**1. What is the definition of Hive? What is the present version of Hive?**

Hive is an open-source data warehouse system built on top of Hadoop. It provides a SQL-like interface to query data stored in Hadoop Distributed File System (HDFS) or other compatible data stores, such as Apache HBase.

Hive allows users to define a schema for their data using a language called HiveQL, which is similar to SQL. HiveQL statements are translated into MapReduce or Apache Tez jobs that run on a Hadoop cluster, making it possible to analyse large amounts of data quickly and efficiently.

Current Version:

**2. Is Hive suitable to be used for OLTP systems? Why?**

**No, not suitable.**

1. Batch processing: Hive is designed for batch processing and handling large volumes of data, whereas OLTP systems require real-time processing for frequent and short transactions.
2. Latency: Hive has a higher query latency. Queries in Hive are translated into a series of MapReduce jobs, which have inherent overhead and latency. Making it inefficient for OLTP systems that need fast response times for individual transactions.
3. Concurrency: Hive's concurrency model is limited because it is based on MapReduce, which is not designed for multiple concurrent users running interactive queries, which is essential for OLTP systems with multiple users performing transactions simultaneously.
4. Data updates: Hive was originally designed for read-heavy analytical workloads rather than frequent data modifications. Its underlying storage system, HDFS, is optimized for write-once, read-many operations. Frequent updates or deletions can lead to a fragmented HDFS and affect performance. While OLTP systems require the ability to frequently insert, update, and delete records.
5. ACID properties: Hive has limited support for ACID (Atomicity, Consistency, Isolation, and Durability) properties, which are crucial for maintaining data integrity in OLTP systems. The reason for limited ACID support is again tied to Hive's reliance on the Hadoop ecosystem, which was not initially designed for transactional consistency and isolation guarantees.

**3. How is HIVE different from RDBMS? Does hive support ACID transactions. If not then give the proper reason.**

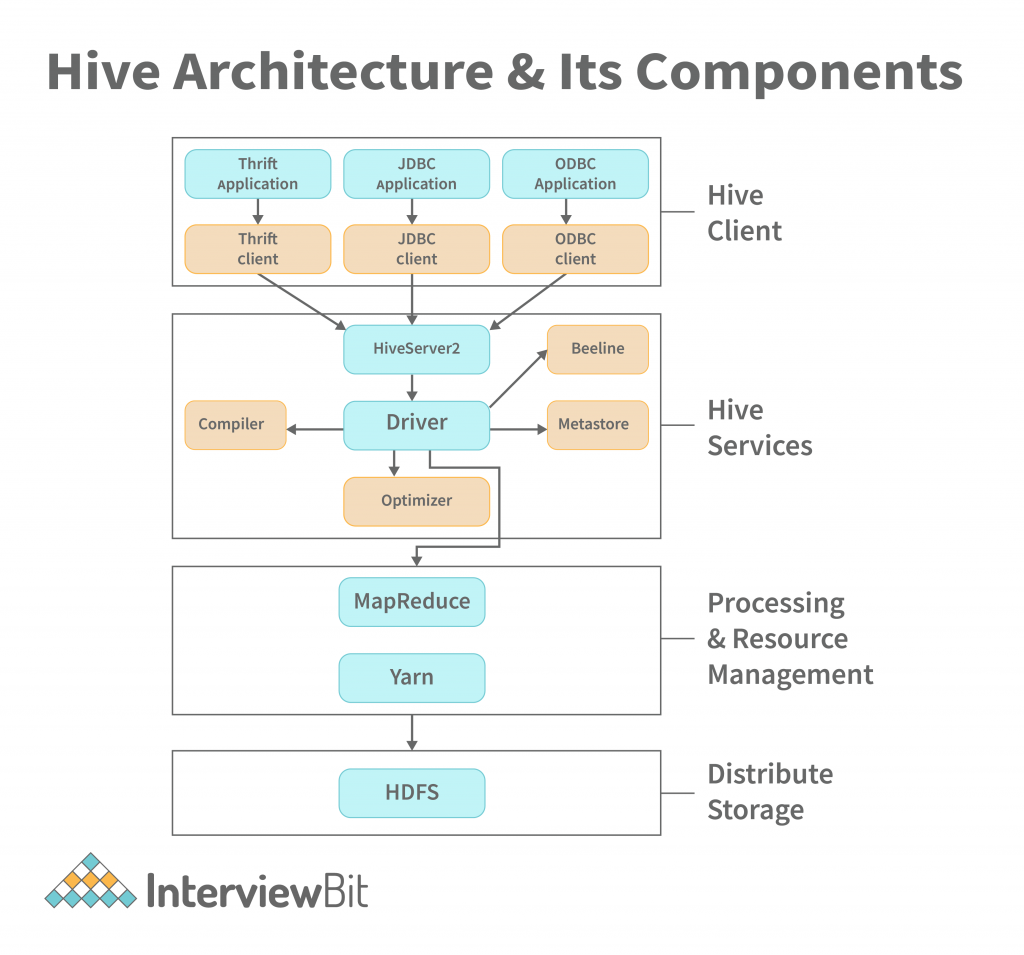
| **Aspect** | **Hive** | **RDBMS** |
| --- | --- | --- |
| Purpose | Designed for batch processing and OLAP workloads. | Designed for OLTP workloads and real-time processing. |
| Query Latency | High query latency due to MapReduce-based execution. | Low query latency, suitable for real-time processing. |
| Data Storage | Hadoop Distributed File System (HDFS) or compatible storage. | Proprietary storage systems, optimized for transactions. |
| Concurrency | Limited support for concurrent users. | High support for concurrent users and transactions. |
| Data Updates | Not optimized for frequent data updates. | Optimized for frequent data updates, inserts, and deletions. |
| ACID Transactions | Limited support for ACID transactions. | Full support for ACID transactions, ensuring data integrity. |

Regarding ACID transactions in Hive, starting with version 0.14, Hive introduced support for ACID transactions. However, this support is limited compared to traditional RDBMS systems. The reasons for this limitation include:

1. Hadoop Ecosystem: Hive relies on the Hadoop ecosystem, which was originally designed for batch processing and not optimized for transactional consistency and isolation guarantees.
2. Storage Model: Hive's underlying storage system, HDFS, is optimized for write-once, read-many operations, which is not well-suited for frequent updates and deletions required by ACID transactions.
3. Concurrency: Hive's concurrency model is limited due to its reliance on MapReduce, which is not designed for multiple concurrent users running interactive queries, affecting transactional isolation.

**4. Explain the hive architecture and the different components of a Hive architecture?**

1. **Hive Client:** Hive Client refers to the applications, tools, or interfaces that allow users to interact with Hive and submit queries. Hive clients can be categorized as follows:
   1. Command Line Interface (CLI): The Hive CLI is a shell prompt that allows users to submit HiveQL queries directly using command line commands. It is a popular way to interact with Hive, especially for developers and administrators.
   2. Web User Interface: The Web UI, like Hue (Hadoop User Experience), provides a graphical interface for users to interact with Hive. Users can submit queries, view results, and manage tables through a web browser, making it more accessible for those who prefer a visual interface.
   3. APIs (Application Programming Interfaces): APIs such as JDBC (Java Database Connectivity) and ODBC (Open Database Connectivity) allow external applications to connect to Hive and interact with it programmatically. This enables developers to integrate Hive queries into their applications or use third-party tools like BI (Business Intelligence) software to analyze data stored in Hive.
2. **Metastore**: The Metastore is a central repository that stores metadata about the structure of the data stored in Hive. It includes information such as the schema of tables, their partitioning schemes, and the location of the data in HDFS.
3. **Driver**: The driver is responsible for receiving queries from the user interface, creating execution plans, and coordinating the execution of queries across other components.
4. **Compiler**: The Hive compiler translates the user's SQL-like queries into a series of MapReduce or Tez jobs that can be executed on a Hadoop cluster. It also optimizes the queries for better performance.
5. **Execution Engine:** The Hive execution engine is responsible for executing the MapReduce or Tez jobs generated by the compiler. It also manages the distributed processing of the data across the Hadoop cluster.
6. **Hadoop Distributed File System (HDFS):** Hive relies on HDFS to store and manage the large datasets that it processes.
7. **User Interface:** Hive provides several user interfaces for users to interact with the system. The most commonly used interface is the Hive CLI, which allows users to submit queries and manage Hive resources.
8. **Query Language:** Hive provides a SQL-like query language called HiveQL, which allows users to write queries that are translated into MapReduce or Tez jobs.
9. **SerDe:** SerDe (Serializer/Deserializer) is a component that helps Hive to read and write data in different file formats, such as CSV, JSON, and Avro.



**5. Mention what Hive query processor does? And Mention what are the components of a Hive query processor?**

The Hive query processor is responsible for translating HiveQL queries into a series of tasks that can be executed on the Hadoop cluster and returning the results to the user. The main components of a Hive query processor are:

1. Parser: The parser checks the syntax of the submitted HiveQL query and converts it into an Abstract Syntax Tree (AST), which is an internal representation of the query.
2. Semantic Analyzer: The semantic analyzer processes the AST to verify if the query is semantically correct. It checks for valid table and column names, performs type checking, and resolves functions and operators used in the query.
3. Logical Plan Generator: This component generates a logical plan from the semantically analyzed AST. The logical plan represents the sequence of operations needed to execute the query, such as filtering, aggregation, and joining.
4. Optimizer: The optimizer applies various optimization techniques to the logical plan to improve query performance. Some optimizations include predicate pushdown, join reordering, and column pruning.
5. Physical Plan Generator: The physical plan generator converts the optimized logical plan into a series of MapReduce jobs or Tez tasks that can be executed on the Hadoop cluster.
6. Execution Engine: The execution engine runs the generated MapReduce jobs or Tez tasks on the Hadoop cluster, processes the data, and returns the results to the user.

In summary, the Hive query processor takes a HiveQL query, checks its syntax and semantics, generates a logical plan, optimizes it, converts it into a physical plan, and finally executes it on the Hadoop cluster using the execution engine.

**Semantic Analyzer**

**Parser**

**Hive Driver**

**Semantic Analyzer**

1. The Hive Driver receives the query and passes it to the Parser.
2. The Parser checks the query syntax and generates an Abstract Syntax Tree (AST).
3. The Semantic Analyzer validates the query semantics and enriches the AST with metadata from the Metastore.
4. The Logical Plan Generator creates a logical plan based on the analyzed AST.
5. The Optimizer applies optimizations to the logical plan for better performance.
6. The Physical Plan Generator translates the optimized logical plan into a series of MapReduce jobs or Tez tasks.
7. The Execution Engine runs the generated tasks on the Hadoop cluster.

**Metastore**

**Logical Plan Generator**

**Physical Plan Generator**

**Optimiser**

**Execution Engine**

**Hadoop Cluster**

**6. What are the three different modes in which we can operate Hive?**

Hive can operate in three different modes:

1. Local Mode: In this mode, Hive runs on a single machine, using the local file system for storage. It is useful for small datasets and testing purposes. The execution engine, by default, is MapReduce, and all tasks run on the same machine as the Hive instance. Local mode is not suitable for processing large datasets due to limited resources and the absence of parallelism.
2. MapReduce Mode: This is the default mode for Hive, where it runs on a Hadoop cluster using the Hadoop Distributed File System (HDFS) for storage. In this mode, Hive translates queries into a series of MapReduce jobs that run on the Hadoop cluster. This mode is suitable for processing large datasets as it leverages the distributed and parallel processing capabilities of Hadoop.
3. Tez Mode: In Tez mode, Hive uses the Apache Tez execution engine instead of MapReduce. Tez is designed to improve the performance of Hive queries by reducing latency and overhead. It allows for more complex and optimized query execution plans, including the ability to process multiple stages of a query in parallel. To use Tez mode, we need to have Apache Tez installed on Hadoop cluster and set the execution engine to Tez in the Hive configuration.

In summary, Hive can operate in Local mode for small datasets and testing, MapReduce mode for distributed processing on a Hadoop cluster, and Tez mode for improved performance and optimized query execution plans.

**7. Features and Limitations of Hive.**

**Features of Hive:**

1. SQL-like language: Hive uses HiveQL, and SQL-like query language, making it easier for those familiar with SQL to work with big data in Hadoop.
2. Schema-on-read: Hive applies schema on read, which means that the schema is applied when the data is queried, allowing for more flexibility in storing and managing data.
3. Scalability: Hive can handle large datasets and scale horizontally across multiple nodes in a Hadoop cluster.
4. Extensibility: Hive supports custom User-Defined Functions (UDFs), SerDes (Serializers and Deserializers), and Storage Handlers, allowing developers to extend Hive's capabilities to fit specific needs.
5. Integration: Hive integrates well with the Hadoop ecosystem and supports various storage systems, including HDFS, Amazon S3, and HBase.

**Limitations of Hive:**

1. High query latency: Hive uses MapReduce (or Tez) for query execution, which can result in high query latency due to the overhead of spinning up tasks on a Hadoop cluster.
2. Limited support for real-time queries: Hive is not suitable for real-time or low-latency queries, as it is designed for batch processing and analytical workloads.
3. Limited support for transactions: While Hive has introduced ACID transaction support, it is not as robust as traditional RDBMS systems in handling complex transactional workloads.
4. Limited concurrency: Hive has limited support for concurrent users and queries due to its reliance on MapReduce or Tez for query execution.
5. Not optimized for frequent updates: Hive is designed for write-once, read-many operations, making it less suitable for workloads that require frequent updates, inserts, or deletions.

**8. How to create a Database in HIVE?**

**HQL : CREATE DATABASE IF NOT EXISTS my\_database;**

**9. How to create a table in HIVE?**

CREATE TABLE IF NOT EXISTS my\_database.my\_table

(

id INT,

name STRING,

age INT

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

STORED AS TEXTFILE;

**10.What do we mean by describe and describe extended and describe formatted with respect to database and table**

**DESCRIBE:** This command displays basic metadata information about the table schema, such as column names, data types, and comments.

Usage: DESCRIBE [database\_name.]table\_name;

**DESCRIBE EXTENDED:** This command displays extended metadata information about the table or database, including basic schema information, table properties, and additional metadata such as the serialization and deserialization libraries.

Usage: DESCRIBE EXTENDED [database\_name.]table\_name;

**DESCRIBE FORMATTED:** This command provides a more human-readable format for displaying extended metadata information about a table or database. The output is similar to DESCRIBE EXTENDED, but it is organized in a structured and formatted manner.

Usage: DESCRIBE FORMATTED [database\_name.]table\_name;

In summary:

* DESCRIBE shows basic schema information of a table.
* DESCRIBE EXTENDED provides additional metadata about a table or database.
* DESCRIBE FORMATTED displays the extended metadata in a more structured and human-readable format.

**11.How to skip header rows from a table in Hive?**

CREATE TABLE my\_database.my\_table

(

id INT,

name STRING,

age INT

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

STORED AS TEXTFILE

**TBLPROPERTIES ('skip.header.line.count'='1');**

**12.What is a hive operator? What are the different types of hive operators?**

A Hive operator is a symbol or a keyword in HiveQL that performs an action or operation on one or more data values, columns, or expressions. They are used to filter, transform, and manipulate data within Hive queries.

Different types of Hive operators include:

1. **Arithmetic Operators:** These operators perform basic arithmetic operations such as addition, subtraction, multiplication, and division.

Examples: +, -, \*, /, %

1. **Comparison Operators:** These operators compare two values and return a boolean result (true or false). They are often used in the WHERE and HAVING clauses for filtering data.

Examples: =, <> or !=, <, >, <=, >=

1. **Logical Operators:** These operators are used to combine two or more conditions in a query. They return a boolean result (true or false).

Examples: AND, OR, NOT

1. **Relational Operators:** These operators are used to manipulate and transform data in the SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY clauses of a query.

Examples: JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, CROSS JOIN, UNION ALL

1. **Special Operators:** These operators are specific to Hive and provide additional functionality for working with complex data types, pattern matching, and null values.

Examples:

* + Complex data type operators: [] (array indexing), . (map key or struct field access)
  + Pattern matching operator: LIKE, RLIKE or REGEXP
  + Null value operators: IS NULL, IS NOT NULL

In summary, Hive operators are symbols or keywords used to perform operations on data values, columns, or expressions. They include arithmetic, comparison, logical, relational, and special operators for various data manipulations and transformations.

**13.Explain about the Hive Built-In Functions**

Hive Built-In Functions are predefined functions provided by Hive that can be used within queries to perform various operations and transformations on the data. These functions can be categorized into several groups:

1. **String Functions:** These functions perform operations on string data, such as concatenation, substring extraction, length calculation, and case manipulation.

Examples: concat(), substr(), length(), lower(), upper()

1. **Mathematical Functions:** These functions perform mathematical calculations, such as rounding, square root, logarithm, and trigonometric operations.

Examples: round(), sqrt(), log(), sin(), cos()

1. **Date and Time Functions:** These functions are used to manipulate and extract information from date and time values, such as adding or subtracting intervals, extracting specific components, and converting between different formats.

Examples: date\_add(), date\_sub(), year(), month(), day(), from\_unixtime(), to\_date()

1. **Aggregate Functions:** These functions perform calculations on a group of rows and return a single value as a result. They are often used with the GROUP BY clause in a query.

Examples: count(), sum(), avg(), min(), max()

1. **Conditional Functions:** These functions return different values or perform different actions based on specific conditions.

Examples: if(), coalesce(), nullif(), case

1. **Type Conversion Functions:** These functions convert values from one data type to another.

Examples: cast(), int(), float(), double(), string()

1. **Miscellaneous Functions:** These functions include a variety of other operations, such as generating hashes, working with arrays, maps, and structs, and querying metadata.

Examples: hash(), array(), map(), struct(), current\_database(), current\_user()

In summary, Hive Built-In Functions are predefined functions used to perform various operations and transformations on data, including string manipulation, mathematical calculations, date and time manipulation, aggregation, conditional logic, type conversion, and more.

**14. Write hive DDL and DML commands.**

In Hive, DDL (Data Definition Language) and DML (Data Manipulation Language) commands are used to define and manipulate data, respectively. Here's a list of common Hive DDL and DML commands:

**DDL Commands:**

1. CREATE DATABASE: Creates a new database.

CREATE DATABASE [IF NOT EXISTS] database\_name;

1. DROP DATABASE: Deletes an existing database.

DROP DATABASE [IF EXISTS] database\_name [CASCADE];

1. CREATE TABLE: Creates a new table with the specified schema.

CREATE [EXTERNAL] TABLE [IF NOT EXISTS] database\_name.table\_name (column1 data\_type1, column2 data\_type2, ...)

[COMMENT 'table\_comment']

[PARTITIONED BY (partition\_column1 data\_type, ...)]

[ROW FORMAT row\_format\_specification]

[STORED AS file\_format]

[LOCATION 'hdfs\_path']

[TBLPROPERTIES ('property\_key'='property\_value')];

1. DROP TABLE: Deletes an existing table.

DROP TABLE [IF EXISTS] database\_name.table\_name;

1. ALTER TABLE: Modifies the properties or schema of an existing table.

ALTER TABLE database\_name.table\_name RENAME TO new\_table\_name;

ALTER TABLE database\_name.table\_name ADD COLUMNS (new\_column1 data\_type1, new\_column2 data\_type2, ...);

1. TRUNCATE TABLE: Removes all data from an existing table.

TRUNCATE TABLE database\_name.table\_name;

**DML Commands:**

1. LOAD DATA: Loads data from an HDFS file or directory into a table.

LOAD DATA [LOCAL] INPATH 'hdfs\_path' [OVERWRITE] INTO TABLE database\_name.table\_name [PARTITION (partition\_spec)];

1. INSERT: Inserts data into a table by selecting data from another table or query.

INSERT [OVERWRITE] [INTO] TABLE database\_name.table\_name [PARTITION (partition\_spec)] SELECT column1, column2, FROM source\_table\_name WHERE condition;

1. UPDATE: Modifies data in a table (requires ACID properties enabled).

UPDATE database\_name.table\_name SET column1 = new\_value1, column2 = new\_value2, ... WHERE condition;

1. DELETE: Removes data from a table (requires ACID properties enabled).

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

**15. Explain about SORT BY, ORDER BY, DISTRIBUTE BY and CLUSTER BY in Hive.**

1. **ORDER BY:** This clause is used to sort the output of a query in a specified order, either ascending (ASC) or descending (DESC). It sorts the data globally, which means the entire dataset is sorted before it is returned. This can be slow for large datasets, as it uses only one reducer.

Example: SELECT name, age FROM students ORDER BY age ASC;

This query will return the 'name' and 'age' columns from the 'students' table, sorted by 'age' in ascending order.

1. **SORT BY:** This clause is similar to ORDER BY, but it sorts the data locally, meaning within each reducer. This can be faster than ORDER BY for large datasets because it uses multiple reducers, but the final output may not be fully sorted.

Example: SELECT name, age FROM students SORT BY age ASC;

This query will return the 'name' and 'age' columns from the 'students' table, sorted by 'age' in ascending order within each reducer.

1. **DISTRIBUTE BY:** This clause is used to control the distribution of data across reducers. It doesn't sort the data but determines how the data is partitioned. The data with the same key will go to the same reducer.

Example: SELECT name, age FROM students DISTRIBUTE BY age;

This query will return the 'name' and 'age' columns from the 'students' table, with records having the same 'age' being processed by the same reducer.

1. **CLUSTER BY:** This is a combination of DISTRIBUTE BY and SORT BY. It first partitions the data using DISTRIBUTE BY and then sorts the data within each partition using SORT BY. It's useful when we want to partition and sort the data at the same time.

Example: SELECT name, age FROM students CLUSTER BY age;

This query will return the 'name' and 'age' columns from the 'students' table, with records having the same 'age' being processed by the same reducer and sorted within each partition.

**In summary, ORDER BY sorts data globally, SORT BY sorts data locally within reducers, DISTRIBUTE BY controls data distribution across reducers, and CLUSTER BY combines both data distribution and sorting within each partition.**

**16.Difference between "Internal Table" and "External Table" and Mention when to choose “Internal Table” and “External Table” in Hive?**

In Hive, there are two types of tables: Internal (Managed) Tables and External Tables. These tables differ in how they store data and manage the data's lifecycle. Here's a simple explanation along with examples:

1. **Internal (Managed) Table:** In an Internal Table, both the data and schema are managed by Hive. When we create an Internal Table, Hive stores the data in its warehouse directory. If we drop an Internal Table, Hive deletes both the data and the metadata associated with the table.

CREATE TABLE students\_internal (

id INT,

name STRING,

age INT

) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

1. **External Table:** In an External Table, Hive manages only the schema (metadata) while the data is stored in an external location specified by the user. When we drop an External Table, Hive only deletes the metadata; the data remains in the external location.

CREATE EXTERNAL TABLE students\_external (

id INT,

name STRING,

age INT

) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LOCATION '/external\_data/students/';

This query creates an External Table called 'students\_external' with 'id', 'name', and 'age' columns, with data stored in the '/external\_data/students/' directory.

**When to choose Internal or External Table:**

1. Choose an Internal Table when:
   * We want Hive to manage both data and schema.
   * The data is exclusive to Hive and not used by other applications.
   * We want to delete the data along with the table when it's dropped.
2. Choose an External Table when:
   * We want Hive to manage only the schema while the data is stored externally.
   * The data is shared with other applications or stored in a specific location.
   * We want to keep the data even if the table is dropped.

In summary, Internal Tables are managed entirely by Hive (data and schema), and deleting the table also deletes the data. External Tables allow we to store data externally while Hive manages the schema, and the data remains intact even when the table is dropped. Choose Internal Tables for data exclusive to Hive and External Tables for data shared with other applications or with specific storage requirements.

**17. Where does the data of a Hive table get stored?**

In Hive, the data of a table gets stored in the Hadoop Distributed File System (HDFS). The location of the data depends on whether the table is an Internal (Managed) Table or an External Table.

1. **Internal (Managed) Table:** The data of an Internal Table is stored in Hive's warehouse directory. By default, this directory is located at /user/hive/warehouse in HDFS. When we create an Internal Table, a subdirectory is created within the warehouse directory using the table's name.

Example: If we create an Internal Table named 'students', the data will be stored in /user/hive/warehouse/students in HDFS.

1. **External Table:** The data of an External Table is stored in an external location specified by the user when the table is created. This location can be any directory within HDFS.

Example: If we create an External Table named 'students' and specify the location as /external\_data/students, the data will be stored in that directory in HDFS.

sql

CREATE EXTERNAL TABLE students (

id INT,

name STRING,

age INT

) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LOCATION '/external\_data/students/';

In summary, the data of a Hive table is stored in HDFS. For Internal Tables, the data is stored in the default warehouse directory, while for External Tables, the data is stored in a user-specified location.

**18. Is it possible to change the default location of a managed table?**

Yes, it is possible to change the default location of a Managed (Internal) Table in Hive. We can do this by specifying the custom location using the LOCATION keyword when creating the table. Here's a simple example:

CREATE TABLE students\_custom\_location (

id INT,

name STRING,

age INT

) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

STORED AS TEXTFILE

LOCATION '/custom\_location/students/';

In this example, a Managed Table named 'students\_custom\_location' is created with 'id', 'name', and 'age' columns. The data is stored in a custom location /custom\_location/students/ in HDFS instead of the default warehouse directory (/user/hive/warehouse).

Please note that even though we changed the location, this table is still considered a Managed (Internal) Table. If we drop the table, both the data and metadata will be deleted.

In summary, we can change the default location of a Managed Table by specifying the custom location with the LOCATION keyword while creating the table.

**19. What is a metastore in Hive? What is the default database provided by Apache Hive for metastore?**

A metastore in Hive is a central repository that stores metadata about the tables, databases, columns, data types, and other information related to the Hive system. Metastore makes it easy for Hive to manage the schema and retrieve metadata when needed. It helps Hive understand the structure of the data and enables it to execute queries efficiently.

The default database provided by Apache Hive for the metastore is the Derby database, which is an embedded database that runs in the same Java Virtual Machine (JVM) as the Hive service. Derby is suitable for small-scale or single-user environments, but it does not support concurrent access from multiple users or clients.

Example: When we create a new table in Hive, the metastore stores the metadata about the table, such as the table name, column names, data types, and storage format. This metadata helps Hive understand the structure of the data and execute queries on the table.

CREATE TABLE students (

id INT,

name STRING,

age INT

) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

In this example, when we create the 'students' table, the metastore will store the table name 'students', column names 'id', 'name', and 'age', their respective data types (INT and STRING), and the storage format (delimited fields terminated by a comma).

In summary, a metastore in Hive is a central repository that stores metadata about tables, databases, columns, and other information. The default database for the Hive metastore is the Derby database, which is suitable for small-scale or single-user environments.

**20. Why does Hive not store metadata information in HDFS?**

Hive does not store metadata information in HDFS because of the following reasons:

1. Latency: HDFS is designed for handling large files and is optimized for high throughput, but it has relatively high latency for small read/write operations. Metadata operations like creating, altering, or dropping tables and databases require low-latency access. Storing metadata in HDFS would make these operations slower, negatively impacting Hive's performance.
2. Concurrency: Metadata operations often require concurrent read and write access, which is not HDFS's strong suit. Traditional databases, such as MySQL or PostgreSQL, are better suited for managing concurrent access to metadata and providing transactional guarantees.
3. Compatibility: Storing metadata in a relational database management system (RDBMS) makes it easier to integrate Hive with other tools and services that expect metadata to be stored in a traditional database format.

**21. What is a partition in Hive? And why do we perform partitioning in Hive?**

A partition in Hive is a technique used to divide a table into smaller, more manageable pieces based on the values of one or more columns. Each partition corresponds to a specific value or range of values and is stored separately. This helps to improve query performance by allowing Hive to read only the relevant partitions while processing a query, instead of scanning the entire table.

Partitioning is performed in Hive for the following reasons:

1. Faster query performance: By dividing the data into smaller, more focused partitions, Hive can read only the necessary partitions when processing a query, reducing the amount of data that needs to be read and speeding up the query.
2. Efficient storage: Partitioning allows we to store data in a more organized way, making it easier to manage and maintain. We can also apply different storage formats or compression algorithms to different partitions based on their characteristics.

Example:

Consider a table named 'sales' with columns 'date', 'product', and 'quantity'. If we partition the 'sales' table based on the 'date' column, each partition will correspond to a specific date, and the data for that date will be stored separately.

sql

CREATE TABLE sales (

date STRING,

product STRING,

quantity INT

) PARTITIONED BY (date);

Now, when we run a query that only needs data for a specific date, Hive can read only the partition corresponding to that date, rather than scanning the entire table.

SELECT product, SUM(quantity) FROM sales

WHERE date = '2023-04-30'

GROUP BY product;

In this example, Hive will only read the partition for '2023-04-30', making the query run faster.

In summary, a partition in Hive is a technique used to divide a table into smaller, more manageable pieces based on the values of one or more columns. Partitioning is performed to improve query performance and enable more efficient storage.

**22. What is the difference between dynamic partitioning and static partitioning?**

In Hive, partitioning is a technique used to divide a table into smaller, more manageable pieces based on the values of one or more columns. There are two types of partitioning: Static Partitioning and Dynamic Partitioning.

1. Static Partitioning: In static partitioning, we manually specify the partition column value while inserting data into the table. This method is more suitable when we have a small number of partitions, and we know the partition values in advance.

Example: Consider a table named 'sales' partitioned by the 'date' column. To insert data using static partitioning, we would specify the partition value:

INSERT INTO TABLE sales PARTITION (date='2023-04-30')

VALUES ('productA', 100);

In this example, the partition value '2023-04-30' is explicitly specified while inserting the data.

1. Dynamic Partitioning: In dynamic partitioning, we don't have to specify the partition column value manually. Instead, Hive automatically determines the partition value based on the input data. This method is more suitable when we have a large number of partitions, or the partition values are unknown in advance.

Example: Consider the same 'sales' table partitioned by the 'date' column. To insert data using dynamic partitioning, we would let Hive determine the partition value:

SET hive.exec.dynamic.partition = true;

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

INSERT INTO TABLE sales PARTITION (date)

SELECT product, quantity, date FROM sales\_input\_data;

In this example, Hive automatically determines the partition values based on the 'date' column in the 'sales\_input\_data' table.

In summary, static partitioning requires manual specification of partition values while inserting data, and it's suitable for tables with a small number of known partition values. Dynamic partitioning allows Hive to automatically determine partition values based on input data, making it more suitable for tables with a large number of partitions or unknown partition values.

**23. How do we check if a particular partition exists?**

In Hive, we can use the SHOW PARTITIONS command to list all the partitions of a table. To check if a particular partition exists, we can filter the output of the SHOW PARTITIONS command using a WHERE clause or by using a Unix command like grep when running the command from the command line.

Example:

Consider a table named 'sales' partitioned by the 'date' column. To check if a partition exists for the date '2023-04-30', we can do the following:

1. Using Hive SQL:

SELECT \* FROM (SHOW PARTITIONS sales) partition\_list

WHERE partition\_list = 'date=2023-04-30';

This query will return a row with 'date=2023-04-30' if the partition exists, and no rows if the partition does not exist.

1. Using the command line with grep:

hive -e "SHOW PARTITIONS sales" | grep 'date=2023-04-30'

This command will print 'date=2023-04-30' if the partition exists, and nothing if the partition does not exist.

**24. How can we stop a partition form being queried?**

**Identify partition:** Determine the specific partition we want to stop from being queried. Partitions are usually based on columns like date, time, or a specific category.

Example: Suppose we have a table 'sales\_data' partitioned by date (partition column 'sales\_date').

**Alter table:** Use the ALTER TABLE statement to modify the partition we want to stop from being queried. We can add a comment to the partition to indicate that it should not be queried.

Example: ALTER TABLE sales\_data PARTITION (sales\_date='2023-05-01') SET TBLPROPERTIES ('comment' = 'Do not query this partition.');

**Use conditional statements:** Whenever we write a query, use conditional statements (like WHERE or CASE) to exclude the specific partition we want to stop from being queried.

Example: SELECT \* FROM sales\_data WHERE sales\_date != '2023-05-01';

**25. Why do we need buckets? How Hive distributes the rows into buckets?**

**Here's why we need buckets:**

1. **Improved performance:** By dividing data into smaller, more manageable pieces (buckets), Hive can process and retrieve data faster, especially when using joins, filters, and aggregations.
2. **Even distribution:** Data is distributed evenly across buckets based on a specified column and hashing function, allowing for balanced data access and query performance.
3. **Clustering:** Buckets enable clustering similar data together, which can improve performance for certain types of queries (e.g., queries with GROUP BY or aggregations).

**Hive distributes rows into buckets using the following process:**

1. **Select bucketing column:** Choose a column that has a good distribution of values and is frequently used in queries (e.g., user\_id, order\_id).
2. **Define number of buckets:** Set the number of buckets to create. A higher number of buckets results in smaller and more manageable partitions.
3. **Specify hashing function:** Hive uses a hashing function on the bucketing column to determine which bucket a row should be placed in. By default, Hive uses the 'MurmurHash' function.

**Create table with buckets:** Use the CREATE TABLE statement with the CLUSTERED BY and INTO BUCKETS keywords to create a bucketed table.

**HQL example:**

CREATE TABLE orders\_bucketed (

order\_id INT,

order\_date STRING,

customer\_id INT,

amount DOUBLE

) CLUSTERED BY (order\_id) INTO 16 BUCKETS;

**Load data:** Insert data into the bucketed table. Hive will automatically distribute rows into the appropriate buckets based on the hashing function applied to the bucketing column.

HQL example: INSERT INTO orders\_bucketed SELECT \* FROM orders;

In summary, buckets in Hive help improve query performance and manage data distribution more effectively by dividing data into smaller, evenly distributed pieces based on a specified column and hashing function.

**26. In Hive, how can we enable buckets?**

In Hive, we can enable buckets by following these steps:

1. **Choose a column**: Select a column with a good distribution of values and is commonly used in queries (e.g., user\_id, order\_id).
2. **Determine the number of buckets:** Decide on the number of buckets we want to create. More buckets lead to smaller and more manageable data chunks.
3. **Create a bucketed table:** Use the CREATE TABLE statement with the CLUSTERED BY and INTO BUCKETS keywords to create a bucketed table.

**HQL example:**

CREATE TABLE orders\_bucketed (

order\_id INT,

order\_date STRING,

customer\_id INT,

amount DOUBLE

) CLUSTERED BY (order\_id) INTO 16 BUCKETS;

Load data into the bucketed table: Insert data into the bucketed table. Hive will automatically distribute rows into the appropriate buckets based on the bucketing column.

HQL example: INSERT INTO orders\_bucketed SELECT \* FROM orders;

By following these steps, we enable buckets in Hive, which help improve query performance and manage data distribution more effectively by dividing data into smaller, evenly distributed chunks based on a specified column.

**27. How does bucketing help in the faster execution of queries?**

1. **Even distribution:** Data is distributed evenly across buckets, ensuring balanced data access and query performance.
2. **Data organization:** Bucketing organizes similar data together, which improves performance for certain types of queries, such as those with GROUP BY or aggregations.
3. **Faster joins:** When joining two bucketed tables on the bucketing column, Hive can perform a more efficient "bucketed map join," which reduces the need for shuffling and sorting, resulting in faster query execution.
4. **Sample data quickly:** With buckets, we can easily sample a subset of data, making it faster to test and validate queries before running them on the full dataset.

**HQL example for creating a bucketed table:**

CREATE TABLE orders\_bucketed (

order\_id INT,

order\_date STRING,

customer\_id INT,

amount DOUBLE

) CLUSTERED BY (order\_id) INTO 16 BUCKETS;

**HQL example for bucketed map join:**

/\* Enable bucketed map join \*/

SET hive.auto.convert.join=true;

SET hive.optimize.bucketmapjoin=true;

SET hive.optimize.bucketmapjoin.sortedmerge=false;

**===========================================================================**

/\* Join bucketed tables \*/

SELECT o.order\_id, o.order\_date, c.customer\_name

FROM orders\_bucketed o

JOIN customers\_bucketed c

ON o.customer\_id = c.customer\_id;

In summary, bucketing in Hive improves query performance by organizing data into smaller, evenly distributed chunks. This enables faster joins, efficient data sampling, and improved performance for certain query types.

**28. How to optimise Hive Performance? Explain in very detail.**

Optimizing Hive performance involves tuning various settings and using best practices in table design and querying. Here are some key points and HQL examples for performance optimization:

1. **Partitioning:** Distribute data in a table based on commonly used columns, such as date or category, to reduce the amount of data scanned during queries.

CREATE TABLE sales\_data (

product\_id INT,

customer\_id INT,

revenue DOUBLE

) PARTITIONED BY (sales\_date STRING);

1. **Bucketing:** Organize data into smaller, evenly distributed chunks based on a specified column to improve query performance.

CREATE TABLE orders\_bucketed (

order\_id INT,

order\_date STRING,

customer\_id INT,

amount DOUBLE

) CLUSTERED BY (order\_id) INTO 16 BUCKETS;

1. **Indexing:** Create indexes on frequently used columns to speed up query performance.

CREATE INDEX sales\_data\_index

ON TABLE sales\_data (product\_id)

AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler';

1. **Use ORC or Parquet file formats:** These file formats provide better compression and improved query performance compared to text-based formats.

CREATE TABLE sales\_data\_orc (

product\_id INT,

customer\_id INT,

revenue DOUBLE

) STORED AS ORC;

1. **Vectorization:** Enable vectorized query execution to process data in batches, improving CPU utilization and query performance.

SET hive.vectorized.execution.enabled = true;

SET hive.vectorized.execution.reduce.enabled = true;

1. **Tez execution engine:** Use the Tez execution engine instead of MapReduce for faster query execution.

SET hive.execution.engine=tez;

1. **Caching:** Cache frequently accessed tables in memory to improve query performance.

CREATE TABLE sales\_data\_cached

TBLPROPERTIES ("hive.cache.expr" = "sales\_date >= '2022-01-01'");

1. **Optimized joins:** Use various join optimization techniques such as map joins, bucketed map joins, and sorted bucketed map joins to speed up join operations.

SET hive.auto.convert.join=true;

SET hive.optimize.bucketmapjoin=true;

1. **Query optimization:** Write efficient queries by using filter conditions, limiting the number of columns, and avoiding nested sub queries when possible.
2. **Tune configuration parameters:** Adjust various Hive and Hadoop configuration settings, such as memory allocation, parallelism, and compression, to optimize performance for specific use case.

In summary, optimizing Hive performance involves using partitioning, bucketing, indexing, file formats, vectorization, execution engines, caching, optimized joins, query optimization, and tuning configuration parameters. By applying these best practices, we can significantly improve the performance of Hive queries.

**29. What is the use of Hcatalog?**

Hive Metastore, often referred to as HCatalog, is a component of the Apache Hive ecosystem that provides a way to store and manage metadata for Hive tables, such as table schema, location, and partitioning information. Here are some uses of HCatalog:

1. **Unified metadata storage:** HCatalog allows we to store metadata for Hive tables in a central repository, making it easy to share and manage across different data processing tools like Hive, Pig, and MapReduce.
2. **Table abstraction:** HCatalog provides a table abstraction layer, enabling users to work with tables without worrying about the underlying data formats or storage details. This makes it easier to switch between different file formats and storage systems.
3. **Data interoperability:** By using HCatalog, we can access and process data stored in various formats (e.g., ORC, Parquet, Avro) and systems (e.g., HDFS, HBase) using different data processing tools like Hive, Pig, and MapReduce.
4. **Simplified data access:** HCatalog allows users to access data using SQL-like queries in Hive or Pig Latin in Pig, simplifying data access and reducing the learning curve for working with data stored in Hadoop.
5. **Schema evolution:** HCatalog supports schema evolution, enabling we to modify table schema over time while maintaining backward compatibility with older data.

HQL example for creating a table in Hive (which uses HCatalog to store metadata):

CREATE TABLE sales\_data (

product\_id INT,

customer\_id INT,

revenue DOUBLE

) PARTITIONED BY (sales\_date STRING) STORED AS ORC;

In summary, HCatalog (Hive Metastore) is a valuable component of the Hive ecosystem that provides a unified metadata storage system, table abstraction, data interoperability, simplified data access, and support for schema evolution. By using HCatalog, we can more effectively manage and access data across different data processing tools and storage systems.

**30. Explain about the different types of join in Hive.**

In Hive, there are several types of joins used to combine data from multiple tables. Here's a brief explanation of each join type along with HQL examples:

**INNER JOIN:** Returns rows where there's a match in both tables.

HQL example:

SELECT o.order\_id, o.order\_date, c.customer\_name

FROM orders o

INNER JOIN customers c

ON o.customer\_id = c.customer\_id;

**LEFT OUTER JOIN (or LEFT JOIN):** Returns all rows from the left table and matching rows from the right table. If no match is found, NULL values are returned for the right table's columns.

HQL example:

SELECT o.order\_id, o.order\_date, c.customer\_name

FROM orders o

LEFT OUTER JOIN customers c

ON o.customer\_id = c.customer\_id;

**RIGHT OUTER JOIN (or RIGHT JOIN):** Returns all rows from the right table and matching rows from the left table. If no match is found, NULL values are returned for the left table's columns.

HQL example:

SELECT o.order\_id, o.order\_date, c.customer\_name

FROM orders o

RIGHT OUTER JOIN customers c

ON o.customer\_id = c.customer\_id;

**FULL OUTER JOIN (or FULL JOIN):** Returns all rows from both tables, with NULL values in columns where there's no match.

HQL example:

SELECT o.order\_id, o.order\_date, c.customer\_name

FROM orders o

FULL OUTER JOIN customers c

ON o.customer\_id = c.customer\_id;

**SEMI JOIN:** Returns rows from the left table that have a match in the right table, but does not return any columns from the right table. This join type is implemented using the IN or EXISTS subqueries.

HQL example (using IN):

SELECT o.\*

FROM orders o

WHERE o.customer\_id IN (SELECT customer\_id FROM customers);

**ANTI JOIN:** Returns rows from the left table that have no match in the right table. This join type is implemented using the NOT IN or NOT EXISTS subqueries.

HQL example (using NOT EXISTS):

SELECT o.\*

FROM orders o

WHERE NOT EXISTS (SELECT 1 FROM customers c WHERE o.customer\_id = c.customer\_id);

**CROSS JOIN:** Returns the Cartesian product of rows from both tables, meaning each row from the left table is combined with each row from the right table.

HQL example:

SELECT o.order\_id, c.customer\_name

FROM orders o

CROSS JOIN customers c;

**31. Is it possible to create a Cartesian join between 2 tables, using Hive?**

Yes, it is possible to create a Cartesian join between two tables using Hive. A Cartesian join, also known as a CROSS JOIN, combines each row from the left table with each row from the right table, resulting in the Cartesian product of the two tables. Here's how to create a Cartesian join in Hive:

Use the CROSS JOIN keyword: In HQL query, use the CROSS JOIN keyword to create a Cartesian join between the two tables.

HQL example:

SELECT o.order\_id, c.customer\_name

FROM orders o

CROSS JOIN customers c;

Keep in mind that Cartesian joins can produce a very large number of rows, as the result will have the number of rows in the first table multiplied by the number of rows in the second table. This can lead to performance issues and memory limitations if the tables involved are large. Be cautious when using CROSS JOIN and ensure that we have a clear understanding of the size of the input tables and the expected output.

**32. Explain the SMB Join in Hive?**

Sorted Merge Bucketed (SMB) join is a type of join optimization technique in Hive that combines data from two bucketed and sorted tables efficiently. SMB join is particularly useful when joining large tables with similar size and data distribution.

Here's how SMB join works:

Tables must be bucketed and sorted on the join key: Both tables involved in the join should be bucketed and sorted on the same join key (column).

Number of buckets must be the same: The number of buckets in both tables must be the same to perform an SMB join.

Merge join optimization: SMB join performs a merge join, which reads one bucket from each table at a time and merges the data in a sorted manner. This avoids loading the entire dataset into memory, making it more efficient for large tables.

HQL example:

Create bucketed and sorted tables:

sql

Copy code

-- Create table orders\_bucketed\_sorted

CREATE TABLE orders\_bucketed\_sorted (

order\_id INT,

order\_date STRING,

customer\_id INT,

amount DOUBLE

) CLUSTERED BY (customer\_id) SORTED BY (customer\_id) INTO 16 BUCKETS;

-- Create table customers\_bucketed\_sorted

CREATE TABLE customers\_bucketed\_sorted (

customer\_id INT,

customer\_name STRING,

customer\_email STRING

) CLUSTERED BY (customer\_id) SORTED BY (customer\_id) INTO 16 BUCKETS;

Enable SMB join optimization:

sql

Copy code

SET hive.auto.convert.sortmerge.join=true;

SET hive.optimize.bucketmapjoin=true;

SET hive.optimize.bucketmapjoin.sortedmerge=true;

Perform the SMB join:

sql

Copy code

SELECT o.order\_id, o.order\_date, c.customer\_name

FROM orders\_bucketed\_sorted o

JOIN customers\_bucketed\_sorted c

ON o.customer\_id = c.customer\_id;

In summary, the Sorted Merge Bucketed (SMB) join in Hive is an optimization technique for joining large, similarly sized tables that are bucketed and sorted on the join key. This approach reads and merges one bucket at a time, improving join efficiency and reducing memory usage.

**33. What is the difference between order by and sort by which one we should use?**

| **Aspect** | **ORDER BY** | **SORT BY** |
| --- | --- | --- |
| **Ordering** | Provides total order of the data | Provides partial order of the data (sorted within each reducer's output) |
| **Reducers** | Uses a single reducer | Uses multiple reducers |
| **Performance** | Can be a bottleneck for large datasets | Better performance for large datasets |
| **Use Case** | Suitable for small datasets where total order is required | Suitable for large datasets where total order is not necessary |

**HQL examples:**

**ORDER BY:**

SELECT customer\_id, revenue

FROM sales\_data

ORDER BY revenue DESC;

**SORT BY:**

SELECT customer\_id, revenue

FROM sales\_data

SORT BY revenue DESC;

**Which one to use?**

* Use ORDER BY when we need a total order of the data and are working with small datasets. This guarantees the correct ordering but may have performance limitations due to the single-reducer constraint.
* Use SORT BY when working with large datasets and a total order of the data is not required. SORT BY offers better performance by utilizing multiple reducers to process the data.
* In summary, choose the appropriate sorting method based on dataset size and ordering requirements. ORDER BY provides a total order of the data but can have performance limitations, while SORT BY provides a partial order of the data and better performance with large datasets.

**34. What is the usefulness of the DISTRIBUTED BY clause in Hive?**

The DISTRIBUTED BY clause in Hive is used to control the distribution of rows in the output of a query across multiple reducers. This is particularly useful when we want to distribute the data evenly among reducers to avoid data skew and improve query performance.

* **Custom distribution:** The DISTRIBUTED BY clause allows we to specify the columns by which the data should be distributed across reducers. This helps ensure a more balanced distribution of data and can improve performance.
* **Data skew reduction:** By distributing the data based on specific columns, we can reduce data skew, which occurs when some reducers receive more data than others. Data skew can lead to longer query execution times, as some reducers take much longer to finish processing their data.
* **Commonly used with SORT BY:** The DISTRIBUTED BY clause is often used with the SORT BY clause to distribute and sort data across multiple reducers. This combination can provide a more efficient sorting process and better performance.

HQL example:

In this example, we want to distribute the data evenly among reducers based on the customer\_id column and sort the data within each reducer's output based on the revenue column.

SELECT customer\_id, revenue

FROM sales\_data

DISTRIBUTE BY customer\_id

SORT BY revenue DESC;

In summary, the DISTRIBUTED BY clause in Hive allows we to control the distribution of rows across reducers based on specific columns, helping to reduce data skew and improve query performance. It is often used in conjunction with the SORT BY clause to distribute and sort data efficiently across multiple reducers.

**35. How does data transfer happen from HDFS to Hive?**

Hive is a data warehousing solution built on top of Hadoop, which uses the Hadoop Distributed File System (HDFS) for storing its data. Data transfer between HDFS and Hive is not a direct transfer process but rather an interaction between the two components. Here's how data flows between HDFS and Hive:

**Data storage:** Hive stores its data in HDFS as the underlying storage system. When we create a Hive table, the data is stored in a directory within HDFS. The table's schema and other metadata are stored in the Hive Metastore.

CREATE EXTERNAL TABLE sales\_data (

product\_id INT,

customer\_id INT,

revenue DOUBLE

) ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

STORED AS TEXTFILE

LOCATION '/user/data/sales\_data';

**Data access:** When we run a query in Hive, it reads the data from HDFS. Hive converts query into a series of MapReduce, Tez, or Spark jobs, depending on the execution engine. These jobs read the data from HDFS, process it, and then write the results back to HDFS.

**HQL example for querying data in Hive:**

SELECT product\_id, SUM(revenue) as total\_revenue

FROM sales\_data

GROUP BY product\_id;

**Data updates:** When we insert, update, or delete data in Hive, the changes are reflected in the HDFS files. Note that Hive is designed for batch processing, and updates and deletes are not as efficient as they are in traditional relational databases.

**HQL example for inserting data into a Hive table:** INSERT INTO sales\_data (product\_id, customer\_id, revenue) VALUES (1, 101, 150.0);

In summary, the data transfer between HDFS and Hive involves storing Hive table data in HDFS, accessing and processing the data using Hadoop processing frameworks (MapReduce, Tez, or Spark), and updating the data stored in HDFS when changes are made in Hive. Hive interacts with HDFS to read and write data, and the data is processed using distributed computing frameworks.

**36. Wherever (Different Directory) I run the hive query, it creates a new metastore\_db, please explain the reason for it?**

When you run a Hive query and notice that a new metastore\_db is created in the current directory, it is because the default configuration for the Hive Metastore is to use an embedded Derby database in the local file system. This happens when the hive.metastore.warehouse.dir configuration parameter is not explicitly set or the Metastore is not configured to use an external database (e.g., MySQL or PostgreSQL).

To avoid creating a new metastore\_db in the current directory, you have two options:

* 1. Set the hive.metastore.warehouse.dir configuration parameter to a fixed location:

You can set this parameter in the hive-site.xml configuration file. By doing this, the metastore\_db will be created in the specified directory, regardless of where you run the Hive query.

Add the following to your hive-site.xml file:

<property>

<name>hive.metastore.warehouse.dir</name>

<value>/path/to/fixed/metastore/directory</value>

</property>

Replace /path/to/fixed/metastore/directory with the desired fixed location for the metastore\_db.

* 1. Configure Hive to use an external database for the Metastore:

Instead of using the default embedded Derby database, you can configure Hive to use an external database, such as MySQL or PostgreSQL, for the Metastore. This will centralize the Metastore and prevent the creation of metastore\_db directories in your local filesystem.

**Here's a high-level overview of the process:**

a. Install and configure the external database (e.g., MySQL or PostgreSQL).

b. Create the necessary Hive Metastore schema in the external database.

c. Update the hive-site.xml configuration file with the necessary properties to connect to the external database.

In summary, the creation of a new metastore\_db directory when running Hive queries is due to the default configuration of the Hive Metastore using an embedded Derby database in the local filesystem. You can avoid this by setting a fixed location for the metastore\_db or configuring Hive to use an external database for the Metastore.

**37. What will happen in case we have not issued the command: ‘SET hive.enforce.bucketing=true;’ before bucketing a table in Hive?**

If you do not set the hive.enforce.bucketing=true; configuration property before creating a bucketed table in Hive, the following outcomes can occur:

* 1. **Bucketing not enforced during data insertion:** Without setting hive.enforce.bucketing=true;, when you insert data into the bucketed table, Hive might not distribute the rows into the specified number of buckets according to the bucketing column. Instead, the data might be inserted without considering the bucketing requirements.
  2. **Bucketing benefits not realized**: Without enforcing bucketing, you might not be able to take advantage of the performance improvements provided by bucketing, such as faster query execution for bucketed map joins, sampling, and partition pruning.

To ensure bucketing is enforced when creating and inserting data into a bucketed table, set the hive.enforce.bucketing configuration property to true. Here's an HQL example:

sql

Copy code

-- Set the configuration property

SET hive.enforce.bucketing=true;

-- Create a bucketed table

CREATE TABLE sales\_data\_bucketed (

product\_id INT,

customer\_id INT,

revenue DOUBLE

)

CLUSTERED BY (product\_id) INTO 8 BUCKETS

STORED AS ORC;

-- Insert data into the bucketed table

INSERT INTO sales\_data\_bucketed

SELECT product\_id, customer\_id, revenue

FROM sales\_data;

In summary, if you do not set the hive.enforce.bucketing=true; configuration property before creating and inserting data into a bucketed table in Hive, bucketing might not be enforced during data insertion, and you may not realize the performance benefits associated with bucketing.

**38. Can a table be renamed in Hive?**

Yes, you can rename a table in Hive. To do so, use the ALTER TABLE statement followed by the RENAME TO clause. Here are the key points:

1. Renaming a table: The ALTER TABLE statement with the RENAME TO clause allows you to change the name of a table in Hive.
2. Updating metadata: When a table is renamed, the Hive Metastore updates the metadata associated with the table to reflect the new name.
3. No data movement: Renaming a table does not involve moving or copying data in the underlying HDFS storage. Only the table name is changed in the metadata.

HQL example:

Suppose you have a table named sales\_data and you want to rename it to sales\_data\_v2. Use the following HQL query: ALTER TABLE sales\_data RENAME TO sales\_data\_v2;

**39. Write a query to insert a new column(new\_col INT) into a hive table at a position before an existing column (x\_col)**

In Hive, you can use the ALTER TABLE statement with the ADD COLUMNS and AFTER clauses to add a new column to an existing table. However, there is no direct way to insert a new column at a specific position before an existing column. Instead, you can follow these steps to achieve the desired outcome:

Create a new table with the desired column order: Create a new table that has the same structure as the original table, but with the new column inserted at the desired position before the existing column.

Copy data from the original table to the new table: Insert the data from the original table into the new table, mapping the columns according to the new structure.

Rename tables: Optionally, rename the original table to a backup name and then rename the new table to the original table name.

Here's an HQL example using a table named sales\_data with columns product\_id, customer\_id, and revenue. We want to insert a new column new\_col INT before the existing column customer\_id.

-- Step 1: Create a new table with the desired column order

CREATE TABLE sales\_data\_new (

product\_id INT,

new\_col INT,

customer\_id INT,

revenue DOUBLE

);

-- Step 2: Copy data from the original table to the new table

INSERT INTO sales\_data\_new (product\_id, new\_col, customer\_id, revenue)

SELECT product\_id, NULL, customer\_id, revenue

FROM sales\_data;

**40. What is serde operation in HIVE?**

In Hive, SerDe (Serializer/Deserializer) is a component that enables you to read and write data in different formats. SerDe is responsible for converting the raw data in HDFS into a format that Hive can process (deserialization) and converting processed data back into the storage format in HDFS (serialization). Here are the key points:

Deserialization: Converting the data from the storage format in HDFS to the internal format used by Hive for processing.

Serialization: Converting the data from the internal format used by Hive to the format suitable for storage in HDFS.

Built-in and custom SerDe: Hive has built-in SerDes for common formats (e.g., Text, ORC, Parquet), but you can also create custom SerDes to handle specific data formats.

Data format flexibility: SerDe allows Hive to work with various data formats, making it easy to query and process structured and semi-structured data.

Here's an HQL example of creating a table using a built-in SerDe for CSV files:

CREATE EXTERNAL TABLE sales\_data (

product\_id INT,

customer\_id INT,

revenue DOUBLE

)

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

WITH SERDEPROPERTIES (

"separatorChar" = ",",

"quoteChar" = "\"",

"escapeChar" = "\\"

)

STORED AS TEXTFILE

LOCATION '/user/data/sales\_data';

In this example, we create an external table sales\_data that uses the org.apache.hadoop.hive.serde2.OpenCSVSerde SerDe to handle CSV data. The ROW FORMAT SERDE clause specifies the SerDe class, and the WITH SERDEPROPERTIES clause sets properties specific to the CSV format, such as the separator, quote, and escape characters.

In summary, SerDe operations in Hive are responsible for serialization and deserialization of data, allowing Hive to work with different data formats. By using built-in or custom SerDes, you can process various structured and semi-structured data formats in Hive.

**41. Explain how Hive Deserializes and serialises the data?**

**Serialization:**

* Data Collection: Hive gathers the data you want to work with. This data could be from various sources like databases, files, or data streams.
* Conversion: Hive then converts this data into a standardized format. This process is known as serialization. It's like taking apart a large 3D puzzle and putting all the pieces into a box.
* Storage or Transmission: The serialized data is now easier to store or send over a network. It's more efficient to work with this "boxed" data than the original "puzzle".

**Deserialization:**

* Retrieval: When you want to analyse or use the data, Hive retrieves the serialized data. This is like getting the box of puzzle pieces.
* Reconstruction: Hive then converts (desterilizes) the data back to its original format. It's like taking the puzzle pieces out of the box and putting them back together to see the whole picture.
* Usage: Now, the data is ready for you to use for your analysis, queries, or other operations.

In short, serialization and deserialization are methods used by Hive to efficiently store and retrieve data. It does this through a system called SerDe (Serializer/Deserializer), which both transforms the data and provides a description (schema) of the data structure.

**42. Write the name of the built-in serde in hive.**

Hive comes with several built-in Serializer/Deserializer (SerDe) libraries that you can use. A few of these include:

* LazySimpleSerDe: This is the default SerDe for most Hive operations. It can handle all primitive types and their collections (arrays, maps, and structs).
* ColumnarSerDe: This SerDe is used to serialize columns of a table into a more efficient format. It's often used with RCFile format.
* AvroSerDe: This is used when working with Avro data files. Avro is a data serialization system that provides rich data structures and a compact, fast binary data format.
* OrcSerde: This SerDe is used with the Optimized Row Columnar (ORC) file format. This file format provides a highly efficient way to store Hive data.
* ParquetHiveSerDe: This SerDe is used when working with Parquet files. Parquet is a columnar storage file format available to any project in the Hadoop ecosystem.
* RegexSerDe: This SerDe is used for parsing data with a regular expression.

**43. What is the need of custom Serde?**

In some cases, the built-in SerDe libraries in Hive may not meet your specific requirements. This is where custom SerDe comes in. Here's why you might need a custom SerDe:

* Unique Data Formats: If your data is in a unique or unusual format that the built-in SerDes can't handle, you'll need to create a custom SerDe to interpret your data correctly.
* Specialized Processing: You might need to perform some specialized processing on your data as it's being read into Hive. A custom SerDe allows you to add this logic.
* Performance Optimization: If the built-in SerDe isn't processing your data efficiently, a custom SerDe could be written to improve performance.
* Proprietary Data: If you're working with proprietary or confidential data, you might need a custom SerDe that can handle any specific security or privacy requirements your data has.

In simple terms, imagine you're trying to assemble a puzzle, but the pieces come in an unusual shape, or you need to paint each piece in a specific color as you assemble it. The built-in tools (SerDes) you have can't handle these tasks, so you need to create your own tool (custom SerDe) to do the job.

**44. Can we write the name of a complex data type(collection data types) in Hive?**

Yes, Hive supports several complex data types, also known as collection data types, which can be used to structure your data more effectively. Here are the names of these types:

* Arrays: An ordered collection of elements. The elements in the array need to be of the same type. For example, an array of strings, integers, etc.
* Maps: A collection of key-value pairs. The key must be a primitive type (like string, int, etc.), and the value can be any type, including complex types.
* Structs: A collection of named fields that each have a type. The fields can be any type, including complex types. Structs are similar to classes in object-oriented languages.

Here's an example of how you might declare a table with complex types:

**CREATE TABLE complex\_data (**

**id INT,**

**name STRING,**

**scores ARRAY<INT>, -- Array of integers**

**address MAP<STRING, STRING>, -- Map with string keys and values**

**properties STRUCT<age: INT, height: FLOAT> -- Struct with named fields**

**);**

In this example, scores is an array of integers, address is a map with string keys and values, and properties is a struct with two named fields: age, which is an integer, and height, which is a float.

**45. Can hive queries be executed from script files? How?**

Yes, Hive queries can be executed from script files. You can write your HiveQL commands in a script file and then execute them using the Hive command line interface. Here's how you can do this:

Create a script file: First, create a script file (for example, my\_script.hql) with your HiveQL queries. For instance, your script file might contain:

USE my\_database;

SELECT \* FROM my\_table;

Execute the script file: You can execute the script file from the command line using the -f option followed by the path to the script file. Here's the syntax:

hive -f /path/to/my\_script.hql

Replace /path/to/my\_script.hql with the actual path to your script file.

This will run the HiveQL commands in the my\_script.hql file. It's a great way to automate and reproduce your Hive operations.

**46. What are the default record and field delimiter used for hive text files?**

Hive uses certain default delimiters for text files (stored as TextFile in Hive). These delimiters are used to differentiate between records (rows) and fields (columns) in the data. The default delimiters are:

Record Delimiter: The record delimiter defines the end of a record (or row). By default, it's a newline character (\n).

Field Delimiter: The field delimiter separates the fields (or columns) within each record. By default, it's a caret (^) symbol (^A in ASCII, represented as \001 in Unicode).

Please note that while these are the default settings, Hive allows you to specify custom delimiters using the ROW FORMAT DELIMITED clause in your HiveQL statements if your data uses different delimiters.

**47.How do we list all databases in Hive whose name starts with s?**

**48.What is the difference between LIKE and RLIKE operators in Hive?**

LIKE and RLIKE are two comparison operators used in Hive to perform pattern matching in strings.

**LIKE Operator:** The LIKE operator in Hive is used to match a specified pattern in a column. It allows two wildcard characters:

% (percent sign): Represents zero, one, or multiple characters.

\_ (underscore): Represents a single character.

For example, SELECT \* FROM table WHERE column LIKE 'A%'; will select all records from the table where the values in the column start with "A".

**RLIKE Operator:** The RLIKE operator in Hive is a more advanced pattern-matching operator that uses regular expressions for matching. This means you can use complex pattern-matching expressions which are not possible with the LIKE operator.

For example, SELECT \* FROM table WHERE column RLIKE 'A.\*[0-9]'; will select all records from the table where the values in the column start with "A" and end with a numeric digit.

In summary, while LIKE is sufficient for simple pattern matching, RLIKE allows for much more complex and flexible pattern matching by using regular expressions.

**49.How to change the column data type in Hive?**

**Create a New Table:** Create a new table that has the same structure as the original table, but with the desired data type for the specified column.

CREATE TABLE new\_table AS

SELECT

column1,

column2,

CAST(column3 AS new\_data\_type) AS column3,

...

FROM old\_table;

Replace column1, column2, column3, etc., with the names of your columns, new\_data\_type with the new data type you want to use, and old\_table and new\_table with the names of your old and new tables.

**Drop the Old Table:** If the new table is correct and you no longer need the old table, you can drop it.

DROP TABLE old\_table;

Rename the New Table: If you want the new table to have the same name as the old table, you can rename it.

ALTER TABLE new\_table RENAME TO old\_table;

Please remember to backup your data before performing these operations, as mistakes may lead to data loss. Also, be aware that creating a new table will require the same amount of storage space as the original table, so make sure you have enough space available.

**50. How will we convert the string ’51.2’ to a float value in the particular column?**

To convert a string value like '51.2' to a float value in a particular column, you can use the CAST() function in Hive. The CAST() function allows you to change the data type of a given expression or column. Here's an example of how to use it:

SELECT

column1,

column2,

CAST(string\_column AS FLOAT) AS float\_column,

...

FROM your\_table;

Replace column1, column2, string\_column, etc., with the names of your columns, and your\_table with the name of your table. In this example, the string\_column is converted to a float and is given the alias float\_column.

If you have a specific string value like '51.2' and want to convert it to a float, you can do this:

SELECT CAST('51.2' AS FLOAT);

**51. What will be the result when we cast ‘abc’ (string) as INT?**

If you try to cast a non-numeric string like 'abc' to an integer in Hive, the result will be NULL. This is because 'abc' does not have a valid integer representation.

Here's an example:

SELECT CAST('abc' AS INT);

The result of this query will be NULL.

This behavior can be important to keep in mind when working with data in Hive, as attempting to cast strings to numeric types without ensuring the strings are properly formatted as numbers can lead to unexpected NULL values in your data.

**52. What does the following query do? a. INSERT OVERWRITE TABLE employees b. PARTITION (country, state) c. SELECT ..., se.cnty, se.st d. FROM staged\_employees se;**

The INSERT OVERWRITE TABLE statement in Hive is used to insert data into a Hive table (in this case, employees). If the table or partition already has data, it will be completely replaced by the data being inserted.

This query is specifically dealing with a partitioned table. In Hive, tables can be partitioned based on certain columns to improve query performance. The PARTITION (country, state) clause in this query indicates that the employees table is partitioned on the country and state columns.

Here's what each part of the query does:

**a. INSERT OVERWRITE TABLE employees:** This part of the query indicates that the query is going to insert data into the employees table. If the specified partitions already contain data, that data will be deleted and replaced with the new data.

**b. PARTITION (country, state):** This indicates the specific partitions into which the data will be inserted. The partitions are determined by the country and state values from the staged\_employees table.

**c. SELECT ..., se.cnty, se.st:** This part of the query is generating the data that will be inserted into the employees table. It's selecting some set of columns (represented by ...) along with the cnty and st columns from the staged\_employees table (which are presumably being used as the country and state partitions).

**d. FROM staged\_employees se;:** This specifies that the data being inserted comes from the staged\_employees table, which is given the alias se for the rest of the query.

So, in summary, this query is taking data from the staged\_employees table and using it to overwrite the data in the country and state partitions of the employees table.

**53. Write a query where we can overwrite data in a new table from the existing table.**

Sure, here is an example of a query that overwrites data in a new table from an existing table using the INSERT OVERWRITE TABLE command in Hive:

INSERT OVERWRITE TABLE new\_table

SELECT \*

FROM existing\_table;

In this query:

INSERT OVERWRITE TABLE new\_table means that the query is going to insert data into new\_table. If new\_table already contains data, it will be completely replaced by the data from existing\_table.

SELECT \* FROM existing\_table; is the part of the query that selects all columns of data (\*) from existing\_table.

Please replace new\_table and existing\_table with your actual table names.

Remember that the structure of new\_table should be compatible with the data being inserted from existing\_table. If new\_table does not exist, you need to create it first using a CREATE TABLE command. If the structures don't match, you will need to specify the columns you want to select from existing\_table instead of using \*.

Also, keep in mind that INSERT OVERWRITE will delete any existing data in new\_table. If you want to keep existing data and add to it, you should use INSERT INTO instead.

**54. What is the maximum size of a string data type supported by Hive? Explain how Hive supports binary formats.**

Maximum Size of a String Data Type: In Hive, the maximum size of a string data type is not explicitly defined, but it is effectively limited by the maximum size of a Java heap. The practical limit can vary depending on the configuration of your Hive and Hadoop cluster. For large strings or binary data, it is generally recommended to use the BINARY data type, which is more suitable for handling large objects.

Binary Formats Support in Hive: Hive supports binary formats through the BINARY data type. The BINARY data type is used to store binary data, like images, audio files, or any other binary data that you don't want to process as text. Binary data is stored as a sequence of bytes and is not interpreted or modified by Hive.

Here's an example of how to create a table with a BINARY data type:

CREATE TABLE my\_binary\_table (

id INT,

file\_name STRING,

file\_data BINARY

);

In this example, the file\_data column uses the BINARY data type to store binary data.

To handle binary data in Hive, you might need to use a custom SerDe (Serializer/Deserializer) if the built-in SerDe doesn't support the specific binary format you're working with. Custom SerDe can be used to read and write binary data in a format that's suitable for your application's needs.

Remember that handling large binary data in Hive can lead to performance and memory

**55. What File Formats and Applications Does Hive Support?**

Apache Hive supports several file formats that can be used to store data. These file formats can be used based on specific use cases, the nature of the data, and performance requirements. Here are a few file formats supported by Hive:

* TextFile: This is the default file format. It is human-readable but can be less efficient for Hive to process compared to other formats.
* SequenceFile: This is a flat file format consisting of binary key-value pairs. It is widely used in Hadoop for its support of splitting (useful for MapReduce jobs).
* RCFile (Record Columnar File): This is a columnar storage file format optimized for Hive queries. It stores data in a columnar manner to allow for better compression and faster selective queries.
* ORC (Optimized Row Columnar): ORC improves upon RCFile by providing a more efficient way of storing and querying data. It supports advanced features like data compression, predicate pushdown, and bloom filters.
* Parquet: Parquet is another columnar storage file format available in Hive. It is often used with Apache Arrow for efficient in-memory processing.
* Avro: Avro is a row-based file format that supports complex data structures. It uses a schema to serialize and deserialize data, which can be included in the file or referenced separately.

In addition to these file formats, Hive can work with custom file formats using InputFormat and OutputFormat interfaces.

In terms of applications, Hive is typically used for data warehousing tasks on large datasets stored in Hadoop's HDFS or other compatible storage systems. It's used for running SQL-like queries (HiveQL), data summarization, and data analysis. Hive can also integrate with other Hadoop applications like HBase, MapReduce, and Spark.

**56. How do ORC format tables help Hive to enhance its performance?**

Optimized Row Columnar (ORC) format is a highly efficient columnar storage file format available in Hive. It significantly improves Hive's performance due to several reasons:

* **Columnar Storage:** As a columnar storage format, ORC stores data by columns, which allows for more efficient disk I/O, compression, and encoding. This is especially beneficial for analytical queries that fetch specific columns from a large dataset.
* **Compression:** ORC file format leverages different compression techniques which significantly reduce the storage footprint. This allows more data to be read from disk in a single operation, speeding up data retrieval.
* **Stripe Structure:** ORC files are divided into large blocks, or "stripes", each of which is around 250MB by default. These stripes are further divided into smaller row groups, enabling efficient reads of a subset of rows.
* **Predicate Pushdown:** This feature allows Hive to skip over unnecessary data blocks. If a query has a filter condition, Hive can apply this filter at the time of reading data, skipping blocks of data that do not satisfy the condition, leading to faster query execution.
* **Bloom Filters:** ORC supports Bloom filters, which are a probabilistic data structure that can be used to test whether an element is a member of a set. They can help speed up queries by quickly checking if a value exists in a set before doing a more expensive disk I/O operation.
* **Indexing:** Each ORC stripe contains a lightweight index that stores min/max values and the count of each column. This allows Hive to quickly skip over data blocks that do not meet the query criteria, further enhancing performance.
* **Schema Evolution:** ORC supports adding, dropping, or changing columns. This can be done without re-writing existing files, saving a significant amount of time and computational resources.

These features make ORC an excellent choice for a Hive table file format, especially when dealing with large-scale data analytics tasks.

**57. How can Hive avoid mapreduce while processing the query?**

Hive can avoid MapReduce while processing queries by using a feature known as "Hive on Tez" or "Hive on Spark", or by using the "Hive Optimized Row Columnar (ORC) format" along with "vectorization". Additionally, small queries can be executed in local mode without using MapReduce, and some operations can be performed by Hive directly on the metadata without scanning the actual data.

**Hive on Tez:** Apache Tez is an extensible framework for building high-performance batch and interactive data processing applications. It allows complex directed-acyclic-graph (DAG) of tasks for processing data, which can be more efficient than MapReduce's two-stage paradigm.

**Hive on Spark:** Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. When configured, Spark can be used as an execution engine instead of MapReduce for Hive queries.

**ORC Format and Vectorization:** The ORC file format provides a highly efficient way to store Hive data. It was designed to overcome limitations of the other Hive file formats. Vectorization in Hive allows operations to be performed on entire columns at once, which can significantly speed up query execution time.

**Local Mode:** For small datasets, Hive can run in local mode, which runs the job in a single JVM and doesn't incur the overhead of launching a separate MapReduce job.

**Direct Operations on Metadata:** Some Hive queries, such as SHOW TABLES, SHOW DATABASES, and DESCRIBE TABLE, can be executed by Hive directly on the metadata stored in the Metastore, without the need for a MapReduce job.

Remember that although these methods can improve the performance of certain types of queries, they may not be suitable for all use cases. The choice of execution engine or file format should be guided by your specific workload, data sizes, and performance requirements.

**58. What is view and indexing in hive?**

**Views in Hive:** Views in Hive are similar to views in traditional SQL databases. A view is a virtual or logical table based on the result set of a SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can create a view using the CREATE VIEW statement. For example:

CREATE VIEW view\_name AS

SELECT column1, column2

FROM table\_name

WHERE condition;

Views can be used for the following purposes:

To focus on specific data and hide the rest. This becomes handy when you have a table with many columns and you only care about a few of them.

To encapsulate complex SQL queries with joins, aggregations, and subqueries. This makes it easier to reuse those queries.

To provide an extra layer of data abstraction, which can be useful for security and simplicity.

However, unlike some traditional databases, views in Hive are read-only and you cannot modify a view's data directly.

**Indexing in Hive:** Indexing in Hive provides a way to accelerate queries. An index is a data structure that allows fast, random lookups and range scans. Hive uses indexes to speed up queries that often filter on a particular column or set of columns.

You can create an index on a table with the CREATE INDEX statement. For example:

CREATE INDEX index\_name

ON TABLE table\_name (column\_name)

AS 'index\_handler\_name'

[WITH DEFERRED REBUILD]

[IDXPROPERTIES (property\_name=property\_value, ...)]

[IN TABLE index\_table\_name]

[PARTITIONED BY (col\_name, ...)]

[COMMENT index\_comment]

[ROW FORMAT ...]

[STORED AS ...]

[LOCATION hdfs\_path]

[TBLPROPERTIES (...)]

Here, index\_handler\_name should be replaced with the name of the index handler that you want to use. Hive currently includes support for bitmap indexes and compact indexes.

However, as of my knowledge cutoff in September 2021, indexing in Hive is not commonly used because of limitations in its design and implementation. Often, other strategies are used to improve Hive query performance, such as partitioning and bucketing data, choosing an appropriate file format (like Parquet or ORC), or using columnar storage formats.

Please check the most recent Hive documentation or resources for any updates or improvements related to indexing in Hive.

**59. Can the name of a view be the same as the name of a hive table?**

No, in Hive, you cannot have a view and a table with the same name within the same database or schema. The naming rules follow a similar pattern to many other databases, where object names within the same schema must be unique.

When you create a view, Hive actually stores metadata about the view, similar to how it stores metadata about a table. Because of this, the view's name must be unique within the same database, just like a table name.

However, you could have a table and a view with the same name if they're in different databases or schemas. For example, you could have a table named my\_table in the default database, and a view named my\_table in a different database, such as test\_database.

**60. What types of costs are associated in creating indexes on hive tables?**

Extra Space: Indexes take up additional storage space on your system. Think of it like having a book (your data) and then creating a detailed index at the back. This index increases the size of the book.

Time and Effort (Computing Power): Creating an index takes time, especially if the table is large. It's like going through every word in a book to build the index. The computer has to go through all your data to create the index, which uses its resources and can slow down other tasks.

Keeping Up-to-Date: If you change the original data, the index needs to be updated too. This means every time you add, change, or remove data, the system also needs to update the index, which again takes time and computing power.

Query Performance Trade-offs: While indexes can speed up some types of data lookup, they can actually slow down others, especially when you're writing data. If you're often changing your data, the constant index updating can slow down these processes.

**61. Give the command to see the indexes on a table.**

SHOW INDEX ON database\_name.table\_name;

**62. Explain the process to access subdirectories recursively in Hive queries.**

In Hive, there's a property called hive.mapred.supports.subdirectories that allows you to enable recursive reads of directories. When this property is set to true, Hive will read all subdirectories recursively when a table or partition is stored in a directory.

Here's how you can set it up:

Firstly, you need to set two configuration properties:

mapred.input.dir.recursive

hive.mapred.supports.subdirectories

Both should be set to true for this feature to work. You can do this using the SET command in Hive:

SET mapred.input.dir.recursive=true;

SET hive.mapred.supports.subdirectories=true;

After setting these properties, you can run your query as usual. Hive will now read data from the directory of the table or partition and all its subdirectories recursively.

Keep in mind, as of my knowledge cut-off in September 2021, enabling these settings might have performance implications as the number of files and directories Hive needs to process can significantly increase, so it's better to use them only when necessary.

**63. If we run a select \* query in Hive, why doesn't it run MapReduce?**

While Hive has traditionally used MapReduce as its execution engine, over time newer execution engines like Tez and Spark have been developed and are often used in Hive because they can process data much faster for certain types of queries.

In the case of a simple SELECT \* FROM table\_name; query, depending on your setup, Hive might not even need to use any of these engines. If you're simply reading data and not doing any transformations or computations on it, Hive can bypass the need for an execution engine and directly stream data from HDFS (Hadoop Distributed File System) back to the client.

This kind of optimization is known as a "Fetch task" and helps to improve performance. When Hive determines that a query can be served by a Fetch task, it skips the MapReduce (or Tez, or Spark) stage entirely, which saves the overhead of job submission and scheduling, and can significantly speed up the query.

However, if your SELECT statement included computations like filtering, joins, aggregations, etc., Hive would need to use an execution engine (MapReduce, Tez, or Spark) to process the query.

It's worth mentioning that the specific behaviors might depend on your Hive version and configuration. As of my knowledge cutoff in September 2021, this is an accurate representation.

**64. What are the uses of Hive Explode?**

The EXPLODE function in Hive is used to convert complex data types into a relational form.

In Hive, you might sometimes have data stored in complex data types like arrays, maps, or structs. While this can be useful for some purposes, there are times when you want to treat each item in a complex data type individually. That's where EXPLODE comes in.

Here are some examples of how you might use EXPLODE:

* 1. **Arrays:** If you have a column that's an array of values, you can use EXPLODE to create a new row for each value in the array. For example, if you have a table of users and each user has an array of hobbies, you can use EXPLODE to create a new row for each hobby a user has. This can be useful for analyzing the data at the level of individual hobbies rather than at the level of users.

SELECT users.name, hobby

FROM users LATERAL VIEW EXPLODE(hobbies) hobbiesTable AS hobby;

* 1. **Maps:** Similar to arrays, if you have a column that's a map, you can use EXPLODE to create a new row for each key-value pair in the map.

SELECT users.name, hobby, frequency

FROM users LATERAL VIEW EXPLODE(hobbies\_frequency) hobbiesTable AS hobby, frequency;

* 1. **Structs:** While you cannot directly explode a struct in Hive, you can first convert it into an array or map and then explode it.

In all these cases, EXPLODE is a way to "flatten" complex data structures into a relational form that's easier to work with for certain types of queries and analyses. Keep in mind, though, that using EXPLODE can greatly increase the number of rows in your output, so be careful with its usage, especially on large datasets.

**65. What is the available mechanism for connecting applications when we run Hive as a server?**

When you run Hive as a server, there are several mechanisms available for connecting applications to it. The most common one is HiveServer2, which is a server interface that enables remote clients to execute queries against Hive and retrieve the results.

Here are some ways to connect to HiveServer2:

1. **JDBC (Java Database Connectivity):** This is a Java API that enables Java programs to execute SQL statements. This provides universal data access from the Java programming language. Using the JDBC driver provided by HiveServer2, client applications can connect to Hive and execute SQL operations.
2. **ODBC (Open Database Connectivity):** This is a standard API for accessing database management systems. Using the ODBC driver provided by HiveServer2, client applications written in languages like Python, R, or even Excel can connect to Hive.
3. **Thrift:** HiveServer2 is built on Apache Thrift, so applications can also connect to it using the Thrift protocol. Thrift supports a variety of programming languages including Java, C++, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml, Delphi, and others.
4. **Beeline:** This is a command line shell that you can use to interact with Hive. Beeline uses JDBC to connect to HiveServer2.

Hue (Hadoop User Experience): This is a web-based interactive query editor that supports Hive, among other tools in the Hadoop ecosystem. It connects to HiveServer2 for query execution.

**66. Can the default location of a managed table be changed in Hive?**

Yes, the default location of managed tables (also known as internal tables) in Hive can be changed, but doing so involves modifying Hive's configuration settings.

By default, Hive stores the data for managed tables in a directory within the Hive warehouse directory. This location is defined by the configuration property hive.metastore.warehouse.dir in the hive-site.xml file.

To change the default location for new managed tables, you would update this hive.metastore.warehouse.dir property to point to the new location. Please note, changing this property only affects new tables that are created after the change; existing tables will remain in their current locations.

Here is how you can change it in hive-site.xml:

<property>

<name>hive.metastore.warehouse.dir</name>

<value>/new/path/to/hive/warehouse</value>

<description>location of default warehouse for managed tables</description>

</property>

After making changes to the hive-site.xml file, you will need to restart Hive services for the changes to take effect.

Keep in mind that changing the warehouse directory could have security and performance implications, and should be done with careful consideration and proper testing. It's also important to ensure that the new location is correctly configured in terms of access permissions and available storage.

Remember that as of my knowledge cutoff in September 2021, the way you manage and configure Hive might vary depending on your specific setup, such as whether you're using a cloud-based service, a specific distribution of Hadoop, etc. Always check the most recent and relevant documentation for your setup.

**67. What is the Hive ObjectInspector function?**

In Apache Hive, the ObjectInspector is an interface that provides a uniform way to access complex data types stored in Hive, such as lists, maps, and structs. Hive uses ObjectInspectors to analyze the internal structure of row objects and also the structure of individual columns and fields.

Hive's ObjectInspectors are important for several reasons:

1. **Type Information:** ObjectInspectors can provide the data type of an object. This is important in Hive because it deals with various complex data types, and it needs to know how to interpret these data types correctly.
2. **Object Access:** ObjectInspectors can access the fields inside an object. For complex types like structs, maps, and arrays, ObjectInspectors can retrieve the data in individual fields or elements.
3. **Object Creation:** ObjectInspectors can create new instances of an object. Hive often needs to create new objects, for example, as the output of a MapReduce task.
4. **Interoperability:** ObjectInspectors allow Hive to work with data stored in various file formats, like TextFile, SequenceFile, Avro, ORC, Parquet, etc. Each file format can provide its own implementation of ObjectInspector that knows how to handle data in that format.
5. **Extensibility:** Because each data type or file format can provide its own implementation of ObjectInspector, it's easy to extend Hive to support new data types or file formats. All you need to do is provide an ObjectInspector (and SerDe, for serialization and deserialization) for the new type or format.

So, in summary, ObjectInspectors are a key part of how Hive handles complex and varied data types and formats, providing flexibility and extensibility.

**68. What is UDF in Hive?**

UDF stands for User-Defined Function in Hive. It's a way for users to define their own functions in programming languages like Java and then use them in SQL statements.

Hive comes with many built-in functions for common operations like string manipulation, date manipulation, and mathematical calculations. However, there might be cases where you need to perform operations that aren't covered by the built-in functions. This is where UDFs come in handy.

There are three types of User-Defined Functions you can create in Hive:

* 1. UDF (User-Defined Functions): These are the simplest kind of custom functions, taking one or more inputs and returning a single output.
  2. UDAF (User-Defined Aggregate Functions): These are functions that perform a calculation on multiple rows and return a single result, similar to built-in aggregate functions like SUM or MAX.
  3. UDTF (User-Defined Table-Generating Functions): These functions can output multiple rows for each input row, similar to the built-in EXPLODE function.

To create a UDF, you generally need to write a Java class that extends one of the UDF classes in the Hive API, then compile it into a JAR file, add the JAR file to Hive's classpath, and then create a function in Hive that points to the Java class.

1. **Here's a simple example of how to use a UDF:**

Create the UDF in Java and compile it into a JAR file.

1. **Add the JAR file to Hive:**

ADD JAR /path/to/my\_udf.jar;

1. **Create a function that points to your UDF:**

CREATE TEMPORARY FUNCTION my\_udf AS 'com.example.MyUDF';

1. **Now you can use your UDF in a query:**

SELECT my\_udf(some\_column) FROM some\_table;

Please note that creating and using UDFs requires knowledge of Java and an understanding of the Hive API. Also, because UDFs are executed on the Hadoop cluster, they can significantly impact performance, and need to be carefully tested and optimized.

**69. Write a query to extract data from hdfs to hive.**

In Hive, you don't typically "extract" data from HDFS in the sense you would extract data from a traditional database or a data source. Instead, you create a table in Hive that is linked to your data in HDFS. This allows you to query the data using Hive's SQL-like interface while the data continues to reside in HDFS.

**To do this, you use a CREATE EXTERNAL TABLE statement in Hive, which might look something like this:**

CREATE EXTERNAL TABLE my\_table

(

col1 INT,

col2 STRING,

col3 DATE

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

LOCATION 'hdfs://namenode/path/to/my/data';

Here's what this does:

* 1. CREATE EXTERNAL TABLE my\_table (...): This creates a new external table in Hive called my\_table.
  2. The part in parentheses (col1 INT, col2 STRING, col3 DATE) defines the schema of your table. You need to replace this with the actual schema of your data.
  3. ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n': This tells Hive how to parse your data. This example assumes your data is in a CSV format, with fields separated by commas and lines separated by newlines. If your data is in a different format, you will need to adjust this accordingly.
  4. LOCATION 'hdfs://namenode/path/to/my/data': This tells Hive where your data is located in HDFS. Replace this with the actual path to your data.

Once you've run this CREATE EXTERNAL TABLE statement, you can query your data in HDFS using Hive. For example, you could do SELECT \* FROM my\_table; to retrieve all data.

**70. What is TextInputFormat and SequenceFileInputFormat in hive?**

TextInputFormat and SequenceFileInputFormat are classes in Hadoop that define how input data should be split up and read for MapReduce processing. They both implement the InputFormat interface in Hadoop.

When you're working with Hive, these classes (and others like them) play a crucial role in helping Hive read data from Hadoop's distributed file system (HDFS).

**Here's a brief description of both:**

**TextInputFormat:** This is the default InputFormat in Hadoop. It reads data line by line. Each line is considered a record, and the output key-value pair is the position in the file (as LongWritable), and the line of text (as Text).

When used in Hive, this is typically used with text data, such as CSV or TSV files. The data is considered as 'delimited' data with fields and lines represented by user-specified delimiters.

**SequenceFileInputFormat:** This InputFormat reads data from SequenceFiles, a flat-file format used in Hadoop. Each record in a SequenceFile consists of a binary key-value pair.

The SequenceFileInputFormat is often used in cases where you have data that was previously the output of a MapReduce job, as MapReduce jobs often output data in SequenceFile format. SequenceFiles are also a good choice for data that needs to be compressed, as they support block-level compression.

In general, you would choose the InputFormat based on the type and format of the data you're working with. Other InputFormats you might encounter in Hive include AvroInputFormat, OrcInputFormat, ParquetInputFormat, etc., each of which is designed to handle a specific type of data.

While these classes primarily concern how data is read for processing in a MapReduce context, in Hive they also have implications for how data is read for queries, as well as how data is written when inserted into Hive tables.

**71. How can we prevent a large job from running for a long time in a hive? 72.When do we use explode in Hive?**

**Preventing large jobs from running for a long time in Hive:** There are several ways to prevent a large job from running for a long time in Hive.

**Limiting the data:** Using the LIMIT keyword, you can restrict the amount of data returned by the query.

**Writing efficient queries:** Avoid using "SELECT \*", instead select only the required columns. Also, make sure to filter data as early as possible using the WHERE clause.

**Partitioning and Bucketing:** By partitioning and bucketing tables in Hive, you can divide your table into smaller, more manageable parts. Partitioning can greatly speed up queries that test the partition column.

**Indexes:** Indexing can help speed up queries. But creating an index has its own cost, so use it judiciously.

**Using appropriate file formats:** Some file formats, like Parquet or ORC, allow faster data retrieval than others. They provide features like data compression and columnar storage.

**Usage of EXPLODE in Hive:** The EXPLODE function is used to transform a single column of array or map type into multiple rows, one for each element in the array or map.

**Here's a simple example:** let's say you have a table called user\_hobbies that has a column hobbies which is an array of strings. If you want to create a new row for each hobby a user has, you could use EXPLODE like this:

SELECT user\_id, hobby

FROM user\_hobbies

LATERAL VIEW EXPLODE(hobbies) exploded\_table AS hobby;

This will return a table where each row contains a user\_id and a single hobby. So if a user has multiple hobbies, there will be multiple rows for that user, one for each hobby.

EXPLODE is useful for transforming complex data types into a format that's easier to analyze using SQL.

**73. Can Hive process any type of data formats? Why? Explain in very detail**

Hive can handle a variety of data formats, both structured and unstructured, because it operates on top of the Hadoop ecosystem, which is designed to work with all types of data. However, to perform operations on the data in an efficient and meaningful way, the data needs to be structured in a way that Hive can understand.

* 1. **There are three primary types of file formats Hive can work with**: text files, sequence files, and other special file formats such as Avro, Parquet, ORC, etc.
  2. **Text files:** Hive can read and write data in text format. When using text files, data fields are typically separated by some delimiter like a comma, tab, or other special character.
  3. **Sequence files:** Sequence files are a flat file format consisting of binary key/value pairs. Hive can read this format, and it's often used for intermediate outputs of MapReduce jobs.
  4. **Special file formats (Avro, Parquet, ORC, and others):** These file formats are often used because they offer better performance characteristics compared to plain text or sequence files.

Avro is a language-neutral data serialization system. It's compact, fast, and binary, which makes it superior to text formats. Avro files contain the schema for the data they hold, which makes them self-describing.

Parquet is a columnar storage file format optimized for use with big data processing frameworks like Hadoop, Hive, and Impala. The columnar nature of Parquet means it can offer better performance on queries that retrieve a subset of the columns in the table.

ORC (Optimized Row Columnar) is a highly efficient way to store Hive data. It was developed specifically for the Hive ecosystem and provides superior compression and improvements in speed for Hive queries.

The ability to read and process these various file formats allows Hive to work with a broad range of data types and structures. However, no matter what the underlying data format is, the data is presented to the user as a set of tables with rows and columns when queried with Hive.

In order to read and write data in these various formats, Hive uses a component called a SerDe (Serializer/Deserializer). A SerDe is responsible for translating data from the file format on disk into a format that Hive can process (rows and columns), and vice versa. Hive provides default SerDes for some file formats, but you can also create custom SerDes if you have unique needs.

So, while Hive can work with a variety of file formats, how it interprets and processes the data within those files depends on the SerDe and InputFormat/OutputFormat specified. Each file format has its own trade-offs in terms of read/write speed, storage efficiency, schema evolution, and so on, so the optimal format to use depends on the specific use case and requirements.

**74.Whenever we run a Hive query, a new metastore\_db is created. Why?**

The metastore\_db is the Hive Metastore, which is a central repository that stores metadata for Hive tables, such as their schema and location. It also includes metadata about partitions, if the tables are partitioned.

When Hive runs, it needs to connect to the Metastore service to fetch this metadata. Hive uses a Derby database for its metastore by default, and this database is stored in a directory called metastore\_db.

Now, a new metastore\_db should not be created every time you run a Hive query. If this is happening, it might be because you're running Hive in embedded mode, which is its default mode. In embedded mode, the Metastore service runs in the same JVM (Java Virtual Machine) as the Hive service, and it connects to a standalone Derby database that only allows one connection at a time.

The problem with this is that if the JVM shuts down after a query, it can leave the Derby database in an inconsistent state. Then, when you start a new Hive session, it might not be able to connect to the existing Derby database and so it creates a new one.

To avoid this issue, you can run Hive in one of the server modes:

* **Local mode:** The Metastore service and the Hive service run in separate JVMs, but they both run on the same machine. They connect to a standalone Derby database.
* **Remote mode:** The Metastore service runs on a separate machine from the Hive service. It can connect to a standalone Derby database or to a traditional database like MySQL or PostgreSQL that can handle multiple connections.

Running Hive in server mode, particularly remote mode, with a traditional database like MySQL or PostgreSQL for the Metastore, is recommended for production environments to ensure stability and reliability. In these modes, the Metastore service should remain up and running consistently, so a new metastore\_db won't be created with each query.

**75. Can we change the data type of a column in a hive table? Write a complete query.**

Hive's ALTER TABLE can be used to add columns, replace columns, rename columns, but it does not support modifying the data type of an existing column.

However, there are workarounds to achieve this. You can create a new temporary table with the desired data type and copy the data over from the existing table. Here's a basic example:

1. **Let's assume we have a table named my\_table:**

CREATE TABLE my\_table (

col1 INT,

col2 STRING

);

1. **You want to change the data type of col1 from INT to STRING. First, create a new temporary table:**

CREATE TABLE temp\_table (

col1 STRING,

col2 STRING

);

1. **Then, insert data from my\_table into temp\_table, casting col1 to a STRING in the process:**

INSERT OVERWRITE TABLE temp\_table SELECT CAST(col1 AS STRING), col2 FROM my\_table;

1. **If you want to preserve the original table name, you can then rename my\_table to something else and rename temp\_table to my\_table:**

ALTER TABLE my\_table RENAME TO old\_table;

ALTER TABLE temp\_table RENAME TO my\_table;

1. **Finally, if you don't need the original table with the old data types, you can drop it:**

DROP TABLE old\_table;

In this way, even though you can't directly change a column's data type in Hive, you can achieve the desired outcome with a few extra steps. Please note that this process can be time-consuming and resource-intensive for large tables, as it involves copying all the data from the original table to a new table.

**76. While loading data into a hive table using the LOAD DATA clause, how do we specify it is a hdfs file and not a local file?**

When you use the LOAD DATA clause in Hive, you specify whether the file is local or in HDFS with the LOCAL keyword.

* If you're loading data from a file in HDFS, you simply omit the LOCAL keyword. Here is an example:

LOAD DATA INPATH 'hdfs://localhost:9000/user/hadoop/mydata.txt' INTO TABLE my\_table;

In this example, 'hdfs://localhost:9000/user/hadoop/mydata.txt' is the HDFS file path.

* If you were loading from a local file, you would use the LOCAL keyword like this:

LOAD DATA LOCAL INPATH '/home/hadoop/mydata.txt' INTO TABLE my\_table;

In this example, '/home/hadoop/mydata.txt' is the local file path.

Remember, the LOAD DATA statement moves the data file into Hive's control and the original file is deleted if it is local. If the data file is in HDFS, then the file's ownership and permissions are changed to Hive's. Be sure to make a copy of your data if you want to keep the original intact.

**77. What is the precedence order in Hive configuration?**

Hive configuration settings are set in multiple places, and the order of precedence from highest (takes precedence) to lowest (overridden by higher levels) is generally as follows:

* **Command line options:** These take the highest precedence. Anything you specify at the command line when starting Hive (using the -hiveconf option) will override all other settings. For example: hive -hiveconf hive.exec.max.dynamic.partitions=1000.
* **Hive CLI Session parameters:** This is where you set the properties in your Hive session using the SET command. This will override the values set in configuration files but can be overridden by command line options.
* **Hive-site.xml in the classpath:** This file is usually located in the Hive's conf directory, and settings here will override those in hive-default.xml.template. But these settings can be overridden by the session parameters and command line options.
* **Hive-default.xml.template:** This file is a template and it contains the default values of all Hive configuration parameters. It's located in the conf directory of the Hive installation. It's usually not meant to be edited, as any changes here can be overridden by hive-site.xml.
* Hardcoded defaults: These are the defaults that are used when no other configuration is specified.

Remember that Hive configuration can be complex, with parameters set at many levels, and some parameters might not follow this exact order of precedence. But, in general, this order is a good guide to how Hive determines the values of its configuration parameters.

**78. Which interface is used for accessing the Hive metastore?**

Hive's metadata is stored in a relational database, and this metadata is accessed through the Hive Metastore service.

The metastore service can run in one of the following modes:

* **Embedded mode (also known as local mode):** In this mode, the metastore service runs in the same JVM (Java Virtual Machine) as the Hive service and accesses the metadata in the relational database directly. The underlying database is accessed using an embedded Derby database instance. This mode is typically used for simple, single-user applications and not recommended for production setups.
* **Local mode:** In this mode, the metastore service runs in the same JVM as the Hive service, but it connects to a standalone relational database such as MySQL or PostgreSQL. The standalone relational database is expected to handle concurrency, unlike the embedded Derby database.
* **Remote mode (also known as standalone mode):** In this mode, the metastore service runs in its own JVM, separate from the Hive service. The metastore service can be accessed by multiple Hive services or other applications (like Spark, Impala, etc.) concurrently. This mode is the most common for production setups.

The Hive Metastore provides a Thrift API, which is used by Hive and other services to interact with the metastore. This is the primary interface used to access the Hive metastore, regardless of which mode the metastore service is running in.

In all modes, the metastore service uses a JDBC (Java Database Connectivity) connection to connect to the underlying relational database where the metadata is stored.

**79. Is it possible to compress json in the Hive external table?**

Yes, Hive does support reading compressed data. When you create an external table, the data is stored on HDFS (Hadoop Distributed File System) and it can be compressed there. Hive will automatically decompress the data when reading it, as long as the correct compression codec is available.

If you want to store JSON data in a compressed format, you can do so using a compression codec like gzip, snappy, bzip2, etc. You just need to ensure the data is compressed using one of the codecs supported by your Hadoop system and the file extension should match the type of compression.

* **For example, if you have JSON data compressed with gzip, you could create an external table like this:**

CREATE EXTERNAL TABLE my\_table (json string)

ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'

LOCATION '/path/to/your/data';

In this case, you would place your gzip-compressed JSON files (\*.json.gz) into the /path/to/your/data directory in HDFS.

Please remember that Hive doesn't support compression at the level of individual columns or data types; it's done at the file level. Also, when writing to tables (not applicable to external tables), the compression is dictated by settings such as hive.exec.compress.output and mapreduce.output.fileoutputformat.compress.

**80. What is the difference between local and remote metastores?**

The primary difference between a local and remote metastore is where the metastore service is run.

* **Local Metastore: In** this configuration, the metastore service and Hive service run in the same JVM (Java Virtual Machine). The metastore service directly accesses the metastore database using JDBC. Note that the term "local" in this context doesn't mean that the metastore database itself is on the same machine as the Hive service; the database can be (and often is) on a separate machine. This mode is generally used for development or testing environments, or simple single-user applications.
* **Remote (or Standalone) Metastore:** In this configuration, the metastore service runs in its own JVM, separate from the Hive service. This allows multiple Hive services, as well as other applications, to connect to the metastore service concurrently. The metastore service is essentially acting as a server that clients connect to via the Thrift network API. This configuration is much more scalable and is the typical choice for production environments.

In both configurations, the metastore service is what interacts directly with the metastore database, but the distinction is whether the metastore service is running in the same process as the Hive service (local) or a separate process (remote). The choice between local and remote will depend on your specific needs for scalability, concurrency, and isolation.

**81. What is the purpose of archiving tables in Hive?**

Archiving in Hive, also referred to as "Hive Data Archiving," is a strategy used for efficient Hadoop storage and query performance. The main purposes of archiving tables in Hive are:

* **Storage Efficiency:** Hadoop's HDFS is not efficient for storing small files because each file, directory, and block in HDFS is represented as an object in the NameNode's memory, each of which occupies approximately 150 bytes. This might not sound like much, but with billions of files, it can quickly fill up the NameNode's memory. Archiving small files into larger files (HAR - Hadoop Archive files) can significantly reduce the memory footprint on the NameNode.
* **Query Performance:** Small files can negatively impact query performance in Hive. MapReduce, which is often used to process data in Hadoop, creates one map task for each file. When you have lots of small files, it results in a large number of map tasks that process little data, and the overhead of task management can make the job slower. By archiving small files into larger ones, the number of map tasks required for processing can be reduced, improving the performance of Hive queries.
* **Data Archival:** If you have older data that is not accessed often but you still need to retain for compliance or other reasons, you can archive it. This allows you to still have the data available without it taking up as much space in your active HDFS storage.

To archive a table in Hive, you generally use the Hadoop Archive tool (hadoop archive) to create a HAR file from your table's files, and then create a new Hive table that points to the HAR file.

However, it's important to note that accessing data from HAR files is slower than accessing data from regular HDFS files. As a result, data archiving is generally best used for data that is not accessed frequently.

**82. What is DBPROPERTY in Hive?**

* **DBPROPERTY is a Hive function that returns the value of a property from a given database. This function takes two parameters:**

1. The name of the database
2. The name of the property you want to retrieve

* **The following is the syntax for using the DBPROPERTY function:**

DBPROPERTY('database\_name', 'property\_name')

* **For example, if you wanted to get the owner of the 'my\_database' database, you would use:**

SELECT DBPROPERTY('my\_database', 'ownerName');

* **DBPROPERTY supports the following property names:**
* ownerName
* ownerType
* createTime

Keep in mind that the availability and functioning of DBPROPERTY and other Hive features can depend on the version of Hive that you're using, so it's a good idea to consult the documentation for your specific version if you have any questions or encounter any issues.

**83. Differentiate between local mode and MapReduce mode in Hive**

| **Feature** | **Local Mode** | **MapReduce Mode** |
| --- | --- | --- |
| **Execution Environment** | Everything runs on a single JVM, doesn't use Hadoop MapReduce or Tez frameworks | Translates SQL queries into a series of MapReduce jobs to be executed on a Hadoop cluster |
| **Data Size** | Suitable for small amounts of data | Suitable for large datasets |
| **Processing** | Processes data sequentially | Processes data in parallel across multiple nodes in the cluster |
| **Storage** | Data can be stored locally | Data is stored in Hadoop Distributed File System (HDFS) |
| **Performance** | Fast for small datasets as there is no overhead of launching MapReduce jobs | Can be slower for small datasets due to overhead, but significantly faster for large datasets due to parallel processing |
| **Scalability** | Limited scalability, depends on the single machine's resources | Highly scalable, can handle large datasets spread across the Hadoop cluster |
| **Use Case** | Useful for testing and development purposes | Suitable for production environments handling large datasets |