo-style replication model with no single point of failure, but adds a more powerful "column family" data model.
- Cassandra is being used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, ebay, Twitter, Netflix, and more.

5.1. FEATURES OF CASSANDRA

Cassandra has become so popular because of its outstanding technical features. Given below are some of the features of Cassandra:

- **Elastic scalability** − Cassandra is highly scalable; it allows to add more hardware to accommodate more customers and more data as per requirement.
- **Always on architecture** − Cassandra has no single point of failure and it is continuously available for business-critical applications that cannot afford a failure.
- **Fast linear-scale performance** − Cassandra is linearly scalable, i.e., it increases your throughput as you increase the number of nodes in the cluster. Therefore it maintains a quick response time.
- **Flexible data storage** − Cassandra accommodates all possible data formats including: structured, semi-structured, and unstructured. It can dynamically accommodate changes to your data structures according to your need.
- **Easy data distribution** − Cassandra provides the flexibility to distribute data where you need by replicating data across multiple data centers.
- **Transaction support** − Cassandra supports properties like Atomicity, Consistency, Isolation, and Durability (ACID).
- **Fast writes** − Cassandra was designed to run on cheap commodity hardware. It performs blazingly fast writes and can store hundreds of terabytes of data, without sacrificing the read efficiency.

5.2. HISTORY OF CASSANDRA

- Cassandra was developed at Facebook for inbox search.
- It was open-sourced by Facebook in July 2008.
- Cassandra was accepted into Apache Incubator in March 2009.
- It was made an Apache top-level project since February 2010.

5.3. ARCHITECTURE

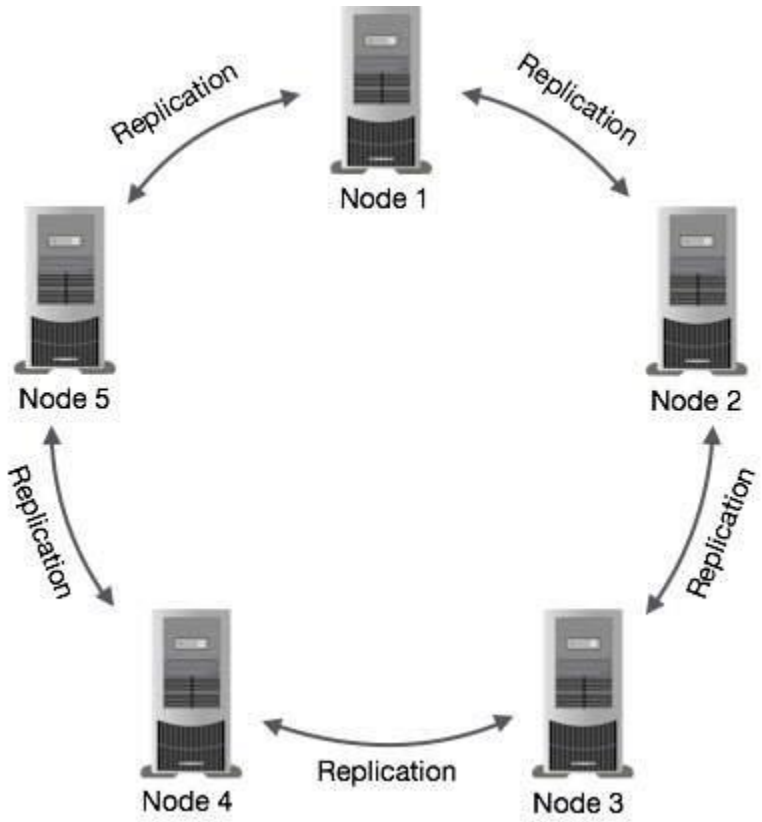
The design goal of Cassandra is to handle big data workloads across multiple nodes without any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster.

- All the nodes in a cluster play the same role. Each node is independent and at the same time interconnected to other nodes.
- Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.
- When a node goes down, read/write requests can be served from other nodes in the network.

DATA REPLICATION IN CASSANDRA

In Cassandra, one or more of the nodes in a cluster act as replicas for a given piece of data. If it is detected that some of the nodes responded with an out-of-date value, Cassandra will return the most recent value to the client. After returning the most recent value, Cassandra performs a **read repair** in the background to update the stale values.

The following figure shows a schematic view of how Cassandra uses data replication among the nodes in a cluster to ensure no single point of failure.



Note − Cassandra uses the **Gossip Protocol** in the background to allow the nodes to communicate with each other and detect any faulty nodes in the cluster.

COMPONENTS OF CASSANDRA

The key components of Cassandra are as follows −

- **Node** − It is the place where data is stored.

- **Data center** − It is a collection of related nodes.
- **Cluster** − A cluster is a component that contains one or more data centers.
- **Commit log** − The commit log is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.
- **Mem-table** − A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.
- **SSTable** − It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- **Bloom filter** − These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

CASSANDRA QUERY LANGUAGE

Users can access Cassandra through its nodes using Cassandra Query Language (CQL). CQL treats the database **(Keyspace)** as a container of tables. Programmers use **cqlsh:** a prompt to work with CQL or separate application language drivers.

Clients approach any of the nodes for their read-write operations. That node (coordinator) plays a proxy between the client and the nodes holding the data.

WRITE OPERATIONS

Every write activity of nodes is captured by the **commit logs** written in the nodes. Later the data will be captured and stored in the **mem-table.** Whenever the mem-table is full, data will be written into the **SStable** data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates the SSTables, discarding unnecessary data.

READ OPERATIONS

During read operations, Cassandra gets values from the mem-table and checks the bloom filter to find the appropriate SSTable that holds the required data.

5.4. DATAMODEL

The data model of Cassandra is significantly different from what we normally see in an RDBMS. This chapter provides an overview of how Cassandra stores its data.

Cluster

Cassandra database is distributed over several machines that operate together. The outermost container is known as the Cluster. For failure handling, every node contains a replica, and in case of a failure, the replica takes charge. Cassandra arranges the nodes in a cluster, in a ring format, and assigns data to them.

Keyspace

Keyspace is the outermost container for data in Cassandra. The basic attributes of a Keyspace in Cassandra are −

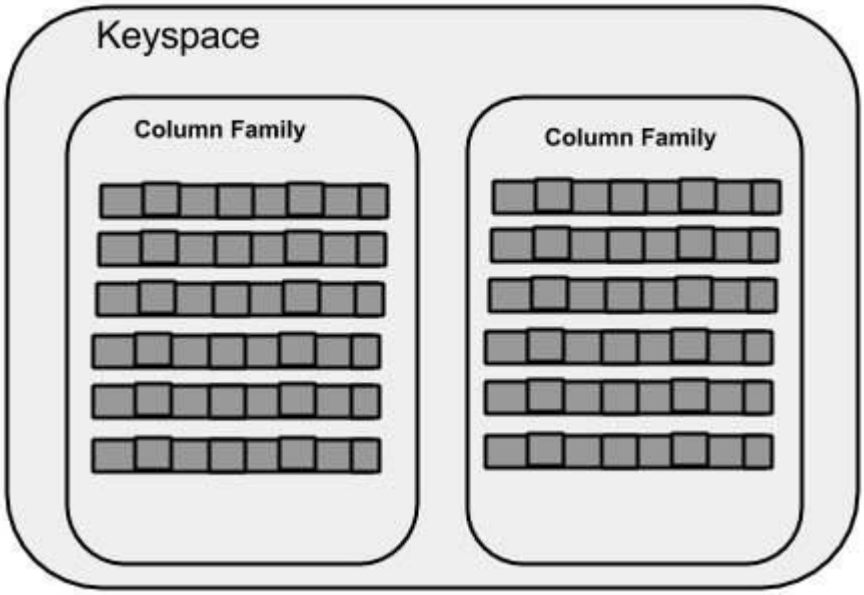
- **Replication factor** − It is the number of machines in the cluster that will receive copies of the same data.
- **Replica placement strategy** − It is nothing but the strategy to place replicas in the ring. We have strategies such as **simple strategy** (rack-aware strategy), **old network topology strategy** (rack-aware strategy), and **network topology strategy** (datacenter-shared strategy).
- **Column families** − Keyspace is a container for a list of one or more column families. A column family, in turn, is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. Each keyspace has at least one and often many column families.

The syntax of creating a Keyspace is as follows −

CREATE KEYSPACE Keyspace name

WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

The following illustration shows a schematic view of a Keyspace.



Column Family

A column family is a container for an ordered collection of rows. Each row, in turn, is an ordered collection of columns. The following table lists the points that differentiate a column family from a table of relational databases.

Relational Table	Cassandra column Family
A schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value.	In Cassandra, although the column families are defined, the columns are not. You can freely add any column to any column family at any time.

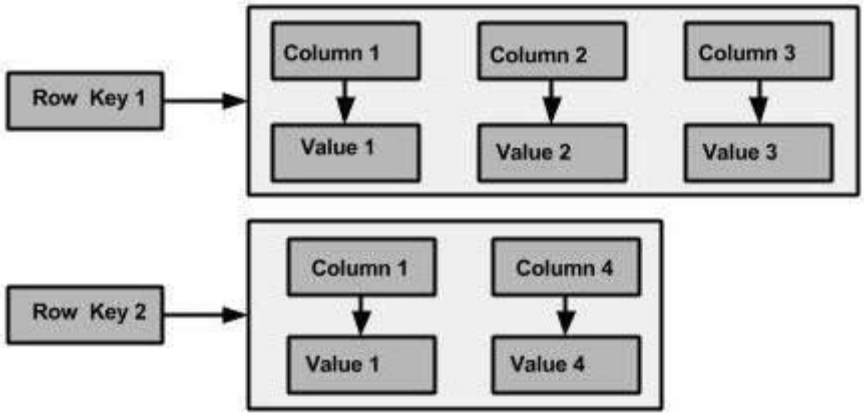
Relational tables define only columns and the user fills in the table with values.	In Cassandra, a table contains columns, or can be defined as a super column family.

A Cassandra column family has the following attributes −

- **keys_cached** − It represents the number of locations to keep cached per SSTable.
- **rows_cached** − It represents the number of rows whose entire contents will be cached in memory.
- **preload_row_cache** − It specifies whether you want to pre-populate the row cache.

Note − Unlike relational tables where a column family's schema is not fixed, Cassandra does not force individual rows to have all the columns.

The following figure shows an example of a Cassandra column family.



Column

A column is the basic data structure of Cassandra with three values, namely key or column name, value, and a time stamp. Given below is the structure of a column.

Column		
name : byte[]	value : byte[]	clock : clock[]

SuperColumn

A super column is a special column, therefore, it is also a key-value pair. But a super column stores a map of sub-columns.

Generally column families are stored on disk in individual files. Therefore, to optimize performance, it is important to keep columns that you are likely to query together in the same column family, and a super column can be helpful here.Given below is the structure of a super column.

Super Column	
name : byte[]	cols : map<byte[], column>

Data Models of Cassandra and RDBMS

The following table lists down the points that differentiate the data model of Cassandra from that of an RDBMS.

RDBMS	Cassandra
RDBMS deals with structured data.	Cassandra deals with unstructured data.
It has a fixed schema.	Cassandra has a flexible schema.
In RDBMS, a table is an array of arrays. (ROW x COLUMN)	In Cassandra, a table is a list of "nested key-value pairs". (ROW x COLUMN key x COLUMN value)
Database is the outermost container that contains data corresponding to an application.	Keyspace is the outermost container that contains data corresponding to an application.
Tables are the entities of a database.	Tables or column families are the entity of a keyspace.
Row is an individual record in RDBMS.	Row is a unit of replication in Cassandra.
Column represents the attributes of a relation.	Column is a unit of storage in Cassandra.
RDBMS supports the concepts of foreign keys, joins.	Relationships are represented using collections.

INSTALLATION AND SETUP OF CASSANDRA

Cassandra can be accessed using cqlsh as well as drivers of different languages. This chapter explains how to set up both cqlsh and java environments to work with Cassandra.

PRE-INSTALLATION SETUP

Before installing Cassandra in Linux environment, we require to set up Linux using **ssh** (Secure Shell). Follow the steps given below for setting up Linux environment.

CREATE A USER

At the beginning, it is recommended to create a separate user for Hadoop to isolate Hadoop file system from Unix file system. Follow the steps given below to create a user.

- Open root using the command **"su"**.
- Create a user from the root account using the command **"useradd username"**.
- Now you can open an existing user account using the command **"su username"**.

Open the Linux terminal and type the following commands to create a user.

```
$ su
password:
# useradd hadoop
# passwd hadoop
New passwd:
Retype new passwd
```

SSH SETUP AND KEY GENERATION

SSH setup is required to perform different operations on a cluster such as starting, stopping, and distributed daemon shell operations. To authenticate different users of Hadoop, it is required to provide public/private key pair for a Hadoop user and share it with different users.

The following commands are used for generating a key value pair using SSH −

- copy the public keys form id_rsa.pub to authorized_keys,
- and provide owner,
- read and write permissions to authorized_keys file respectively.

```
$ ssh-keygen -t rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

- Verify ssh:

```
ssh localhost
```

INSTALLING JAVA

Java is the main prerequisite for Cassandra. First of all, you should verify the existence of Java in your system using the following command −

```
$ java -version
```

If everything works fine it will give you the following output.

java version "1.7.0_71"

Java(TM) SE Runtime Environment (build 1.7.0_71-b13)

Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)

If you don't have Java in your system, then follow the steps given below for installing Java.

Step 1

Download java (JDK <latest version> - X64.tar.gz) from the following link:

Then jdk-7u71-linux-x64.tar.gz will be downloaded onto your system.

Step 2

Generally you will find the downloaded java file in the Downloads folder. Verify it and extract the **jdk-7u71-linux-x64.gz** file using the following commands.

```
$ cd Downloads/
$ ls
jdk-7u71-linux-x64.gz
$ tar zxf jdk-7u71-linux-x64.gz
$ ls
jdk1.7.0_71 jdk-7u71-linux-x64.gz
```

Step 3

To make Java available to all users, you have to move it to the location "/usr/local/". Open root, and type the following commands.

```
$ su
password:
# mv jdk1.7.0_71 /usr/local/
# exit
```

Step 4

For setting up **PATH** and **JAVA_HOME** variables, add the following commands to **~/.bashrc** file.

```
export JAVA_HOME = /usr/local/jdk1.7.0_71
export PATH = $PATH:$JAVA_HOME/bin
```

Now apply all the changes into the current running system.

```
$ source ~/.bashrc
```

Step 5

Use the following commands to configure java alternatives.

```
# alternatives --install /usr/bin/java java usr/local/java/bin/java 2
# alternatives --install /usr/bin/javac javac usr/local/java/bin/javac 2
# alternatives --install /usr/bin/jar jar usr/local/java/bin/jar 2

# alternatives --set java usr/local/java/bin/java
# alternatives --set javac usr/local/java/bin/javac
# alternatives --set jar usr/local/java/bin/jar
```

Now use the **java -version** command from the terminal as explained above.

SETTING THE PATH

Set the path of Cassandra path in "/.bashrc" as shown below.

```
[hadoop@linux ~]$ gedit ~/.bashrc
```

```
export CASSANDRA_HOME = ~/cassandra
export PATH = $PATH:$CASSANDRA_HOME/bin
```

DOWNLOAD CASSANDRA

Apache Cassandra is available at Download Link Cassandra using the following command.

```
$ wget http://supergsego.com/apache/cassandra/2.1.2/apache-cassandra-2.1.2-bin.tar.gz
```

Unzip Cassandra using the command **zxvf** as shown below.

```
$ tar zxvf apache-cassandra-2.1.2-bin.tar.gz.
```

Create a new directory named cassandra and move the contents of the downloaded file to it as shown below.

```
$ mkdir Cassandra
$ mv apache-cassandra-2.1.2/* cassandra.
```

CONFIGURE CASSANDRA

Open the **cassandra.yaml:** file, which will be available in the **bin** directory of Cassandra.

```
$ gedit cassandra.yaml
```

Note − If you have installed Cassandra from a deb or rpm package, the configuration files will be located in **/etc/cassandra** directory of Cassandra.

The above command opens the **cassandra.yaml** file. Verify the following configurations. By default, these values will be set to the specified directories.

- data_file_directories **"/var/lib/cassandra/data"**
- commitlog_directory **"/var/lib/cassandra/commitlog"**
- saved_caches_directory **"/var/lib/cassandra/saved_caches"**

Make sure these directories exist and can be written to, as shown below.

CREATE DIRECTORIES

As super-user, create the two directories **/var/lib/cassandra** and **/var./log/cassandra** into which Cassandra writes its data.

```
[root@linux cassandra]# mkdir /var/lib/cassandra
[root@linux cassandra]# mkdir /var/log/cassandra
```

Give Permissions to Folders

Give read-write permissions to the newly created folders as shown below.

```
[root@linux /]# chmod 777 /var/lib/cassandra
[root@linux /]# chmod 777 /var/log/cassandra
```

START CASSANDRA

To start Cassandra, open the terminal window, navigate to Cassandra home directory/home, where you unpacked Cassandra, and run the following command to start your Cassandra server.

```
$ cd $CASSANDRA_HOME
$./bin/cassandra -f
```

Using the –f option tells Cassandra to stay in the foreground instead of running as a background process. If everything goes fine, you can see the Cassandra server starting.

PROGRAMMING ENVIRONMENT

To set up Cassandra programmatically, download the following jar files −

- slf4j-api-1.7.5.jar
- cassandra-driver-core-2.0.2.jar
- guava-16.0.1.jar
- metrics-core-3.0.2.jar
- netty-3.9.0.Final.jar

Place them in a separate folder. For example, we are downloading these jars to a folder named **"Cassandra_jars"**.

Set the classpath for this folder in **".bashrc"**file as shown below.

```
[hadoop@linux ~]$ gedit ~/.bashrc
```

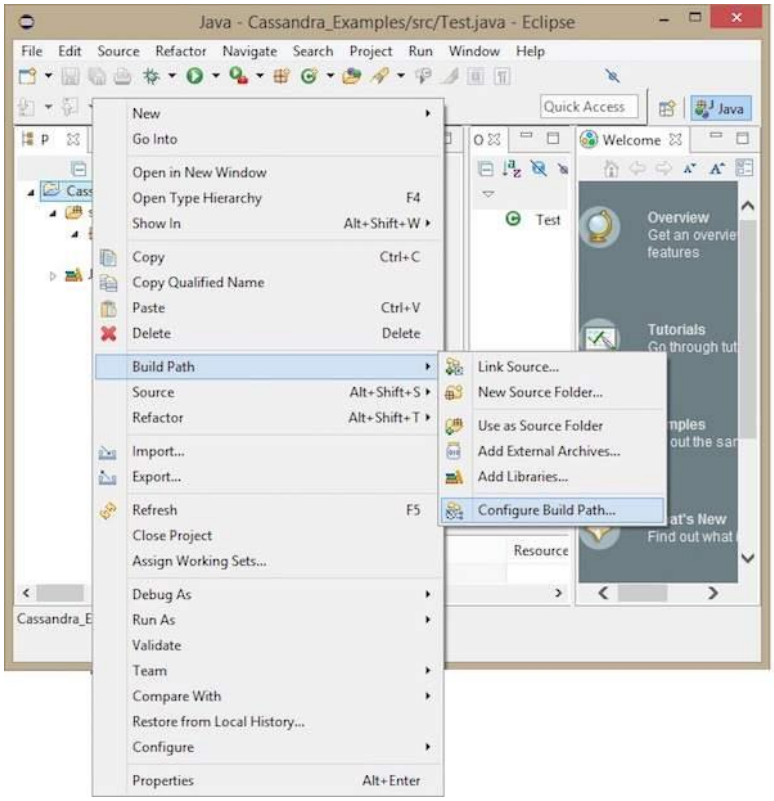
```
//Set the following class path in the .bashrc file.

export CLASSPATH = $CLASSPATH:/home/hadoop/Cassandra_jars/*
```

ECLIPSE ENVIRONMENT

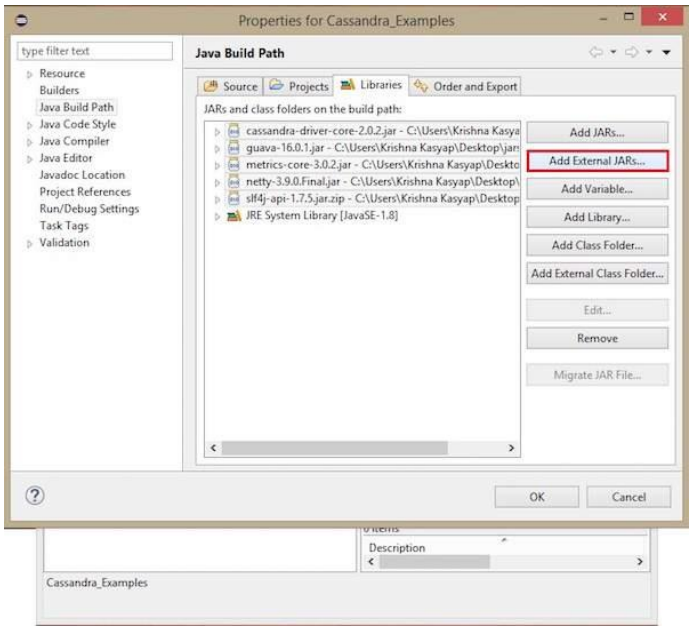
Open Eclipse and create a new project called Cassandra _Examples.

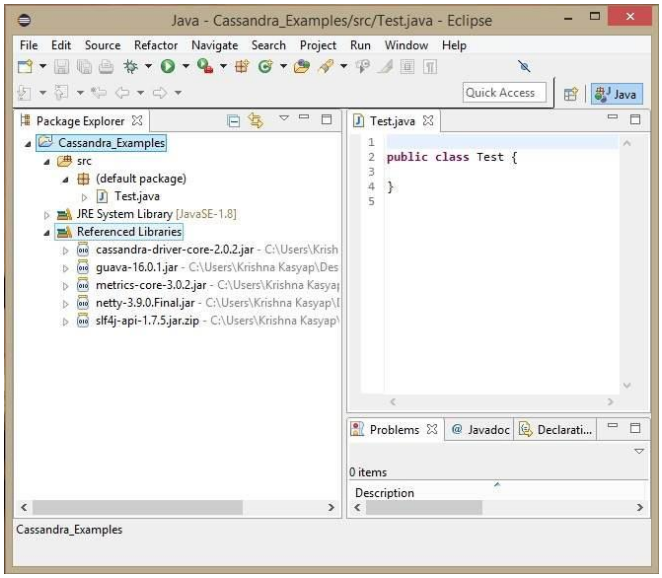
Right click on the project, select **Build Path→Configure Build Path** as shown below.



It will open the properties window. Under Libraries tab, select **Add External JARs**. Navigate to the directory where you saved your jar files. Select all the five jar files and click OK as shown below.

Under Referenced Libraries, you can see all the required jars added as shown below −





MAVEN DEPENDENCIES

Given below is the pom.xml for building a Cassandra project using maven.

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>

          <configuration>
            <source>1.7</source>
            <target>1.7</target>
          </configuration>

      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.slf4j</groupId>
```

```
    <artifactId>slf4j-api</artifactId>
    <version>1.7.5</version>
   </dependency>

   <dependency>
    <groupId>com.datastax.cassandra</groupId>
    <artifactId>cassandra-driver-core</artifactId>
    <version>2.0.2</version>
   </dependency>

   <dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>16.0.1</version>
   </dependency>

   <dependency>
    <groupId>com.codahale.metrics</groupId>
    <artifactId>metrics-core</artifactId>
    <version>3.0.2</version>
   </dependency>

   <dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty</artifactId>
    <version>3.9.0.Final</version>
   </dependency>
  </dependencies>

</project>
```

5.5. CASSANDRA KEYSPACE OPERATIONS:

A keyspace in Cassandra is a namespace that defines data replication on nodes. A cluster contains one keyspace per node. Given below is the syntax for creating a keyspace using the statement CREATE KEYSPACE.

Let us go through some keyspace operations in Cassandra along with examples and their corresponding output.

1. CREATING A KEYSPACE:

To create a keyspace in Cassandra, you can use the `CREATE KEYSPACE` statement. Here's an example:

```
cql
CREATE KEYSPACE my_keyspace
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 3
};
```

This example creates a keyspace named "my_keyspace" with a replication strategy of "SimpleStrategy" and a replication factor of 3. The replication factor determines the number of replicas of each data item across the cluster.

Output:

```
No output. The keyspace "my_keyspace" is created if the query is successful.
```

2. ALTERING A KEYSPACE:

You can alter an existing keyspace in Cassandra using the `ALTER KEYSPACE` statement. Here's an example:

```
cql
ALTER KEYSPACE my_keyspace
WITH replication = {
  'class': 'NetworkTopologyStrategy',
  'DC1': 2,
  'DC2': 2
```

```
};
```

This example alters the keyspace "my_keyspace" to use the "NetworkTopologyStrategy" replication strategy and sets the replication factor to 2 in data centers "DC1" and "DC2".

Output:

```
No output. The keyspace "my_keyspace" is altered if the query is successful.
```

3. DESCRIBING A KEYSPACE:

To retrieve information about a keyspace, you can use the `DESCRIBE KEYSPACE` statement. Here's an example:

```
cql
DESCRIBE KEYSPACE my_keyspace;
```

This example retrieves the metadata and configuration details of the "my_keyspace" keyspace.

Output:

```
Keyspace 'my_keyspace'
----------------------
Replication class: org.apache.cassandra.locator.NetworkTopologyStrategy
Durable Writes: true
Options: [DC1:2, DC2:2]
```

The output provides information such as the replication class, durable writes setting, and replication options of the keyspace.

4. DROPPING A KEYSPACE:

To delete a keyspace in Cassandra, you can use the `DROP KEYSPACE` statement. Here's an example:

```
cql
DROP KEYSPACE my_keyspace;
```

This example drops the keyspace named "my_keyspace" permanently, along with all its associated tables and data.

Output:

```
No output. The keyspace "my_keyspace" is dropped if the query is
successful.
```

These examples demonstrate various keyspace operations in Cassandra, including creating, altering, describing, and dropping a keyspace. Performing these operations allows you to manage the logical containers for your data and configure replication settings according to your application's needs.

5.6. CASSANDRA TABLE OPERATIONS

Let us explore some table operations in Cassandra with examples and their corresponding output.

1. CREATING A TABLE:

To create a table in Cassandra, you can use the `CREATE TABLE` statement. Here's an example:

```
cql
CREATE TABLE my_keyspace.users (
   id UUID PRIMARY KEY,
```

```
    name TEXT,
    age INT,
    email TEXT
);
```

This example creates a table named "users" in the "my_keyspace" keyspace. It has columns for "id" (with UUID data type), "name" (with TEXT data type), "age" (with INT data type), and "email" (with TEXT data type). The "id" column is specified as the primary key.

Output:

```
No output. The table "users" is created if the query is successful.
```

2. ALTERING A TABLE:

You can alter an existing table in Cassandra using the `ALTER TABLE` statement. Here's an example:

```
cql
ALTER TABLE my_keyspace.users
ADD address TEXT;
```

This example adds a new column named "address" to the "users" table in the "my_keyspace" keyspace. The column is specified with the TEXT data type.

Output:

```
No output. The table "users" is altered if the query is successful.
```

3. DESCRIBING A TABLE:

To retrieve information about a table, you can use the `DESCRIBE TABLE` statement. Here's an example:

```
cql
DESCRIBE TABLE my_keyspace.users;
```

This example retrieves the metadata and schema details of the "users" table in the "my_keyspace" keyspace.

Output:

```
CREATE TABLE my_keyspace.users (
    id UUID PRIMARY KEY,
    address TEXT,
    age INT,
    email TEXT,
    name TEXT
)
```

The output displays the CREATE TABLE statement used to create the table, along with its column names and data types.

4. DROPPING A TABLE:

To delete a table in Cassandra, you can use the `DROP TABLE` statement. Here's an example:

```
cql
DROP TABLE my_keyspace.users;
```

This example drops the "users" table from the "my_keyspace" keyspace permanently.

Output:

```
No output. The table "users" is dropped if the query is successful.
```

These examples illustrate various table operations in Cassandra, including creating, altering, describing, and dropping a table. By performing these operations, you can define the structure of your tables, modify them as needed, and manage their lifecycle within a keyspace.

5.7. CASSASNDRA CRUD OPERATIONS:

Certainly! Let's explore the CRUD operations (Create, Read, Update, Delete) in Cassandra with relevant examples and their corresponding output.

1. CREATE (INSERT) OPERATION

To insert data into a Cassandra table, you can use the `INSERT` statement. Here's an example:

```
cql
INSERT INTO my_keyspace.users (id, name, age, email)
VALUES (uuid(), 'John Doe', 25, 'johndoe@example.com');
```

This example inserts a new row into the "users" table in the "my_keyspace" keyspace. The values for the "id", "name", "age", and "email" columns are provided.

Output:

```
No output. The row is inserted if the query is successful.
```

2. READ (SELECT) OPERATION:

To retrieve data from a Cassandra table, you can use the `SELECT` statement. Here is an example

```
SELECT * FROM my_keyspace.users;
```

This example retrieves all rows and columns from the "users" table in the "my_keyspace" keyspace.

Output:

```
 id                                   | address | age | email
| name
 --------------------------------------+---------+-----+-----------------
----+----------
 3f8d76ec-4871-4ff7-9e84-7bbcb8d27ae3 |  null   | 25  |
johndoe@example.com | John Doe
```

The output displays the selected columns ("id", "address", "age", "email", "name") and their corresponding values for each row in the table.

3. UPDATE OPERATION:

To update data in a Cassandra table, you can use the `UPDATE` statement. Here's an example:

cql

```
UPDATE my_keyspace.users
SET age = 30
WHERE id = 3f8d76ec-4871-4ff7-9e84-7bbcb8d27ae3;
```

This example updates the "age" column of the row with the specified "id" in the "users" table.

Output:

```
No output. The row is updated if the query is successful.
```

4. DELETE OPERATION:

To delete data from a Cassandra table, you can use the `DELETE` statement. Here is an example:

```
DELETE FROM my_keyspace.users
WHERE id = 3f8d76ec-4871-4ff7-9e84-7bbcb8d27ae3;
```

This example deletes the row with the specified "id" from the "users" table.

Output:

```
No output. The row is deleted if the query is successful.
```

These examples demonstrate the basic CRUD operations in Cassandra. By performing these operations, you can create new rows, retrieve data, update existing rows, and delete rows from your tables. Remember to adapt the queries based on your specific table structure and requirements.

5.8. CQL DATA TYPE

CQL provides a rich set of built-in data types, including collection types. Along with these data types, users can also create their own custom data types. The following table provides a list of built-in data types available in CQL.

Data Type	Constants	Description
ascii	strings	Represents ASCII character string
bigint	bigint	Represents 64-bit signed long
blob	blobs	Represents arbitrary bytes
Boolean	booleans	Represents true or false
counter	integers	Represents counter column
decimal	integers, floats	Represents variable-precision decimal
double	integers	Represents 64-bit IEEE-754 floating point
float	integers, floats	Represents 32-bit IEEE-754 floating point
inet	strings	Represents an IP address, IPv4 or IPv6
int	integers	Represents 32-bit signed int
text	strings	Represents UTF8 encoded string
timestamp	integers, strings	Represents a timestamp
timeuuid	uuids	Represents type 1 UUID

uuid	uuids	Represents type 1 or type 4
		UUID
varchar	strings	Represents uTF8 encoded string
varint	integers	Represents arbitrary-precision integer

Collection Types

Cassandra Query Language also provides a collection data types. The following table provides a list of Collections available in CQL.

Collection	Description
list	A list is a collection of one or more ordered elements.
map	A map is a collection of key-value pairs.
set	A set is a collection of one or more elements.

User-defined datatypes

Cqlsh provides users a facility of creating their own data types. Given below are the commands used while dealing with user defined datatypes.

- **CREATE TYPE** − Creates a user-defined datatype.
- **ALTER TYPE** − Modifies a user-defined datatype.
- **DROP TYPE** − Drops a user-defined datatype.
- **DESCRIBE TYPE** − Describes a user-defined datatype.
- **DESCRIBE TYPES** − Describes user-defined datatypes.

6. HIVE

The term 'Big Data' is used for collections of large datasets that include huge volume, high velocity, and a variety of data that is increasing day by day. Using traditional data management systems, it is difficult to process Big Data. Therefore, the Apache Software Foundation introduced a framework called Hadoop to solve Big Data management and processing challenges.

6.1. HADOOP

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

- **MapReduce:** It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.
- **HDFS:**Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

- **Sqoop:** It is used to import and export data to and from between HDFS and RDBMS.
- **Pig:** It is a procedural language platform used to develop a script for MapReduce operations.
- **Hive:** It is a platform used to develop SQL type scripts to do MapReduce operations.

Note: There are various ways to execute MapReduce operations:

- The traditional approach using Java MapReduce program for structured, semi-structured, and unstructured data.
- The scripting approach for MapReduce to process structured and semi structured data using Pig.
- The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

6.2. WHAT IS HIVE?

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

6.3. FEATURES OF HIVE

Hive is a data warehouse infrastructure built on top of Hadoop that provides several features to facilitate the management and analysis of large datasets. Here are some key features of Hive:

1. Schema on Read: Hive follows a schema-on-read approach, which means that the schema is applied when the data is queried, rather than when it is stored. This allows for flexibility in working with structured, semi-structured, and unstructured data, as the schema can be inferred or defined during query execution.

2. SQL-Like Query Language: Hive provides a query language called HiveQL, which is similar to SQL. This makes it easy for users familiar with SQL to write queries for data retrieval, filtering, joining, and aggregation. HiveQL supports a wide range of SQL-like operations, including SELECT, JOIN, GROUP BY, ORDER BY, and more.

3. Data Type Support: Hive supports various data types, including primitive types like INT, STRING, BOOLEAN, FLOAT, and DOUBLE, as well as complex types like ARRAY, MAP, and STRUCT. This flexibility allows users to define the structure of their data and handle nested and complex data formats.

4. Partitioning: Hive supports partitioning, which enables users to divide data into logical partitions based on one or more columns. Partitioning helps improve query performance by allowing queries to access only relevant partitions instead of scanning the entire dataset. It is particularly useful for large datasets where filtering based on certain criteria can be optimized.

5. Data Compression: Hive provides built-in support for data compression, allowing users to compress their data to reduce storage requirements and improve query performance. It supports various compression codecs like Gzip, Snappy, and LZO, which can be specified when creating tables or loading data.

6. Extensibility: Hive is extensible and allows users to integrate custom functionality and libraries. It provides User-Defined Functions (UDFs) and User-Defined Aggregations (UDAs) that allow users to define their own functions to perform specialized computations and transformations on data.

7. Metadata Management: Hive maintains metadata about tables, partitions, columns, and other database objects in a metastore. The metastore provides a central repository for storing and retrieving metadata, making it easier to manage the schema and access control for large datasets.

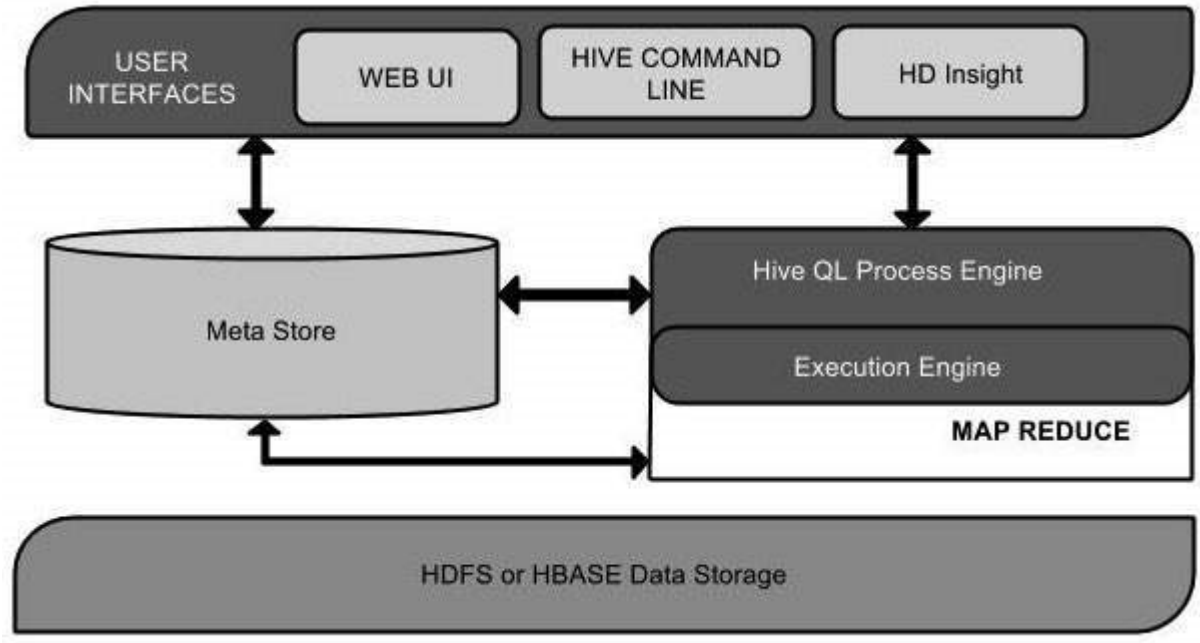
8. Integration with Hadoop Ecosystem: Hive seamlessly integrates with other components of the Hadoop ecosystem. It can leverage the scalability and fault tolerance provided by Hadoop Distributed File System (HDFS) for storing and processing large datasets. Hive queries can also be executed using Apache Tez or Apache Spark, enabling faster and more efficient query processing.

9. Data Processing Optimization: Hive incorporates various optimizations to improve query performance. It includes query optimization techniques like predicate pushdown, join optimization, and map-side processing. Hive also supports the concept of bucketing, which allows data to be organized into buckets based on a hash function, enabling efficient sampling and data skew handling.

These features make Hive a powerful tool for managing and analyzing large datasets in a distributed computing environment. It provides a familiar SQL-like interface, flexibility in data handling, and integration with the Hadoop ecosystem, making it suitable for big data processing and analytics tasks.

6.4. ARCHITECTURE OF HIVE

The following component diagram depicts the architecture of Hive:



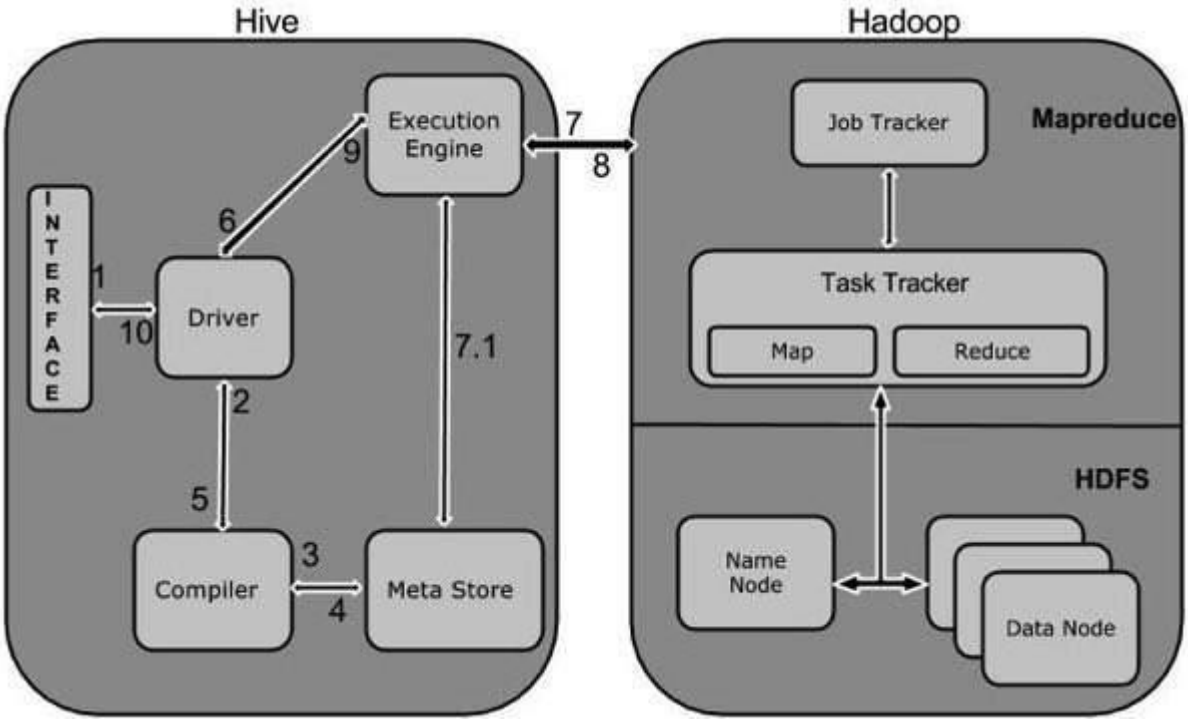
This component diagram contains different units. The following table describes each unit:

Unit Name	Operation
User Interface	Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive

	supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation

1	**Execute Query** The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.
2	**Get Plan** The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	**Get Metadata** The compiler sends metadata request to Metastore (any database).
4	**Send Metadata** Metastore sends metadata as a response to the compiler.
5	**Send Plan** The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	**Execute Plan** The driver sends the execute plan to the execution engine.
7	**Execute Job** Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.
7.1	**Metadata Ops** Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	**Fetch Result** The execution engine receives the results from Data nodes.
9	**Send Results** The execution engine sends those resultant values to the driver.

10	**Send Results** The driver sends the results to Hive Interfaces.

6.5. HIVE TABLE OPERATIONS

In Hive, database operations refer to the actions performed on databases, tables, and data within Hive. Let's explore some common database operations in Hive along with examples and potential outputs:

1. Creating a Database:

To create a database in Hive, you can use the `CREATE DATABASE` statement. Here's an example:

```
CREATE DATABASE mydatabase;
```

Output: The database named "mydatabase" will be created in Hive.

2. Listing Databases:

To list all the databases available in Hive, you can use the `SHOW DATABASES` statement. Here's an example:

```
SHOW DATABASES;
```

Output: A list of all the databases present in Hive will be displayed.

3. Switching Databases:

To switch to a specific database in Hive, you can use the `USE` statement. Here's an example:

```
USE mydatabase;
```

Output: The current database will be changed to "mydatabase".

4. Creating Tables:

To create a table in Hive, you can use the `CREATE TABLE` statement. Here's an example:

```
CREATE TABLE employees (
  id INT,
  name STRING,
  age INT
);
```

Output: The "employees" table with three columns (id, name, age) will be created in the current database.

5. Loading Data into Tables:

To load data into a table in Hive, you can use the `LOAD DATA` statement. Here's an example:

```
LOAD DATA INPATH '/user/hive/data/employees.csv' INTO TABLE employees;
```

Output: The data from the "employees.csv" file located at the specified path will be loaded into the "employees" table.

6. Querying Data:

To query data from a table in Hive, you can use the HiveQL SELECT statement. Here's an example:

```
SELECT * FROM employees;
```

Output: All the records stored in the "employees" table will be returned.

7. Inserting Data into Tables:

To insert data into a table in Hive, you can use the `INSERT INTO` statement. Here's an example:

```
INSERT INTO employees VALUES (1, 'John Doe', 30);
```

```

```

Output: A new record with the provided values will be inserted into the "employees" table.

8. Updating Data:

To update data in a table in Hive, you can use the `UPDATE` statement. Here's an example:

```
UPDATE employees SET age = 31 WHERE id = 1;
```

Output: The age of the record with id 1 in the "employees" table will be updated to 31.

9. Deleting Data:

To delete data from a table in Hive, you can use the `DELETE` statement. Here's an example:

```
DELETE FROM employees WHERE id = 1;
```

Output: The record with id 1 will be deleted from the "employees" table.

These are some of the common database operations in Hive. Depending on your use case, you can perform various other operations like altering tables, dropping databases, renaming tables, and more.

6.6. PARTITIONING

Hive organizes tables into partitions. It is a way of dividing a table into related parts based on the values of partitioned columns such as date, city, and department. Using partition, it is easy to query a portion of the data.

Tables or partitions are sub-divided into buckets, to provide extra structure to the data that may be used for more efficient querying. Bucketing works based on the value of hash function of some column of a table.

For example, a table named Tab1 contains employee data such as id, name, dept, and yoj (i.e., year of joining). Suppose you need to retrieve the details of all employees who joined in 2012. A query searches the whole table for the required information. However, if you partition the employee data with the year and store it in a separate file, it reduces the query processing time. The following example shows how to partition a file and its data:

The following file contains employeedata table.

/tab1/employeedata/file1

```
id, name, dept, yoj
1, gopal, TP, 2012
2, kiran, HR, 2012
3, kaleel,SC, 2013
4, Prasanth, SC, 2013
```

The above data is partitioned into two files using year.

/tab1/employeedata/2012/file2

```
1, gopal, TP, 2012
2, kiran, HR, 2012
```

/tab1/employeedata/2013/file3

```
3, kaleel,SC, 2013
4, Prasanth, SC, 2013
```

ADDING A PARTITION

We can add partitions to a table by altering the table. Let us assume we have a table called employee with fields such as Id, Name, Salary, Designation, Dept, and yoj.

Syntax:

```
ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec
[LOCATION 'location1'] partition_spec [LOCATION 'location2'] ...;

partition_spec:
: (p_column = p_col_value, p_column = p_col_value, ...)
```

The following query is used to add a partition to the employee table.

```
hive> ALTER TABLE employee
> ADD PARTITION (year='2012')
> location '/2012/part2012';
```

RENAMING A PARTITION

The syntax of this command is as follows.

```
ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION
partition_spec;
```

The following query is used to rename a partition:

```
hive> ALTER TABLE employee PARTITION (year='1203')
  > RENAME TO PARTITION (Yoj='1203');
```

DROPPING A PARTITION

The following syntax is used to drop a partition:

```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec, PARTITION
partition_spec,...;
```

The following query is used to drop a partition:

```
hive> ALTER TABLE employee DROP [IF EXISTS]
  > PARTITION (year='1203');
```

6.7. HIVE QL

HiveQL is the query language used in Hive, which is similar to SQL. It allows users to interact with Hive and perform various operations on the data stored in Hive tables. Let's explore some common operations and functions in HiveQL along with example snippets and potential outputs:

1. Creating Tables:

To create a table in Hive, you can use the `CREATE TABLE` statement. Here's an example:

```
CREATE TABLE employees (
  id INT,
  name STRING,
  age INT
);
```

Output: The "employees" table will be created with three columns: id, name, and age.

2. Reading Data:

To read data from a table in Hive, you can use the `SELECT` statement. Here's an example:

```
SELECT * FROM employees;
```

Output: All the records stored in the "employees" table will be returned.

3. Arithmetic Operations:

HiveQL supports various arithmetic operations like addition, subtraction, multiplication, and division. Here's an example:

```
SELECT id, age + 5 AS increased_age FROM employees;
```

Output: The "increased_age" column will be added to the result set, which will contain the original age incremented by 5 for each record.

4. Relational Operations:

HiveQL supports relational operations like equality, inequality, greater than, less than, etc. Here's an example:

```
SELECT * FROM employees WHERE age > 30;
```

Output: Only the records where the age is greater than 30 will be returned.

5. Joining Tables:

HiveQL allows you to join multiple tables using the `JOIN` statement. Here's an example:

```
SELECT e.id, e.name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

Output: The result set will contain the id and name from the "employees" table and the corresponding department_name from the "departments" table, joined on the department_id column.

6. Grouping Data:

To perform grouping operations, you can use the `GROUP BY` statement in HiveQL. Here's an example:

```
SELECT department_id, COUNT(*) as employee_count
FROM employees
GROUP BY department_id;
```

Output: The result set will show the department_id and the count of employees in each department.

7. Ordering Data:

To order the result set based on one or more columns, you can use the `ORDER BY` statement. Here is an example:

```
SELECT * FROM employees
ORDER BY age DESC;
```

Output: The records from the "employees" table will be sorted in descending order based on the age column.

These are just a few examples of the operations and functions you can perform using HiveQL. HiveQL supports a wide range of operations and functions, including aggregation functions, subqueries, window functions, and more, allowing you to manipulate and analyze your data effectively within Hive.

7. ORIENT DATABASE

OrientDB is an Open Source NoSQL Database Management System. **NoSQL Database** provides a mechanism for storing and retrieving NO-relation or NON-relational data that refers to data other than tabular data such as document data or graph data. NoSQL databases are increasingly used in Big Data and real-time web applications. NoSQL systems are also sometimes called "Not Only SQL" to emphasize that they may support SQL-like query languages.

OrientDB also belongs to the NoSQL family. OrientDB is a second generation Distributed Graph Database with the flexibility of Documents in one product with an open source of Apache 2 license. There were several NoSQL databases in the market before OrientDB, one of them being MongoDB.

FEATURES:

The main feature of OrientDB is to support multi-model objects, i.e. it supports different models like Document, Graph, Key/Value and Real Object. It contains a separate API to support all these four models.

DOCUMENT MODEL

The terminology Document model belongs to NoSQL database. It means the data is stored in the Documents and the group of Documents are called as **Collection**. Technically, document means a set of key/value pairs or also referred to as fields or properties.

OrientDB uses the concepts such as classes, clusters, and link for storing, grouping, and analyzing the documents.

The following table illustrates the comparison between relational model, document model, and OrientDB document model −

Relational Model	**Document Model**	**OrientDB Document Model**
Table	Collection	Class or Cluster
Row	Document	Document
Column	Key/value pair	Document field
Relationship	Not available	Link

Graph Model

A graph data structure is a data model that can store data in the form of Vertices (Nodes) interconnected by Edges (Arcs). The idea of OrientDB graph database came from property graph. The vertex and edge are the main artifacts of the Graph model. They contain the properties, which can make these appear similar to documents.

The following table shows a comparison between graph model, relational data model, and OrientDB graph model.

Relational Model	Graph Model	OrientDB Graph Model
Table	Vertex and Edge Class	Class that extends "V" (for Vertex) and "E" (for Edges)
Row	Vertex	Vertex
Column	Vertex and Edge property	Vertex and Edge property
Relationship	Edge	Edge

The Key/Value Model

The Key/Value model means that data can be stored in the form of key/value pair where the values can be of simple and complex types. It can support documents and graph elements as values.

The following table illustrates the comparison between relational model, key/value model, and OrientDB key/value model.

Relational Model	Key/Value Model	OrientDB Key/Value Model
Table	Bucket	Class or Cluster
Row	Key/Value pair	Document
Column	Not available	Document field or Vertex/Edge property
Relationship	Not available	Link

THE OBJECT MODEL

This model has been inherited by Object Oriented programming and supports **Inheritance** between types (sub-types extends the super-types), **Polymorphism** when you refer to a base class and **Direct binding** from/to Objects used in programming languages.

The following table illustrates the comparison between relational model, Object model, and OrientDB Object model.

Relational Model	Object Model	OrientDB Object Model
Table	Class	Class or Cluster

Row	Object	Document or Vertex
Column	Object property	Document field or Vertex/E
Relationship	Pointer	Link

Before go ahead in detail, it is better to know the basic terminology associated with OrientDB. Following are some of the important terminologies.

RECORD

The smallest unit that you can load from and store in the database. Records can be stored in four types.

- Document
- Record Bytes
- Vertex
- Edge

RECORD ID

When OrientDB generates a record, the database server automatically assigns a unit identifier to the record, called RecordID (RID). The RID looks like #<cluster>:<position>. <cluster> means cluster identification number and the <position> means absolute position of the record in the cluster.

DOCUMENTS

The Document is the most flexible record type available in OrientDB. Documents are softly typed and are defined by schema classes with defined constraint, but you can also insert the document without any schema, i.e. it supports schema-less mode too.

Documents can be easily handled by export and import in JSON format. For example, take a look at the following JSON sample document. It defines the document details.

```
{
  "id"       : "1201",
  "name"     : "Jay",
  "job"      : "Developer",
  "creations" : [
    {
      "name"    : "Amiga",
      "company" : "Commodore Inc."
    },
```

```
    {
      "name"    : "Amiga 500",
      "company" : "Commodore Inc."
    }
  ]
}
```

RECORDBYTES

Record Type is the same as BLOB type in RDBMS. OrientDB can load and store document Record type along with binary data.

VERTEX

OrientDB database is not only a Document database but also a Graph database. The new concepts such as Vertex and Edge are used to store the data in the form of graph. In graph databases, the most basic unit of data is node, which in OrientDB is called a vertex. The Vertex stores information for the database.

EDGE

There is a separate record type called the Edge that connects one vertex to another. Edges are bidirectional and can only connect two vertices. There are two types of edges in OrientDB, one is regular and another one lightweight.

CLASS

The class is a type of data model and the concept drawn from the Object-oriented programming paradigm. Based on the traditional document database model, data is stored in the form of collection, while in the Relational database model data is stored in tables. OrientDB follows the Document API along with OPPS paradigm. As a concept, the class in OrientDB has the closest relationship with the table in relational databases, but (unlike tables) classes can be schema-less, schema-full or mixed. Classes can inherit from other classes, creating trees of classes. Each class has its own cluster or clusters, (created by default, if none are defined).

CLUSTER

Cluster is an important concept which is used to store records, documents, or vertices. In simple words, Cluster is a place where a group of records are stored. By default, OrientDB will create one cluster per class. All the records of a class are stored in the same cluster having the same name as the class. You can create up to 32,767(2^15-1) clusters in a database.

The CREATE class is a command used to create a cluster with specific name. Once the cluster is created you can use the cluster to save records by specifying the name during the creation of any data model.

RELATIONSHIPS

OrientDB supports two kinds of relationships: referenced and embedded. **Referenced relationships** means it stores direct link to the target objects of the relationships. **Embedded**

relationships means it stores the relationship within the record that embeds it. This relationship is stronger than the reference relationship.

DATABASE

The database is an interface to access the real storage. IT understands high-level concepts such as queries, schemas, metadata, indices, and so on. OrientDB also provides multiple database types. For more information on these types, see Database Types.

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/

REFERENCES: