

* **Apache Hive** is a data warehouse system designed for data summarization, analysis, and querying of large datasets.
* It translates SQL-like queries into **MapReduce jobs**, enabling efficient execution and processing in distributed environments.
* Hive can handle both **structured and semi-structured data** using its SQL-like query language, known as **HiveQL (HQL)**.
* **Hive is not a traditional database**; rather, it is a query engine that runs on top of Hadoop.
* **Hive is not designed for OLTP (Online Transaction Processing) systems** and is best suited for batch processing and analytical workloads.
* **Hive does not support real-time queries or row-level updates**, as it is optimized for large-scale data processing.

**Ways to Run Hive**

Hive can be accessed through three primary interfaces:

1. **Hive CLI (Command Line Interface)**
   * Simply type hive in the command line of your **edge node**, and you will enter the Hive terminal.



A black screen with white text

Description automatically generated

1. **HUE (Hadoop User Experience) - Web UI**
   * Hive can also be accessed via **HUE**, a web-based interface for querying and managing data.
   * The default address for HUE in a typical setup is: http://192.168.21.3:8888.
   * You can execute the same Hive queries in HUE’s **Query Editor** by selecting the **Hive** option.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **Thrift Server (for Remote Connections)**
   * Hive can be accessed remotely using the **Thrift Server**, which allows external clients to connect.
   * The recommended way to connect to Hive via the Thrift Server is through **Beeline**, a JDBC-based command-line client.
   * Once inside the Beeline terminal (e.g., in a **Cloudera setup**), you can execute SQL-like queries just as you would in the Hive CLI.

**3. Thrift Server (For Remote Connections Using Beeline)**

* The **HiveServer2 (Thrift Server)** allows remote clients to connect and execute queries.
* The recommended client for this is **Beeline**, a JDBC-based command-line tool.
* To connect to Hive using Beeline, use the following steps:

**Step 1: Launch Beeline**

bash cmd 🡺 beeline

**Step 2: Connect to HiveServer2**

!connect jdbc:hive2://<hive-server-host>:10000 default;user=<your-username>;password=<your-password>

Replace <hive-server-host> with the actual Hive server hostname or IP address.  
If authentication is not required, you can use:

!connect jdbc:hive2://<hive-server-host>:10000/default

**Hive Warehousing**

**Listing Databases**

To list all available databases in Hive, use the following command:

SHOW DATABASES;

A black screen with white text

Description automatically generated

**Note:** In this guide, we will create a database named hive\_db (if not existed) and use it throughout for practice.

A black background with white text

Description automatically generated

**Creating a Database**

To create a new database in Hive, use:

CREATE DATABASE hive\_db;

A black background with white text

Description automatically generated

**Checking the Current Database**

To see which database you are currently working in, run:



**Switching to a Specific Database**

Since we want to use hive\_db, we need to switch to it using:

USE hive\_db;

A black and white text

Description automatically generated

Now, all subsequent operations will be executed within the hive\_db database.

**Loading Data into Hive Tables**

Hive provides two primary methods for loading data into tables:

**1️) Loading Data Using the LOAD Command**

In this approach, you first **create a Hive table** and then load data into it using the LOAD command.

🔹 **Loading Data from the Local File System:**

* Use the **LOCAL** keyword in the LOAD command to specify that the file is in the local file system.

LOAD DATA LOCAL INPATH '/path/to/local/file.csv' INTO TABLE table\_name;

This moves the file from the local system to Hive’s warehouse directory in HDFS.

🔹 **Loading Data from HDFS:**

* Do **not** use the LOCAL keyword when loading data from HDFS.

LOAD DATA INPATH '/hdfs/path/to/file.csv' INTO TABLE table\_name;

Here, the file remains in HDFS and is simply **moved** to the Hive table directory.

**2️) Loading Data Without Using the LOAD Command**

Alternatively, if you already have data stored in **a specific directory**, you can **link the table directly to the data’s location** while creating the Hive table.

🔹 **Defining External Table with Data in HDFS or Local File System**

CREATE EXTERNAL TABLE table\_name (

column1 STRING,

column2 INT

)

STORED AS TEXTFILE

LOCATION '/hdfs/path/to/data/';

**1️) Loading Data into a Hive Table Using the LOAD Command**

To load data into a Hive table using the LOAD command, follow these steps:

**Step 1: Create a Sample Data File**

Use the touch or vi command to create a text file (emp.txt) and add data to it:

vi emp.txt



(Add data inside the file and save it.)

**Step 2: Verify the Data**

Display the file contents using the cat command:

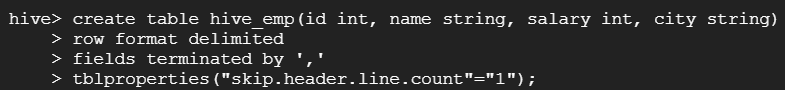
cat emp.txt

A screenshot of a computer screen

Description automatically generated

**Step 3: Create a Hive Table and Exclude Headers**

If the dataset contains **header rows**, they can be skipped using **table properties** while creating the table:



A black background with white text

AI-generated content may be incorrect.

🔹 **"skip.header.line.count"="1"** ensures that Hive **ignores the first row** if it contains column headers.

**Step 4: Load Data into Hive Table**

Depending on the data location, use one of the following commands:

**🔹 Loading Data from the Local File System**

LOAD DATA LOCAL INPATH '/path/to/emp.txt' INTO TABLE employee;

* **Moves** the file from the local system to Hive’s warehouse in HDFS.

**🔹 Loading Data from HDFS**

LOAD DATA INPATH '/hdfs/path/to/emp.txt' INTO TABLE employee;

* **Moves** the file within HDFS but does not copy it from a local machine.

**Step 5: Verify the Loaded Data**

After loading, check if the data is inserted correctly:

SELECT \* FROM employee;

* This method allows **Hive to read data directly** from the specified location **without moving it**.
* The data can reside in either **HDFS** or a **local file system**.
* This approach is commonly used when integrating **Hive with external data sources**.

**Key Differences Between the Two Approaches**

| **Method** | **Data Movement** | **Best Used For** |
| --- | --- | --- |
| **LOAD Command** | Moves data to Hive’s internal storage (HDFS warehouse) | When you want Hive to manage the data |
| **External Table (LOCATION)** | Keeps data in the original location | When data is shared across multiple applications or already exists in HDFS |

**Verifying the Table Location Using the NameNode UI**

To check the location of the recently created Hive table using the **Hadoop NameNode UI**, follow these steps:

1️. **Open the NameNode UI**

* Access the Hadoop NameNode UI in your web browser (e.g., http://<namenode-host>:9870).

2️. **Navigate to the File System Browser**

* Go to **Utilities** → **Browse the File System**.

3️. **Locate the Table Directory**

* Navigate to the Hive warehouse directory:

/user/hive/warehouse/hive\_db.db

* Inside this directory, you will find a subdirectory for your table (e.g., /user/hive/warehouse/hive\_db.db/employee).

This confirms that the table data is stored in HDFS under the Hive warehouse directory.



A green rectangular object with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

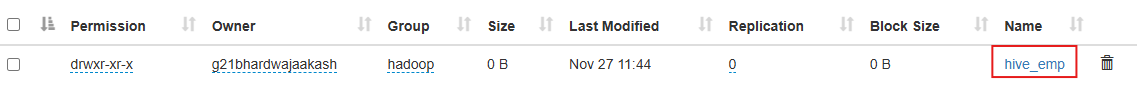
Description automatically generated

A close up of words

Description automatically generated

A screenshot of a computer

Description automatically generated



**Checking Table Metadata in Hive**

To retrieve metadata information about a Hive table, use the following command:

DESCRIBE FORMATTED table\_name;



This command provides **detailed metadata**, including:

**Columns and their data types**  
**Partitioned column name (if applicable)**  
**Bucketed column name (if applicable)**  
**Table location in HDFS**  
**Table type** (Managed/Internal or External)  
**SerDe (Serializer/Deserializer) library used**

A black background with white text

Description automatically generated

A computer screen with a red line

Description automatically generated

**Alternative Metadata Commands**

1️.**Basic Table Schema**

DESCRIBE table\_name;

🔹 Displays **column names and their data types**.

2️. **Extended Table Metadata**

DESCRIBE EXTENDED table\_name;

🔹 Provides **more detailed metadata**, including **storage format, table properties, and location**.

3️. **Metadata for Partitioned Tables**

SHOW PARTITIONS table\_name;

🔹 Lists all partitions in a partitioned table.

NOTE: **When a table is created in Hive, a corresponding directory is automatically generated in HDFS** under the Hive warehouse location.

* The default location for **managed tables** in HDFS is:

/user/hive/warehouse/hive\_db.db/table\_name

**A screenshot of a computer screen

Description automatically generated**

* Similarly, any **database created in Hive** is mapped to an HDFS directory:

/user/hive/warehouse/hive\_db.db

**Key Points:**

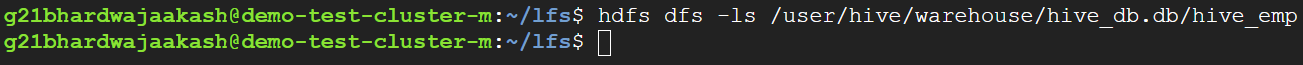
**Managed tables** store both metadata and data inside Hive’s warehouse directory in HDFS.  
 **External tables** do **not** move data into the Hive warehouse; instead, they reference an existing HDFS location.  
**Deleting a managed table** deletes both its **metadata and data**, whereas deleting an **external table** only removes the metadata, keeping the data intact in HDFS.

**Verifying Table Data in HDFS**

**If a Hive table has just been created but no data has been inserted, its corresponding directory in HDFS will be empty.**

To check the contents of the directory, use the following command in the Hadoop CLI:

hdfs dfs -ls /user/hive/warehouse/hive\_db.db/table\_name



**Expected Behavior:**

* If the table **has no data**, the command will **return no files** inside the directory.
* Once data is inserted using LOAD DATA or INSERT INTO, the directory will contain **actual data files**.

**Loading Data into a Hive Table**

There are **two ways** to load data into a Hive table using the LOAD DATA command:

1️. **From the Edge Node to a Hive Table**

* This method loads data from the local filesystem into Hive.



**Viewing Data in a Hive Table**

After loading data, you can retrieve and display it using:

SELECT \* FROM hive\_emp;

A black screen with white text

Description automatically generated

If headers were disabled during table creation, only the data will be displayed.

**Displaying Column Headers**

If you want to display column headers along with the data, use the following command

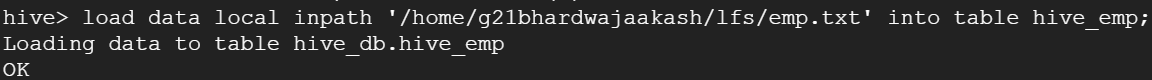


A black screen with white text

Description automatically generated

**Note:** Beeline formats the output in a tabular structure, making it easier to read.

**Data is not automatically formatted in tabular form** in the Hive CLI. Beeline provides a better display.  
**Reloading the same file using LOAD DATA will cause duplication** in the table, as Hive appends data instead of overwriting it.  
**To avoid duplication, consider using INSERT OVERWRITE instead of LOAD DATA if needed.**



A screen shot of a computer

Description automatically generated



A black screen with white text

Description automatically generated

**Loading Data from HDFS to Hive Table**

To load data from HDFS into a Hive table, follow these steps:

1️. **Create a New Table in Hive**

* First, create the table in Hive using a CREATE TABLE statement.
* A screen shot of a computer

  Description automatically generated

2️. **Push Data from Edge Node to HDFS**

* Before loading data into Hive, you need to place the file in HDFS.
* Create a directory in HDFS and upload the file from the edge node:

hdfs dfs -mkdir /user/hive/input

hdfs dfs -put /local\_path/your\_file.txt /user/hive/input

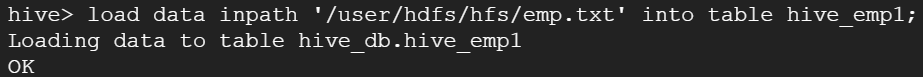




3️. **Load Data from HDFS into Hive Table**

* After the file is available in HDFS, you can load the data into the Hive table:

LOAD DATA INPATH '/user/hive/input/your\_file.txt' INTO TABLE hive\_emp;



* **Note:** Since you are loading from HDFS, **exclude the LOCAL keyword**.

4️. **Verify Data in Hive Table**

* Use the following query to check the loaded data:

SELECT \* FROM hive\_emp;

A black background with white text

Description automatically generated



**Key Notes on HDFS Data Loading**

**Data Transfer Type:**

* **From Edge Node to Hive Table:** Data is **copied** from the edge node to Hive’s warehouse.
* **From HDFS to Hive Table:** Data is **moved** from HDFS to the Hive table. This means it is **cut** from the original location in HDFS and **pasted** into the table.

**Verify the HDFS Directory After Loading Data**

* If you check the directory in HDFS where the input file was originally stored, you will notice that the file has disappeared.
* The data has been moved to the Hive table and can be found at:

/user/hive/warehouse/hive\_db.db/hive\_emp1

**Introduction to Hive Collection Data Types**

Apache Hive provides a **powerful SQL-like interface** for handling large-scale data stored in **Hadoop Distributed File System (HDFS)**. When working with structured or semi-structured data, Hive supports **complex data types** that allow efficient data organization and retrieval.

Among these, **Collection Data Types**—such as **ARRAY, MAP, and STRUCT**—enable users to store **multiple values, key-value pairs, or nested records** within a single column. These data types are particularly useful for handling hierarchical, JSON-like, or denormalized datasets without requiring multiple tables.

**Hive Data Types : ARRAY**

In Hive, an **ARRAY** is a collection of elements that share the same data type. Arrays are useful when storing multiple values in a single column.

**Steps to Work with Arrays in Hive**

**Create a Source File**

* Create a text file (array\_file.txt) that contains sample data with an array-type column.(Please create a separate directory on your Edge Node with name *lfs* or any other name to create sample data)



A screenshot of a computer

Description automatically generated

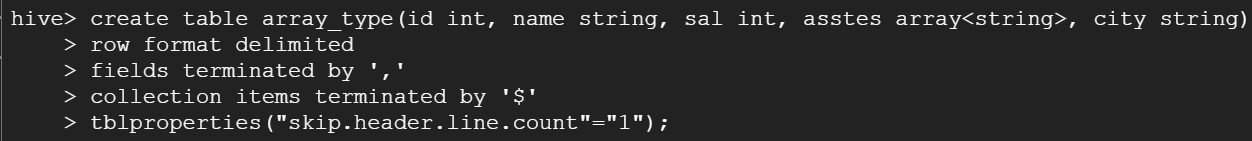
**Create a Hive Table with an Array Column**

* Define the column as an **ARRAY** in the table schema.



* Use the correct **collection item delimiter** to specify how elements inside the array are separated.

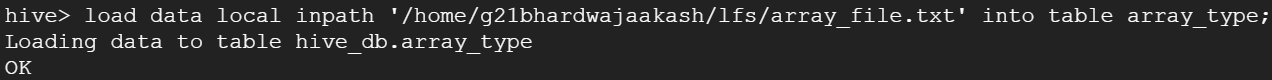




🔹 In this example, assets are an **ARRAY of strings**, and items inside the array are separated by a $ symbol.

**Load and Query Data in the Array Column**

* Load data into the table:



* To display the data

A screen shot of a computer code

Description automatically generated

* Query the table to retrieve array elements:

A computer screen with white text

Description automatically generated

* To access a **specific index** inside the array:



A black screen with white text

Description automatically generated

🔹 **Note:** Hive arrays are **zero-indexed**, meaning the first element is at index **0**.

**Hive Data Types: MAP**

In Hive, a **MAP** is a collection of **key-value pairs**, where:

* The **key** must be a **unique** and **non-null** value.
* The **value** can be of any data type.

Maps are useful when storing **associative data** where each key maps to a specific value (e.g., employee ID to department, product ID to price, etc.).

**Steps to Work with MAP Columns in Hive**

**Create a Source File**

* Create a text file (array\_map.txt) that contains sample data with a **map-type** column.



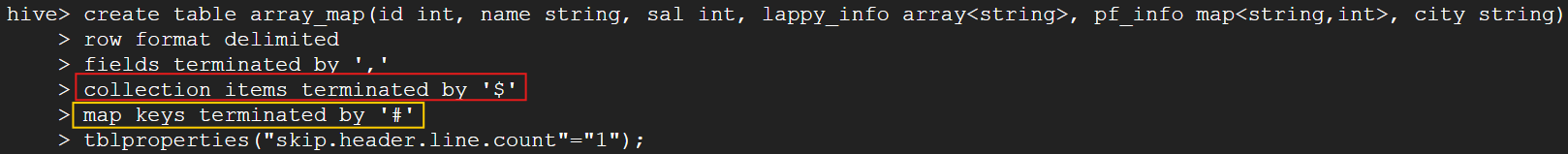
A screen shot of a computer

Description automatically generated



**Create a Hive Table with a MAP Column**

* Define the column as a **MAP** in the table schema.
* Specify how **key-value pairs** and their **entries** are separated.



Here we have 2 complex columns i.e. array-type and map-type. So, we will handle both the columns during table creation.

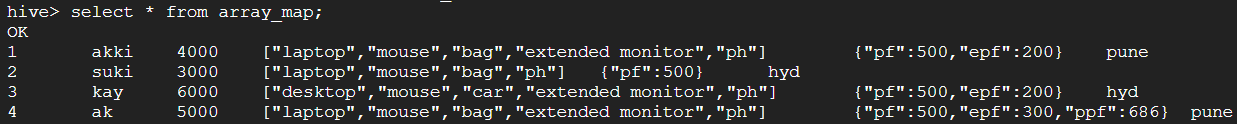
🔹 Here, lappy\_info is an **ARRAY** of strings where items inside the array are separated by $ symbol. pf\_info is a **MAP** where the **keys** are pf type names (e.g., pf,epf) and the **values** as integer   
🔹 **Entries inside the MAP are separated by ‘$’**, while **keys and values are separated by ‘#**’

**Load and Query Data in the MAP Column**

* **Load data into the table:**



* **Retrieve all MAP data:**



**Access a specific key inside the MAP column:**



A black screen with white text

Description automatically generated

🔹 This query retrieves the **value**  corresponding to the "pf" key inside the **pf\_info** column.

**Hive Data Types: STRUCT**

In Hive, a **STRUCT** is a complex data type that allows storing **multiple attributes of different data types** within a **single column**. It is useful when handling **nested** or **hierarchical** data.

**Key Characteristics of STRUCT in Hive**

Stores **multiple fields of different types** within a single column.  
Each field can be accessed using **dot notation (.)**.  
Unlike **MAPs**, field names are **predefined** in the schema.

**Steps to Work with STRUCT Columns in Hive**

**Create a Source File**

Create a file (struct\_file.txt) with sample structured data.

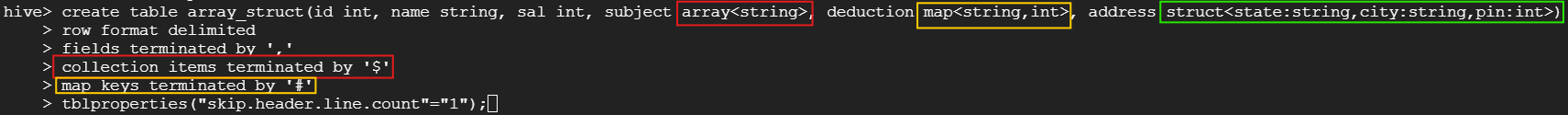


A black background with white text and red squares

Description automatically generated

**Create a Hive Table with a STRUCT Column**

Define a table with a **STRUCT** column in the schema.

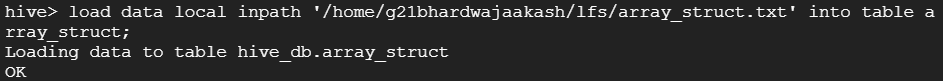


🔹 Here, the column **address** is a **STRUCT** with fields:

* state (String)
* city (String)
* pin (Integer)

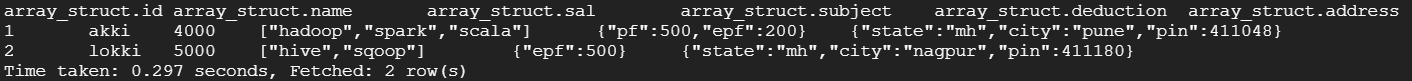
**Load and Query Data in a STRUCT Column**

* **Load Data into the Table:**



* **Retrieve Entire STRUCT Data:**

SELECT \* FROM array\_struct;



* **Access Specific Fields inside STRUCT:**



🔹 This extracts the **pin** from the **address** STRUCT column.

**Additional Notes**

**Collection Data Types in Hive**

* **ARRAY:** Stores multiple values of the same type.
* **MAP:** Stores key-value pairs.
* **STRUCT:** Stores nested fields with different data types.

**Delimiters Matter**

* Use COLLECTION ITEMS TERMINATED BY to define how elements inside an array are separated in the source file.
* Choose delimiters carefully to avoid conflicts with actual data values.

**Key Considerations for MAP Type Columns**

* Keys **must be unique** within a single row.
* Choose **appropriate delimiters** to avoid conflicts with actual data values.
* **Indexing in MAPs starts with the key, not a numeric index** (unlike arrays).

**Difference Between STRUCT and MAP in Hive**

* **STRUCT:** Field names are **predefined** and accessed using **dot notation**.
* **MAP:** Field names (**keys**) are **dynamic** and accessed using **square brackets ([])**.

**Common Use Cases for STRUCT**

* Storing **nested user details** (e.g., name, age, address).
* Capturing **product attributes** (e.g., weight, price, category).
* Grouping related fields **without creating additional tables**.

**Types of Tables in Hive**

Hive supports two types of tables, each serving distinct purposes in data management and storage:

* 1. Managed or Internal Tables
  2. External Tables

**Managed Tables in Hive**

A **Managed Table** (also known as an Internal Table) in Hive is created by default when no table type is specified. Managed tables are stored in the Hive warehouse directory:

/user/hive/warehouse/

1. By default, tables created in Hive are Managed (Internal) tables.
2. Managed tables are stored in the /user/hive/warehouse/ directory in HDFS.
3. If you want to store the table in a different location, use the LOCATION argument during table creation.
4. When you drop a Managed table, both the table schema and the underlying data (HDFS directory) are deleted.

**Note:** When a Hive Managed table is created, Hive automatically creates a directory in HDFS with the same name as the table under /user/hive/warehouse/hive\_db.db

**Key Characteristics of Managed Tables:**

* **Default Table Type:** If you don't specify EXTERNAL during table creation, Hive creates a Managed table by default.
* **Storage Location:** Managed tables are stored under the default Hive warehouse directory.
* **Custom Location:** You can specify a custom location using the LOCATION clause during table creation.
* **Drop Behavior:** Dropping a Managed Table deletes both the table metadata and the data stored in HDFS.

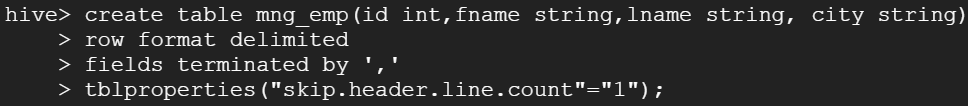
**Creating a Managed Table with Default Location:**

**On your Edge node, inside** *lfs* **folder, create a sample file** *int\_emp.txt* **for this exercise**

A screenshot of a computer

Description automatically generated

**Create Hive table to hold the data**



**Inspect for additional details :** Verifies that *mng\_emp* is an Manage Table.

A screenshot of a computer

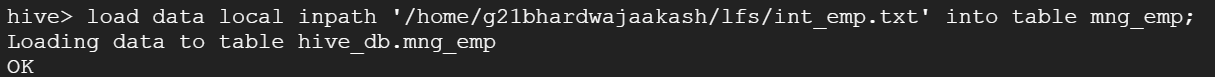
Description automatically generated

Following fig shows that a directory got created with the name same as the name of the table under */user/hive/warehouse/hive\_db.db*

A black screen with white text

Description automatically generated

**Loading the data into the Managed Table:** *mng\_emp*

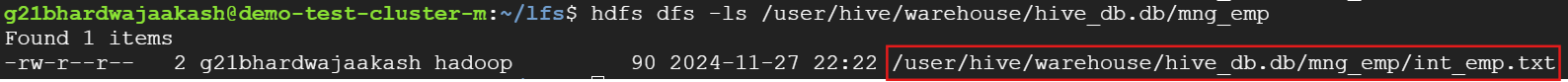


**Fetch data from the table :** This shows that data has been loaded successfully

A black screen with white text

Description automatically generated

Data can also be seen under the path: */user/hive/warehouse/hive\_db.db/mng\_emp*



**Now let’s drop the table**

A black background with white text

Description automatically generated

Let’s see what happened under the path where data was present before :

*/user/hive/warehouse/hive\_db.db/mng\_emp*



**Conclusion**

 Managed tables are ideal when Hive is responsible for managing the entire data lifecycle.

 Use Managed tables when you do not require the data outside of Hive after the table is dropped.

 Dropping a Managed table will remove both the table and its underlying data from HDFS.

 When creating a Managed table, Hive automatically creates a directory in HDFS at the path /user/hive/warehouse/hive\_db.db, with the directory name matching the table name. This directory stores the table's data files and metadata.

**External Tables in Hive**

An External Table in Hive allows you to manage the schema, while the data remains external. Hive does not manage the underlying data for External Tables.

* External tables are created by specifying the EXTERNAL keyword during table creation.
* When you create an External table, Hive stores metadata in the Hive metastore, but the data remains in the specified HDFS location.
* After loading data into an External table, the data is stored in the specified directory (e.g., ext\_emp directory in HDFS).

**Important:** Dropping an External table only removes the table metadata from Hive but retains the underlying data in HDFS. This allows you to access the data directly from HDFS even after the table is dropped.

* **Key Characteristics:**
  + Data is stored at a location specified by the user, often outside the Hive warehouse directory.
  + When an External Table is dropped, only the schema is removed; the data remains intact.
  + Useful for sharing data across multiple Hive tables or systems.

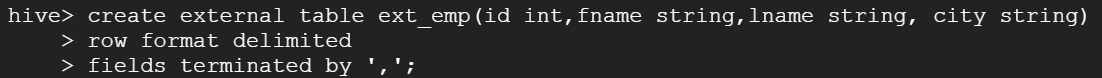
**Creating an External Table with Default Location:**

**On your Edge node, inside** *lfs* **folder, create a sample file** *ext\_file.txt* **for this exercise**

A screenshot of a computer

Description automatically generated

**Create Hive table to hold the data**



**Inspect for additional details :** Verifies that *ext\_emp* is an Manage Table

A screenshot of a computer

Description automatically generated

Following fig shows that a directory got created with the name same as the name of the table under */user/hive/warehouse/hive\_db.db*

A black and white screen with numbers

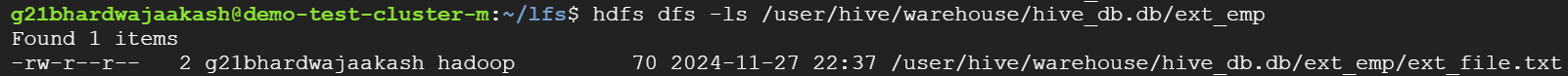
Description automatically generated

**Loading the data into the External Table:** *mng\_emp*



**Fetch data from the table :** use *select \* mng\_emp,* command

Data can also be seen under the path: */user/hive/warehouse/hive\_db.db/mng\_emp*



**Now let’s drop the table**

A black background with white text

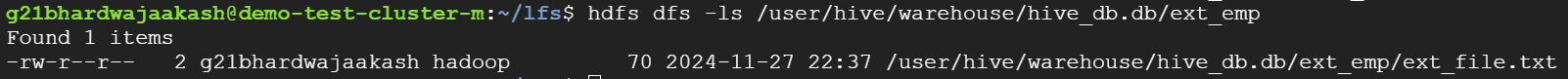
Description automatically generated

Following fig shows that the directory is still there under */user/hive/warehouse/hive\_db.db*

A black and white screen with numbers

Description automatically generated

Also, you would be able to see the data



**Conclusion**

 External tables are ideal when you need Hive to manage the schema but not the data itself.

 Use External tables when you want to retain the data even after dropping the table from Hive.

 Dropping an External table removes the table schema from Hive but does not affect the underlying data in HDFS or other storage systems.

 When creating an External table, you must specify the location of the data, which could be in HDFS or an external storage system, as Hive does not automatically manage the data directory.

**Creating Managed and External Tables with a Specified Location**

When creating a table in Hive, you can specify a custom storage location using the LOCATION keyword. The provided path must be enclosed in single quotes and can point to either a local filesystem path or an HDFS path.

**Syntax:** LOCATION 'path'

**Path Types:**

* **Local Path**: Refers to a directory on the local filesystem.
* **HDFS Path**: Refers to a directory in Hadoop Distributed File System (HDFS).

**Key Considerations:**

1. You must manually create the target directory before using it in the LOCATION clause.
2. The specified path should point to a directory, not an individual file.
3. Data files should be placed inside this directory before querying.
4. Unlike standard Managed tables, when using LOCATION, Hive does not move or delete data during table operations.
5. There is no need to use the LOAD command to import data, as the data is already present in the specified location.

**Example Paths:**

* **Local Filesystem:** /home/g21bhardwajaakash/lfs/mgd\_dir
* **HDFS Filesystem:** /bootcamp/mng\_dir

**Steps to Create a Table with a Custom Location:**

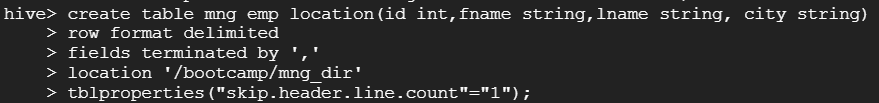
1. **Create the Directory in HDFS**



1. **Load Data from Local Filesystem into HDFS**



1. **Create the Table with the LOCATION Clause**



1. **Query the Data**

A black screen with white text

Description automatically generated

This approach ensures that Hive references the data without moving or altering it.

**Why You Should Avoid Using LOCATION for Managed Tables**

1. **Inconsistent Management**: If the specified path is outside Hive’s control (e.g., local filesystem or an edge node), you might face issues when querying the data.
2. **Unexpected Data Deletion**: If you drop the table, all data inside the custom LOCATION will be deleted, which may not be desirable.
3. **Best Practice**: If you need a **custom location**, it's recommended to use an **External Table**, which preserves the data even if the table is dropped.

**Key Differences Between Managed and External Tables:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Managed Table** | **External Table** |
| **Storage Location** | Hive warehouse directory | User-specified location |
| **Data Management** | Managed by Hive | Managed externally by the user |
| **Data Deletion** | Data is deleted when the table is dropped | Data is retained when the table is dropped |
| **Use Cases** | Temporary/intermediate data | Persistent data shared across systems |

**Additional Insights:**

* Managed Tables are best suited for data that is exclusive to Hive and can be safely managed within the Hive environment.
* External Tables are ideal when the data is shared across different tools, systems, or processing frameworks, providing flexibility and preventing accidental data loss.

**Hive Partitioning**

Tables, partitions, and buckets are fundamental components of Hive data modeling.

**What is Partitioning?**

Partitioning in Hive is a technique used to divide a table into smaller, more manageable parts based on specific column values, known as **partition keys**. This helps in organizing large datasets efficiently by storing data in separate directories based on the partition column(s).

For example, partitioning a table by **date** or **country** ensures that records are stored in corresponding directories, making data retrieval more efficient.

Hive also supports **multi-level partitioning**, where data is further subdivided based on multiple partition keys. For instance, after partitioning records by **date**, they can be further split by **country**, leading to a hierarchical directory structure.

**Advantages of Partitioning**

1. **Improved Query Performance:** Since data is stored in smaller segments, queries scanning specific partitions execute faster compared to searching the entire dataset.
   * Example: If a table is partitioned by country, querying records for country='IN' will only scan the directory country=IN, instead of reading the entire table.
2. **Optimized Storage and Processing:** Partitioning distributes data **horizontally**, reducing the load on query execution by limiting the number of files being processed.

**Disadvantages of Partitioning**

1. **Metadata Overhead:** A large number of partitions creates numerous files and directories in HDFS, increasing the burden on the **NameNode**, as it must store all metadata in memory.
2. **Query Optimization Trade-offs:** While partitioning improves performance for queries using **WHERE** conditions on partition keys, it may slow down other queries, such as those involving **GROUP BY** operations, due to the distributed storage structure.

**Types of Partitioning in Hive**

Hive supports two types of partitioning:

**1. Static Partitioning**

* In **static partitioning**, the partition column values are explicitly provided during data loading.
* The user manually specifies the partition key, and the data is inserted into the corresponding partitioned directory.
* This method is efficient when dealing with smaller datasets or when the partition values are known beforehand.

**2. Dynamic Partitioning**

* In **dynamic partitioning**, Hive automatically determines partition column values based on the data being inserted.
* This is useful when dealing with large datasets where partition values are not known in advance.
* Dynamic partitioning simplifies the ETL process by automatically segregating data into partitions without manual intervention.

1. **Static Partition**

**Create 3 different files on the Edge-node**

A screenshot of a computer

Description automatically generated

**Displaying the data**

A black screen with green text

Description automatically generated

A black screen with green text

Description automatically generated

A screenshot of a computer

Description automatically generated

Now that we have data so next step is to create **Hive Partitioned Table**

While creating the table specify partition column and its datatype on which you would want to do partition. That column should not be the part of the table definition. Example if we are partitioning the table based on country so it should not be the part of the table schema

A computer screen with white text

Description automatically generated

If you check additional information using describe formatted command, you will see the partitioned information

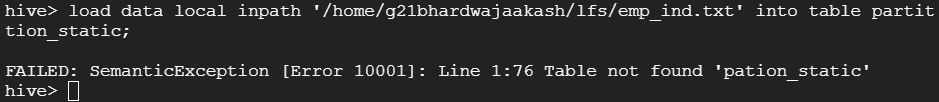
A screen shot of a computer

Description automatically generated

A screen shot of a computer

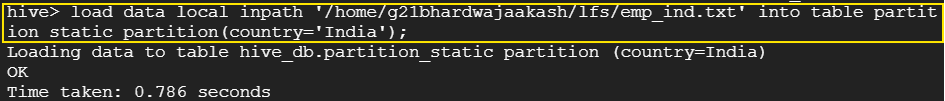
Description automatically generated

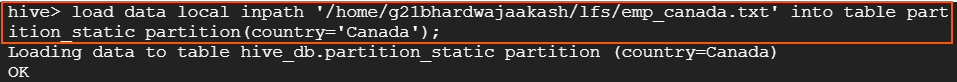
**Scenario 1**: If the client is providing you with 3 different files and they are asking to maintain the data in 1 table then we will have to execute 3 different load commands. Suppose in a real-time we have 100 of 1000s of partitions then it would become difficult to load the data by load command.

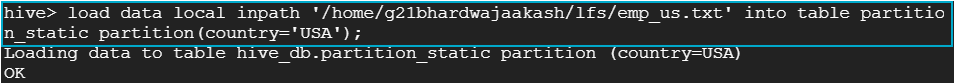


This will fail. To fix this you have to explicitly mention in the load command that data needs to go to what partition of partitioned column.



***country=India***

***country = Canada***

***country=US***

**Displaying the data**

A screen shot of a computer

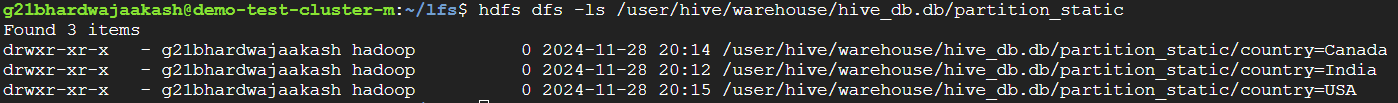
Description automatically generated

Also, whatever partitions you are using, a separate directory will be created for each partition. Therefore, the number of directories that will be created = number of partitions = number of times you have to use the load command.

*/user/hive/warehouse/hive\_db.db/partition\_static/*

A black screen with numbers and symbols

Description automatically generated



**Scenario 2 :** Client is sending just 1 file

* Create a file on the Edge-Node *emp\_all.txt*
* Create a Hive Table to hold the data

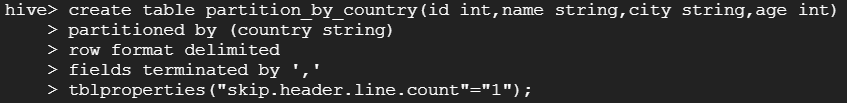


**Displaying the data**

A screenshot of a computer

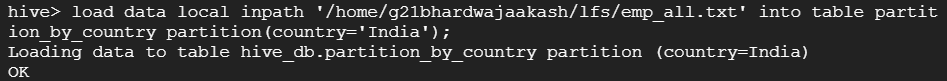
Description automatically generated

**Note**: The partition column should not be the part of the create table statement



**Loading the Data**

Now let’s load the data into the table but in just 1 partition



The problem with this approach is that there will be all the India values when you the read the data from Hive table

A screen shot of a computer

Description automatically generated

**To check how the data look like in the hefts**



So, under the directory *partition\_by\_country* there is only one directory with the name *country=India* and under that there is one file.



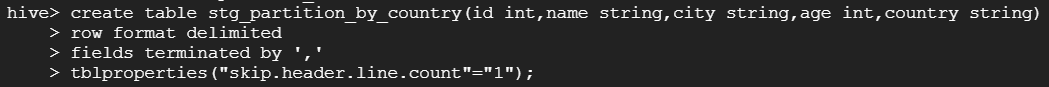
The data is appropriately stored on HDFS but if you go to the Hive table you are able to see the data but for the country you are only able to see the partition as India. Because Hive implements Schema-on-read, meaning it will check and validate schema when you are trying to fetch the results/query the results

A black screen with white text

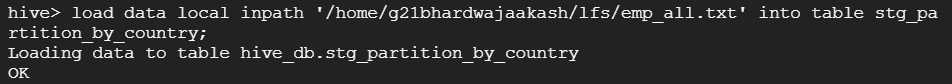
Description automatically generated

To handle this issue, you will create an intermediate table and from there you will load the data into the partitioned table.

So, create a simple staging table



**Loading the data**



A screen shot of a computer

Description automatically generated

Now we will load the data from this Staging table to the Hive Partitioned Table

First truncate the old table

A black and white text

Description automatically generated

No data is present

A black background with white text

Description automatically generated

**Now we want to insert the data in the Hive Partitioned table so**



**And where is data coming from**



Note: In the select statement you cannot use \* because you cannot specify the partition column in the select clause while reading the data





Data in HDFS





Just one file



Data only belongs to India

A black background with white text

Description automatically generated

Same data in Hive table as well

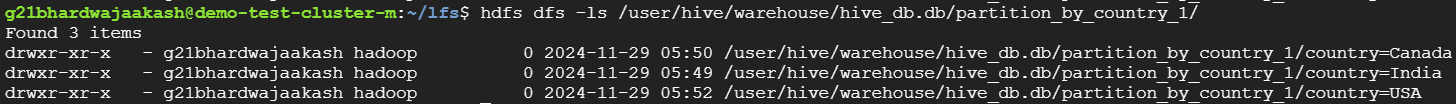
**Loading data for Canada**



**Loading data for USA**



**Data inside HDFS**



**Adding and Dropping Partitions**

**A black screen with white text

Description automatically generated**

**Create a new file**

****

**Load the data into the dedicated partition**

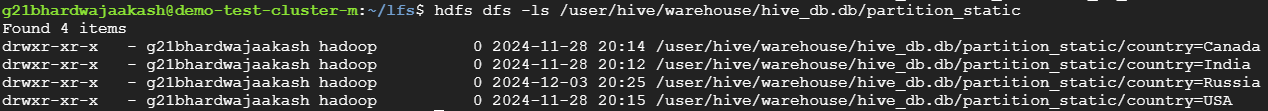
****

**A black screen with white text

Description automatically generated**

Hence, you can see that another partition with the name *country=Russia* has been added

Verify if a separate directory got created for the new partition



Dropping the partition

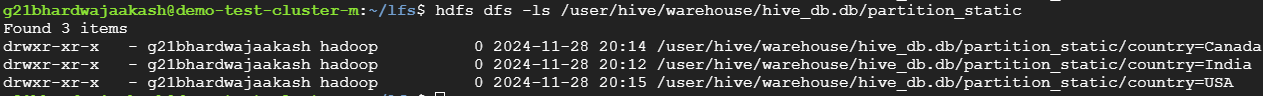


Confirmation

A black screen with white text

Description automatically generated

Confirmation at HDFS level



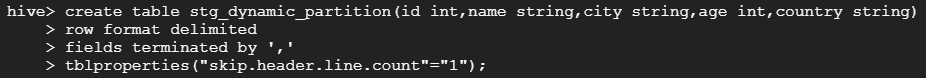
**Dynamic Partitioning**

**Our data is**

**A screenshot of a computer

Description automatically generated**

Create an Intermediary table



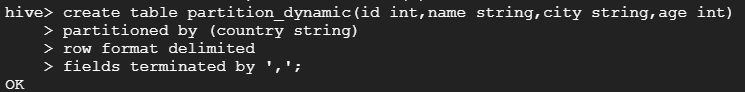
Loading the data into the table



This is how data looks inside Hive Staging table



Now create a partitioned table on country column.

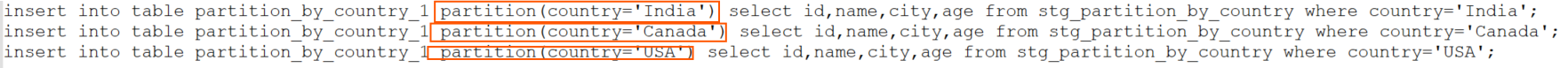


A screen shot of a computer

Description automatically generated

**Loading the data from Staging table to the Partitioned table**

In the static partition while loading the data you specify the partition value. But in the dynamic partition you do not provide the partition column value but the partition column

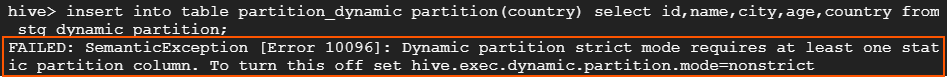




In static partition you also mention the where clause but in dynamic partition you don’t have to mention where clause.



In static partition you do not specify the partition column name in select clause while loading the data using SQL command but in case of dynamic partition you specify the partition column at the last of the select clause



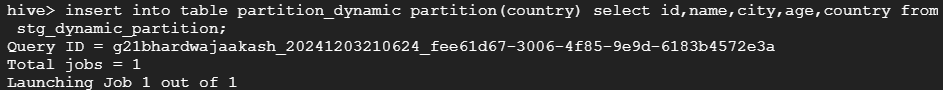
You have to set some properties to make dynamic partitioning work

A close up of words

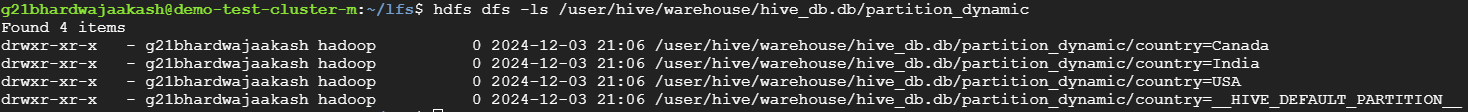
Description automatically generated



After setting the above property the insert statement will work



Therefore, here you are not explicitly specifying partition column value neither you are specifying the where clause automatically on the basis of country it will load the data into the Hive table and also at the HDFS level it will create different directories.



**NOTE:**

**For the Bucketed table, the number of reducers = no. of buckets you specify during table creation**

**in most cases**, the number of reducers **should be equal to the number of buckets** specified during table creation, but **this is not always enforced automatically** in Hive.

**Understanding Bucketing and Reducers in Hive**

When you create a **bucketed table** in Hive, you specify the number of **buckets** as follows:

sql

CopyEdit

CREATE TABLE bucketed\_table (

id INT,

name STRING

)

CLUSTERED BY (id) INTO 4 BUCKETS;

* Here, the table is **bucketed into 4 buckets** based on id.

**When does the number of reducers match the number of buckets?**

1. If Hive is responsible for writing the bucketed data **(using INSERT OVERWRITE or INSERT INTO)**, then **you should set the number of reducers equal to the number of buckets** to ensure proper bucketing:

sql

CopyEdit

SET hive.enforce.bucketing = true; -- Required for bucketing to work

* + If this setting is enabled, Hive **tries** to set the number of reducers **equal to the number of buckets**.
  + However, **Hive does not always guarantee an exact match**—it depends on workload, data size, and settings.

1. If you are using **INSERT OVERWRITE**, and the setting **hive.enforce.bucketing=true** is enabled, Hive **tries** to use the same number of reducers as buckets.

**When might reducers NOT be equal to the number of buckets?**

* If **hive.enforce.bucketing=false** (default in some versions), Hive might not enforce this rule.
* If the **data size is too small**, Hive’s optimizer might **reduce the number of reducers**.
* If there is **skewed data**, reducers may be adjusted dynamically.

**Conclusion**

✅ **Typically, number of reducers = number of buckets**, but Hive does not always enforce this unless hive.enforce.bucketing=true.  
✅ To ensure correct bucketing, always enable bucketing explicitly.  
✅ If reducers do not match buckets, bucketed table operations might not work as expected (e.g., bucket join optimizations).

**SerDe in Hive**

**Serialization and Deserialization in Hive**

**Serialization**

Serialization is the process of transforming structured objects into a byte stream for transmission over a network or for writing data to persistent storage.

In Hive, serialization refers to converting data from Hive’s internal format into a specific format (e.g., ORC, Parquet, Text, etc.).

Think of it as a scenario where two entities speak different languages and cannot understand each other. To enable communication, they need a common method of translation. Similarly, in the case of a **MySQL database and Hadoop**, their data types and formats differ. However, since both systems ultimately run on a machine that understands only **binary data**, the conversion process ensures compatibility. This conversion process is known as **serialization**.

**Deserialization**

Deserialization is the reverse process—converting a byte stream into structured objects so that the data can be read and processed.

Deserialization in Hive happens when reading data—i.e., converting stored data into a readable format.

**Important Notes on Serialization and Deserialization**

* **Serialization in Apache Hive occurs when using the INSERT command.**
* **When using LOAD DATA**, Hive does not modify the data. It simply moves the file to the target table’s location without serialization or deserialization.
* **When using INSERT**, Hive writes data into the table and applies the configured SerDe (Serializer/Deserializer), meaning serialization happens here.

**Example: Converting CSV Data to ORC/Parquet Format**

**Scenario:**

A client sends data in **CSV format**, but you need to store it in **ORC or Parquet format**.

**Steps:**

1. **Create a Hive table** and specify the input and output format using the STORED AS clause. (e.g., STORED AS PARQUET for Parquet format).
2. **Create a staging table** to load the CSV data. This table will be a simple Hive table.
3. **Use the INSERT statement** to load data from the staging table into the Parquet/ORC Hive table.

**Row-Based vs. Column-Based File Formats in Hive**

**Row-Based File Format**

* Data is stored **row by row**.
* **Write operations are efficient** because new records are simply appended to the end.
* **Reading performance suffers** when only a few columns are required, as the entire row must be scanned.
* **Compression is not very effective** due to mixed data types within rows.
* **Example:** Suppose an **employee table** has **1000 records**, and you need to compute the **sum of salaries**. The system must scan **every row** to extract the **salary column**, which is inefficient.

**Column-Based File Format**

* Data is stored **column by column**, with all values of a single column placed sequentially.
* **Write operations are slower**, as a new record must be appended to each column separately.
* **Read performance is significantly better**, especially for queries that retrieve only a subset of columns.
* **Compression is more efficient**, as all values in a column typically have the same data type.
* **Example:** If data is stored **column-wise**, compression works well because similar data types are stored together, reducing redundancy.

**CSV SerDe (Serializer/Deserializer) in Hive**

**Example: orders\_data.csv**

A screen shot of a computer

Description automatically generated

A CSV file containing orders data with the following columns:

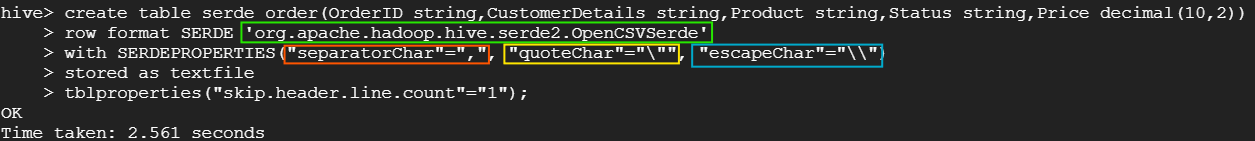
* **CustomerDetails:** Contains first and second names, separated by a comma ,. Since the comma is part of the column value, values must be enclosed in **double quotes** (" ").
* **Product:** Stored as a string.
* **Status:** Values should be displayed with double quotes. An **escape character** is used to handle this.

**Handling CSV Data in Hive**

When dealing with such data, you **cannot** use the following clause:

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

Instead, a **SerDe** is used during table creation to handle special cases.



A screenshot of a computer

Description automatically generated

**Storage in HDFS**

Each Hive table corresponds to a **dedicated directory in HDFS** where the data is stored.

A screen shot of a computer

Description automatically generated

**Steps to Load CSV Data into Hive**

**Load data into the table using LOAD DATA.**



**Query the table to verify the data.**

A screen shot of a computer

Description automatically generated

**JSON SerDe in Hive**

**Example: orders\_data.json**

A screen shot of a computer

Description automatically generated

A black screen with white text

Description automatically generated

* Before working with JSON in Hive, analyze its structure using an **online JSON parser**.
* Identify key elements such as:
  + **customer column**
  + **products column**
  + **tags column**

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

AI-generated content may be incorrect.

***customer column***



***products column***

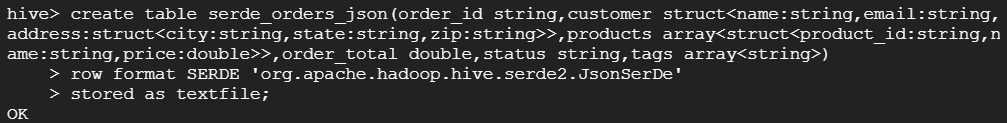


***tags column***

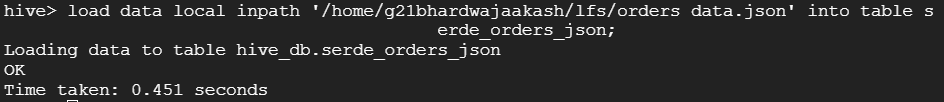


**Steps to Handle JSON Data in Hive**

1. **Create a Hive table** with the JSON SerDe.



1. **Load JSON data into the table.**



1. **Query the table column by column.**

A black background with white text

AI-generated content may be incorrect.



A black background with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

**ORC SerDe in Hive**

**Steps to Implement ORC Format in Hive**

1. **Create a staging table** to initially load the CSV data.

A screen shot of a computer

Description automatically generated

1. **Move the data to the Edge Node.**

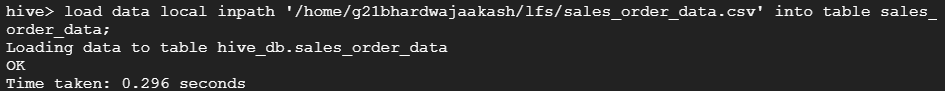
A screenshot of a computer

AI-generated content may be incorrect.

1. **Display the data** to understand the column structure.



1. Load the data to the Staging table



1. **Create an ORC Hive table** (Hive automatically detects the ORC library when STORED AS ORC is specified).

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **Insert data from the staging table into the ORC table.**

A screen shot of a computer

Description automatically generated

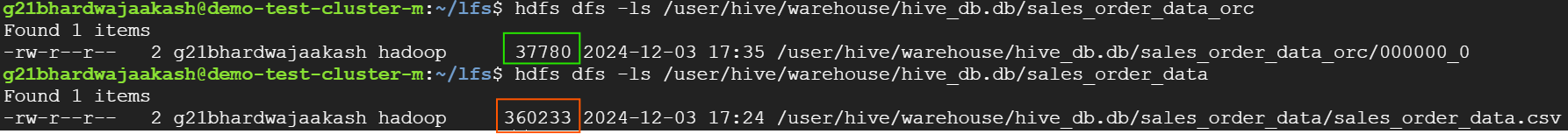
1. **Query and analyze data in the ORC format.**

A screen shot of a computer

AI-generated content may be incorrect.

**Benefit of ORC Format**

* Compared to simple text storage, ORC offers **significant compression**, improving storage efficiency.



**Parquet SerDe in Hive**

**Steps to Implement Parquet Format in Hive**

1. **Create a Parquet-based Hive table.**

**A screen shot of a computer

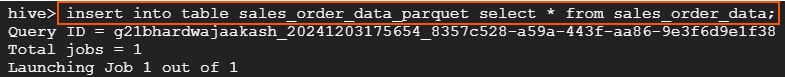
Description automatically generated**

1. **Use the default Parquet SerDe library provided by Hive.**

A screen shot of a computer

Description automatically generated

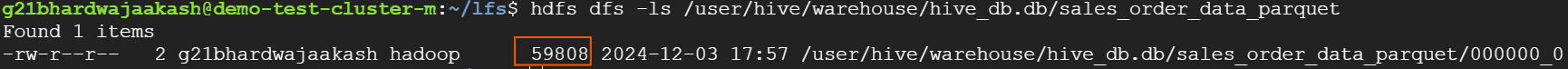
1. **Load data from the staging table into the Parquet table.**



1. **Query the Parquet table to verify the data.**

A black background with white text

Description automatically generated



**File Size Comparison Across Table Types**

| **Table Type** | **Size (Bytes)** |
| --- | --- |
| Simple Hive Table | **360,233** |
| ORC Hive Table | **37,780** |
| Parquet Hive Table | **59,808** |

This demonstrates that **ORC and Parquet** formats achieve **higher compression and better storage efficiency** than a simple Hive table.

**Conclusion**

* **Row-based formats (e.g., CSV)** are easy to write but slow to read.
* **Column-based formats (e.g., ORC, Parquet)** improve read performance and compression efficiency.
* **SerDes** play a critical role in **converting data formats** between external sources and Hive’s internal representation.
* **Serialization happens when inserting data**, and **deserialization happens when reading data** in Hive.