PRACTICAL NO :04

HIVE

Unit Structure :

4.0 OBJECTIVE

Hive allows users to **read, write, and manage petabytes of data using SQL**. Hive is built on top of Apache Hadoop, which is an open-source framework used to efficiently store and process large datasets. As a result, Hive is closely integrated with Hadoop, and is designed to work quickly on petabytes of data.

4.1 INTRODUCTION

Hive is a data warehouse system which is used to analyze structured data. It is built on the top of Hadoop. It was developed by Facebook.

Hive provides the functionality of reading, writing, and managing large datasets residing in distributed storage. It runs SQL-like queries called HQL (Hive query language) which gets internally converted to MapReduce jobs.

Using Hive, we can skip the requirement of the traditional approach of writing complex MapReduce programs. Hive supports Data Definition Language (DDL), Data Manipulation Language (DML), and User Defined Functions (UDF).

Features of Hive

These are the following features of Hive:

- Hive is fast and scalable.
- It provides SQL-like queries (i.e., HQL) that are implicitly transformed to MapReduce or Spark jobs.
- It is capable of analyzing large datasets stored in HDFS.
- It allows different storage types such as plain text, RCFile, and HBase.
- It uses indexing to accelerate queries.
- It can operate on compressed data stored in the Hadoop ecosystem.
- It supports user-defined functions (UDFs) where users can provide its functionality.

Limitations of Hive

- Hive is not capable of handling real-time data.
- It is not designed for online transaction processing.
- Hive queries contain high latency.
- Hive is a database technology that can define databases and tables to analyze structured data. The theme for structured data analysis is to store the data in a tabular manner, and pass queries to analyze it. This chapter explains how to create a Hive database. Hive contains a default database named **default**.

Create Database Statement

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]
```

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 userdb:

```
hive> CREATE DATABASE [IF NOT EXISTS] userdb;
or
hive> CREATE SCHEMA userdb;
```

The following query is used to verify a databases list:

```
hive> SHOW DATABASES;
default
userdb
```

Hive - Partitioning

Hive organizes tables into partitions. It is a way of dividing a table into related parts based on the values of partitioned columns such as date, city, and department. Using partition, it is easy to query a portion of the data.

Tables or partitions are sub-divided into buckets, to provide extra structure to the data that may be used for more efficient querying. Bucketing works based on the value of hash function of some column of a table.

For example, a table named Tab1 contains employee data such as id, name, dept, and yoj (i.e., year of joining). Suppose you need to retrieve the details of all employees who joined in 2012. A query searches the whole table for the required information. However, if you partition the employee data with the year and store it in a separate file, it reduces the query processing time. The following example shows how to partition a file and its data:

The following file contains employeedata table.
/tab1/employeedata/file1
id, name, dept, yoj
1, gopal, TP, 2012
2, kiran, HR, 2012
3, kaleel,SC, 2013
4, Prasanth, SC, 2013

The above data is partitioned into two files using year.

/tab1/employeedata/2012/file2
1, gopal, TP, 2012
2, kiran, HR, 2012

/tab1/employeedata/2013/file3
3, kaleel,SC, 2013
4, Prasanth, SC, 2013

Adding a Partition
We can add partitions to a table by altering the table. Let us assume we have a table called employee with fields such as Id, Name, Salary, Designation, Dept, and yoj.

Syntax:

```
ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec [LOCATION 'location1'] partition_spec [LOCATION 'location2'] ...;
partition_spec: : (p_column = p_col_value, p_column = p_col_value, ...)
```

The following query is used to add a partition to the employee table.

```
hive> ALTER TABLE employee
 > ADD PARTITION (year=’2012’)
> location '/2012/part2012';
```

Renaming a Partition
The syntax of this command is as follows.

```
ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION partition_spec;
```

The following query is used to rename a partition:

```
hive> ALTER TABLE employee PARTITION (year=’1203’) > RENAME TO PARTITION (Yoj=’1203’);
```

Dropping a Partition The following syntax is used to drop a partition:

```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec, PARTITION partition_spec,...;
```

The following query is used to drop a partition:

```
hive> ALTER TABLE employee DROP [IF EXISTS] > PARTITION (year='1203');
```

4.2 SUMMARY

Hive - Built-in Functions

Here we are explaining the built-in functions available in Hive. The functions look quite similar to SQL functions, except for their usage.

Built-In Functions

Hive supports the following built-in functions:

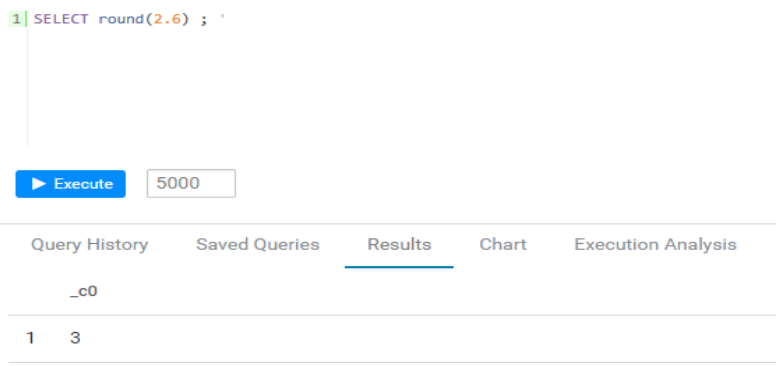
Return Type	Signature	Description
BIGINT	round(double a)	It returns the rounded BIGINT value of the double.
BIGINT	floor(double a)	It returns the maximum BIGINT value that is equal or less than the double.
BIGINT	ceil(double a)	It returns the minimum BIGINT value that is equal or greater than the double.
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.

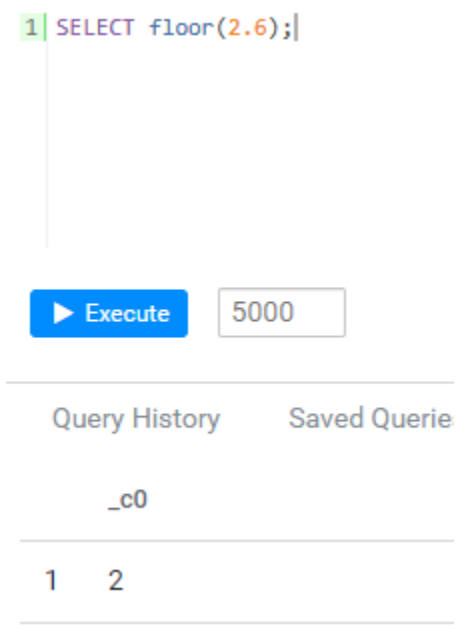
Example
The following queries demonstrate some built-in functions:
round() function

```
hive> SELECT round(2.6) from temp;
```



floor() function

```
hive> SELECT floor(2.6) from temp;
```



ceil() function

```
hive> SELECT ceil(2.6) from temp;
```

On successful execution of the query, you get to see the following response: 3.0

Aggregate Functions

Hive supports the following built-in aggregate functions. The usage of these functions is the same as the SQL aggregate functions.

Return Type	Signature	Description
BIGINT	count(*), count(expr),	count(*) - Returns the total number of retrieved rows.
DOUBLE	sum(col), sum(DISTINCT col)	It returns the sum of the elements in the group or the sum of the distinct values of the column in the group.
DOUBLE	avg(col), avg(DISTINCT col)	It returns the average of the elements in the group or the average of the distinct values of the column in the group.
DOUBLE	min(col)	It returns the minimum value of the column in the group.
DOUBLE	max(col)	It returns the maximum value of the column in the group.

Hive - Built-in Operators

Here we are explaining the operators available in Hive.

- Relational Operators
- Arithmetic Operators
- Logical Operators
- Complex Operators

Relational Operators

These operators are used to compare two operands. The following table describes the relational operators available in Hive:

Operator	Operand	Description
A = B	all primitive types	TRUE if expression A is equivalent to expression B otherwise FALSE.
A != B	all primitive types	TRUE if expression A is not equivalent to expression B otherwise FALSE.
A < B	all primitive types	TRUE if expression A is less than expression B otherwise FALSE.
A <= B	all primitive types	TRUE if expression A is less than or equal to expression B otherwise FALSE.
A > B	all primitive types	TRUE if expression A is greater than expression B otherwise FALSE.
A >= B	all primitive types	TRUE if expression A is greater than or equal to expression B otherwise FALSE.
A IS NULL	all types	TRUE if expression A evaluates to NULL otherwise FALSE.
A IS NOT NULL	all types	FALSE if expression A evaluates to NULL otherwise TRUE.
A LIKE B	Strings	TRUE if string pattern A matches to B otherwise FALSE.

Example

Let us assume the employee table is composed of fields named Id, Name, Salary, Designation, and Dept as shown below. Generate a query to retrieve the employee details whose Id is 1205.

```
+-----+--------------+--------+---------------------------+------+
|        Id        |        Name        |        Salary        |        Designation        |        Dept        |
+-----+--------------+-----------------------------------+------+
 |1201 | Gopal | 45000 | Technical manager | TP |
|1202 | Manisha | 45000 | Proofreader | PR |
|1203 | Masthanvali | 40000 | Technical writer | TP |
1204 | Krian | 40000 | Hr Admin | HR |
|1205 | Kranthi | 30000 | Op Admin | Admin|
+-----+--------------+--------+---------------------------+------+
```

The following query is executed to retrieve the employee details using the above table:

```
hive> SELECT * FROM employee WHERE Id=1205;
```

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

```
+-----+-----------+-----------+---------------------------------+
| ID | Name | Salary | Designation | Dept |
 +-----+--------------+-------+---------------------------------+
|1205 | Kranthi | 30000 | Op Admin | Admin |
+-----+-----------+-----------+---------------------------------+
```

The following query is executed to retrieve the employee details whose salary is more than or equal to Rs 40000.

```
hive> SELECT * FROM employee WHERE Salary>=40000;
```

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

```
+-----+------------+--------+-------------------------+------+
| ID | Name | Salary | Designation | Dept |
+-----+------------+--------+-------------------------+------+
|1201 | Gopal | 45000 | Technical manager | TP |
|1202 | Manisha | 45000 | Proofreader | PR |
|1203 | Masthanvali| 40000 | Technical writer | TP |
|1204 | Krian | 40000 | Hr Admin | HR |
 +-----+------------+--------+-------------------------+------+
```

Arithmetic Operators

These operators support various common arithmetic operations on the operands. All of them return number types. The following table describes the arithmetic operators available in Hive:

Operators	Operand	Description
A + B	all number types	Gives the result of adding A and B.
A - B	all number types	Gives the result of subtracting B from A.
A * B	all number types	Gives the result of multiplying A and B.
A / B	all number types	Gives the result of dividing B from A.
A % B	all number types	Gives the reminder resulting from dividing A by B.
A & B	all number types	Gives the result of bitwise AND of A and B.
A | B	all number types	Gives the result of bitwise OR of A and B.
A ^ B	all number types	Gives the result of bitwise XOR of A and B.
~A	all number types	Gives the result of bitwise NOT of A.

Example

The following query adds two numbers, 20 and 30.

```
hive> SELECT 20+30 ADD FROM temp;
```

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

```
+--------+
| ADD |
+--------+
| 50 |
+--------+
```

Logical Operators
The operators are logical expressions. All of them return either TRUE or FALSE.

Operators	Operands	Description
A AND B	boolean	TRUE if both A and B are TRUE, otherwise FALSE.
A && B	boolean	Same as A AND B.
A OR B	boolean	TRUE if either A or B or both are TRUE, otherwise FALSE.
A || B	boolean	Same as A OR B.
NOT A	boolean	TRUE if A is FALSE, otherwise FALSE.
!A	boolean	Same as NOT A.

Example
The following query is used to retrieve employee details whose Department is TP and Salary is more than Rs 40000.

```
hive> SELECT * FROM employee WHERE Salary>40000 && Dept=TP;
```

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

```
+------+--------------+-------------+-------------------+--------+
 | ID | Name | Salary | Designation | Dept |
+------+--------------+-------------+-------------------+--------+
 |1201 | Gopal | 45000 | Technical manager | TP |
+------+--------------+-------------+-------------------+--------+
```

Complex Operators
These operators provide an expression to access the elements of Complex Types.

Operator	Operand	Description
A[n]	A is an Array and n is an int	It returns the nth element in the array A. The first element has index 0.
M[key]	M is a Map<K, V> and key has type K	It returns the value corresponding to the key in the map.
S.x	S is a struct	It returns the x field of S.

Hive - View and Indexes

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 ...
```

Example Let us take an example for view. Assume employee table as given below, with the fields Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 30000. We store the result in a view named emp_30000.

```
+------+--------------+------------+------------------+--------+
 | ID | Name | Salary | Designation | Dept |
+------+--------------+------------+------------------+--------+
|1201 | Gopal | 45000 | Technical manager | TP |
|1202 | Manisha | 45000 | Proofreader | PR |
 |1203 | Masthanvali | 40000 | Technical writer | TP |
|1204 | Krian | 40000 | Hr Admin | HR |
|1205 | Kranthi | 30000 | Op Admin | Admin |
+------+--------------+------------+------------------+--------+
```

The following query retrieves the employee details using the above HIVE scenario:

```
hive> CREATE VIEW emp_30000 AS SELECT * FROM employee WHERE salary>30000;
```

Dropping a View

Use the following syntax to drop a view:

```
DROP VIEW view_name ;
```

The following query drops a view named as emp_30000:

```
hive> DROP VIEW emp_30000;
```

Creating an Index 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. Its syntax is as follows:

```
CREATE INDEX index_name
ON TABLE base_table_name (col_name, ...) AS 'index.handler.class.name'
 [WITH DEFERRED REBUILD]
[IDXPROPERTIES (property_name=property_value, ...)]
[IN TABLE index_table_name]
 [PARTITIONED BY (col_name, ...)]
[ [ ROW FORMAT ...] STORED AS ... | STORED BY ... ]
 [LOCATION hdfs_path] [TBLPROPERTIES (...)]
```

Example Let us take an example for index. Use the same employee table that we have used earlier with the fields Id, Name, Salary, Designation, and Dept. Create an index named index_salary on the salary column of the employee table.

The following query creates an index:

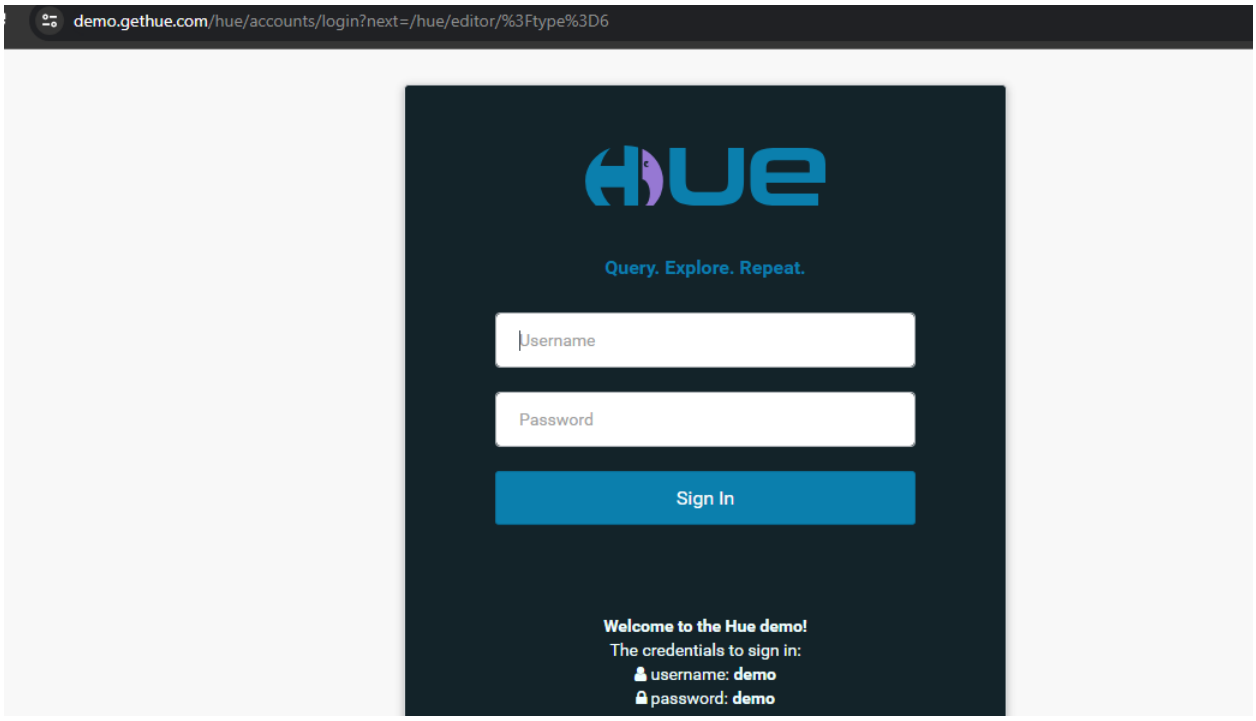
```
hive> CREATE INDEX inedx_salary ON TABLE employee(salary) AS
'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler';
```

It is a pointer to the salary column. If the column is modified, the changes are stored using an index value. Dropping an Index The following syntax is used to drop an index:

```
DROP INDEX ON The following query drops an index named index_salary: hive> DROP
INDEX index_salary ON employee;
```

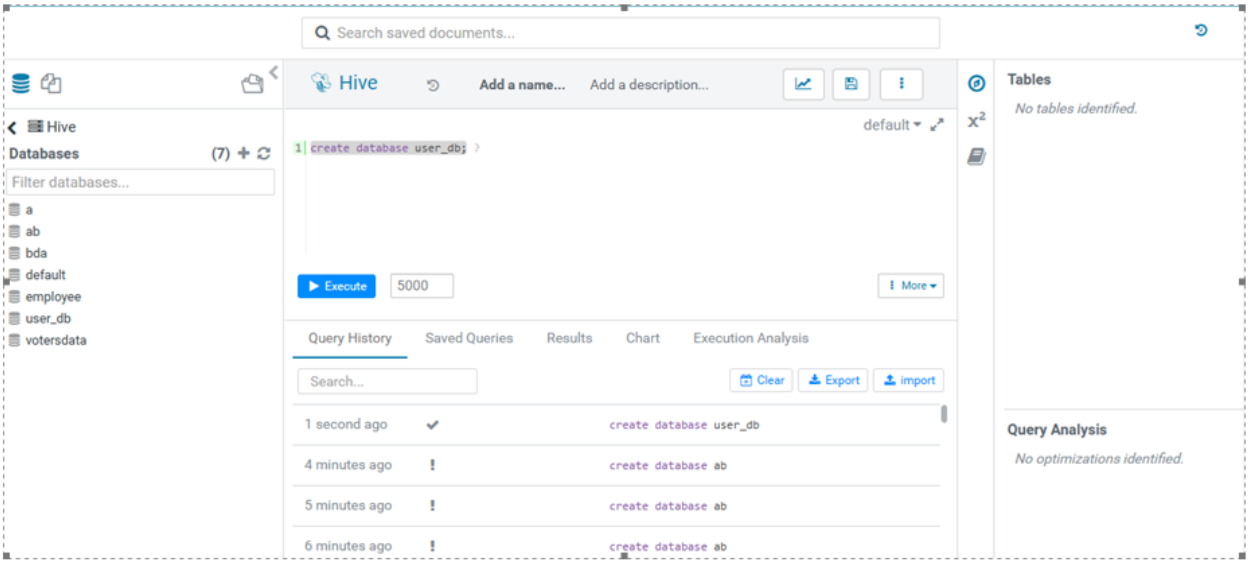
Steps of lab session:

1. **Open https://demo.gethue.com/hue/editor/?type=6**

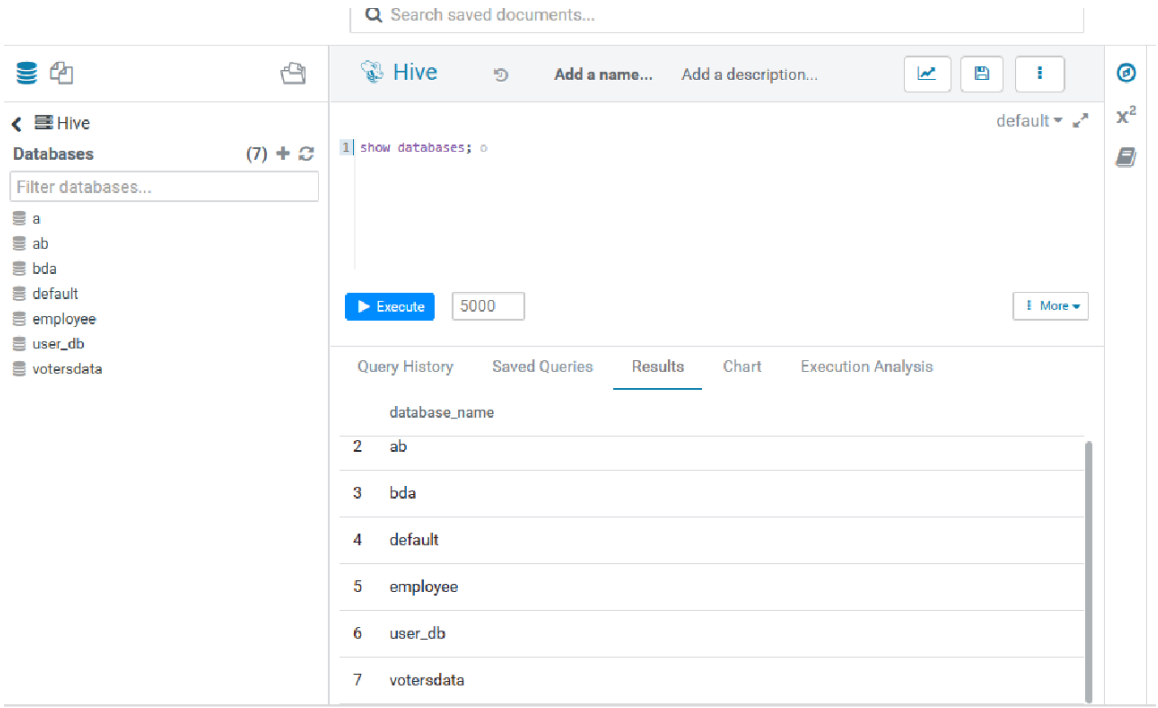


Login using username: **demo** and password : **demo**

2. **create database user_db;**



3 .**show databases;**



4. Execute following queries

create table user_db.student (id int, name String, mobile String, address String);

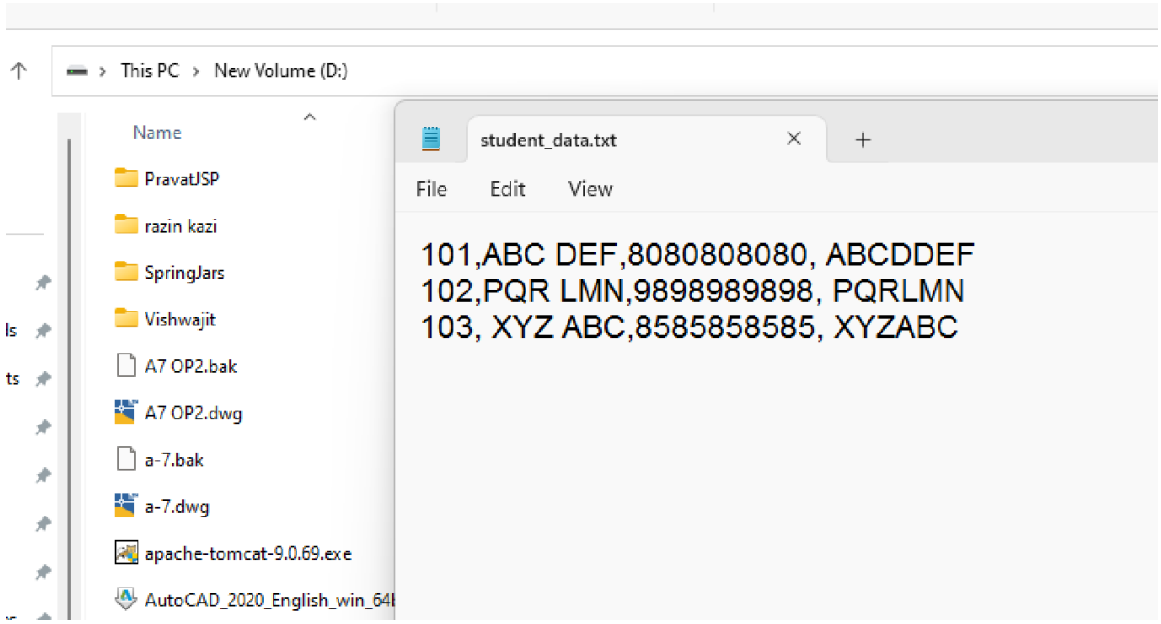
drop table user_db.student;

create table user_db.student (id int, name String, mobile String, address String)
tblproperties ('creator'='asn', 'created_at'='30-10-2023');

describe user_db.student;

	col_name	data_type	comment
1	id	int	
2	name	string	
3	mobile	string	
4	address	string	

For loading following data :



Load data in path D:\student_data.txt into table user_db.student;

insert into user_db.student values(501,'ppp qqq rrr','1234567890','pqrs');

insert into user_db.student values(502,'abc qqq rrr','1234567890','abcrs');

insert into user_db.student values(503,'lmn qqq rrr','1234567890','lmns');

insert into user_db.student values(504,'xyz qqq rrr','1234567890','xyzs');

insert into user_db.student values(505,'tuv qqq rrr','1234567890','tuvs');

or

insert into user_db.student values(503,'lmn qqq rrr','1234567890','lmns'), (504,'xyz qqq rrr','1234567890','xyzs') , (505,'tuv qqq rrr','1234567890','tuvs');

select * from student;

```
1 select * from student; =
2
```

Execute 5000

Query History | Saved Queries | Results | Chart | Execution Analysis

	student.id	student.name	student.mobile	student.address
1	501	ppp qqq rrr	1234567890	pqrs
2	502	abc qqq rrr	1234567890	abcrs
3	503	lmn qqq rrr	1234567890	lmns
4	504	xyz qqq rrr	1234567890	xyzs
5	505	tuv qqq rrr	1234567890	tuvs

View:

1. Creating a view

*CREATE VIEW stud_503 AS SELECT * FROM student WHERE id >503;*

```
1 CREATE VIEW stud_503 AS SELECT * FROM student WHEREiid>503;
```

Execute 5000

Query History | Saved Queries | Results | Chart | Execution Analysis

2. *Viewing a view*

*select * from stud_503;*

```
1 select * from stud_503; ,
```

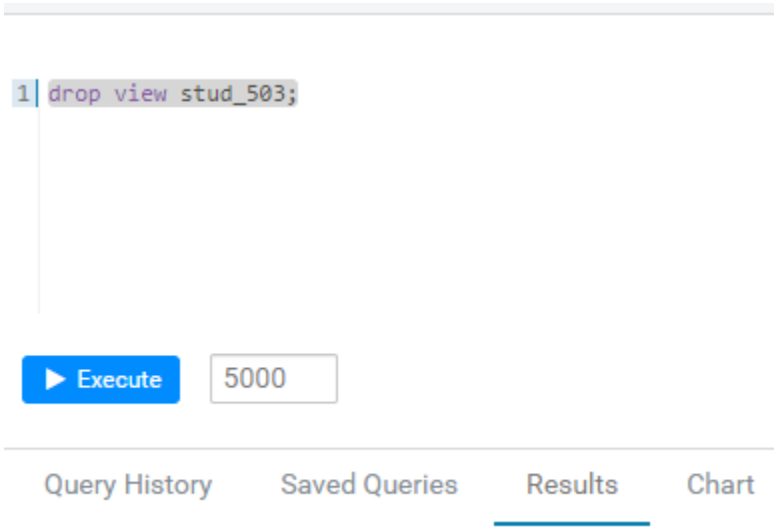
Execute 5000

Query History | Saved Queries | Results | Chart | Execution Analysis

	stud_503.id	stud_503.name	stud_503.mobile	stud_503.address
1	504	xyz qqq rrr	1234567890	xyzs
2	505	tuv qqq rrr	1234567890	tuvs

3. Dropping a view

drop view stud_503;



4.3 REFERENCES

https://demo.gethue.com/hue/editor?editor=845644

"The Visual Display of Quantitative Information" by Edward R. ...

"Storytelling With Data: A Data Visualization Guide for Business Professionals" by Cole Nussbaumer Knaflic.

"Data Visualization – A Practical Introduction" by Kieran Healy.

4.4 EXERCISES

Write a program to perform Table Operations such as Creation, Altering, and Dropping tables in Hive.