**Hive interview Questions & Answers**

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

Apache Hive is an **open-source data warehousing tool** for performing distributed processing and data analysis. It was developed by **Facebook** to reduce the work of writing the Java MapReduce program.

Apache Hive uses a **Hive Query language**, which is a declarative language similar to SQL. Hive translates the hive queries into MapReduce programs.

It supports developers to perform processing and analyses on structured and semi-structured data by replacing complex java MapReduce programs with hive queries.

One who is familiar with SQL commands can easily write the hive queries.

Hive makes the job easy for performing operations like

* + Analysis of huge datasets
  + Ad-hoc queries
  + Data encapsulation

present version of Hive: **hive-4.0.0-alpha-1**

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

No.

Hive is mainly used for batch processing i.e. OLAP and it is not used for OLTP because of the realtime operations of the database.

OLAP – online analytical process- data processing toanalyse large volumes of data – analysing customer preferences based on orders placed

OLTP – online transactional process – designed to support transaction related operation – ticket booking

Hive is not suitable for OLTP because of its architecture. Hive is built on top of HDFS and uses map reduce for processing which is used for batch processing and analytical operations on large datasets. It is designed for data warehousing purpose and to store and process high volumes of data.

On the other hand OLTP systems require high speed data processing, efficient transactional operations , frequent read and write operations.

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

|  |  |
| --- | --- |
| **RDBMS** | **HIVE** |
| It is used to maintain database. | It is used to maintain data warehouse. |
| It uses SQL (Structured Query Language). | It uses HQL (Hive Query Language). |
| Schema is fixed in RDBMS. Schema on write. Data is structured, schema is checked before insertion in table. | Schema varies in it. Schema on read. Data is unstructured or semi structured ,schema is checked at the time of query execution. |
| Normalized data is stored. | Normalized and de-normalized both type of data is stored. |
| Used for transactional operations requires ACID properties and querying structured data, | Querying analytical queries on large datasets  Requires high throughput and parallel processing capabilities |
| Vertical scaling – more powerful cpu | Horizontal scaling – more nodes on Hadoop cluster |
| expensive to maintain and scale | Cost efficient |

Older versions of Hive doesn’t support ACID transactions on tables. Though in newer versions it supports by default ACID transactions are disabled and you need to enable it before start using it.

Below are the properties you need to enable ACID transactions.

SET hive.support.concurrency=true;

SET hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;

# The follwoing are not required if you are using Hive 2.0

SET hive.enforce.bucketing=true;

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

# The following parameters are required for standalone hive metastore

SET hive.compactor.initiator.on=true;

SET hive.compactor.worker.threads=1

Besides this, you also need to create a Transactional table by using TBLPROPERTIES (‘transactional’=’true’) at the time of creating Managed table.

Managed Table should use ORC format.

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

The major components of Hive and its interaction with the Hadoop is demonstrated in the figure below and all the components are described further:

# • User Interface (UI) –

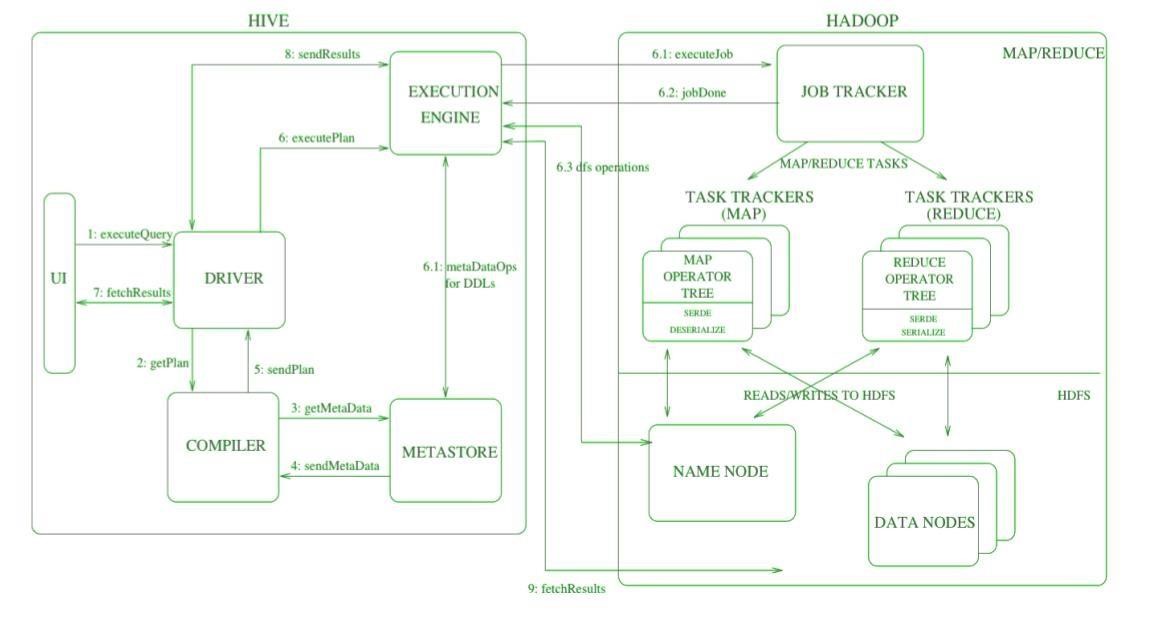
As the name describes User interface provide an interface between user and hive. It enables user to submit queries and other operations to the system. Hive web UI, Hive command line, and Hive HD Insight (In windows server) are supported by the user interface.

* **Hive Server** – It is referred to as Apache Thrift Server. It accepts the request from different clients and provides it to Hive Driver.
* **Driver** – Queries of the user after the interface are received by the driver within the Hive. Concept of session handles is implemented by driver. Execution and Fetching of APIs modelled on JDBC/ODBC interfaces is provided by the user.
* **Compiler** – Queries are parses, semantic analysis on the different query blocks and query expression is done by the compiler. Execution plan with the help of the table in the database and partition metadata observed from the metastore are generated by the compiler eventually.

* **Metastore –** All the structured data or information of the different tables and partition in the warehouse containing attributes and attributes level information are stored in the metastore. Sequences or de-sequences necessary to read and write data and the corresponding HDFS files where the data is stored. Hive selects corresponding database servers to stock the schema or Metadata of databases, tables, attributes in a table, data types of databases, and HDFS mapping.

* **Execution Engine –** Execution of the execution plan made by the compiler is performed in the execution engine. The plan is a DAG of stages. The dependencies within the various stages of the plan is managed by execution engine as well as it executes these stages on the suitable system components

.



**Diagram –** Architecture of Hive that is built on the top of Hadoop

In the above diagram along with architecture, *job execution flow in Hive with Hadoop is demonstrated step by step*.

# • Step-1: Execute Query –

Interface of the Hive such as Command Line or Web user interface delivers query to the driver to execute. In this, UI calls the execute interface to the driver such as ODBC or JDBC.

# • Step-2: Get Plan –

Driver designs a session handle for the query and transfer the query to the compiler to make execution plan. In other words, driver interacts with the compiler.

# • Step-3: Get Metadata –

In this, the compiler transfers the metadata request to any database and the compiler gets the necessary metadata from the metastore.

• **Step-4: Send Metadata –**

Metastore transfers metadata as an acknowledgment to the compiler.

# • Step-5: Send Plan –

Compiler communicating with driver with the execution plan made by the compiler to execute the query.

# • Step-6: Execute Plan –

Execute plan is sent to the execution engine by the driver.

* Execute Job
* Job Done
* Dfs operation (Metadata Operation)
* **Step-7: Fetch Results –**

Fetching results from the driver to the user interface (UI).

# • Step-8: Send Results –

Result is transferred to the execution engine from the driver. Sending results to Execution engine. When the result is retrieved from data nodes to the execution engine, it returns the result to the driver and to user interface (UI).

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

Hive query processor is responsible for executing the generated plan on the Hadoop cluster. It coordinates execution of map reduce jobs, tracks the progress of query and handles errors that occur during query execution.

Following are the components of a Hive Query Processor:

* + Parse and Semantic Analysis (ql/parse)
  + Metadata Layer (ql/metadata)
  + Type Interfaces (ql/typeinfo)
  + Sessions (ql/session)
  + Map/Reduce Execution Engine (ql/exec)
  + Plan Components (ql/plan)
  + Hive Function Framework (ql/udf)
  + Tools (ql/tools)
  + Optimizer (ql/optimizer)

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

1. Local mode: In local mode hive runs on a single machine and the data is read and written to local file system. This mode is primarily used for testing and debugging small datasets or HiveQL queries. In local mode hive does not require Hadoop cluster or distributed file system. Instead Hive uses local system to store data and metadata. It uses derbyDB.
2. MapReduce mode:

Here hive runs on top of Hadoop in a distributed mode. Hive generates MapReduce jobs for each query and submits them to the Hadoop cluster for execution. This mode is suitable for large scale data processing and analysis. Data stored in HDFS is processed in parallel by multiple nodes in the Hadoop cluster.

1. Tez mode:

Tez is an alternative execution engine for hive that provides a faster and more efficient way of executing than MapReduce. In Tez Mode, hive generates DAG – direct acyclic graph of the query and submits it to the Tez execution engine for execution. This mode is suitable for large-scale data processing and analysis where performance is critical.

Hive can also run as a combination of map reduce and Tez mode called Hybrid Execution Mode, and it allows Hive to switch between MapReduce and Tez execution engines based on the query type and data size. Overall, the choice of mode depends on the size of the data, the complexity of the query, and the performance requirements. Local mode is suitable for small-scale testing and hive MapReduce and Tez modes are ideal for large-scale data processing and analysis.

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

* **Features:**

|  |  |
| --- | --- |
| **Features** | **Explanation** |
| Supported Computing Engine | Hive supports MapReduce, Tez, and Spark computing engine. |
| Framework | Hive is a stable batch-processing framework built on top of the Hadoop Distributed File system and can work as a data warehouse. |
| Easy To Code | Hive uses HIVE query language to query structure data which is easy to code. The 100 lines of java code we use to query a structure data can be minimized to 4 lines with HQL. |
| Declarative | HQL is a declarative language like SQL means it is non-procedural. |
| Structure Of Table | The table, the structure is similar to the RDBMS. It also supports partitioning and bucketing. |
| Supported data structures | Partition, Bucket, and tables are the 3 data structures that hive supports. |
| Supports ETL | Apache hive supports ETL i.e. Extract Transform and Load. Before Hive python is used for ETL. |
| Storage | Hive supports users to access files from HDFS, Apache HBase, Amazon S3, etc. |
| Capable | Hive is capable to process very large datasets of Petabytes in size. |
| Helps in processing unstructured data | We can easily embed custom MapReduce code with Hive to process unstructured data. |
| Drivers | JDBC/ODBC drivers are also available in Hive. |
| Fault Tolerance | Since we store Hive data on HDFS so fault tolerance is provided by Hadoop. |
| Area of uses | We can use a hive for data mining, predictive modelling, and document indexing. |

* **Limitations:**

|  |  |
| --- | --- |
| **Limitation** | **Explanation** |
| Does not support OLAP | Apache Hive doesn’t support online transaction  processing (OLTP) but Online Analytical Processing(OLAP) is supported. |
| No updation and Deletion | Hive does not support update and delete operation on tables. |
| Doesn’t support subqueries | Subqueries are not supported. |
| Latency | The latency in the apache hive query is very high. |
| Only non-real or cold data is supported | Hive is not used for real-time data querying since it takes a while to produce a result. |
| Transaction processing is not supported | HQL does not support the Transaction processing feature. |

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

Create Database is a statement used to create a database in Hive. A database in Hive is a **namespace** or a collection of tables. The **syntax** for this statement is as follows:

# CREATE DATABASE|SCHEMA [IF NOT EXISTS] <database name>

Here, IF NOT EXISTS is an optional clause, which notifies the user that a database with the same name already exists. We can use SCHEMA in place of DATABASE in this command. The following query is executed to create a database named **archi**:

**hive> CREATE DATABASE [IF NOT EXISTS] archi; OR hive> CREATE SCHEMA archi;**

The following query is used to verify a databases list:

# hive> SHOW DATABASES;

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

create table customers

(

id int,

name string, age int,

address string,

salary int

)

row format delimited fields terminated by ','

tblproperties ("skip.header.line.count" = "1");

1. **What do you mean by describe and describe extended and describe formatted with respect to database and table.**

**Describe**- This will show table columns

**Describe extended** - This will show table columns, data types, and other details of the table. Other details will be displayed in single line.

**Describe formatted** - This will show table columns, data types, and other details of the table. Other details will be displayed into multiple lines.

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

TBLPROPERTIES("skip.header.line.count"="1");

ALTER TABLE tablename

SET TBLPROPERTIES ("skip.header.line.count"="1");

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

**Apache Hive** provides various Built-in operators for data operations to be implemented on the tables present inside Apache Hive warehouse.

**Hive operators** are used for mathematical operations on operands. It returns specific value as per the logic applied.

Types of Hive Built-in Operators

* + **Relational Operators**
  + **Arithmetic Operators** • **Logical Operators**

# • String Operators • Operators on Complex Types

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

Hive supports the following built-in functions:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Return Type** | **Signature** | **Description** |
| BIGINT | round(double a) | It returns the rounded BIGINT value of the double. |
| BIGINT | floor(double a) | It returns the maximum BIGINT value that is equal or less than the double. |
| BIGINT | ceil(double a) | It returns the minimum BIGINT value that is equal or greater than the double. |
| double | rand(), rand(int seed) | It returns a random number that changes from row to row. |
| string | concat(string A, string  B,...) | It returns the string resulting from concatenating B after A. |
| string | substr(string A, int start) | It returns the substring of A starting from start position till the end of string A. |
| string | substr(string A, int start, int length) | It returns the substring of A starting from start position with the given length. |
| string | upper(string A) | It returns the string resulting from converting all characters of A to upper case. |
| string | ucase(string A) | Same as above. |
| string | lower(string A) | It returns the string resulting from converting all characters of B to lower case. |
| string | lcase(string A) | Same as above. |

|  |  |  |
| --- | --- | --- |
| string | trim(string A) | It returns the string resulting from trimming spaces from both ends of A. |
| string | ltrim(string A) | It returns the string resulting from trimming spaces from the beginning (left hand side) of A. |
| string | rtrim(string A) | rtrim(string A) It returns the string resulting from trimming spaces from the end (right hand side) of A. |
| string | regexp\_replace(string A, string B, string C) | It returns the string resulting from replacing all substrings in B that match the Java regular expression syntax with C. |
| int | size(Map<K.V>) | It returns the number of elements in the map type. |
| int | size(Array<T>) | It returns the number of elements in the array type. |
| value of <type> | cast(<expr> as <type>) | It converts the results of the expression expr to <type> e.g. cast('1' as BIGINT) converts the string '1' to it integral representation. A NULL is returned if the conversion does not succeed. |
| string | from\_unixtime(int unixtime) | convert the number of seconds from Unix epoch  (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of  "1970-01-01 00:00:00" |
| string | to\_date(string timestamp) | It returns the date part of a timestamp string: to\_date("1970-01-01 00:00:00") = "1970-01-01" |
| int | year(string date) | It returns the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970 |
| int | month(string date) | It returns the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11 |
| int | day(string date) | It returns the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1 |
| string | get\_json\_object(string json\_string, string path) | It extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It returns NULL if the input json string is invalid. |

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

Hive DDL commands:

* 1. CREATE
  2. SHOW
  3. DESCRIBE
  4. USE
  5. DROP
  6. ALTER
  7. TRUNCATE

Hive DML commands:

* 1. [LOAD](https://data-flair.training/blogs/hive-dml-commands/#LOAD-command)
  2. [SELECT](https://data-flair.training/blogs/hive-dml-commands/#SELECT-command)
  3. [INSERT](https://data-flair.training/blogs/hive-dml-commands/#INSERT-command)
  4. [DELETE](https://data-flair.training/blogs/hive-dml-commands/#DELETE-command)
  5. [UPDATE](https://data-flair.training/blogs/hive-dml-commands/#UPDATE-command)
  6. [EXPORT](https://data-flair.training/blogs/hive-dml-commands/#EXPORT-command)
  7. [IMPORT](https://data-flair.training/blogs/hive-dml-commands/#IMPORT-command)

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

* **ORDER BY x:** guarantees global ordering, but does this by pushing all data through just one reducer. This is basically unacceptable for large datasets. You end up one sorted file as output.
* **SORT BY x:** orders data at each of N reducers, but each reducer can receive overlapping ranges of data. You end up with N or more sorted files with overlapping ranges.
* **DISTRIBUTE BY x:** ensures each of N reducers gets non-overlapping ranges of x, but doesn't sort the output of each reducer. You end up with N or more unsorted files with non-overlapping ranges.
* **CLUSTER BY x:** ensures each of N reducers gets non-overlapping ranges, then sorts by those ranges at the reducers. This gives you global ordering, and is the same as doing (DISTRIBUTE BY x and SORT BY x). You end up with N or more sorted files with non-overlapping ranges.

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 tables and external tables. The main difference between them lies in how they handle data storage and management.

1. Internal Tables:
   * Internal tables, also known as managed tables, are the default table type in Hive.
   * With internal tables, Hive assumes full control over the lifecycle of the table and its data.
   * The data associated with internal tables is managed by Hive, which means that when you drop an internal table, Hive also removes the associated data.
   * The data for internal tables is typically stored within the Hive warehouse directory (configured by **hive.metastore.warehouse.dir**).
   * Internal tables are best suited for scenarios where you want Hive to have complete control over the data and manage it on your behalf.
2. External Tables:
   * External tables, as the name suggests, are tables where the data is stored externally to Hive.
   * The data for external tables is typically stored in a location of your choice, which can be on Hadoop Distributed File System (HDFS), Amazon S3, Azure Blob Storage, or any other compatible storage system.
   * External tables provide a way to access and query data that is managed outside of Hive.
   * When you drop an external table, Hive only removes the metadata associated with the table, but the data itself remains intact in the external storage location.
   * External tables are useful when you have data that is generated or managed by processes external to Hive, and you want Hive to provide a structured way to query that data without moving or managing it.

To summarize, internal tables manage the data lifecycle and storage within Hive, while external tables reference data stored externally to Hive and allow querying without moving or managing the data. The choice between internal and external tables depends on factors such as data ownership, data management requirements, and the need for data persistence when the table is dropped.

**Use an internal table if:**

1. Data is temporary and doesn’t affect businesses in real time.
2. If you want the hive to manage the data and the tables.

**Use an external table if:**

1. You want to use data outside HIVE for performing a different operation such as loading and merging.
2. The data is of production quality.

When you drop an internal table both metadata and data is lost but when you drop an external table only data is lost but metadata is preserved.

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

Hive data are stored in one of [**Hadoop compatible filesystem**:](https://cwiki.apache.org/confluence/display/HADOOP2/HCFS)

S3, HDFS or other compatible filesystem.

HDFS path : user/hive/warehouse

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

The LOCATION keyword, we can change the default location of Managed tables while creating the managed table in hive. However, to do so, the user needs to specify the storage path of the managed table as the value to the LOCATION keyword, that will help to change the default location of a managed table.

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

In Hive, a metastore is a centralized repository that stores metadata about Hive tables, partitions, columns, and other related information. It serves as a catalog or a data dictionary for Hive, allowing users to define, manage, and access structured data stored in various file formats in a distributed storage system like Hadoop HDFS.

The metastore stores information such as table schemas, column names, data types, table locations, and partitioning information. It provides Hive with the necessary metadata to execute queries, optimize data retrieval, and perform other operations efficiently.

The default database provided by Apache Hive for the metastore is called "Derby." Derby is an embedded relational database management system written in Java, and it is included with the Hive distribution. Derby is suitable for small-scale deployments or testing purposes, but for larger and more production-oriented deployments, it is recommended to use a more robust database like MySQL, PostgreSQL, or Oracle as the metastore. These external databases can be configured as the metastore backend by modifying Hive's configuration files.

Top of Form

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

Hive does not store metadata information in HDFS (Hadoop Distributed File System) for a few reasons:

1.Scalability: HDFS is designed for storing and processing large data files in a distributed manner. It is optimized for handling massive amounts of data and provides high throughput for sequential data access. On the other hand, metadata management involves small-sized files and frequent random access operations. Storing metadata in HDFS would introduce unnecessary overhead and could potentially impact performance due to the different access patterns and characteristics.

2.Performance: Storing metadata in a distributed file system like HDFS would require additional disk I/O and network overhead for each metadata operation. This can lead to slower query execution times and reduced overall performance. By separating the metadata storage from the data storage, Hive can optimize and cache metadata operations in a more efficient manner, improving the query performance.

3.Flexibility: Hive is designed to work with various data storage systems, not just HDFS. It can interface with different file systems, such as local file systems, Amazon S3, Azure Blob Storage, and more. Storing metadata separately from the underlying data storage allows Hive to be more flexible and agnostic to the specific storage system being used.

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

In Hive, a partition is a way to divide a table into smaller, more manageable segments based on specific column values. It allows you to organize and store data in a structured manner, making it easier to query and analyze subsets of the data efficiently.

Partitioning in Hive offers several benefits:

1. Improved Performance: Partitioning can significantly improve query performance by limiting the amount of data that needs to be scanned. When querying a partitioned table, Hive can skip scanning irrelevant partitions based on the query predicates. This reduces the I/O and processing overhead, resulting in faster query execution times.
2. Data Organization: Partitioning helps organize data based on specific criteria, such as dates, regions, or categories. It provides a logical structure to the data, making it easier to understand and manage. Partitioning also enables selective loading and dropping of data partitions, facilitating efficient data ingestion and maintenance workflows.
3. Simplified Data Lifecycle Management: Partitioning can simplify data lifecycle management tasks, such as archiving or deleting old data. Instead of managing the entire dataset, you can target specific partitions for archival or deletion based on predefined criteria, such as partition age or retention policies.

Overall, partitioning in Hive provides performance optimizations, efficient data filtering, data organization, and simplified data lifecycle management. It is particularly beneficial when dealing with large datasets, where dividing the data into smaller partitions based on relevant criteria can significantly enhance query performance and data management capabilities.

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

**Static partitioning :**

In static partitioning, we need to specify the partition column value in each and every LOAD statement.

suppose we are having partition on column country for table t1(userid, name,occupation, country), so each time we need to provide country value

hive>LOAD DATA INPATH '/hdfs path of the file' INTO TABLE t1 PARTITION(country="US") hive>LOAD DATA INPATH '/hdfs path of the file' INTO TABLE t1 PARTITION(country="UK")

**Dynamic Partitioning:**

Dynamic partition allow us not to specify partition column value each time. the approach we follows is as below: create a non-partitioned table t2 and insert data into it. now create a table t1 partitioned on intended column(say country). load data in t1 from t2 as below:

hive> INSERT INTO TABLE t2 PARTITION(country) SELECT \* from T1;

make sure that partitioned column is always the last one in non partitioned table(as we are having country column in t2)

**When to use static partitioning and when to use dynamic partitioning?**

* Use static partitioning when data is already physically categorized/grouped/partitioned and ready to be added as a partition to a table.
* Static partitioning will not result in MapReduce job execution since the data is already physically categorized/partitioned.
* Use dynamic partitioning when data is not already physically categorized/grouped/partitioned.
* Dynamic partitioning will result in MapReduce job execution to group the data first and then partitions will be added to the table.

1. **How do you check if a particular partition exists?**

we can check whether a particular partition exists or not using below query:

*SHOW PARTITIONS table\_name*

*PARTITION(partitioned\_column=’partition\_value’)*

1. **How can you stop a partition form being queried?**

By using the ENABLE OFFLINE clause with ALTER TABLE statement.

Syntax:

*ALTER TABLE t1 PARTITION (PARTITION\_SPEC) ENABLE OFFLINE;*  Example:

*Hive> ALTER TABLE TownsList\_Dynamic*

*PARTITION (country=’England’) ENABLE OFFLINE;*  Now, let’s issue the SELECT statement.

*Hive> select \* from TownsList\_Dynamic where country=’England’;*

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

Bucketing is a method to evenly distributed the data across many files.

Create multiple buckets and then place each record into one of the buckets based on some logic mostly some hashing algorithm.

Bucketing feature of Hive can be used to distribute/organize the table/partition data into multiple files such that similar records are present in the same file.

While creating a Hive table, a user needs to give the columns to be used for bucketing and the number of buckets to store the data into. Which records go to which bucket are decided by the Hash value of columns used for bucketing.

This is mainly useful when all values in the table are unique ie partitioning cant be done.

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

*set.hive.enforce.bucketing=true;*

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

In bucketing, the partitions can be subdivided into buckets based on the hash function of a column thereby avoiding full table scan. It gives extra structure to the data which can be used for more efficient queries.

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

Optimizing Hive performance involves various strategies and techniques that can enhance query execution speed, resource utilization, and overall efficiency of the Hive data warehouse system. Below, provide a detailed explanation of some key areas you can focus on to optimize Hive performance:

1. Data Organization:
   * Partitioning: Partitioning involves dividing data into logical partitions based on specific columns. It helps improve query performance by eliminating the need to scan the entire dataset for every query. Choose appropriate columns for partitioning based on the query patterns.
   * Bucketing: Bucketing involves dividing data into more manageable and evenly distributed files based on a hash function applied to a column's values. It enables faster data retrieval by reducing the amount of data to be scanned during query execution.
2. Schema Design:
   * Denormalization: Denormalizing the schema can improve performance by reducing the number of joins required. However, it should be balanced with the need for data consistency and maintainability.
   * Appropriate Data Types: Choose the most suitable data types for your columns to ensure efficient storage and query processing. Avoid using excessively large data types if smaller ones can fulfill the requirements.
3. Query Optimization:
   * Predicate Pushdown: Hive supports predicate pushdown, which means pushing the filter conditions closer to the data source. Enable this feature to minimize the amount of data read from disk, thereby improving query performance.
   * Join Optimization: Consider using techniques like bucketing and sorting for the join columns to enhance join performance. Additionally, enable map-side joins when possible, which avoid shuffling data between nodes.
   * Limiting Data Skew: Data skew refers to uneven data distribution, which can impact query performance. Techniques such as data skew optimization, sampling, and bucketing can help mitigate the effects of data skew.
4. Performance Tuning:
   * Memory Configuration: Optimize Hive's memory settings, such as the heap size and garbage collection parameters, based on the available system resources and workload requirements. This can prevent out-of-memory errors and improve query execution speed.
   * Parallel Execution: Configure Hive to run multiple tasks concurrently, leveraging the available cluster resources effectively. Adjust the level of parallelism using parameters like "mapreduce.job.reduces" and "hive.exec.parallel" based on the workload characteristics.
   * Compression: Enable compression on Hive tables to reduce storage footprint and improve I/O performance. However, consider the trade-off between compression ratios and CPU overhead during decompression.
5. Hardware Considerations:
   * Storage Selection: Choose appropriate storage systems for Hive, such as HDFS or cloud-based storage options like S3. Optimize the block size and replication factor based on the workload requirements and available storage infrastructure.
   * Network Bandwidth: Ensure sufficient network bandwidth between Hive nodes and data sources, as slow network connections can impact query performance, especially in distributed environments.
6. Hive Configuration:
   * Configuration Parameters: Analyze and tune various Hive configuration parameters based on your specific workload and cluster setup. Parameters like "hive.vectorized.execution.enabled," "hive.cbo.enable," and "hive.stats.autogather" can significantly impact query execution plans and performance.

Remember that the effectiveness of these optimizations can vary depending on your specific use case, data size, hardware infrastructure, and query patterns. It's important to analyze and profile your workload to identify the areas that require optimization and iterate on the tuning process for continuous improvement.

.

.

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

HCatalog is a tool that allows you to access Hive metastore tables within Pig, Spark SQL, and/or custom MapReduce applications. HCatalog has a REST interface and command line client that allows you to create tables or do other operations. You then write your applications to access the tables using HCatalog libraries.

Overall, HCatalog simplifies data management, promotes data sharing, and enables seamless integration across different data processing frameworks in the Hadoop ecosystem. It provides a unified view of the data stored in Hadoop, making it easier for users to work with and analyze large datasets using their preferred tools.

1. **Explain about the different types of joins in Hive.**

JOIN clause is used to combine and retrieve the records from multiple tables. JOIN is same as OUTER JOIN in SQL. A JOIN condition is to be raised using the primary keys and foreign keys of the tables.

The following query executes JOIN on the CUSTOMER and ORDER tables, and retrieves the records:

hive> SELECT c.ID, c.NAME, c.AGE, o.AMOUNT FROM CUSTOMERS c JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+

LEFT OUTER JOIN

The HiveQL LEFT OUTER JOIN returns all the rows from the left table, even if there are no matches in the right table. This means, if the ON clause matches 0 (zero) records in the right table, the JOIN still returns a row in the result, but with NULL in each column from the right table.

A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDER tables: hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE

FROM CUSTOMERS c

LEFT OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+----------+--------+---------------------+

RIGHT OUTER JOIN

The HiveQL RIGHT OUTER JOIN returns all the rows from the right table, even if there are no matches in the left table. If the ON clause matches 0 (zero) records in the left table, the JOIN still returns a row in the result, but with NULL in each column from the left table.

A RIGHT JOIN returns all the values from the right table, plus the matched values from the left table, or NULL in case of no matching join predicate.

The following query demonstrates RIGHT OUTER JOIN between the CUSTOMER and ORDER tables. hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

FULL OUTER JOIN

The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.

The following query demonstrates FULL OUTER JOIN between CUSTOMER and ORDER tables: hive>

SELECT c.ID, c.NAME, o.AMOUNT, o.DATE

FROM CUSTOMERS c

FULL OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

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

join\_condition

| table\_reference [CROSS] JOIN table\_reference join\_condition

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

SMB (Sort-Merge-Bucket) join is a specific type of join strategy available in Apache Hive. It is designed to optimize join operations by leveraging the benefits of both bucketing and sorting techniques.

When two tables are bucketed and sorted on the join columns, the SMB join can be used to efficiently merge the sorted data from both tables, reducing the need for expensive shuffling and sorting operations during the join process.

Here's how the SMB join works in Hive:

1. Bucketing: Both tables involved in the join operation need to be bucketed on the join columns. Bucketing is a technique where data is divided into multiple buckets based on a hash function applied to the join columns. Bucketing ensures that rows with the same join column values are allocated in the same bucket across different tables.
2. Sorting: Within each bucket, the data is sorted based on the join columns. Sorting the data ensures that the rows with the same join column values are adjacent to each other, making it easier to merge the data during the join.
3. Join Execution: When a query involving an SMB join is executed, Hive scans the buckets of both tables in parallel. Since the data within each bucket is already sorted, Hive can efficiently merge the data from both tables without the need for a full shuffle and sort operation. The merge process combines rows with the same join column values, producing the joined result.

Benefits of SMB Join in Hive:

1. Reduced Data Shuffling: By leveraging bucketing and sorting, the SMB join minimizes the need for data shuffling during the join operation. This significantly reduces the network overhead and improves the join performance, especially for large datasets.
2. Efficient Memory Usage: SMB join is memory-efficient because it avoids loading and storing large amounts of data during the join process. Instead, it reads and merges the data in small chunks, allowing for better memory management.
3. Scalability: SMB join is particularly beneficial for large-scale data processing. It takes advantage of parallelism and distributed processing capabilities of Hive, enabling efficient joins even in distributed environments.

Considerations for SMB Join in Hive:

1. Bucketing and Sorting Overhead: The benefits of SMB join come with the overhead of bucketing and sorting the data. Preparing the tables by bucketing and sorting can be time-consuming and may require additional planning and maintenance.
2. Join Column Selection: The performance of SMB join heavily relies on the appropriate selection of join columns. Choosing the right columns to bucket and sort on is crucial for achieving efficient join execution.
3. Data Skew: Data skew, where certain bucket values are significantly larger than others, can impact the effectiveness of SMB join. Skewed data distribution may result in uneven workload distribution among nodes, reducing the performance gains from the SMB join strategy.

Overall, SMB join in Hive is a powerful technique that optimizes join operations by combining bucketing and sorting. It is especially beneficial for large-scale data processing scenarios, where it can significantly improve join performance and reduce data shuffling overhead.

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

The main difference between ORDER BY and SORT BY in Hive is the scope of sorting. ORDER BY provides a global order for the entire result set, but it requires collecting and sorting all the data in memory, which may not be efficient for large datasets. SORT BY, on the other hand, performs a local sort within each reducer task, allowing for distributed sorting and reducing memory requirements. However, it does not guarantee a global order for the entire result..

Note: It may be confusing as to the difference between SORT BY alone of a single column and CLUSTER BY. The difference is that CLUSTER BY partitions by the field and SORT BY if there are multiple reducers partitions randomly in order to distribute data (and load) uniformly across the reducers. Basically, the data in each reducer will be sorted according to the order that the user specified.

The following example shows :

SELECT key, value FROM src SORT BY key ASC, value DESC

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

In Hive, the "DISTRIBUTED BY" clause is used in conjunction with the "CLUSTERED BY" clause to control the distribution of data across the nodes of a cluster. The "DISTRIBUTED BY" clause specifies the columns that determine the data distribution, helping optimize query performance and resource utilization. Here's the usefulness of the "DISTRIBUTED BY" clause in Hive:

1. Data Distribution: By using the "DISTRIBUTED BY" clause, you can control how the data is distributed across the nodes of a cluster. Hive uses a hash function applied to the specified columns to determine which data goes to which node. Distributing data evenly across nodes helps balance the workload and facilitates parallel processing.
2. Load Balancing: When data is evenly distributed across the cluster, it ensures that the processing load is distributed evenly as well. This prevents hotspots or imbalances that can lead to resource contention and degrade query performance. Load balancing is particularly important in distributed environments to utilize the cluster resources efficiently.
3. Query Performance: Distributing data based on the "DISTRIBUTED BY" clause can enhance query performance. When performing operations like joins or aggregations, having data co-located on the same node can minimize data movement and reduce network overhead, resulting in faster query execution.
4. Data Skew Handling: The "DISTRIBUTED BY" clause can help address data skew, which occurs when certain data values are significantly more common than others. By distributing data based on a set of columns, you can mitigate the effects of data skew by spreading the skewed data across multiple nodes, allowing for better parallelism and load balancing.
5. Join Optimization: When performing joins in Hive, the "DISTRIBUTED BY" clause can be used in conjunction with the "CLUSTERED BY" clause to optimize the join operation. By specifying the same distribution columns in both clauses, Hive can co-locate data from both tables with the same join column values on the same nodes, minimizing data movement and improving join performance.

It's important to note that the effectiveness of the "DISTRIBUTED BY" clause depends on the data characteristics, the nature of the queries, and the overall cluster configuration. Consider the data distribution patterns, query patterns, and system resources when deciding which columns to use in the "DISTRIBUTED BY" clause. Additionally, analyzing query execution plans and profiling the system's performance can help fine-tune the data distribution strategy for optimal results

The syntax of the DISTRIBUTE BY clause in hive is as below:

*SELECT Col1, Col2,……ColN FROM TableName DISTRIBUTE BY Col1, Col2, ….. ColN*

**SELECT** SalesYear, Amount **FROM** tbl\_Sales DISTRIBUTE **BY** SalesYear;

Note:

Key Differences:

"ORDER BY" provides a global order for the entire result set, while "SORT BY" and "CLUSTER BY" ensure local order within each reducer output.

* "CLUSTER BY" combines the functionality of sorting and data distribution, whereas "DISTRIBUTE BY" focuses solely on data distribution.
* "ORDER BY" requires collecting and sorting all data in memory, potentially requiring significant memory resources. "SORT BY" and "CLUSTER BY" perform partial sorting within each reducer, reducing memory requirements.
* "CLUSTER BY" attempts to co-locate data with the same values for the specified columns, while "DISTRIBUTE BY" determines data distribution using a hash function.
  + "ORDER BY" and "SORT BY" are typically used on the final query results, while "CLUSTER BY" and "DISTRIBUTE BY" are used during the processing phase of a query.

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

When data is transferred from Hadoop Distributed File System (HDFS) to Hive, the process involves several steps. Here's a high-level overview of how data transfer occurs:

1. Data Ingestion: The first step is to ingest data into HDFS. This can be done through various means, such as uploading files directly to HDFS, using data replication tools, or running ETL (Extract, Transform, Load) processes to import data into HDFS. The data is stored in HDFS in a distributed and fault-tolerant manner.
2. Metadata Creation: Once the data is in HDFS, Hive needs to be made aware of its existence and structure. This is achieved by creating a Hive table that represents the data. The table definition includes information such as the schema, column names, data types, and storage location of the data within HDFS.
3. Metadata Storage: Hive maintains a metadata store, known as the Hive Metastore, which stores information about the tables, partitions, and their associated metadata. When a table is created, Hive updates the metastore with the table's metadata, including the table's structure and the location of its data in HDFS.
4. Data Registration: Once the table is created and the metadata is stored in the metastore, Hive associates the table metadata with the actual data in HDFS. This registration process links the logical representation of the table in Hive with the physical data stored in HDFS.
5. Query Processing: With the table registered, users can interact with the data using Hive's query language, HiveQL. When a query is executed, Hive's query processor translates the HiveQL statements into a series of MapReduce, Tez, or Spark tasks, depending on the execution engine used.
6. Data Retrieval and Processing: The query tasks are responsible for retrieving the relevant data blocks from HDFS based on the table's location and any applied filters or transformations. The data is processed according to the query operations such as filtering, aggregation, or joining.
7. Result Presentation: Once the data processing is complete, the query results can be presented to the user. The results can be written to a file in HDFS, displayed on the console, or stored in an external system.

It's important to note that data transfer from HDFS to Hive doesn't involve physically moving the data. Instead, Hive provides a logical layer on top of HDFS, allowing users to define and interact with structured data using a SQL-like interface. Hive leverages the data stored in HDFS and operates on it by reading the data directly from HDFS during query execution.

The metadata stored in the Hive Metastore serves as a catalog that maps the Hive tables to the corresponding data in HDFS, enabling efficient data retrieval and processing. This separation of metadata and data allows Hive to provide a high-level abstraction while leveraging the scalability and fault-tolerance of HDFS.

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

The creation of a new metastore\_db directory in different directories when running Hive queries is a deliberate mechanism to maintain metadata isolation and separation between different Hive instances or environments, ensuring data integrity and independent operation.

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

If the command "SET hive.enforce.bucketing=true;" is not issued before bucketing a table in Hive, it can lead to unexpected behavior and potential data inconsistencies. Here's what could happen:

1. Bucketing Error: Without setting "hive.enforce.bucketing=true," Hive does not enforce the bucketing property on the table. When you try to bucket the table using the "CLUSTERED BY" clause, Hive will not check whether the table has been properly bucketed or not. This means that you may be able to bucket the table even if it doesn't meet the bucketing requirements. However, this can result in incorrect query results when bucketing assumptions are violated.
2. Data Skew: Bucketing is designed to distribute data evenly across buckets based on a hash function. When "hive.enforce.bucketing=true" is not set, Hive does not ensure proper data distribution among buckets. This can lead to data skew, where some buckets may contain significantly more data than others. Data skew can impact query performance, as the workload may not be evenly distributed across the cluster, and some nodes may be overloaded.
3. Inconsistent Results: If bucketing is performed without enforcing the bucketing property, it can result in inconsistent query results. Bucketing assumes that data with the same bucketing key values are co-located within the same bucket. When bucketing is not properly enforced, this assumption may not hold true, causing queries to retrieve incorrect or incomplete data.
4. Maintenance Challenges: Without enforcing bucketing, it becomes challenging to ensure data consistency and maintain bucketing properties. If subsequent operations or modifications are performed on the table without considering proper bucketing, it can lead to further data inconsistencies and performance issues.

To avoid these issues, it is crucial to set "hive.enforce.bucketing=true" before bucketing a table in Hive. This setting enforces the bucketing property, ensuring that tables are correctly bucketed and maintaining the assumptions required for efficient query processing. By enforcing bucketing, you can achieve better query performance, data distribution, and consistency when working with bucketed tables in Hive.

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

You can rename the table name in the hive.

You need to use the alter command.

This command allows you to change the table name as shown below.

$ ALTER TABLE name RENAME TO new\_name

**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)**

*ALTER TABLE table\_name*

*CHANGE COLUMN new\_col*

*INT BEFORE x\_col;*

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

In Hive, SerDe (Serializer/Deserializer) stands for Serialization/Deserialization. It is a fundamental operation that enables Hive to read data from various formats or data sources and convert it into a structured format that can be processed and queried.

SerDe acts as an interface between Hive and external data formats. It defines how data is serialized (written) and deserialized (read) by Hive, allowing Hive to interact with different types of data sources, including text files, CSV files, JSON, Avro, Parquet, ORC, and more. Each data format requires a specific SerDe implementation to handle its serialization and deserialization.

When a table is created in Hive, a SerDe is specified to define how the data in the table should be interpreted and processed. The SerDe determines how the table's data is serialized when written to disk and how it is deserialized when read back into Hive for querying.

The SerDe operation in Hive involves the following steps:

1. Serialization: When data is inserted or loaded into a table, the SerDe is responsible for converting the data from its internal representation in Hive to a serialized format that can be stored on disk. This includes encoding the data according to the specified data format, handling delimiters, and applying any necessary transformations or conversions.
2. Storage: The serialized data produced by the SerDe is stored in the underlying storage system, such as the Hadoop Distributed File System (HDFS) or cloud storage like Amazon S3.
3. Deserialization: When a query is executed on the table, the SerDe is used to read and deserialize the data from its serialized format back into Hive's internal representation. This involves parsing the serialized data, applying any necessary transformations or conversions, and presenting it in a structured format that can be processed by Hive.
4. Query Processing: Once the data is deserialized, Hive processes the data using its query engine and executes the requested operations, such as filtering, aggregating, joining, or sorting.

Different SerDe implementations support different data formats and have specific configurations and options. Hive provides a set of built-in SerDe implementations, and it is also possible to create custom SerDes to handle specific data formats or requirements.

SerDe operations are crucial for enabling Hive to interact with a wide range of data formats and integrate with various external systems. They play a vital role in enabling data interoperability and flexibility in Hive, allowing users to query and process data from diverse sources using the familiar SQL-like interface provided by HiveQL.

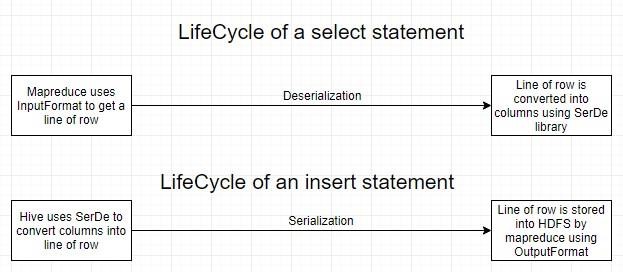
**41. Explain how Hive Deserializes and serialises the data?**

In Hive, data serialization and deserialization (SerDe) are crucial processes that enable reading data from external sources and writing data to external targets. Hive uses SerDe libraries to deserialize data from its original format into a tabular structure that can be processed by Hive, and to serialize processed data back into the original format for storage or output.

1. Data Deserialization: When reading data into Hive, the process of deserialization involves converting the data from its original format (such as CSV, JSON, Avro, etc.) into a structured format that Hive can work with. Here's an overview of how Hive performs data deserialization:
   * Hive relies on the SerDe libraries, which are responsible for parsing the data and converting it into a row-based format.
   * The SerDe libraries interpret the data based on the specified data format, schema, and serialization/deserialization rules.
   * The deserialized data is transformed into a tabular structure, typically represented as rows and columns.
   * Hive then applies any configured transformations, such as column projections, filtering, and aggregations, on the deserialized data.
2. Data Serialization: After processing the data within Hive, serialization is performed to convert the processed data back into the original format or any desired output format. Here's how Hive accomplishes data serialization:
   * The processed data, represented in tabular form, is serialized using the configured SerDe libraries.
   * The SerDe libraries apply the specified serialization rules to transform the tabular data into the desired output format, such as CSV, JSON, Avro, etc.
   * The serialized data is then written to the target storage or output location, which can be a file in HDFS, an external storage system, or any other supported destination.

Hive provides a variety of built-in and third-party SerDe libraries to handle different data formats. These libraries define the rules for deserializing and serializing data according to the specific format's syntax and schema. Additionally, Hive allows users to define custom SerDe libraries if the desired data format is not supported out of the box.

By leveraging the SerDe capabilities, Hive enables seamless integration with various data formats, making it possible to process and analyze data from diverse sources while maintaining compatibility with the original formats for storage and interoperability.



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

The Hive SerDe library is in org.apache.hadoop.hive.serde2. (The old SerDe library in org.apache.hadoop.hive.serde is deprecated.)

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

The need for a custom SerDe (Serialization/Deserialization) in Hive arises in scenarios where the built-in SerDe libraries do not support the specific data format or structure you want to work with. Here are a few reasons why you might require a custom SerDe:

1. Unsupported Data Format: If you have data stored in a format that is not supported by the built-in SerDe libraries in Hive, you'll need a custom SerDe to handle that format. For example, if you have data in a proprietary or specialized format, a custom SerDe can be developed to parse and process that data correctly within Hive.
2. Complex Data Structure: Hive's built-in SerDe libraries are designed to handle common data structures, such as delimited text, JSON, or Avro. However, if your data has a complex structure, nested fields, or a custom schema, a custom SerDe can be created to deserialize and serialize the data accurately. This is especially useful when dealing with data formats like XML or binary files.
3. Performance Optimization: In some cases, the built-in SerDe libraries may not provide optimal performance for certain data formats or processing requirements. By developing a custom SerDe, you can fine-tune the deserialization and serialization process to improve performance, enhance efficiency, and meet specific performance objectives.
4. Data Transformation and Integration: A custom SerDe can be used to transform data during deserialization or serialization. This can involve manipulating the data structure, performing data cleansing, applying business rules, or integrating data from multiple sources. A custom SerDe allows you to customize the data processing logic to meet your specific requirements.
5. Compliance and Regulatory Needs: If your data has specific compliance or regulatory requirements, such as data masking, encryption, or anonymization, a custom SerDe can be developed to enforce these requirements during data deserialization or serialization.

Developing a custom SerDe requires understanding the data format, defining the rules for deserialization and serialization, and implementing the necessary logic to handle the custom format or requirements. By creating a custom SerDe, you can leverage the flexibility and extensibility of Hive to work with diverse data formats and meet your specific data processing needs.

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

In addition to primitive data types, Hive also supports a few complex data types: Struct, MAP , and Array,union . Complex data types are also known as collection data types.

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

Yes.

*Hive> source /path/to/file/file\_with\_query.hql*

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

In Hive, when working with text files, the default record and field delimiters used are as follows:

1. Record Delimiter (Row Delimiter): The default record delimiter, also known as the row delimiter, is the newline character (**\n**). Hive considers each occurrence of the newline character as the end of a record or row in a text file.
2. Field Delimiter (Column Delimiter): The default field delimiter, also known as the column delimiter, is a tab character (**\t**). Hive considers each occurrence of the tab character as the separation between fields or columns within a record or row in a text file.

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

*SHOW (DATABASES|SCHEMAS) [LIKE identifier\_with\_wildcards];*

*SHOW DATABASES LIKE 's%';*

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

LIKE is an operator similar to LIKE in SQL. We use LIKE to search for string with similar text.

In Hive, the **RLIKE** operator is used for pattern matching using regular expressions. It allows you to search for a pattern within a string column and return rows that match the specified regular expression.

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

*ALTER TABLE table\_name CHANGE column\_name column\_name new\_datatype;*

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

select cast ( ’51.2’ as float)

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

It prints NULL

**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 provided query performs an **INSERT OVERWRITE** operation in Hive to populate the **employees** table using data from the **staged\_employees** table. Here's a breakdown of the query:

a. **INSERT OVERWRITE TABLE employees**: This line specifies that the data will be inserted into the **employees** table. The **OVERWRITE** keyword indicates that any existing data in the **employees** table will be overwritten by the new data being inserted.

b. **PARTITION (country, state)**: This line specifies that the **employees** table is partitioned by two columns: **country** and **state**. Partitioning allows data to be organized and stored in separate directories based on the values in the specified partition columns. It helps with performance optimization and query efficiency.

c. **SELECT ..., se.cnty, se.st**: This line defines the columns to be selected from the **staged\_employees** table. The **...** indicates that additional columns are being selected, but their specific names are not provided in the query. The **se.cnty** and **se.st** columns are explicitly selected from the **staged\_employees** table.

d. **FROM staged\_employees se**: This line specifies the source table from which the data will be selected. In this case, the **staged\_employees** table is referenced with the table alias **se**.

In summary, the query inserts data from the **staged\_employees** table into the **employees** table, overwriting any existing data. The data is selected from the **staged\_employees** table, and the resulting rows are partitioned based on the values in the **country** and **state** columns.

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

*create table B like A;*

*INSERT OVERWRITE TABLE B SELECT A.value FROM A;*

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

Regarding how Hive supports binary formats, Hive provides support for binary data through various means:

1. Binary Serialization: Hive supports binary serialization/deserialization of data using specialized SerDe libraries. These libraries define the rules for converting data between its binary representation and a structured format that can be processed by Hive. Examples of such SerDe libraries include Avro, Thrift, and Apache ORC.
2. Binary File Formats: Hive supports binary file formats such as Apache ORC (Optimized Row Columnar) and Apache Parquet. These file formats are designed to store and process structured data efficiently, providing advantages like compression, columnar storage, and predicate pushdown.

When using binary file formats, Hive leverages the capabilities of these formats to efficiently read, write, and process binary data. This allows for improved performance and storage efficiency compared to traditional text-based formats.

By supporting binary serialization, binary file formats, and integration with specialized SerDe libraries, Hive enables efficient processing of binary data, making it suitable for handling a wide range of data formats and scenarios.

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

Hive supports a variety of file formats and applications, providing flexibility in handling different data formats and integrating with various tools and ecosystems. Here are some of the file formats and applications supported by Hive:

1. File Formats: Hive supports both text-based and binary file formats, including:
   * Text File: Plain text files with configurable delimiters (default delimiter is tab-separated).
   * SequenceFile: A binary file format optimized for storing key-value pairs.
   * RCFile (Record Columnar File): A columnar storage file format designed for efficient data retrieval.
   * Avro: A compact, schema-based binary format used for data serialization.
   * Parquet: A columnar storage file format that provides efficient compression and predicate pushdown capabilities.
   * ORC (Optimized Row Columnar): A high-performance columnar file format with advanced compression and predicate pushdown features.
2. Data Integration: Hive integrates with various data integration and processing frameworks, including:
   * Apache Hadoop: Hive is tightly integrated with the Hadoop ecosystem, leveraging HDFS for storage and MapReduce or Apache Tez for data processing.
   * Apache Spark: Hive provides a Spark SQL interface, allowing users to execute Hive queries within Spark applications.
   * Apache Flink: Hive can be used as a data source and sink for Flink jobs, enabling seamless integration with Flink's stream processing capabilities.
   * Apache Kafka: Hive supports integrating with Apache Kafka, allowing data ingestion and analysis of streaming data.
   * Apache NiFi: Hive can be integrated with Apache NiFi for data ingestion, transformation, and routing.
3. Ecosystem Integration: Hive can be integrated with various tools and ecosystems, enabling interoperability and compatibility, such as:
   * Apache Zeppelin: Hive integrates with Zeppelin notebooks, providing an interactive data analysis and visualization environment.
   * Tableau: Hive can be used as a data source for Tableau, allowing users to create visualizations and reports based on Hive tables.
   * Apache Superset (incubating): Hive can serve as a data source for Superset, enabling data exploration and interactive dashboards.
   * Business Intelligence (BI) Tools: Hive can be used with popular BI tools like MicroStrategy, QlikView, and Power BI for data analysis and reporting.

By supporting a wide range of file formats and integrating with various tools and ecosystems, Hive enables seamless data processing, analysis, and integration within the larger big data ecosystem.

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

Using the ORC format leads to a reduction in the size of the data stored, as this file format has high compression ratios. As the data size is reduced, the time to read and write the data is also reduced.

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

When you perform a "select \* from <tablename>", Hive fetches the whole data from file as a FetchTask rather than a mapreduce task which just dumps the data as it is without doing anything on it. This is similar to "hadoop dfs -text <filename>"

However, while using "select <column> from <tablename>", Hive requires a map-reduce job since it needs to extract the 'column' from each row by parsing it from the file it loads.

Hive provides an alternative execution engine called Apache Tez, which allows queries to be processed without relying on the MapReduce framework. By leveraging Tez, Hive can achieve faster query execution and improved performance. Here's how Hive can avoid MapReduce and utilize Tez for query processing:

1. Apache Tez Execution Engine: Tez is a data processing framework built on top of Apache Hadoop YARN. It provides a flexible execution model that allows for efficient and optimized data processing. Hive can be configured to use Tez as the execution engine instead of MapReduce.
2. DAG-based Execution: Unlike MapReduce, which executes jobs in a linear fashion, Tez uses a Directed Acyclic Graph (DAG) model. Hive generates a DAG representation of the query plan, which allows for parallel execution of multiple tasks and optimized data flows.
3. Vertex-Parallel Processing: In Tez, the query plan is divided into multiple vertices, each representing a task. These vertices can be executed in parallel, enabling better utilization of resources and reducing the overall execution time.
4. Data Pipelining: Tez supports efficient data movement and pipelining between tasks. Intermediate data can be streamed directly from one task to another, reducing disk I/O and improving data processing performance.
5. Dynamic Task Scheduling: Tez optimizes task scheduling based on data locality, workload balancing, and other factors. It dynamically schedules tasks to minimize data shuffling and network overhead, leading to improved performance.
6. Optimization Techniques: Tez incorporates various optimization techniques such as data skew handling, vectorization, and dynamic partition pruning. These optimizations further enhance query performance and reduce execution time.

To utilize Tez as the execution engine in Hive, you need to ensure that Tez is properly configured and enabled. This involves setting appropriate configuration properties in Hive to specify the execution engine as Tez and configuring other Tez-related parameters.

By leveraging Apache Tez, Hive can avoid the overhead of MapReduce and achieve faster and more efficient query processing. Tez's optimized execution model, parallel processing, data pipelining, and optimization techniques contribute to improved performance and reduced latency in Hive queries.

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

In Hive, a view and indexing are two different concepts related to data organization and querying:

1. View in Hive: A view in Hive is a virtual table that does not store any data itself but provides a logical representation of the underlying data in other tables. It is created based on a query that defines the desired columns and rows from one or more existing tables. Views are used to simplify complex queries, provide a consistent and controlled access to data, and encapsulate business logic.

When a query is executed against a view, Hive translates the query into the underlying tables' query and retrieves the data dynamically. Views can be used for data abstraction, security, and query optimization. They enable users to work with a simplified and predefined structure without directly accessing the underlying tables.

1. Indexing in Hive: Indexing in Hive refers to the creation and usage of indexes to improve query performance. An index is a data structure that allows for efficient data retrieval based on specific columns or expressions. By creating an index on one or more columns of a table, Hive can locate the desired data more quickly during query execution, reducing the amount of data scanned.

The goal of Hive indexing is to improve the speed of query lookup on certain columns of a table. Without an index, queries with predicates like

'WHERE tab1. col1 = 10' loads the entire table or partition and process all the rows;

Views are generated based on user requirements. You can save any result set data as a view. The usage of view in Hive is same as that of the view in SQL. It is a standard RDBMS concept. We can execute all DML operations on a view.

Creating a View

You can create a view at the time of executing a SELECT statement.

The syntax is as follows:

CREATE VIEW [IF NOT EXISTS] view\_name [(column\_name [COMMENT column\_comment], ...) ] [COMMENT table\_comment] AS SELECT ...

An Index is nothing but a pointer on a particular column of a table.

Creating an index means creating a pointer on a particular column of a table.

CREATE INDEX inedx\_salary ON TABLE employee(salary) AS

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

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

No, the name of a view cannot be the same as the name of a Hive table. In Hive, the names of tables and views are stored in the same namespace, and they must be unique within that namespace.

When you create a view in Hive, you provide a name for the view. This name must be unique among all the views and tables in the same database or schema. Hive does not allow views and tables to have the same name within the same namespace.

If you attempt to create a view with the same name as an existing table, Hive will throw an error indicating that the name is already in use. Similarly, if you try to create a table with the same name as an existing view, Hive will raise an error.

It's good practice to use distinct and meaningful names for both tables and views to avoid any naming conflicts and ensure clarity in your data structures and query operations within Hive

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

1. Storage Cost: Indexes require additional storage space. The size of the index depends on the number of indexed columns and the cardinality of the data. More indexes or larger indexes will consume more storage space, which can be a significant cost consideration, especially for large datasets.
2. Processing Cost: Indexing involves extra processing during data insertion, updates, and deletions. When new data is added or existing data is modified or deleted, the index needs to be updated accordingly. This extra processing can impact the overall performance of data manipulation operations and can increase the processing time and resource utilization.
3. Maintenance Cost: Indexes need to be maintained to stay in sync with the underlying data. As the data changes over time, the indexes must be updated to reflect those changes. Regular maintenance tasks include index rebuilding or reorganization, which can be time-consuming and resource-intensive, especially for large datasets.
4. Overhead Cost: Indexes introduce some overhead during query execution. While indexes can speed up query performance by enabling faster data retrieval, they also add overhead due to the need for index lookup and potential index range scans. This overhead is typically manageable, but it can become significant if indexes are misused or too many indexes are created.
5. Development Cost: Creating and managing indexes requires additional development effort. Indexes need to be defined and maintained appropriately, considering the query patterns and performance requirements of the application. Designing optimal indexes requires understanding the data and query workload, which may involve analyzing query execution plans and tuning index configurations.

It's essential to carefully evaluate the costs and benefits of creating indexes on Hive tables based on the specific use case and workload characteristics. While indexes can improve query performance, they also come with storage and processing overhead, which should be considered when deciding whether to create indexes or not

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

*SHOW INDEX ON table\_name;*

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

We can use following commands in Hive to recursively access sub-directories:

*hive> Set mapred.input.dir.recursive=true;*

*hive> Set hive.mapred.supports.subdirectories=true;*

Once above options are set to true, Hive will recursively access sub-directories of a directory in MapReduce.

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

In Hive, when you run a simple **SELECT \*** query without any additional transformations or aggregations, it does not trigger a MapReduce job. Instead, Hive utilizes a mechanism called "Hive Optimization" to optimize and execute the query efficiently.

When you execute a **SELECT \*** query in Hive, it is treated as a metadata-only operation. Hive checks the table metadata, such as the schema and location of the data, and retrieves the necessary metadata to satisfy the query without actually processing the data through a MapReduce job.

The metadata of the table contains information about the data's location, format, and partitioning. Hive leverages this metadata to determine the columns and their corresponding data locations to be fetched when executing a **SELECT \*** query. It directly reads the data from the underlying storage (HDFS, S3, etc.) without the need for a MapReduce job.

By avoiding a MapReduce job for simple **SELECT \*** queries, Hive improves query performance and minimizes the overhead associated with launching and managing MapReduce tasks. However, when you perform complex queries involving transformations, aggregations, joins, or conditions, Hive will generate an optimized execution plan and execute the query using MapReduce or other execution engines like Tez or Spark, depending on the Hive configuration.

It's important to note that Hive's behavior may vary based on the version and configuration of Hive you are using. Newer versions of Hive, such as Hive on Tez or Hive on Spark, may have different optimizations and execution strategies in place.

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

Explode is a **User Defined Table generating Function**(UDTF) in Hive. It takes an array (or a map) as an input and outputs the elements of the array (or a map) as separate rows. UDTFs can be used in the SELECT expression list and as a part of LATERAL VIEW.

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

When running Hive as a server, there are several mechanisms available for connecting applications and interacting with Hive programmatically. Here are some common mechanisms:

1. Hive JDBC: Hive provides a JDBC (Java Database Connectivity) driver that allows applications to connect to Hive using standard JDBC APIs. Applications can use JDBC to establish a connection to Hive, execute queries, and retrieve results. The JDBC driver provides a convenient way to integrate Hive with applications written in programming languages that support JDBC, such as Java, Python, or Scala.
2. Hive Thrift API: Hive Thrift API is a multi-language API that allows applications to interact with Hive using Thrift, a remote procedure call (RPC) framework. It provides a set of client libraries for various programming languages, including Java, Python, C++, and more. By using the Thrift API, applications can connect to Hive, execute queries, fetch results, and perform administrative tasks programmatically.
3. HiveServer2: HiveServer2 is a service that provides a Thrift interface to access Hive. It allows multiple clients to connect simultaneously and execute queries. Applications can connect to HiveServer2 using the Thrift API or other compatible clients. HiveServer2 provides more robust concurrency and security features compared to the previous version, HiveServer1.
4. Hive Beeline: Beeline is a command-line interface (CLI) for Hive that supports both JDBC and HiveServer2 connections. It provides a shell-like environment for executing Hive queries and commands interactively. Applications can launch Beeline and communicate with Hive programmatically by executing Beeline commands or scripts.
5. Hive Web Interface: Hive also offers a web-based user interface, known as the Hive Web Interface or HiveServer2 Web UI. It provides a browser-based interface for executing queries, monitoring job progress, and managing Hive resources. Applications can interact with Hive programmatically by making HTTP requests to the HiveServer2 Web UI REST API.

These mechanisms allow applications to connect to Hive as a server and perform operations such as executing queries, retrieving results, and managing Hive resources programmatically. The choice of the mechanism depends on the programming language, the specific requirements of the application, and the preferred mode of interaction (e.g., JDBC, Thrift, CLI, or web-based).

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

In Hive, the default location of a managed table is determined by the underlying storage system. By default, managed tables are stored in the Hive warehouse directory specified in the Hive configuration (**hive.metastore.warehouse.dir**). The warehouse directory is typically an HDFS (Hadoop Distributed File System) path.

While it's not recommended to change the default location of a managed table, it is possible to do so by altering the table properties.

Using the LOCATION keyword, we can change the default location of Managed tables while creating the managed table in Hive.

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

In Hive, the ObjectInspector is a fundamental component of the Hive infrastructure that enables the interpretation and manipulation of data stored in Hive tables. It is a set of classes and interfaces that provide a standardized way to inspect the internal structure of objects in Hive, including their types, fields, and values.

The ObjectInspector function in Hive serves two main purposes:

1. Data Deserialization: When Hive reads data from its underlying storage system (e.g., HDFS), it needs to deserialize the data into a format that can be processed and analyzed. The ObjectInspector is responsible for deserializing the data and converting it into a Hive-internal representation. It examines the serialized data, interprets its structure, and generates the appropriate ObjectInspector instances to represent the data's fields and types.
2. Data SerDe (Serialization/Deserialization): The ObjectInspector is also used in the serialization process when writing data to the underlying storage system. It enables Hive to serialize data from its internal representation to the format expected by the storage system. The ObjectInspector provides methods to access the fields and values of an object, allowing Hive to extract the relevant data and serialize it accordingly.

ObjectInspector provides various methods to access the fields and values of an object, based on its type. These methods allow Hive to perform operations like reading or writing specific fields, comparing values, and performing type-specific operations on the data.

Hive provides different implementations of ObjectInspectors to handle different data types, such as primitive types (e.g., int, string), complex types (e.g., arrays, maps, structs), and user-defined types (e.g., custom SerDes). These implementations ensure that Hive can handle a wide range of data types and efficiently work with structured and semi-structured data.

Overall, the ObjectInspector function in Hive plays a crucial role in data deserialization, serialization, and accessing the internal structure of objects, enabling Hive to process and manipulate data effectively.

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

UDF stands for User Defined Functions.

Hive is a powerful tool that allows us to provision sql queries on top of stored data for basic querying and/or analysis. And on top of an already rich set of built-in functions, it allows us to extend its functionality by writing custom functions of our own.

**Types of UDFs available in Hive:**

1. **UDF** (User Defined Function): Operates on a single row and generates a single output.
2. **UDAF** (User Defined Aggregate Function): Operates on a group of rows and generates a single output. Generally, used with an accompanying ‘group by’ clause.
3. **UDTF** (User Defined Tabular Function): Operates on a single row and generates multiple rows as an output.

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

*load data inpath '<hdfs location>' into table <hive table name>;*

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

**TextInputFormat :**

TEXTFILE format is a famous input/output format used in [**Hadoop**.](https://acadgild.com/big-data/big-data-development-training-certification) In Hive if we define a table as

TEXTFILE it can load data of from CSV (Comma Separated Values), delimited by Tabs, Spaces, and JSON data. This means fields in each record should be separated by comma or space or tab or it may be JSON(JavaScript Object Notation) data.

By default, if we use TEXTFILE format then each line is considered as a record.

We can create a TEXTFILE format in Hive as follows:

*create table table\_name (schema of the table) row format delimited fields terminated by ',' | stored as TEXTFILE.*

At the end, we need to specify the type of **file format**. If we do not specify anything it will consider the file format as TEXTFILE format.

The TEXTFILE input and TEXTFILE output format are present in the Hadoop package as shown below:

*org.apache.hadoop.mapred.TextInputFormat org.apache.hadoop.mapred.TextOutputFormat*  **SequenceFileInputFormat:**

We know that Hadoop's performance is drawn out when we work with a small number of files with big size rather than a large number of files with small size. If the size of a file is smaller than the typical block size in Hadoop, we consider it as a small file. Due to this, a number of metadata increases which will become an overhead to the NameNode. To solve this problem sequence files are introduced in Hadoop. Sequence files act as a container to store the small files.

Sequence files are flat files consisting of binary key-value pairs. When Hive converts queries to MapReduce jobs, it decides on the appropriate key-value pairs to be used for a given record. Sequence files are in the binary format which can be split and the main use of these files is to club two or more smaller files and make them as a one sequence file.

In Hive we can create a sequence file by specifying STORED AS SEQUENCEFILE in the end of a CREATE TABLE statement.

There are three types of sequence files:

* Uncompressed key/value records.
* Record compressed key/value records - only 'values' are compressed here
* Block compressed key/value records - both keys and values are collected in 'blocks' separately and compressed. The size of the 'block' is configurable.

Hive has its own SEQUENCEFILE reader and SEQUENCEFILE writer libraries for reading and writing through sequence files.

In Hive we can create a sequence file format as follows:

*create table table\_name (schema of the table) row format delimited fileds terminated by ',' | stored as SEQUENCEFILE*

Hive uses the SEQUENCEFILE input and output formats from the following packages:

*org.apache.hadoop.mapred.SequenceFileInputFormat org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat*

**71.How can you prevent a large job from running for a long time in a hive?**

Preventing a large Hive job from running for an excessively long time involves optimizing the query and adjusting various configuration parameters. Here are several approaches to consider:

1. Query Optimization: Improve the efficiency of your Hive query by optimizing its structure and execution plan. Consider the following techniques:
   * Reduce data scanned: Minimize the amount of data being processed by using appropriate filtering, partitioning, and bucketing techniques. Apply predicates or WHERE clauses to limit the dataset to only the necessary data.
   * Optimize joins: Use appropriate join techniques (e.g., map-side join, broadcast join) and ensure that join conditions are efficient. Enable Hive's Cost-Based Optimizer (CBO) to choose the most optimal join strategy.
   * Avoid unnecessary operations: Eliminate unnecessary operations such as unnecessary sorting, redundant aggregations, or excessive data shuffling.
   * Utilize appropriate data types and functions: Choose the appropriate data types and use built-in functions efficiently to perform calculations or transformations.
2. Resource Allocation: Adjust the resource allocation for Hive to ensure it has sufficient resources to execute the job efficiently. Consider the following aspects:
   * Configure memory settings: Adjust the memory-related configuration parameters like **hive.execution.engine**, **hive.tez.container.size**, **hive.tez.java.opts**, or **hive.auto.convert.join.noconditionaltask** to allocate sufficient memory for query execution.
   * Adjust parallelism: Tune parameters like **hive.exec.parallel**, **hive.exec.parallel.thread.number**, or **hive.exec.reducers.bytes.per.reducer** to control the parallelism and the number of reducers used in the job.
3. Cluster Configuration: Ensure that your Hadoop cluster is appropriately configured to handle large jobs. Consider the following:
   * Adjust resource allocation for YARN: Configure YARN's resource allocation settings (e.g., **yarn.scheduler.maximum-allocation-mb**, **yarn.scheduler.minimum-allocation-mb**, **yarn.nodemanager.resource.memory-mb**) to allocate sufficient resources for Hive jobs.
   * Enable dynamic resource allocation: Enable dynamic resource allocation in YARN to allow Hive to request additional resources when necessary.
4. Data Partitioning and Indexing: Organize your data using partitioning and indexing techniques to enable faster data retrieval and avoid unnecessary scanning of large datasets.
5. Incremental Processing: Consider breaking down the large job into smaller incremental tasks to process data in batches. This approach can help in achieving better performance and enables checkpoints or restartability if the job fails.
6. Data Sampling: Use data sampling techniques to analyze a smaller subset of your data, allowing you to optimize the query and evaluate its performance before running it on the entire dataset.
7. Monitor and Tune: Monitor the job execution using Hive's built-in monitoring tools or external monitoring systems. Analyze the query execution plan, identify bottlenecks, and tune the query or cluster configuration accordingly.

By implementing these approaches, you can optimize the query execution, allocate appropriate resources, and ensure efficient processing of large jobs in Hive, reducing their overall execution time.

**72.When do we use explode in Hive?**

We need to use Explode in Hive to convert complex data types into desired table formats. explode UDTF basically emits all the elements in an array into multiple rows.

Ex: map, arrays

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

Hive is designed to be a versatile data processing tool that can handle a wide range of data formats. While Hive has its own default storage format called ORC (Optimized Row Columnar), it can process various data formats through the use of input/output formats, known as SerDes (Serializer/Deserializer).

Here's why Hive can process different data formats:

1. Pluggable SerDes: Hive supports pluggable SerDes, which are responsible for serializing data from a Hive table into a storage format and deserializing data from the storage format back into a Hive table. SerDes enable Hive to interact with various data formats by providing the necessary conversion logic. By specifying the appropriate SerDes, Hive can process data in formats such as text, CSV, JSON, Avro, Parquet, ORC, SequenceFile, and more.
2. File Formats and Input/Output Formats: Hive leverages the Hadoop ecosystem, which provides a wide range of file formats and input/output formats. Hive utilizes these formats and formats-specific input/output classes, such as TextInputFormat, SequenceFileInputFormat, AvroSerDe, ParquetSerDe, etc., to read and write data in different formats. This allows Hive to work with data stored in various formats supported by the underlying Hadoop ecosystem.
3. Custom SerDes: In addition to the built-in SerDes, Hive allows users to develop custom SerDes for handling specific data formats. Users can implement their own SerDes to handle proprietary or specialized data formats that are not natively supported by Hive. This extensibility enables Hive to adapt to unique data formats and process them efficiently.
4. Data Manipulation and Querying: Hive's query language, HiveQL, provides a SQL-like interface for data manipulation and querying. Hive abstracts the underlying data format complexities from users, allowing them to write queries in HiveQL and perform operations on the data regardless of its format. Users can use HiveQL to filter, aggregate, join, and transform data stored in different formats without needing to understand the intricacies of each format.

Overall, Hive's ability to process different data formats is achieved through its support for pluggable SerDes, utilization of Hadoop's file formats and input/output formats, and the flexibility to create custom SerDes. This flexibility makes Hive a versatile tool for working with a variety of data formats, allowing users to process and analyze data stored in different formats using a common SQL-like interface.

HiveQL handles structured data only.

By default, Hive has derby database to store the data in it. We can configure Hive with MySQL database. As mentioned, HiveQL can handle only structured data. Data is eventually stored in files.

Hive supports several file formats:

* Text File
* SequenceFile
* [RCFile](https://cwiki.apache.org/confluence/display/Hive/RCFile)
* [Avro Files](https://cwiki.apache.org/confluence/display/Hive/AvroSerDe)
* [ORC Files](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC)
* [Parquet](https://cwiki.apache.org/confluence/display/Hive/Parquet)
* Custom INPUTFORMAT and OUTPUTFORMAT

The [hive.default.fileformat c](https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties#ConfigurationProperties-hive.default.fileformat)onfiguration parameter determines the format to use if it is not specified in a [CREATE TABLE o](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-CreateTable)[r ALTER TABLE s](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-AlterEitherTableorPartition)tatement. Text file is the parameter's default value.

# Hive Text File Format

**Hive Text file format** is a default storage format. You can use the text format to interchange the data with other client application. The text file format is very common most of the applications. Data is stored in lines, with each line being a record. Each lines are terminated by a newline character (\n).

The text format is simple plane file format. You can use the compression (*BZIP2*) on the text file to reduce the storage spaces.

# Hive Sequence File Format

**Sequence files** are Hadoop flat files which stores values in binary key-value pairs. The sequence files are in binary format and these files are able to split. The main advantages of using sequence file is to merge two or more files into one file.

# Hive RC File Format

**RCFile** is row columnar file format. This is another form of Hive file format which offers high row level compression rates. If you have requirement to perform multiple rows at a time then you can use RCFile format.

The RCFile are very much similar to the sequence file format. This file format also stores the data as key-value pairs.

# Hive AVRO File Format

**AVRO** is open source project that provides data serialization and data exchange services for Hadoop. You can exchange data between Hadoop ecosystem and program written in any programming languages. Avro is one of the popular file format in Big Data Hadoop based applications.

# Hive ORC File Format

The **ORC file** stands for Optimized Row Columnar file format. The ORC file format provides a highly efficient way to store data in Hive table. This file system was actually designed to overcome limitations of the other Hive file formats. The Use of ORC files improves performance when Hive is reading, writing, and processing data from large tables.

# Hive Parquet File Format

**Parquet** is a column-oriented binary file format. The parquet is highly efficient for the types of largescale queries. Parquet is especially good for queries scanning particular columns within a particular table. The Parquet table uses compression Snappy, gzip; currently Snappy by default.

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

Whenever you run the hive in embedded mode, it creates the local metastore. And before creating the metastore it looks whether metastore already exist or not. This property is defined in configuration file hive-site.xml. Property is “javax.jdo.option.ConnectionURL” with default value “jdbc:derby:;databaseName=metastore\_db;create=true”. So to change the behavior change the location to absolute path, so metastore will be used from that location.

**75.Can we change the data type of a column in a hive table? Write a complete query.** *ALTER TABLE table\_name CHANGE column\_name column\_name new\_datatype;*

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

Hive provides us the functionality to load pre-created table entities either from our local file system or from HDFS. The LOAD DATA statement is used to load data into the hive table.

Syntax:

*LOAD DATA [LOCAL] INPATH '' [OVERWRITE] INTO TABLE ;*

Note: The LOCAL Switch specifies that the data we are loading is available in our Local File System. If the LOCAL switch is not used, the hive will consider the location as an HDFS path location. The OVERWRITE switch allows us to overwrite the table data.

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

In Hive we can use following precedence order to set the configurable properties.

* Hive SET command has the highest priority
* -hiveconf option from Hive Command Line
* hive-site.xml file
* hive-default.xml file
* hadoop-site.xml file
* hadoop-default.xml file

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

WebHCat API web interface can be used for Hive commands. It is a REST API that allows applications to make HTTP requests to access the Hive metastore (HCatalog DDL). It also enables users to create and queue Hive queries and commands.

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

Just gzip your files and put them as is (\*.gz) into the table location.

1. **What is the difference between local and remote metastores?** 
   * **Local Metastore**: Here metastore service still runs in the same JVM as Hive but it connects to a database running in a separate process either on same machine or on a remote machine.
   * **Remote Metastore**: Metastore runs in its own separate JVM not on hive service JVM.

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

Due to the design of HDFS, the number of files in the filesystem directly affects the memory consumption in the namenode. While normally not a problem for small clusters, memory usage may hit the limits of accessible memory on a single machine when there are >50-100 million files. In such situations, it is advantageous to have as few files as possible.

The use of [Hadoop Archives i](http://hadoop.apache.org/docs/stable1/hadoop_archives.html)s one approach to reducing the number of files in partitions. Hive has built-in support to convert files in existing partitions to a Hadoop Archive (HAR) so that a partition that may once have consisted of 100's of files can occupy just ~3 files (depending on settings). However, the trade-off is that queries may be slower due to the additional overhead in reading from the HAR.

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

**DBPROPERTIES** – Optional but used to specify any properties of database in the form of (key, value) separated pairs.

*CREATE DATABASE IF NOT EXISTS test\_db*

*COMMENT "Test Database created for tutorial"*

*WITH DBPROPERTIES(*

*'Date' = '2014-12-03',*

*'Creator' = 'Bala G',*

*'Email' = 'bala@somewhere.com'*

*);*

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

**MapReduce mode**:

In MapReduce mode, Hive script is executed on Hadoop cluster. The Hive scripts are converted into MapReduce jobs and then executed on Hadoop cluster (hdfs)

**Local mode**:

In this mode, Hive script runs on a Single machine without the need of Hadoop cluster or hdfs. Local mode is used for development purpose to see how the script would behave in an actual environment.