**BDA 2nd Section**

**CHAPTER 5 – Introduction to MongoDB**

**1) Explain different CRUD operations in MongoDB.**

MongoDB is a NoSQL database that stores data in a flexible, JSON-like format called BSON (Binary JSON). CRUD operations refer to Create, Read, Update, and Delete operations that can be performed on data in a database. Here's an explanation of each CRUD operation in MongoDB with examples:

**### 1. Create (Insert):**

\*\*Syntax:\*\*

db.collectionName.insert(document)

\*\*Example:\*\*

// Insert a new document into the "users" collection

db.users.insert({

name: "John Doe",

age: 30,

email: "john.doe@example.com"

})

**### 2. Read (Query):**

\*\*Syntax:\*\*

db.collectionName.find(query, projection)

\*\*Example:\*\*

// Find all documents in the "users" collection

db.users.find()

// Find documents with a specific condition

db.users.find({ age: { $gt: 25 } })

**### 3. Update:**

\*\*Syntax:\*\*

db.collectionName.update(query, update, options)

\*\*Example:\*\*

// Update a document in the "users" collection

db.users.update({ name: "John Doe" }, { $set: { age: 31 } })

// Update multiple documents

db.users.update({ age: { $lt: 25 } }, { $inc: { age: 2 } }, { multi: true })

**### 4. Delete:**

\*\*Syntax:\*\*

db.collectionName.remove(query)

\*\*Example:\*\*

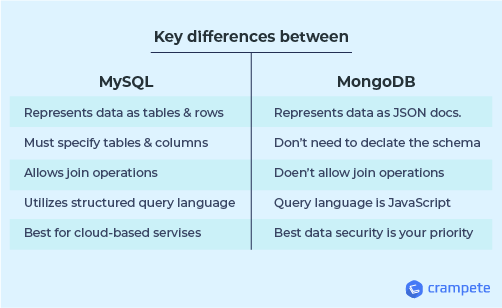
```javascript

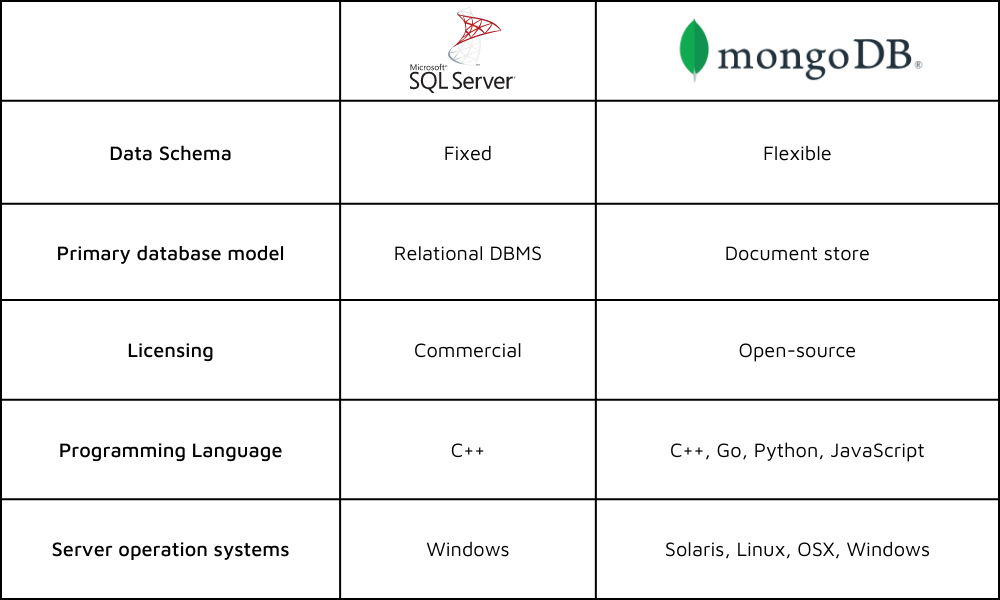
db.users.remove({ name: "John Doe" })

// Remove all documents that match a condition

db.users.remove({ age: { $gt : 40 } }) // age greater than 40

**2) Write difference between MongoDB and SQL**





**3) What is MongoDB ?**

MongoDB is a popular open-source NoSQL (non-relational) database management system. It falls under the category of document-oriented databases, which means it stores and retrieves data in a flexible, JSON-like format called BSON (Binary JSON). MongoDB is designed to handle large volumes of unstructured or semi-structured data, making it well-suited for a variety of applications and use cases.

Key features and characteristics of MongoDB include:

**1. \*\*Document-Oriented:\*\***

- MongoDB stores data in documents, which are JSON-like BSON objects. Each document represents a record and can have a flexible schema, meaning different documents in the same collection can have different fields.

**2. \*\*Scalability:\*\***

- MongoDB is designed to scale horizontally by distributing data across multiple servers. This makes it suitable for handling large amounts of data and traffic.

**3. \*\*Schema Flexibility:\*\***

- Unlike traditional relational databases, MongoDB does not require a predefined schema. Fields can be added or removed from documents without affecting other documents in the same collection.

**4. \*\*Query Language:\*\***

- MongoDB uses a rich query language that supports a wide range of queries and operations. It includes support for filtering, sorting, and aggregation. It uses JS(JavaScript) language for query purpose .

**5. \*\*Indexing:\*\***

- MongoDB supports the creation of indexes on fields, which can significantly improve query performance. Indexes can be created on single fields or compound indexes on multiple fields.

**6. \*\*High Performance:\*\***

- MongoDB is designed for high-performance reads and writes. It uses techniques like memory-mapped files for efficient storage and retrieval of data.

**7. \*\*Replication and High Availability:\*\***

- MongoDB supports automatic data replication, allowing data to be mirrored across multiple servers. This provides fault tolerance and high availability in case of server failures.

**8. \*\*Sharding:\*\***

- Sharding is the process of distributing data across multiple servers. MongoDB supports sharding to horizontally scale databases and improve performance.

**9. \*\*Aggregation Framework:\*\***

- MongoDB provides a powerful aggregation framework for performing data transformations and computations on the server side. It supports a variety of aggregation stages and operators.

**4) Explain Map reduce programming in MongoDB with example.**

<https://www.youtube.com/watch?v=T3393e-fHY0>

MapReduce is a programming model and processing technique used for large-scale data processing and computation. MongoDB provides support for MapReduce as a way to perform complex aggregations and transformations on data stored in collections. The MapReduce process in MongoDB involves two main functions: the map function and the reduce function.

Here's an overview of how MapReduce works in MongoDB:

1. \*\*Map Function:\*\*

- The map function is applied to each document in the collection, producing a set of key-value pairs as output. The map function is responsible for extracting and emitting the relevant data for further processing.

2. \*\*Reduce Function:\*\*

- The reduce function processes the output of the map function, grouping data by key and performing aggregation or computation on each group. The result is a reduced set of key-value pairs.

**CHAPTER 6 – Introduction to Cassendra**

**1) Explain CRUD operations in Cassandra with example query.**

Cassandra is a highly scalable and distributed NoSQL database that is designed to handle large amounts of data across multiple commodity servers. CRUD operations in Cassandra stand for Create, Read, Update, and Delete operations. Here's an explanation of each operation along with an example query:

**1. \*\*Create (Insert):\*\***

To insert data into a Cassandra table, you use the `INSERT` statement. In Cassandra, you need to specify the primary key columns and the values for those columns.

Example:

INSERT INTO keyspace\_name.table\_name (column1, column2, column3)

VALUES (value1, value2, value3);

INSERT INTO users (user\_id, username, email)

VALUES (1, 'john\_doe', 'john.doe@example.com');

**2. \*\*Read (Select):\*\***

Reading data is done using the `SELECT` statement. You can retrieve specific columns or all columns from a table.

Example:

SELECT column1, column2 FROM keyspace\_name.table\_name WHERE condition;

SELECT username, email FROM users WHERE user\_id = 1;

**3. \*\*Update:\*\***

To update existing data, you use the `UPDATE` statement. In Cassandra, you need to specify the columns you want to update and the new values.

Example:

UPDATE keyspace\_name.table\_name SET column1 = value1, column2 = value2 WHERE condition;

UPDATE users SET email = 'new\_email@example.com' WHERE user\_id = 1;

**4. \*\*Delete:\*\***

The `DELETE` statement is used to delete data from a table. You can delete specific rows based on a condition or delete all rows in a table.

Example:

DELETE FROM keyspace\_name.table\_name WHERE condition;

DELETE FROM users WHERE user\_id = 1;

Remember that in Cassandra, the primary key is crucial for data distribution and retrieval, and queries should be designed to align with the data model and access patterns. Additionally, Cassandra supports various consistency levels, allowing you to control the trade-off between consistency and availability in a distributed environment.

**2) Explain features of Cassandra in details.**

Cassandra is a highly scalable, distributed NoSQL database designed for handling large volumes of data across multiple commodity servers. It provides a number of features that make it suitable for high-performance, fault-tolerant, and highly available applications. Here are some key features of Cassandra:

**1. \*\*Distributed and Decentralized:\*\***

Cassandra is designed to be distributed and decentralized. It uses a master-less architecture with no single point of failure. Data is distributed across multiple nodes in a cluster, and each node can serve read and write requests independently.

**2. \*\*Scalability:\*\***

Cassandra provides linear scalability, allowing you to easily add or remove nodes to meet the demands of your application. As your data and traffic increase, you can horizontally scale the cluster by adding more nodes without significant performance degradation.

**3. \*\*High Availability:\*\***

Cassandra is built for high availability, ensuring that your data remains accessible even in the presence of node failures. Data is replicated across multiple nodes, and the system can continue to function even if some nodes are unavailable.

**4. \*\*Fault Tolerance:\*\***

With its decentralized architecture and data replication, Cassandra is fault-tolerant. If a node goes down, data can still be retrieved from other replicas. Cassandra uses strategies like consistent hashing and partitioning to ensure that data is distributed evenly across the cluster.

**5. \*\*Schema-Free:\*\***

Cassandra is a NoSQL database, which means it is schema-free. You can dynamically add columns to a table without affecting the existing data. This flexibility is particularly useful when dealing with evolving data models.

**6. \*\*CQL (Cassandra Query Language):\*\***

While Cassandra is schema-free, it provides a SQL-like language called CQL for interacting with the database. CQL allows users to create, update, and query the database using a syntax similar to SQL, making it more accessible to developers familiar with relational databases.

**7. \*\*Tunable Consistency:\*\***

Cassandra allows you to tune the consistency level on a per-query basis. This means you can choose the level of consistency required for each operation, balancing between consistency and availability based on your application's needs.

**8. \*\*Write and Read Performance:\*\***

Cassandra is optimized for write and read performance. It supports high write throughput and can efficiently handle large amounts of data writes and reads across the cluster.

**9. \*\*Compression and Compaction:\*\***

Cassandra uses compression to reduce storage space requirements, and it employs compaction processes to merge and clean up data on disk, ensuring efficient storage and retrieval.

**10. \*\*Security:\*\***

Cassandra provides security features such as authentication, authorization, and encryption. It supports role-based access control to restrict access to specific resources.

**11. \*\*Integrated Caching:\*\***

Cassandra includes an integrated caching mechanism to improve read performance. Caches can be configured to store frequently accessed data in memory, reducing the need to read from disk.

**12. \*\*Built-in Replication:\*\***

Data replication is a fundamental aspect of Cassandra's design. It allows for data redundancy and ensures that multiple copies of data are stored on different nodes for increased fault tolerance and availability.

These features collectively make Cassandra suitable for use cases that require horizontal scalability, high availability, and fault tolerance, such as in large-scale web applications, time-series data, and other data-intensive scenarios.

**3) What is concept of hinted handoffs, consistency and replication factor in Cassandra.**

In Cassandra, the concepts of hinted handoffs, consistency, and replication factor are important aspects that contribute to the database's fault tolerance, consistency model, and high availability. Let's explore each concept:

**1. \*\*Hinted Handoffs:\*\***

- \*\*Definition:\*\* Hinted handoff is a mechanism in Cassandra that helps ensure data consistency in the face of temporary node failures. When a node in the cluster is temporarily unreachable or down, hinted handoff allows another node to temporarily store a hint or a record of the write operation on behalf of the unreachable node.

- \*\*How it Works:\*\*

- When a write request is made to a node, and the replica for that data on another node is unavailable, the coordinator node will write a hint for the unavailable node.

- The hint is stored locally on the coordinator until the unavailable node becomes reachable again.

- Once the node is back online, it retrieves the hinted data during the hinted handoff process, ensuring that the write operation is eventually applied.

- \*\*Purpose:\*\*

- Hinted handoffs help maintain write availability in scenarios where some nodes are temporarily unavailable, allowing the cluster to continue functioning and serving read and write requests even during node outages.

**2. \*\*Consistency:\*\***

- \*\*Definition:\*\* In Cassandra, consistency refers to the level of agreement between replicas when performing read and write operations. Cassandra provides tunable consistency, allowing developers to choose the level of consistency required for each operation.

- \*\*Consistency Levels:\*\*

- \*\*Consistency Levels for Reads:\*\*

- \*\*ONE:\*\* Reads from at least one replica.

- \*\*QUORUM:\*\* Reads from a majority of replicas.

- \*\*ALL:\*\* Reads from all replicas.

- \*\*Consistency Levels for Writes:\*\*

- \*\*ONE:\*\* Writes to at least one replica.

- \*\*QUORUM:\*\* Writes to a majority of replicas.

- \*\*ALL:\*\* Writes to all replicas.

- \*\*Trade-off:\*\* The choice of consistency level involves a trade-off between consistency and availability. Higher consistency levels (e.g., ALL) ensure stronger consistency but may impact availability, especially in the presence of node failures.

**3. \*\*Replication Factor:\*\***

- \*\*Definition:\*\* The replication factor in Cassandra determines the number of copies (replicas) of each piece of data that are stored across the cluster. It is set at the keyspace level when defining a Cassandra table.

- \*\*How it Works:\*\*

- If the replication factor is set to N, each piece of data is replicated to N nodes in the cluster.

- Replicas are distributed across the cluster using a consistent hashing algorithm.

- Replication factor contributes to fault tolerance by ensuring that data is stored on multiple nodes, allowing the system to continue functioning even if some nodes are down.

- \*\*Example:\*\*

- If the replication factor is set to 3, each piece of data is stored on three nodes in the cluster, providing redundancy and fault tolerance.

These concepts collectively contribute to Cassandra's ability to provide high availability, fault tolerance, and tunable consistency in distributed and decentralized environments. The appropriate configuration of hinted handoffs, consistency levels, and replication factor depends on the specific requirements and characteristics of the application and workload.

**4) With help of example explain set collection, list collection and map collection in Cassendra. when collection should not be used?**

In Cassandra, collections are used to store multiple values within a single column. There are three main types of collections: sets, lists, and maps. Let's explore each type with examples:

**1. \*\*Set Collection:\*\***

- \*\*Definition:\*\* A set in Cassandra is an unordered collection of unique elements. Sets are useful when you need to store a collection of items where the order doesn't matter, and you want to ensure that each item is unique.

- \*\*Example:\*\*

CREATE TABLE user\_preferences (

user\_id UUID PRIMARY KEY,

languages\_set set<text>

);

INSERT INTO user\_preferences (user\_id, languages\_set)

VALUES (uuid1, {'English', 'French', 'Spanish'});

- In this example, the `languages\_set` column stores a set of unique language values for each user.

**2. \*\*List Collection:\*\***

- \*\*Definition:\*\* A list in Cassandra is an ordered collection of elements where duplicates are allowed. Lists are useful when you need to maintain the order of the elements and allow duplicates.

- \*\*Example:\*\*

CREATE TABLE user\_activities (

user\_id UUID PRIMARY KEY,

activities\_list list<text>

);

INSERT INTO user\_activities (user\_id, activities\_list)

VALUES (uuid2, ['Reading', 'Running', 'Swimming']);

- Here, the `activities\_list` column stores a list of activities in a specific order for each user.

**3. \*\*Map Collection:\*\***

- \*\*Definition:\*\* A map in Cassandra is a collection of key-value pairs, where each key is unique. Maps are useful when you need to associate values with specific keys.

- \*\*Example:\*\*

CREATE TABLE user\_contacts (

user\_id UUID PRIMARY KEY,

contact\_info map<text, text>

);

INSERT INTO user\_contacts (user\_id, contact\_info)

VALUES (uuid3, {'email': 'user@example.com', 'phone': '123-456-7890'});

```

- In this example, the `contact\_info` column stores a map of contact information with keys like 'email' and 'phone'.

**\*\*When Collections Should Not Be Used:\*\***

While collections in Cassandra provide flexibility, they should be used judiciously. Here are some scenarios where collections might not be the best choice:

1. \*\*High Cardinality:\*\*

- Collections with high cardinality (many unique elements) can lead to performance issues and increased storage requirements. In such cases, consider alternative data modeling approaches.

2. \*\*Frequent Updates:\*\*

- Collections are not optimal for scenarios with frequent updates, especially in the case of sets and lists. Updates might require reading and rewriting the entire collection, leading to inefficiencies.

3. \*\*Querying Within Collections:\*\*

- Cassandra is designed for efficient querying based on primary keys. If you find yourself needing to query within collections extensively, it might be a sign that a different data model or denormalization approach is more suitable.

4. \*\*Large Collections:\*\*

- Large collections can impact performance, especially during writes. If collections become too large, it might be more efficient to break them into separate tables or use a different data modeling strategy.

5. \*\*Complex Nesting:\*\*

- Excessive nesting of collections within collections can make queries and updates more complex. It's important to strike a balance between flexibility and simplicity in your data model.

**CHAPTER 6: INTRODUCTION TO HIVE**

**1) List out features of Hive.**

Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis. It enables users to write SQL-like queries, known as HiveQL, to analyze large datasets stored in Hadoop Distributed File System (HDFS). Here are some key features of Apache Hive:

**1. \*\*SQL-Like Query Language (HiveQL):\*\***

- Hive provides a SQL-like language called HiveQL, which allows users to query and analyze data using familiar SQL syntax. This makes it easier for users who are already familiar with SQL to work with big data.

**2. \*\*Schema on Read:\*\***

- Hive follows a schema-on-read approach, allowing users to structure the data at the time of querying rather than at the time of storage. This flexibility is beneficial when dealing with semi-structured or unstructured data.

**3. \*\*Data Storage in HDFS:\*\***

- Hive stores its data in Hadoop Distributed File System (HDFS), making it a scalable solution for handling large volumes of data across a distributed storage infrastructure.

**4. \*\*Integration with Hadoop Ecosystem:\*\***

- Hive seamlessly integrates with other components of the Hadoop ecosystem, including Hadoop MapReduce, HBase, Spark, and more. This integration allows users to leverage the strengths of different tools within the Hadoop ecosystem.

**5. \*\*Partitioning and Bucketing:\*\***

- Hive supports data partitioning, allowing users to organize data in Hive tables based on one or more partition keys. This can significantly improve query performance by minimizing the amount of data that needs to be scanned.

- Bucketing is another feature that allows users to group data into buckets based on a hash function, which can improve query performance.

**6. \*\*Extensibility:\*\***

- Hive is extensible, and users can define their own custom functions (UDFs), serializers, and deserializers to process data. This extensibility enables users to integrate Hive with their specific use cases and requirements.

**7. \*\*Metadata Store (Hive Metastore):\*\***

- Hive uses a metadata store, known as the Hive Metastore, to store information about tables, partitions, and schemas. This metadata allows Hive to manage and organize data efficiently.

**8. \*\*Optimization and Query Execution Plans:\*\***

- Hive performs optimization of queries and generates query execution plans. It includes a cost-based optimizer that helps in selecting the most efficient execution plan for a given query.

**9. \*\*User-Defined Functions (UDFs):\*\***

- Users can define their own functions in Hive, known as User-Defined Functions (UDFs), to perform custom processing on the data. This flexibility allows users to extend Hive's functionality.

**10. \*\*JDBC and ODBC Support:\*\***

- Hive provides support for JDBC (Java Database Connectivity) and ODBC (Open Database Connectivity), enabling users to connect to Hive using a variety of programming languages and tools.

**11. \*\*Security:\*\***

- Hive supports authentication and authorization mechanisms, including integration with Apache Ranger and Apache Sentry, to control access to data and metadata.

**12. \*\*Data Serialization and Deserialization:\*\***

- Users can specify custom serialization and deserialization formats for data stored in Hive tables. This feature allows Hive to work with various data formats, including Avro, ORC, Parquet, and more.

These features collectively make Hive a powerful tool for processing and querying large datasets in a Hadoop ecosystem, providing a SQL-like interface for data analysis on top of distributed storage.

**2) Draw the architecture of hive and explain its components.**

<https://www.javatpoint.com/hive-architecture>

**3) What is static and dynamic partitioning in HIVE? Illustrate with example.**

In Hive, partitioning is a mechanism for organizing data in a table into sub-directories based on the values of one or more columns. This can significantly improve query performance by limiting the amount of data that needs to be scanned. There are two types of partitioning in Hive: static partitioning and dynamic partitioning.

**1. \*\*Static Partitioning:\*\***

- \*\*Definition:\*\* Static partitioning involves explicitly specifying the partition values for each record at the time of insertion.

- \*\*Example:\*\*

Consider a table called `sales` with the following schema:

CREATE TABLE sales (

product\_id INT,

date STRING,

revenue DOUBLE

)

PARTITIONED BY (country STRING);

To insert data into this table with static partitioning, you explicitly specify the partition value for each record:

-- Insert data into the 'sales' table with static partitioning

INSERT INTO TABLE sales PARTITION (country='USA') VALUES (101, '2023-01-01', 100.0);

INSERT INTO TABLE sales PARTITION (country='Canada') VALUES (102, '2023-01-01', 150.0);

```

**2. \*\*Dynamic Partitioning:\*\***

- \*\*Definition:\*\* Dynamic partitioning allows Hive to automatically determine the partition values based on the data being inserted.

- \*\*Example:\*\*

Using the same `sales` table, you can use dynamic partitioning as follows:

-- Enable dynamic partitioning

SET hive.exec.dynamic.partition=true;

SET hive.exec.dynamic.partition.mode=nonstrict;

-- Insert data into the 'sales' table with dynamic partitioning

INSERT INTO TABLE sales PARTITION (country) VALUES (103, '2023-01-01', 200.0, 'UK');

INSERT INTO TABLE sales PARTITION (country) VALUES (104, '2023-01-01', 120.0, 'France');

```

In this example, the `country` partition value is determined dynamically based on the values provided in the `INSERT` statement.

**\*\*Important Notes:\*\***

- For dynamic partitioning, it's important to set the Hive configuration properties `hive.exec.dynamic.partition` and `hive.exec.dynamic.partition.mode` appropriately.

- Dynamic partitioning is generally more flexible and convenient, especially when dealing with a large number of partitions or when the partition values are not known in advance.

- Static partitioning is explicit and allows for better control over the partition values, but it may be less convenient in scenarios where the number of partitions is large or changes frequently.

The choice between static and dynamic partitioning depends on the specific requirements of the use case and the nature of the data being processed in Hive. Both approaches help optimize query performance by organizing data into directories based on partition values.

**CHAPTER 7: INTRODUCTION TO PIG**

**1) What is pig? Explain its features.**

Apache Pig is a high-level platform and scripting language built on top of Hadoop. It provides a simple abstraction over the MapReduce programming model, making it easier for developers to process and analyze large datasets. Pig is particularly useful for data processing tasks where the complexity of writing MapReduce programs in Java is not justified. Here are some key features of Apache Pig:

**1. \*\*Abstraction over MapReduce:\*\***

- Pig provides a higher-level scripting language (Pig Latin) that abstracts away the complexity of writing low-level MapReduce programs. This abstraction allows developers to express data transformations in a more concise and readable manner.

**2. \*\*Ease of Use:\*\***

- Pig is designed to be easy to use, especially for users familiar with scripting languages. Pig Latin scripts resemble SQL queries, making it accessible to a broader audience.

**3. \*\*Extensibility:\*\***

- Pig is extensible, and users can write User Defined Functions (UDFs) in Java, Python, or other languages to perform custom processing on data. This extensibility allows users to incorporate their own logic into Pig scripts.

**4. \*\*Optimization Opportunities:\*\***

- Pig automatically optimizes and parallelizes data processing tasks. It performs logical and physical optimization to improve the efficiency of the execution plan.

**5. \*\*Schema Flexibility:\*\***

- Pig is schema-flexible, meaning it can handle semi-structured and unstructured data. While Pig supports a schema-on-read approach, where the schema can be inferred at runtime, users can also define schemas explicitly.

**6. \*\*Rich Set of Operators:\*\***

- Pig provides a rich set of built-in operators to perform various data manipulations, including filtering, grouping, sorting, and joining. These operators can be combined to express complex data transformations.

**7. \*\*Multi-Query Execution:\*\***

- Pig supports multi-query execution, allowing users to express a sequence of data processing steps in a single script. This can improve overall efficiency by avoiding unnecessary intermediate data storage.

**8. \*\*Parallel Execution:\*\***

- Pig automatically parallelizes data processing tasks across a Hadoop cluster. This parallel execution enables Pig to scale horizontally and process large datasets efficiently.

**9. \*\*Built-In Functions:\*\***

- Pig includes a variety of built-in functions that can be used for common data manipulation tasks. These functions cover a wide range of operations, from mathematical functions to string manipulations.

**10. \*\*Integration with Hadoop Ecosystem:\*\***

- Pig seamlessly integrates with other components of the Hadoop ecosystem, such as HDFS (Hadoop Distributed File System), HBase, and Hive. This integration allows users to leverage the strengths of different tools within the Hadoop ecosystem.

**11. \*\*Community and Support:\*\***

- Pig has an active open-source community and is supported by the Apache Software Foundation. Users can find documentation, tutorials, and community support to help with their Pig-related questions and challenges.

Overall, Apache Pig simplifies the development of data processing applications on Hadoop by providing a high-level scripting language and abstracting away many of the complexities associated with writing MapReduce programs. Its ease of use, extensibility, and optimization capabilities make it a valuable tool for large-scale data processing tasks.

**2) What is pig philosophy? Explain ETL processing in Pig.**

The philosophy of Apache Pig can be summarized with the following principles:

**1. \*\*Abstraction:\*\***

- Pig provides a high-level abstraction over the complexities of MapReduce programming. Users write Pig Latin scripts to express data transformations, and Pig takes care of translating those scripts into a series of MapReduce jobs.

**2. \*\*Optimization:\*\***

- Pig is designed to automatically optimize the execution plan of a Pig Latin script. It performs logical optimization to simplify the script and physical optimization to improve the efficiency of the MapReduce jobs it generates.

**3. \*\*Extensibility:\*\***

- Pig is extensible, allowing users to write custom User Defined Functions (UDFs) in Java, Python, or other languages. This enables users to incorporate their own processing logic into Pig scripts.

**4. \*\*Flexibility:\*\***

- Pig is schema-flexible, accommodating both structured and semi-structured data. Users can define schemas explicitly or use a schema-on-read approach where the schema is inferred at runtime.

**5. \*\*Parallel Execution:\*\***

- Pig is designed for parallel execution. It automatically parallelizes data processing tasks across a Hadoop cluster, allowing for scalable and efficient processing of large datasets.

**6. \*\*Ease of Use:\*\***

- Pig emphasizes ease of use, providing a scripting language (Pig Latin) that is simple, readable, and resembles SQL. This makes Pig accessible to a broader audience, including those who may not be proficient in Java.

**### ETL Processing in Pig:**

ETL (Extract, Transform, Load) is a common data integration process, and Apache Pig is well-suited for ETL tasks on Hadoop. Here's how ETL processing is typically done in Pig:

**1. \*\*Extract:\*\***

- The extraction phase involves loading data from various sources into Hadoop, often into the Hadoop Distributed File System (HDFS). Sources can include log files, databases, or other data repositories.

**2. \*\*Transform:\*\***

- The transformation phase is where Pig plays a crucial role. Pig Latin scripts are used to express the transformations needed on the raw data. This can include filtering, cleaning, aggregating, and any other data manipulation tasks.

- Example Pig Latin script for transformation:

-- Load data from HDFS

raw\_data = LOAD '/input/data' USING PigStorage(',');

-- Filter out records based on a condition

filtered\_data = FILTER raw\_data BY column\_name == 'desired\_value';

-- Perform some transformations

transformed\_data = FOREACH filtered\_data GENERATE column1, column2 \* 2 AS doubled\_column;

-- Store the transformed data back to HDFS

STORE transformed\_data INTO '/output/data' USING PigStorage(',');

**3. \*\*Load:\*\***

- In the loading phase, the transformed data is stored back into HDFS or another data store for further analysis or reporting. This step completes the ETL process, making the processed data available for downstream applications.

- The final output may be used for analysis with tools like Hive, or it may be loaded into a data warehouse for reporting and business intelligence purposes.

In summary, Pig simplifies the ETL process on Hadoop by providing a high-level scripting language (Pig Latin) that abstracts away the complexities of MapReduce. Its philosophy of abstraction, optimization, and extensibility makes it a valuable tool for users involved in large-scale data processing and ETL tasks on Hadoop clusters.

**3) Write word count example using PIG.**

Sure, let's create a simple Word Count example using Apache Pig. In this example, we'll process a text file, tokenize the words, and count the occurrences of each word.

Assuming you have a text file named `input.txt` with the following content:

Simple text

Hello world

Apache Pig is awesome

Pig Latin is easy

Word count example with Pig

**Here's a Word Count example in Pig Latin:**

-- Load the input data from HDFS

input\_data = LOAD '/path/to/input.txt' USING PigStorage('\t') AS (line:chararray);

-- Tokenize the words

tokenized\_data = FOREACH input\_data GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- Group by word and count occurrences

word\_counts = GROUP tokenized\_data BY word;

word\_count\_result = FOREACH word\_counts GENERATE group AS word, COUNT(tokenized\_data) AS count;

-- Store the result back to HDFS

STORE word\_count\_result INTO '/path/to/output' USING PigStorage('\t');

-- Display the result

DUMP word\_count\_result;

```

In this Pig Latin script:

1. `LOAD`: Loads the input data from HDFS into the `input\_data` relation.

2. `FOREACH`: Tokenizes each line into words using the `TOKENIZE` function and `FLATTEN` operator.

3. `GROUP`: Groups the tokenized data by word.

4. `COUNT`: Counts the occurrences of each word within the groups.

5. `STORE`: Stores the result into HDFS in the specified output directory.

6. `DUMP`: Displays the word count result on the console.

Save this script to a file, for example, `word\_count.pig`, and execute it using the Pig command:

pig -f word\_count.pig

Make sure to replace `/path/to/input.txt` with the actual path to your input file and `/path/to/output` with the desired output directory.

After execution, you can check the output in HDFS or the specified output directory to see the word count result.