What is Hive?

➞ Hive is an open-source data warehousing and SQL-like querying tool built on top of Hadoop. It was created by Facebook to enable SQL-like access to Hadoop Distributed File System (HDFS) data, making it easier for non-programmers to work with big data.

➞ Hive is a data warehouse system for data summarization, analysis and for querying of large data.
➞ It converts SQL-like queries into MapReduce jobs for easy execution and processing of extremely large volumes of data.

➞ Hive can process structured and semi-structured data by using a SQL-like language called HQL (Hive Query Language).

➞ It allows you to define schemas and data models, and run queries using a variety of data formats, including CSV, JSON, and more.

➞ Hive also supports data partitioning, indexing, and compression for faster query performance.

➞ One of the biggest advantages of Hive is its scalability. It can handle massive amounts of data, making it an ideal tool for large-scale data processing and analysis.

Different data types in Hive?

➞ Hive supports various data types that can be used to define columns in tables. Many of the datatypes you find in Relational Databases can be found in Hive as well. They are called Primitive Datatypes. Along with these , Hive also has Complex Datatypes.

➞ HIVE Primitive Data Types

☞ Numeric Data Types : Hive supports several numeric data types, including TINYINT, SMALLINT, INT, BIGINT, FLOAT, and DOUBLE. These data types are used to store numeric values and can be used to perform arithmetic operations.

☞ String Data Types : Hive supports several string data types, including STRING, VARCHAR, and CHAR. These data types are used to store character data.

☞ Date/Time Data Types : Hive supports several date and time data types, including TIMESTAMP, DATE, and INTERVAL. These data types are used to store time-related data.

☞ Miscellaneous Data Types : Hive supports 2 miscellaneous data types Boolean and Binary.

➞ HIVE Complex Data Types

☞ ARRAY : An ordered collection of elements of the same type. Just like other programming languages it is a collection of elements of similar data type. The elements are maintained in an index , you can retrieve the value like column_name[index_num].

☞ MAP: An unordered collection of key-value pairs. It is a collection of Key Value pairs. Here the values can be accessed by providing the column_name[keys].

☞ STRUCT: A collection of named fields of different types. This is a collection of named fields where each field can be of any primitive datatype. Fields in Struct can be accessed using Dot(.) operator. Column_name.Field_name

Different types of tables in Hive?

➞ In this post, we will discuss different types of tables in Hive.

➞ **Managed/Internal Table:**

☞ Managed table is also called as Internal table. Managed Tables in Hive are the default type of tables.

☞ When you create a managed table, Hive creates a directory in HDFS and manages the data stored in it.

☞ All the operations related to the table, such as inserting data, dropping the table, or altering the schema, are managed by Hive itself.

☞ Managed Tables are best suited for small to medium-sized datasets.

☞ If we delete a Managed table, both the table data and metadata for that table will be deleted from the HDFS.

☞ Stored in a directory based on settings in hive.metastore.warehouse.dir, by default internal tables are stored in the following directory "/user/hive/warehouse" you can change it by updating the location in the config file

☞ SYNTAX:

```
CREATE TABLE employees (

id INT,

name STRING,

age INT )

COMMENT 'Employee details'

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ',' STORED AS TEXTFILE


LOCATION '/user/hive/warehouse/employees';
```

➞ **External Tables:**

☞ External Tables in Hive are similar to Managed Tables, but the data is not managed by Hive.

☞ When you create an External Table, you have to specify the location of the data in HDFS.

☞ External Tables are best suited for large datasets that are managed by external systems.

☞ If you drop an External Table, the data stored in HDFS will not be deleted, as it is not managed by Hive.

☞ Whenever we want to delete the table's metadata and we want to keep the table's data as it is, we use External table.

☞ External table only deletes the schema of the table.

☞ External table stores files on the HDFS server but tables are not linked to the source file completely. ☞

SYNTAX:

```
CREATE EXTERNAL TABLE sales (
id INT, product STRING, price DOUBLE
)
COMMENT 'Sales details'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE LOCATION '/user/data/sales';
```

Explain Hive Partitioning?

➞ One of the key features of Hive is the ability to partition data in a table. Partitioning can greatly improve the performance of queries and make it easier to manage large datasets.

➞ A partition in Hive is a way of dividing a table into smaller, more manageable pieces based on a specific column or set of columns. Each partition contains a subset of the data in the table, and each partition is stored in a separate directory or folder on the Hadoop Distributed File System (HDFS).

➞ Partitioning is particularly useful when working with large datasets that are frequently queried based on a specific column. For example, if you have a table of sales data and you frequently run queries to aggregate sales by date, you can partition the table by date. This way, each partition will contain all the sales data for a specific date range, and queries that only need to access a specific date range can be executed more quickly.

➞ Partitioning can be done based on more than one column which will impose multi-dimensional structure on directory storage. For e.g. in addition to partitioning records by date column, we can also sub-divide the single day records into country wise separate files by including country column into partitioning.

➞ To create a partitioned table in Hive, you first need to specify the partition columns in the table definition. For example, the following SQL statement creates a partitioned table called sales with two partition columns, year and month:

```
CREATE TABLE sales (sale_id INT,
sale_date TIMESTAMP,
sale_amount DECIMAL(10,2),
customer_id INT)
PARTITIONED BY (year INT, month INT);
```

➞ Once the partitioned table has been created, you can load data into it by specifying the partition columns in the LOAD DATA statement. For example, the following command loads sales data for January 2022 into the sales table:

```
LOAD DATA INPATH '/path/to/january-2022-sales.csv' INTO TABLE sales PARTITION (year=2022, month=1);
```

➞ To query a partitioned table in Hive, you can use the WHERE clause to filter the results based on the partition columns. For example, the following query returns all sales data for the year 2022:

```
SELECT * FROM sales WHERE year=2022;
```

Difference between static Partitioning and dynamic Partitioning in hive?

➞ Hive provides two types of partitioning methods: static partitioning and dynamic partitioning.

➞ **Static Partitioning:**

☞ Static partitioning is a technique where data is partitioned based on predefined values.

☞ In static partitioning, the values for partition columns are known beforehand, and partitions are created manually using the ALTER TABLE statement.

☞ In this mode, input data should contain the columns listed in table definition but not the columns defined in partition by clause.

☞ For example, if you have a dataset that contains sales data for different regions, you can partition the data by region using static partitioning.

☞ Static partitioning has some advantages over dynamic partitioning. It is easier to manage and optimize because the partitions are known beforehand, and it allows for more precise control over data placement.

☞ However, it is not very flexible, and you need to manually create partitions for every new value of the partition column.

➞ **Dynamic Partitioning:**

☞ Dynamic partitioning is a technique where Hive automatically creates partitions based on the data.

☞ In dynamic partitioning, the values for partition columns are determined dynamically based on the data being loaded into the table.

☞ Hive will automatically splits our data into separate partitions files based on the values of partition keys present in input files.

☞ For example, if you have a sales dataset that contains information about different regions, you can use dynamic partitioning to create a partition for each region without having to manually create the partition directories.

☞ Dynamic partitioning has some advantages over static partitioning. It is more flexible and can automatically handle new values of the partition column. It also simplifies the ETL process by eliminating the need for manual partition creation.

☞ However, dynamic partitioning can be slower than static partitioning because it involves more overhead to determine the partition values.

☞ By default, dynamic partitioning is disabled in HIVE to prevent accidental partition creations. To use dynamic partitioning in Hive, you need to enable it using the set hive.exec.dynamic.partition.mode configuration property. You can also specify the partition columns using the set hive.exec.dynamic.partition property.

Explain Bucketing in Hive

➞ Hive buckets are a mechanism used in Apache Hive to improve query performance by organizing data in a more efficient way.

➞ At times, even after partitioning on a particular field or fields, the partitioned file size doesn’t match with the actual expectation and remains huge and we want to manage the partition results into different parts. To overcome this problem of partitioning, Hive provides Bucketing concept, which allows user to divide table data sets into more manageable parts.

➞ Hive buckets are essentially a way of dividing data into smaller, more manageable pieces called buckets. Each bucket is a subset of the data that is stored separately on disk, making it easier for Hive to read and process the data. The number of buckets used to partition the data is configurable and can be set to any value that suits the data being processed.

➞ The Bucketing concept is based on Hash function, which depends on the type of the bucketing column. Records which are bucketed by the same column will always be saved in the same bucket.

➞ One of the primary benefits of using Hive buckets is that they allow for more efficient querying of data. When a query is executed, Hive can limit the amount of data it needs to read by only accessing the relevant buckets. This can significantly reduce the amount of time it takes to execute a query, particularly when working with large datasets.

➞ Another advantage of using Hive buckets is that they can be used in conjunction with Hive's partitioning functionality. By combining partitioning with bucketing, developers can create a more fine-grained system for organizing data, which can further improve query performance.

➞ To use Hive buckets, developers need to specify the number of buckets to be used when creating tables in Hive. This can be done using the "CLUSTERED BY" clause in the CREATE TABLE statement. For example, the following statement creates a table with four buckets: CREATE TABLE mytable (id INT, name STRING) CLUSTERED BY (id) INTO 4 BUCKETS;

➞ Once the table is created, data can be inserted into it using the "INSERT INTO" statement. Hive will automatically distribute the data across the specified number of buckets based on the clustering column.

Explain Map Side Joins in Hive?

➞ Map-side join is an optimization technique used in Apache Hive to improve the performance of join operations by minimizing data shuffling between the mappers and reducers.
➞ Map-side join works by loading a small table into memory and using it to perform the join operation with the larger table in the mapper phase.
➞ This reduces the amount of data that needs to be shuffled to the reducers, which can greatly reduce the overall processing time and resource usage.
➞ Map side join is a process where joins between two tables are performed in the Map phase without the involvement of Reduce phase.
➞ Map-side Joins allows a table to get loaded into memory ensuring a very fast join operation, performed entirely within a mapper and that too without having to use both map and reduce phases.
➞ Map-side joins work best when the small table can fit entirely in memory on each node. If the small table is too large to fit in memory, then a traditional reduce-side join may be more appropriate.
➞ Additionally, map-side joins may not always be faster than reduce-side joins, and the optimal approach depends on the specifics of the data and the join operation being performed.

➞ Steps to perform Map-side joins:

👉 By specifying the keyword, /*+ MAPJOIN (b) */ in the join statement.
👉 By setting the following property to true. hive.auto.convert.join = true
👉 For performing Map-side joins, there should be two files, one is of larger size and the other is of smaller size. You can set the small file size by using the following property: hive.mapjoin.smalltable.filesize = (default it will be 25MB)
👉Now, let us perform Map-side joins and join the two datasets based on their IDs. 👉 SYNTAX: SELECT /*+ MAPJOIN(small_table) */ * FROM large_table TABLESAMPLE(1 PERCENT) JOIN small_table ON large_table.id = small_table.id;

👉 The Hive query is modified to enable map-side join using the MAPJOIN hint. The MAPJOIN hint tells Hive to perform a map-side join using the small table. Additionally, the TABLESAMPLE clause is used to randomly sample a portion of the larger table to reduce the amount of data processed.

Explain Bucket-Map Join in Hive?

➞ Bucket-Map Join is a type of Map Side Join in Hive that combines the benefits of bucketing and Map Side Joins.

➞ Bucketing is a method of partitioning data into multiple buckets based on a specific column or set of columns. This ensures that data with the same value for the bucketing column(s) are stored together, allowing for faster access and processing.

➞ Map Side Joins, on the other hand, allow for smaller tables to be loaded into memory and joined with larger tables, reducing the amount of data shuffling and improving performance.

➞ In Bucket-Map Join, both tables involved in the join are bucketed on the join keys. This ensures that the data for each join key value is stored in the same bucket for both tables. Then, the smaller table is loaded into memory and broadcasted to all the mappers in the cluster, while the larger table is read from disk.

➞ During the join operation, each mapper processes a specific portion of the larger table and joins it with the entire smaller table in memory. Since both tables are bucketed on the join keys, the join operation only needs to be performed on the data within each bucket, reducing the amount of data that needs to be processed and improving performance.

➞ For performing Bucket-Map join, we need to set this property in the Hive shell.

```
set hive.optimize.bucketmapjoin = true
```

➞ The syntax for a Bucket-Map Join in Hive is as follows

```
SELECT /*+ MAPJOIN(small_table) */ *

FROM large_table

JOIN small_table

ON large_table.join_key = small_table.join_key

CLUSTER BY large_table.join_key;
```

➞ In the above query, both tables are bucketed on the join_key column, and the CLUSTER BY keyword ensures that the data is stored in the same bucket for both tables. The MAPJOIN hint tells Hive to use a Map Side Join with the small_table and load it into memory.

Explain Sort Merge Bucket (SMB) Map Join in Hive?

➞ Sort Merge Bucket (SMB) Map Join is a type of Map Side Join in Hive that improves the performance of join operations by reducing the amount of data shuffling between mappers and reducers. It is particularly useful for joining large tables where conventional Map Side Joins may not be efficient.

➞ SMB Map Join is based on the idea of bucketing, which involves partitioning data into buckets based on the values of a specific column or set of columns. In SMB Map Join, both tables involved in the join operation are bucketed on the join keys. The buckets are then sorted based on the join keys, which enables the join operation to be performed using a merge-sort algorithm.

➞ For performing the SMB-Map join, we need to set the following properties:

```
set hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;

set hive.optimize.bucketmapjoin = true;

set hive.optimize.bucketmapjoin.sortedmerge = true;
```

➞ To perform the Sort-Merge-Bucket Map join, we need to have two tables with the same number of buckets on the join column and the records are to be sorted on the join column.

➞ The syntax for an SMB Map Join in Hive is as follows:

```
SELECT /*+ MAPJOIN(small_table) */ *

FROM large_table J

OIN small_table

ON large_table.join_key = small_table.join_key

CLUSTER BY large_table.join_key SORT BY large_table.join_key;
```

➞ In the above query, both tables are bucketed on the join_key column and sorted within each bucket. The CLUSTER BY keyword ensures that the data is stored in the same bucket for both tables, and the SORT BY keyword sorts the data within each bucket. The MAPJOIN hint tells Hive to use a Map Side Join with the small_table and load it into memory.

Explain Views in Hive?

➞ Hive views are virtual tables that are created using a SELECT statement in HiveQL.

➞ They allow users to save complex queries as a named view, which can then be queried like a regular table.

➞ Hive views are a powerful tool for data analysts and developers to simplify query complexity, improve query performance, and maintain data security.

➞ Creating a View in Hive To create a view in Hive, users can use the CREATE VIEW statement followed by a SELECT statement that defines the view's columns and data. The syntax is as follows:

CREATE VIEW view_name AS SELECT column1, column2, ... FROM table_name WHERE condition;

➞ Querying a View

Once the view has been created, users can query it as if it were a regular table.

The syntax is as follows:

SELECT * FROM view_name;

➞ Updating a View

Hive views can be updated with a new SELECT statement to change the view's definition.

The syntax is as follows:

ALTER VIEW view_name AS SELECT column1, column2, ... FROM table_name WHERE condition;

➞ Benefits of Using Hive Views

☞ Simplify Query Complexity: Hive views can simplify the complexity of queries by encapsulating complex joins and aggregations into a single view. This makes queries easier to read and maintain.
☞ Improve Query Performance: Hive views can improve query performance by pre-computing complex joins and aggregations, reducing the need for repetitive calculations.

☞ Enhance Data Security: Hive views can be used to restrict access to sensitive data by providing a level of abstraction between the raw data and the users. This allows users to query the view without accessing the underlying data.

☞ Reusability: Hive views can be reused across different queries and applications, which saves time and reduces errors by centralizing the definition of complex queries.

When should we use SORT BY instead of ORDER BY?

➞ In Hive, the SORT BY and ORDER BY clauses are used to sort the results of a query in ascending or descending order based on one or more columns. However, there is a subtle difference between the two clauses that can have a significant impact on the performance of a query.

➞ ORDER BY is used to sort the output of a query based on one or more columns. It sorts the entire dataset, which means that all data must be shuffled and sorted, which can be very expensive in terms of performance and memory usage. For large datasets, ORDER BY can cause performance issues and can even result in out-of-memory errors.

➞ SORT BY, on the other hand, is used to sort the data only within each reducer. It does not sort the entire dataset, but only the data that is processed by each reducer. This means that SORT BY can be much more efficient than ORDER BY in terms of performance and memory usage.

➞ we should use SORT BY whenever possible, especially for large datasets. Here are some scenarios where SORT BY is a better choice than ORDER BY:

☞ Large Datasets: For large datasets, ORDER BY can be very expensive in terms of performance and memory usage. In contrast, SORT BY can be much more efficient, as it only sorts the data processed by each reducer.

☞ Non-Unique Data: If the data contains duplicates or non-unique values, ORDER BY can result in unpredictable results, as it sorts the entire dataset. SORT BY is a better choice in such scenarios, as it sorts the data within each reducer.

☞ Intermediate Sorting: If the data is already sorted or partially sorted, using SORT BY can be more efficient than using ORDER BY. This is because SORT BY can take advantage of the already sorted data and minimize the amount of sorting required.

How can you check if a specific partition exists in Hive?

➞ When working with partitions, it is sometimes necessary to check whether a specific partition exists or not.

➞ To check if a specific partition exists in Hive, we can use the SHOW PARTITIONS command. This command lists all the partitions that exist in a table, and we can use it to check if a specific partition is present.

➞ The syntax of the SHOW PARTITIONS command is as follows:

SHOW PARTITIONS table_name [PARTITION(column_name=value)];

➞ If we specify a partition, only that partition will be displayed. If we do not specify a partition, all the partitions for the table will be displayed. For example, suppose we have a table called sales, partitioned by year, month, and day. To check if the partition for sales in the year 2022, month 5, and day 1 exists, we can use the following command:

SHOW PARTITIONS sales PARTITION(year=2022,month=5,day=1);

➞ If the partition exists, we will see the partition information in the output. If the partition does not exist, we will get an empty result.

➞ Another way to check if a partition exists is to query the partition directory directly. In Hive, each partition is stored as a separate directory under the table directory. We can check if a specific partition directory exists using standard file system commands like ls, ls -l, or hadoop fs -ls. For example, to check if the partition for sales in the year 2022, month 5, and day 1 exists, we can use the following command:

hadoop fs -ls /user/hive/warehouse/sales/year=2022/month=5/day=1

➞ If the partition directory exists, we will see the directory information in the output. If the partition directory does not exist, we will get an error message indicating that the directory does not exist.

Explain Components in Hive Architecture?

➞ The Hive architecture consists of several components that work together to support its functionality:

➞ User Interface (UI) As the name describes User interface provide an interface between user and hive. It enables user to submit queries and other operations to the system. Hive web UI, Hive command line, and Hive HD Insight (In windows server) are supported by the user interface.

➞ Hive Server Hive Server is responsible for handling client requests and managing connections. It provides a Thrift API that allows external applications to access Hive. Hive Server is important for ensuring that Hive queries can be executed from external applications and tools.

➞ Driver Queries of the user after the interface are received by the driver within the Hive. Concept of session handles is implemented by driver. Execution and Fetching of APIs modelled on JDBC/ODBC interfaces is provided by the user.

➞ Query Compiler The Query Compiler is responsible for compiling the HiveQL (Hive Query Language) queries into MapReduce jobs. The Query Compiler consists of two main components: the Parser and the Semantic Analyzer. The Parser is responsible for parsing the HiveQL queries and creating a parse tree. The Semantic Analyzer validates the parse tree and generates the logical plan. The logical plan is then optimized and converted into a physical plan that consists of MapReduce jobs. The Query Compiler is essential for ensuring that Hive queries are optimized for efficient execution.

➞ Metastore The Metastore is the central repository that stores metadata information about the data stored in Hive. It contains information about the database schema, tables, columns, partitions, and their corresponding data types, location, and format. The Metastore uses a relational database (such as MySQL or PostgreSQL) to store this metadata information. The Metastore is critical for ensuring data consistency and reliability.

➞ Execution Engine The Execution Engine is responsible for executing the physical plan generated by the Query Compiler. The Execution Engine consists of two main components: the Job Scheduler and the Task Tracker. The Job Scheduler is responsible for scheduling MapReduce jobs and managing resources. The Task Tracker is responsible for executing individual MapReduce tasks on the cluster. The Execution Engine is crucial for executing Hive queries efficiently and effectively.

Explain Hive Architecture?

The following architecture explains the flow of submission of query into Hive.

➞ Step-1: Execute Query – Interface of the Hive such as Command Line or Web user interface delivers query to the driver to execute. In this, UI calls the execute interface to the driver such as ODBC or JDBC.

➞ Step-2: Get Plan – Driver designs a session handle for the query and transfer the query to the compiler to make execution plan. In other words, driver interacts with the compiler or Request is sent to the driver program (client write SQL like queries) driver takes queries and convert the query into map reduce program with the help of required APIs.

➞ Step-3: Get Metadata – In this, the compiler transfers the metadata request to any database and check the Symantic & Syntactical error then it compiles the code, the compiler gets the necessary metadata from the Metastore.

➞ Step-4: Send Metadata – Metastore transfers metadata as an acknowledgment to the compiler.

➞ Step-5: Send Plan – Compiler communicating with driver with the execution plan made by the compiler to execute the query.

➞ Step-6: Execute Plan – Execute plan is sent to the execution engine by the driver. Execute Job Job Done Dfs operation (Metadata Operation)

➞ Step-7: Fetch Results – Fetching results from the driver to the user interface (UI).

➞ Step-8: Send Results – Result is transferred to the execution engine from the driver. Sending results to Execution engine. When the result is retrieved from data nodes to the execution engine, it returns the result to the driver and to user interface (UI).

Explain Different optimization techniques in Hive?

The amount of data processed by Hive grows, performance can begin to suffer. In this post, we'll explore some techniques for optimizing Hive queries to improve performance.

➞ Partitioning and Bucketing One of the best ways to improve Hive query performance is to partition the data. Partitioning breaks up the data into smaller, more manageable pieces that can be processed separately, reducing the amount of data that needs to be scanned for each query. Bucketing is a similar technique that groups data into buckets based on a hash function applied to one or more columns. This can further improve performance by reducing the number of rows that need to be read for each query.

➞ Use Vectorization Vectorization is a technique that enables Hive to process data in batches instead of one row at a time. This can improve query performance by up to 10x. Vectorization is enabled by default in recent versions of Hive, but it may need to be explicitly enabled in older versions.

➞ Use Cost-Based Optimization Cost-Based Optimization (CBO) is a feature in Hive that uses statistics about the data to optimize queries. It works by estimating the cost of each query plan and selecting the most efficient one. CBO can be enabled by setting the hive.cbo.enable property to true.
➞ Use ORC File Format ORC (Optimized Row Columnar) is a file format that is optimized for Hive queries. ORC files are highly compressed and support predicate pushdown, column pruning, and other optimizations. Converting data to the ORC format can significantly improve query performance.

➞ Reduce the Data Skew Data skew is a condition where some partitions or buckets contain significantly more data than others. This can cause performance problems because the queries need to scan more data than necessary. One way to address data skew is to use bucketing, as discussed earlier. Another approach is to use dynamic partitioning to evenly distribute data across partitions.
➞ Tune the Hive Configuration Hive's performance can be greatly affected by its configuration settings. By tuning the configuration, users can improve query performance. Some key settings to consider include the size of the memory heap, the number of map and reduce tasks, and the size of the input buffer.

How to Implement SCD1 in Hive?

Here is a step-by-step guide on how to implement Slowly Changing Dimension Type 1 (SCD1) in Hive:
➞ Create a staging table in Hive to hold the incoming data. For example, let's say you have a table customer with columns id, name, and address, and you want to apply SCD1 on the address column. You can create a staging table as follows:

```
CREATE TABLE customer_stg ( id INT, name STRING, address STRING );
```

➞ Load the incoming data into the staging table. You can use Hive's LOAD DATA command to load data from a file, or INSERT INTO to insert data from another table.

➞ Create a target table in Hive to hold the final result. This table should have the same structure as the staging table.

```
CREATE TABLE customer_tgt ( id INT, name STRING, address STRING );
```

➞ Insert the data from the staging table into the target table. This will overwrite any existing data in the target table.

```
INSERT OVERWRITE TABLE customer_tgt

SELECT id, name, address FROM customer_stg;
```

➞ Add a timestamp column to the target table to track when each record was last updated. You can use the CURRENT_TIMESTAMP function to get the current timestamp.

```
ALTER TABLE customer_tgt ADD COLUMN last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP; ➞
```

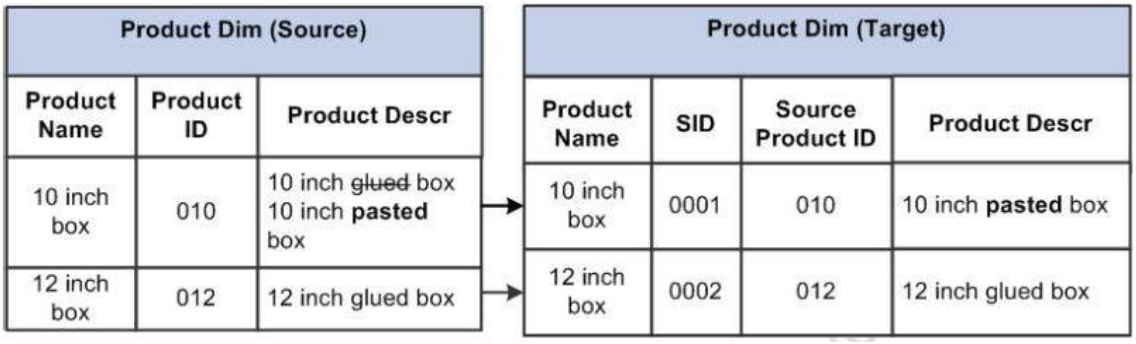
Whenever new data arrives, update the target table with the new values using a MERGE statement. The MERGE statement compares the incoming data with the existing data in the target table and updates the rows that have changed. In this example, we want to update the address column whenever it changes.

```
MERGE INTO customer_tgt tgt

USING customer_stg stg

 ON tgt.id = stg.id

WHEN MATCHED AND tgt.address <> stg.address THEN

UPDATE SET

tgt.address = stg.address,

tgt.last_updated = CURRENT_TIMESTAMP;
```

The MERGE statement joins the customer_tgt and customer_stg tables on the id column. For each matching row, it checks if the address column has changed. If it has, it updates the address column and the last_updated column with the current timestamp.

➞ That's it! You've now implemented SCD1 in Hive using a staging table, a target table, and a MERGE statement.

Type 1 Slowly Changing Dimension



Product Dim (Source)		
Product Name	Product ID	Product Descr
10 inch box	010	10 inch ~~glued~~ box 10 inch **pasted** box
12 inch box	012	12 inch glued box

Product Dim (Target)			
Product Name	SID	Source Product ID	Product Descr
10 inch box	0001	010	10 inch **pasted** box
12 inch box	0002	012	12 inch glued box

Explain the difference between Hive and traditional RDBMS?

Here are some of the key differences between Hive and traditional RDBMS:

➞ Architecture: Hive is built on top of the Hadoop ecosystem and runs on distributed clusters of commodity hardware. Traditional RDBMS systems typically run on a single server or a small cluster of servers.

➞ Data model: Hive supports a schema-on-read data model, which means that the data schema is not defined until the data is accessed. This allows for more flexibility and agility in handling unstructured or semi-structured data. Traditional RDBMS systems use a schema-on-write model, where the schema is defined before the data is inserted.

➞ Language: Hive uses a SQL-like language called HiveQL, which supports many of the same features as SQL but also includes extensions for working with unstructured data and Hadoop-specific features. Traditional RDBMS systems use SQL as their primary query language.

➞ Performance: Hive is designed to handle large-scale data processing and is optimized for batch processing rather than real-time queries. Traditional RDBMS systems are typically optimized for transactional processing and real-time queries.

➞ Security: Hive integrates with Hadoop's security features, including Kerberos authentication and encryption. Traditional RDBMS systems typically have their own built-in security features.

↪ In summary, Hive is designed for large-scale data processing on distributed clusters using a SQL-like language, while traditional RDBMS systems are optimized for transactional processing on single servers or small clusters using SQL. Hive is more flexible in handling unstructured data and can handle larger volumes of data, while traditional RDBMS systems are more suited for real-time queries and transactions.

Describe the process of data serialization and deserialization in Hive?

Data serialization and deserialization play a crucial role in the data processing workflow of Hive. Serialization is the process of converting structured data into a binary or textual format that can be easily stored or transmitted, while deserialization is the reverse process of converting the serialized data back into its original form. In Hive, data serialization and deserialization are handled by SerDe (Serializer/Deserializer) libraries, which provide the necessary functionality to convert data between Hive's internal representation and external formats.

➞ Here is an overview of the process of data serialization and deserialization in Hive:

➞ SerDe Libraries: Hive supports various SerDe libraries that enable serialization and deserialization for different data formats. Examples include JSON SerDe, Avro SerDe, ORC SerDe, and Parquet SerDe. Each SerDe library is designed to handle a specific data format and provides the necessary logic to serialize and deserialize data in that format.

➞ Table Definition: In Hive, data is organized into tables. When creating a table, you specify the SerDe library to be used for serialization and deserialization. This is done by specifying the ROW FORMAT clause in the CREATE TABLE statement, along with the corresponding SerDe library and any additional configuration parameters.

➞ Serialization: When data is inserted into a table or queried in Hive, the serialization process takes place. Hive reads the input data and uses the specified SerDe library to serialize the data into a binary or textual format. This format is optimized for efficient storage and processing within Hive.

➞ Storage Format: The serialized data is stored in a specific storage format, such as text files, ORC (Optimized Row Columnar) files, or Parquet files. Each storage format has its own advantages in terms of compression, columnar storage, and query performance. The choice of storage format depends on the specific requirements of the data and the desired query performance.

➞ Deserialization: When retrieving data from a table or executing a query, Hive performs the deserialization process. It reads the serialized data from the storage format and uses the SerDe library to deserialize the data back into its original form. This allows Hive to work with the data in a structured and queryable format.

How does Hive handle data skewness?

Data skewness refers to an uneven distribution of data across partitions or buckets in a Hive table, which can lead to performance issues and inefficiencies in query processing. Hive provides several mechanisms to handle data skewness and optimize query execution in such scenarios.

➞ Dynamic Partitioning: Hive supports dynamic partitioning, which allows data to be partitioned based on specific column values during the data loading process. By partitioning the data dynamically, Hive can distribute the data evenly across multiple partitions, reducing the impact of data skewness.

➞ Bucketing: Bucketing is another technique provided by Hive to handle data skewness. It involves dividing the data into equal-sized buckets based on a hash function applied to a specific column. By bucketing the data, Hive distributes the records evenly across buckets, ensuring a more balanced distribution and efficient query processing.

➞ Sampling: Hive provides the ability to sample data from a table, allowing users to analyze a representative subset of the data. Sampling can help identify data skewness by revealing the distribution of values within a column. With this information, users can make informed decisions on how to handle the skewness, such as adjusting partitioning or bucketing strategies.

➞ Skewed Join Optimization: Hive offers a Skewed Join Optimization feature to handle data skewness during join operations. This optimization technique detects skewed keys and performs special handling for them. It involves creating separate files for skewed keys and optimizing the join process by using different techniques, such as broadcasting the smaller side of the join or using map-side joins.

➞ Custom Scripts and Techniques: Hive provides the flexibility to implement custom scripts and techniques to handle data skewness. Users can leverage user-defined functions (UDFs) or custom scripts to identify and address data skewness specific to their datasets.

➞ It's important to note that the effectiveness of these techniques depends on the specific characteristics of the data and the nature of the queries being executed. Analyzing the data distribution, understanding the query patterns, and experimenting with different optimization strategies are crucial for effectively handling data skewness in Hive.

What are the limitations of Hive?

While Hive offers many advantages, it also has certain limitations that users should be aware of. In this post, we will explore the limitations of Hive:

➞ High Latency: Hive is designed for batch processing and is optimized for handling large-scale data processing tasks. However, this batch processing nature introduces high latency, making it less suitable for real-time or interactive data analysis scenarios. Hive's query response time may not be suitable for applications that require near real-time or low-latency data processing.

➞ Limited Support for Updates and Deletes: Hive primarily focuses on read-heavy workloads and is designed for data warehousing and analytics. As a result, it has limited support for data updates and deletes. Hive is optimized for data ingestion and batch processing, making it challenging to perform frequent updates or deletes on individual records.

➞ Lack of Full ACID Compliance: Hive does not provide full ACID (Atomicity, Consistency, Isolation, Durability) compliance like traditional relational databases. While Hive supports ACID operations within a partition, it does not support cross-partition ACID transactions. This limitation makes it challenging to maintain strong consistency across distributed transactions in Hive.

➞ Limited Indexing Support: Hive relies on the use of partitions and bucketing for efficient data processing. While these techniques can improve query performance, Hive has limited support for secondary indexes. This means that querying on non-partitioned or non-bucketed columns may result in slower query execution times.

➞ Lack of Real-Time Streaming Support: Hive is not well-suited for real-time streaming data processing. It is primarily designed for batch processing of static data sets. While Hive can integrate with streaming frameworks like Apache Kafka, it may not provide the same level of real-time processing capabilities as specialized streaming platforms.

➞ Steep Learning Curve: Hive uses a SQL-like language called HiveQL for querying data, but it also introduces additional concepts and syntax specific to Hive. Users with a background in traditional SQL databases may need to familiarize themselves with Hive's unique features and optimizations, which can result in a steeper learning curve.

How can you control the number of reducers in Hive?

In Hive, the number of reducers determines the degree of parallelism during data processing and influences the performance of query execution. By default, Hive automatically determines the number of reducers based on the size of the input data. However, there are scenarios where you may want to control the number of reducers manually to optimize query performance or adjust resource utilization.

➞ Setting the Number of Reducers in the Query: Hive provides the flexibility to specify the number of reducers directly in the query using the SET command. You can use the mapreduce.job.reduces configuration property to set the desired number of reducers for a specific query. For example, SET mapreduce.job.reduces=10; will enforce 10 reducers for the query execution.

➞ Using the CLUSTER BY or DISTRIBUTE BY Clauses: Hive's CLUSTER BY or DISTRIBUTE BY clauses allow you to control the number of reducers indirectly. These clauses define the columns by which the data should be partitioned or distributed, influencing the number of reducers based on the number of distinct values in the specified columns. By choosing appropriate columns, you can indirectly control the number of reducers.

➞ Setting Hive Configuration Properties: Hive provides various configuration properties related to reducers that can be adjusted to control their behavior. Properties such as hive.exec.reducers.bytes.per.reducer and hive.exec.reducers.max.tasks.per.node allow you to fine-tune the number of reducers based on data size and available resources.

➞ Considering Hardware and Cluster Capacity: The number of reducers should align with the hardware capabilities and cluster capacity. If the cluster has limited resources or the hardware is not sufficient to handle a large number of reducers simultaneously, it's important to consider these factors while controlling the number of reducers. Adjusting the number based on the available resources can prevent resource contention and optimize overall cluster performance.

What is the difference between a local mode and a MapReduce mode in Hive?

➞ Hive, a data warehousing and analytics system built on top of Hadoop, provides two execution modes: local mode and MapReduce mode. These modes differ in how Hive processes and executes queries.

➞ Local Mode:

☞ In local mode, Hive executes queries on a single machine using the local file system. It is primarily used for development, testing, and small-scale data processing.

☞ Local mode is convenient for running queries on small datasets without the need for a Hadoop cluster. It allows developers to quickly test queries and analyze small data samples.

☞ In local mode, Hive does not leverage the parallel processing capabilities of Hadoop. Instead, it operates on a single machine, utilizing the available resources like CPU and memory.

☞ Local mode is not suitable for large-scale data processing or handling massive datasets as it lacks the distributed computing power of a Hadoop cluster.

➞ MapReduce Mode:

☞ MapReduce mode is the default and most commonly used execution mode in Hive. It leverages the power of the Hadoop Distributed File System (HDFS) and the MapReduce framework for distributed data processing. ☞

In MapReduce mode, Hive translates queries written in HiveQL into MapReduce jobs, which are executed in parallel across the nodes of a Hadoop cluster.

☞ MapReduce mode provides scalability and fault tolerance, making it suitable for processing large volumes of data across a cluster of machines. It allows for distributed storage, processing, and fault recovery, which are key advantages of Hadoop.

☞ With MapReduce mode, Hive takes advantage of Hadoop's distributed computing capabilities, enabling parallel execution of queries across multiple nodes. This parallelism improves query performance and reduces the time required for data processing.

➞ In summary, the key difference between local mode and MapReduce mode in Hive lies in their execution environment and scalability. Local mode operates on a single machine and is suitable for small-scale data processing, while MapReduce mode leverages a Hadoop cluster for distributed processing of large datasets. Local mode is useful for development and testing, while MapReduce mode is the preferred choice for production deployments where scalability and distributed computing power are essential.

Explain the concept of input/output formats in Hive?

➞ What are Input/Output Formats?

Input/Output formats in Hive define the structure and layout of data files when reading or writing data. They determine how data is organized, encoded, and serialized, allowing Hive to interact with different data sources efficiently. Hive supports multiple input/output formats, providing flexibility to work with a wide range of data formats and storage systems.

➞ Built-in Input/Output Formats: Hive comes with a set of built-in input/output formats, including:

☞ TextFile: This is the default input/output format in Hive, which treats data as plain text files. Each line in the file is treated as a separate record, making it suitable for processing structured and unstructured textual data.

☞ SequenceFile: SequenceFile is a binary file format in which data is stored as key-value pairs. It provides high-performance storage and compression, making it ideal for processing large datasets efficiently.

☞ ORC (Optimized Row Columnar): ORC is a columnar storage format in Hive, designed for high-performance data processing. It offers improved compression and predicate pushdown capabilities, reducing the amount of data to be read during query execution.

☞ Parquet: Parquet is another columnar storage format that provides efficient data compression and column pruning. It is widely used in Hive for its compatibility with various processing frameworks, including Apache Spark and Apache Impala.

➞ Custom Input/Output Formats: In addition to the built-in formats, Hive allows users to define and use custom input/output formats tailored to their specific needs. This flexibility enables integration with external systems, proprietary data formats, or specialized storage solutions.

Explain schema evolution in Hive?

➞ In the world of big data analytics, dealing with evolving data structures is a common challenge. Hive, a powerful data warehousing and query tool, offers mechanisms to handle schema evolution effectively.

➞ Understanding Schema Evolution: Schema evolution refers to the changes that occur in the structure of data over different stages of its lifecycle. These changes can include adding or removing columns, modifying data types, or altering the overall schema design. Handling schema evolution is critical for ensuring data compatibility and enabling seamless data processing and analysis.

➞ Best Practices for Handling Schema Evolution in Hive: To effectively handle schema evolution in Hive, consider the following best practices:

☞ Define a Versioning Strategy: Implement a versioning strategy for your data to keep track of schema changes over time. This can help ensure data lineage and facilitate the mapping of different schema versions to queries.

☞ Use Structured Data Formats: Leverage structured data formats like ORC or Parquet to enforce stricter schema validation and optimize storage and query performance.

☞ Handle Compatibility: Implement data transformation mechanisms to handle compatibility between different schema versions. Use tools like Hive's SerDe (Serializer/Deserializer) to handle schema evolution gracefully.

☞ Document Schema Changes: Maintain proper documentation of schema changes, including the reasons behind the modifications and the impact on data processing and analytics.

How can you handle schema evolution in Hive?

➞ Approaches to Schema Evolution in Hive: Hive provides two main approaches to manage schema evolution: schema-on-read and schema-on-write. Let's explore each of these approaches in detail.

Schema-on-Read:

In Hive's schema-on-read approach, the schema is applied dynamically during the reading phase. It allows Hive to handle evolving schemas without requiring explicit schema changes in the underlying data. Here's how schema-on-read works:

☞ Lazy Evaluation: Hive reads data in a lazy manner, allowing it to adapt to changes in schema during runtime. When executing queries, Hive infers the schema from the data itself, using the provided metadata or schema information.

☞ Flexibility: Schema-on-read provides flexibility in handling different versions of data with varying schemas. It allows querying data without the need for upfront schema definitions, making it suitable for scenarios where data structures may change frequently.

☞ Data Compatibility: Hive's schema-on-read approach enables reading and processing of data with different schema versions, as long as the necessary transformations or mappings are applied during query execution.

Schema-on-Write: While Hive primarily follows the schema-on-read approach, it also supports schema-on-write through the use of structured data formats like ORC (Optimized Row Columnar) and Parquet. These columnar storage formats allow for more structured and optimized data storage, which can improve query performance. Here's how schema-on-write works:

☞ Strict Data Validation: When writing data in Hive, structured data formats enforce stricter schema validation rules. The data is serialized and stored in a columnar format, ensuring adherence to a predefined schema.

☞ Schema Evolution Control: Schema-on-write provides more control over schema changes during data ingestion. It requires explicit schema modifications when adding or modifying columns, ensuring that the new data conforms to the updated schema.

☞ Enhanced Performance: By enforcing a fixed schema during write operations, Hive can optimize data storage and retrieval, resulting in improved query performance and reduced data processing overhead.

How to Load Files with Multiple Delimiters in Hive?

➞ Loading data into Hive is a common task in big data processing. However, sometimes the data files may have multiple delimiters, which can pose a challenge when defining the table schema and correctly parsing the data. In this post, we will explore how to load files with multiple delimiters in Hive and handle the data effectively.

➞ Step 1: Understand the Data Structure:

☞ Before loading the file into Hive, it's crucial to understand the structure of the data and identify the delimiters used. Take a closer look at the file and note down all the delimiters present.

☞ Input_file:

linkedin,2900:India|Country

Instagram,400:India|Country

Others,244:India|Country

➞ Step 2: Create the Hive Table:

☞ To load the data, we need to define a Hive table that matches the structure of the file. Determine the appropriate column types and order based on your data. For example, if your file has three columns, the table creation statement would look like this:

☞ Hive provides the RegexSerDe #serde (Serializer/Deserializer), which allows us to specify a regular expression pattern to parse the data. We can leverage this to handle multiple delimiters.

```
CREATE EXTERNAL TABLE test (

column1 STRING,

 column2 STRING,

 column3 STRING,

column4 STRING

)

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'

WITH SERDEPROPERTIES ( 'input.regex'='^(.*?),(.*?):(.*?)\\|(.*?)$' )

LOCATION 'your_input_file';
```

☞ In the above example, we assume the delimiters are a comma (,), colon (:), pipe (|), and any other characters are part of the values.

Step 3: Verify the Results:

☞ To verify that the data has been loaded correctly, run a select query on the table:

```
SELECT * FROM your_table;
```

☞ Output:

linkedin 2900 India Country

Instagram 400 India Country

Others 244 India Country

➞ Conclusion:

☞ Loading files with multiple delimiters in Hive can be achieved by leveraging the #RegexSerDe serde and defining the appropriate regular expression pattern. By following the steps outlined in this post, you can successfully load and parse data files with multiple delimiters in Hive, enabling efficient data processing and analysis.

☞ Remember, understanding the structure of your data and choosing the correct delimiters are essential for accurately defining the table schema and successfully loading the data into Hive.

Meaning of keywords:

PARTITIONED BY: Specifies partition columns for the table.

CLUSTERED BY: Specifies bucketing columns and the number of buckets.

ROW FORMAT: Defines how data rows are formatted.

STORED AS: Specifies the storage format of the table (e.g., TEXTFILE, ORC,PARQUET).

LOCATION: Specifies the HDFS path where external table data resides.

TBLPROPERTIES: Allows setting additional table properties like serde properties,

table comments, etc.

Hive with cloud:

Amazon Web Services (AWS):

Amazon EMR (Elastic MapReduce): Amazon EMR supports Hive out-of-the-box,allowing users to deploy Hive clusters on AWS infrastructure quickly. EMR providesmanaged Hadoop clusters, simplifying the setup and management of Hive deployments.

Amazon S3 (Simple Storage Service): Hive can directly interact with data stored in Amazon S3, allowing users to query and analyze data without needing to provision and manage HDFS clusters. This provides flexibility and cost-effectiveness in storing and processing data on AWS.

AWS Glue: AWS Glue provides a fully managed extract, transform, and load (ETL) service that integrates with Hive metastore, allowing users to define, orchestrate, and monitor data pipelines for data preparation and transformation tasks.

Microsoft Azure:

Azure HDInsight: Azure HDInsight offers managed Hadoop clusters that supportApache Hive. Users can deploy Hive clusters on Azure and leverage Azure Blob Storage as the underlying storage for data processing. HDInsight provides integration with other Azure services for seamless data workflows.

Azure Data Lake Storage: Hive can interact with data stored in Azure Data Lake Storage (ADLS), enabling users to run Hive queries on data stored in ADLS without needing to move or replicate data to other storage systems.

Google Cloud Platform (GCP):

Google Dataproc: Google Dataproc provides managed Apache Hadoop and Apache Spark clusters on GCP. Users can deploy Hive clusters on Dataproc and leverage Google

Cloud Storage (GCS): as the underlying storage for data processing.

BigQuery: While not a direct integration with Hive, Google BigQuery offers a serverless, highly scalable data warehouse solution. Users can load data from Hive or other sources into BigQuery for analysis using SQL queries, providing an alternative to traditional Hive deployments on GCP.

Hive also supports generic JDBC and ODBC connectors, allowing users to connect to external data sources and cloud-based databases for data querying and analysis.

Complex Data-types in Hive

Array

```
create table mobilephones ( id string,

title string,

cost float,

colors array<string>,

screen_size array<float>

);

insert into table mobilephones

select "redminote7", "Redmi Note 7", 300, array ("white", "silver", "black"),array(float(4.5))

UNION ALL

select "motoGplus", "Moto G Plus", 200, array ("black", "gold"),

array (float (4.5), float (5.5));

select id,color[0] from mobilephones;

create table mobilephones_new (

id string,

title string,

cost float,

colors array<string>,

screen_size array<float>

)

row format delimited

fields terminated by ','

collection items terminated by '#';

load data local inpath '/home/cloudera/Desktop/shared1/mobilephones.csv' into table
mobilephones_new;
```

Map

```
create table mobilephones (
id string,
title string,
cost float,
colors array<string>,
screen_size array<float>,
features map<string, boolean>
)
row format delimited
fields terminated by ','
collection items terminated by '#'
map keys terminated by ':';
```

Example of Map –

```
features map<string, boolean>
String Boolean
Key Value
Camera True
Dualsim False
load data local inpath '/home/cloudera/Desktop/shared1/mobilephones.csv' into table mobilephones;
```

Struct

```
create table mobilephones (
id string,
title string,
cost float,
colors array<string>,
screen_size array<float>,
features map<string, boolean>,
information struct<battery: string, camera:string>
)
row format delimited
fields terminated by ','
collection items terminated by '#'
map keys terminated by ':';
load data local inpath '/home/cloudera/Desktop/shared1/mobilephones.csv' into table mobilephones;
select id,features['camera'],information.battery from mobilephones;
```