What is HIVE

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 user can provide its functionality.

## Limitations of Hive

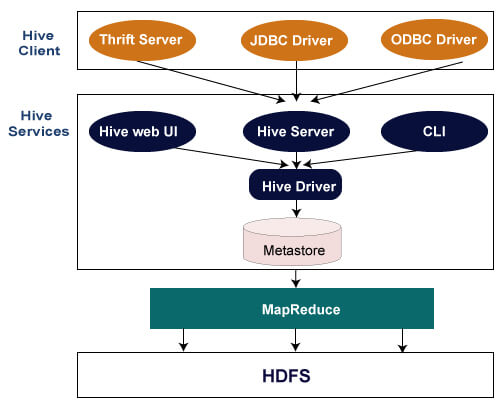
* Hive is not capable of handling real-time data.
* It is not designed for online transaction processing.
* Hive queries contain high latency.

## Differences between Hive and Pig

|  |  |
| --- | --- |
| **Hive** | **Pig** |
| Hive is commonly used by Data Analysts. | Pig is commonly used by programmers. |
| It follows SQL-like queries. | It follows the data-flow language. |
| It can handle structured data. | It can handle semi-structured data. |
| It works on server-side of HDFS cluster. | It works on client-side of HDFS cluster. |
| Hive is slower than Pig. | Pig is comparatively faster than Hive. |

Hive Architecture

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



Hive Client

Hive allows writing applications in various languages, including Java, Python, and C++. It supports different types of clients such as:-

* Thrift Server - It is a cross-language service provider platform that serves the request from all those programming languages that supports Thrift.
* JDBC Driver - It is used to establish a connection between hive and Java applications. The JDBC Driver is present in the class org.apache.hadoop.hive.jdbc.HiveDriver.
* ODBC Driver - It allows the applications that support the ODBC protocol to connect to Hive.

Hive Services

The following are the services provided by Hive:-

* Hive CLI - The Hive CLI (Command Line Interface) is a shell where we can execute Hive queries and commands.
* Hive Web User Interface - The Hive Web UI is just an alternative of Hive CLI. It provides a web-based GUI for executing Hive queries and commands.
* Hive MetaStore - It is a central repository that stores all the structure information of various tables and partitions in the warehouse. It also includes metadata of column and its type information, the serializers and deserializers which is used to read and write data and the corresponding HDFS files where the data is stored.
* Hive Server - It is referred to as Apache Thrift Server. It accepts the request from different clients and provides it to Hive Driver.
* Hive Driver - It receives queries from different sources like web UI, CLI, Thrift, and JDBC/ODBC driver. It transfers the queries to the compiler.
* Hive Compiler - The purpose of the compiler is to parse the query and perform semantic analysis on the different query blocks and expressions. It converts HiveQL statements into MapReduce jobs.
* Hive Execution Engine - Optimizer generates the logical plan in the form of DAG of map-reduce tasks and HDFS tasks. In the end, the execution engine executes the incoming tasks in the order of their dependencies.

HIVE Data Types

Hive data types are categorized in numeric types, string types, misc types, and complex types. A list of Hive data types is given below.

Integer Types

|  |  |  |
| --- | --- | --- |
| **Type** | **Size** | **Range** |
| TINYINT | 1-byte signed integer | -128 to 127 |
| SMALLINT | 2-byte signed integer | -32,768 to 32,767 |
| INT | 4-byte signed integer | -2,147,483,648 to 2,147,483,647 |
| BIGINT | 8-byte signed integer | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |

Decimal Type

|  |  |  |
| --- | --- | --- |
| **Type** | **Size** | **Range** |
| FLOAT | 4-byte | Single precision floating point number |
| DOUBLE | 8-byte | Double precision floating point number |

Date/Time Types

**TIMESTAMP**

* It supports traditional UNIX timestamp with optional nanosecond precision.
* As Integer numeric type, it is interpreted as UNIX timestamp in seconds.
* As Floating point numeric type, it is interpreted as UNIX timestamp in seconds with decimal precision.
* As string, it follows java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff" (9 decimal place precision)

**DATES**

The Date value is used to specify a particular year, month and day, in the form YYYY--MM--DD. However, it didn't provide the time of the day. The range of Date type lies between 0000--01--01 to 9999--12--31.

**00:00/05:29**

String Types

**STRING**

The string is a sequence of characters. It values can be enclosed within single quotes (') or double quotes (").

**Varchar**

The varchar is a variable length type whose range lies between 1 and 65535, which specifies that the maximum number of characters allowed in the character string.

**CHAR**

The char is a fixed-length type whose maximum length is fixed at 255.

Complex Type

|  |  |  |
| --- | --- | --- |
| **Type** | **Size** | **Range** |
| Struct | It is similar to C struct or an object where fields are accessed using the "dot" notation. | struct('James','Roy') |
| Map | It contains the key-value tuples where the fields are accessed using array notation. | map('first','James','last','Roy') |
| Array | It is a collection of similar type of values that indexable using zero-based integers. | array('James','Roy') |

**Create Database**

In Hive, the database is considered as a catalog or namespace of tables. So, we can maintain multiple tables within a database where a unique name is assigned to each table. Hive also provides a default database with a name **default**.

* 1. create database<database name>:- will Create new database for user.
  2. Show database: -Will display the names of all databases.
  3. to suppress the warning generated by Hive on creating the database with the same name “create a database if not exists <database name>;  “
  4. Hive also allows assigning properties with the database in the form of key-value pair.

create the database <database name>

**>**WITH DBPROPERTIES ('creator' = 'User name', 'date' = 'creation date');

**5.** retrieve the information associated with the database.

describe database extended <database name>;

6. drop database :- will drop the database. If database not found will give an error

7. drop database if exists <tablename> :- it will drop the database if database not found if not then surpass the warning.

8. drop database if exists <tablename> cascade; :- this command will also drop the database it is prefer when there are tables in database we awant delete tables also because in Hadoop if there tables in database then the database is not removed

# Hive - Create Table

In Hive, we can create a table by using the conventions similar to the SQL. It supports a wide range of flexibility where the data files for tables are stored. It provides two types of table: -

* Internal table
* External table

Internal Table

The internal tables are also called managed tables as the lifecycle of their data is controlled by the Hive. By default, these tables are stored in a subdirectory under the directory defined by hive.metastore.warehouse.dir (i.e. /user/hive/warehouse). The internal tables are not flexible enough to share with other tools like Pig. If we try to drop the internal table, Hive deletes both table schema and data.

1. create table databse name.tablename (Id int, Name string , Salary float)

row format delimited

fields terminated by ',' ;

Here, the command also includes the information that the data is separated by ','.

1. describe databasename.tablename:- It will give the information about the database.

## External Table

The external table allows us to create and access a table and a data externally. The **external** keyword is used to specify the external table, whereas the **location** keyword is used to determine the location of loaded data.

As the table is external, the data is not present in the Hive directory. Therefore, if we try to drop the table, the metadata of the table will be deleted, but the data still exists.

To create an external table, follow the below steps: -

* Create a directory on HDFS by using the following command: -

1. hdfs dfs -mkdir /HiveDirectory

* Now, store the file on the created directory.

1. hdfs dfs -put hive/emp\_details /HiveDirectory

* Let's create an external table using the following command: -

1. create external table <Tablename> (Id ient, Name string , Salary float)
2. row format delimited
3. fields terminated by ','
4. location '/HiveDirectory';

Using Command “**select \* from emplist;** “

# Hive - Load Data

Once the internal table has been created, the next step is to load the data into it. So, in Hive, we can easily load data from any file to the database.

* Let's load the data of the file into the database by using the following command: -

1. load data local inpath '/home/codegyani/hive/emp\_details' into table demo.employee;

Here, **emp\_details** is the file name that contains the data.

* Now, we can use the following command to retrieve the data from the database.

1. select \* from demo.employee;
2. In Hive, if we try to load unmatched data (i.e., one or more column data doesn't match the data type of specified table columns), it will not throw any exception. However, it stores the Null value at the position of unmatched tuple.

# Hive - Drop Table

Hive facilitates us to drop a table by using the SQL **drop table** command.

* drop the table by using the following command: -

1. drop table new\_employee;

# Hive - Alter Table

In Hive, we can perform modifications in the existing table like changing the table name, column name, comments, and table properties. It provides SQL like commands to alter the table.

### **Rename a Table**

If we want to change the name of an existing table, we can rename that table by using the following signature: -

1. Alter table old\_table\_name rename to new\_table\_name;

### **Adding column**

In Hive, we can add one or more columns in an existing table by using the following signature: -

1. Alter table table\_name add columns(column\_name datatype);
2. Alter table employee\_data add columns (age int);  this will add new column to our employee\_data table

### **Change Column**

In Hive, we can rename a column, change its type and position. Here, we are changing the name of the column by using the following signature: -

1. Alter table table\_name change old\_column\_name new\_column\_name  datatype;
2. Alter table employee\_data change name first\_name string;  It will change name attribute to first\_name

### **Delete or Replace Column**

Hive allows us to delete one or more columns by replacing them with the new columns. Thus, we cannot drop the column directly.

* drop a column from the table.

1. alter table employee\_data replace columns( id string, first\_name string, age int);

# Partitioning in Hive

The partitioning in Hive means dividing the table into some parts based on the values of a particular column like date, course, city or country. The advantage of partitioning is that since the data is stored in slices, the query response time becomes faster.

As we know that Hadoop is used to handle the huge amount of data, it is always required to use the best approach to deal with it. The partitioning in Hive is the best example of it.

Let's assume we have a data of 10 million students studying in an institute. Now, we have to fetch the students of a particular course. If we use a traditional approach, we have to go through the entire data. This leads to performance degradation. In such a case, we can adopt the better approach i.e., partitioning in Hive and divide the data among the different datasets based on particular columns.

The partitioning in Hive can be executed in two ways -

* [Static partitioning](https://www.javatpoint.com/partitioning-in-hive#Static)
* [Dynamic partitioning](https://www.javatpoint.com/dynamic-partitioning-in-hive)

## Static Partitioning

In static or manual partitioning, it is required to pass the values of partitioned columns manually while loading the data into the table. Hence, the data file doesn't contain the partitioned columns.

**Example of Static Partitioning**

* First, select the database in which we want to create a table.

1. hive**>** use test;

* Create the table and provide the partitioned columns by using the following command: -

1. hive**>** create table student (id int, name string, age int,  institute string)
2. partitioned by (course string)
3. row format delimited
4. fields terminated by ',';

* Load the data into the table and pass the values of partition columns with it by using the following command: -

1. hive**>** load data local inpath '/home/codegyani/hive/student\_details1' into table student
2. partition(course= "java");

Here, we are partitioning the students of an institute based on courses.

* Load the data of another file into the same table and pass the values of partition columns with it by using the following command: -

1. hive**>** load data local inpath '/home/codegyani/hive/student\_details2' into table student
2. partition(course= "hadoop");

# Dynamic Partitioning

In dynamic partitioning, the values of partitioned columns exist within the table. So, it is not required to pass the values of partitioned columns manually.

* First, select the database in which we want to create a table.

1. hive**>** use show;

* Enable the dynamic partition by using the following commands: -

1. hive**>** set hive.exec.dynamic.partition=true;
2. hive**>** set hive.exec.dynamic.partition.mode=nonstrict;

* Create a dummy table to store the data.

1. hive**>** create table stud\_demo(id int, name string, age int, institute string, course string)
2. row format delimited
3. fields terminated by ',';
4. Now, load the data into the table.
5. hive**>** load data local inpath '/home/codegyani/hive/student\_details' into table stud\_demo;

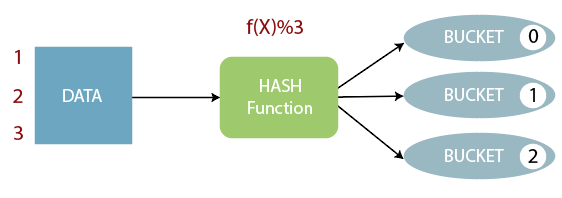
* Create a partition table by using the following command: -

1. hive**>** create table student\_part (id int, name string, age int, institute string)
2. partitioned by (course string)
3. row format delimited
4. fields terminated by ',';

# Bucketing in Hive

The bucketing in Hive is a data organizing technique. It is similar to partitioning in Hive with an added functionality that it divides large datasets into more manageable parts known as buckets. So, we can use bucketing in Hive when the implementation of partitioning becomes difficult. However, we can also divide partitions further in buckets.

## Working of Bucketing in Hive



* The concept of bucketing is based on the hashing technique.
* Here, modules of current column value and the number of required buckets is calculated (let say, F(x) % 3).
* Now, based on the resulted value, the data is stored into the corresponding bucket.

### **Example of Bucketing in Hive**

* First, select the database in which we want to create a table.

1. hive**>** use showbucket;

* Create a dummy table to store the data.

1. hive**>** create table emp\_demo (Id int, Name string , Salary float)
2. row format delimited
3. fields terminated by ',' ;
4. Now, load the data into the table.
5. hive**>** load data local inpath '/home/codegyani/hive/emp\_details' into table emp\_demo;

* Enable the bucketing by using the following command: -

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

* Create a bucketing table by using the following command: -

1. hive**>** create table emp\_bucket(Id int, Name string , Salary float)
2. clustered by (Id) into 3 buckets
3. row format delimited
4. fields terminated by ',' ;
5. Now, insert the data of dummy table into the bucketed table.
6. hive**>** insert overwrite table emp\_bucket select \* from emp\_demo;

# HiveQL - Operators

The HiveQL operators facilitate to perform various arithmetic and relational operations.

## Arithmetic Operators in Hive

In Hive, the arithmetic operator accepts any numeric type. The commonly used arithmetic operators are:

|  |  |
| --- | --- |
| **Operators** | **Description** |
| A + B | This is used to add A and B. |
| A – B | This is used to subtract B from A. |
| A \* B | This is used to multiply A and B. |
| A / B | This is used to divide A and B and returns the quotient of the operands. |
| A % B | This returns the remainder of A / B. |
| A | B | This is used to determine the bitwise OR of A and B. |
| A & B | This is used to determine the bitwise AND of A and B. |
| A ^ B | This is used to determine the bitwise XOR of A and B. |
| ~A | This is used to determine the bitwise NOT of A. |

### **Examples of Arithmetic Operator in Hive**

* Let's see an example to increase the salary of each employee by 50.

1. hive**>** select id, name, salary + 50 from employee;

## Relational Operators in Hive

In Hive, the relational operators are generally used with clauses like Join and Having to compare the existing records. The commonly used relational operators are: -

|  |  |
| --- | --- |
| **Operator** | **Description** |
| A=B | It returns true if A equals B, otherwise false. |
| A <> B, A !=B | It returns null if A or B is null; true if A is not equal to B, otherwise false. |
| A<B | It returns null if A or B is null; true if A is less than B, otherwise false. |
| A>B | It returns null if A or B is null; true if A is greater than B, otherwise false. |
| A<=B | It returns null if A or B is null; true if A is less than or equal to B, otherwise false. |
| A>=B | It returns null if A or B is null; true if A is greater than or equal to B, otherwise false. |
| A IS NULL | It returns true if A evaluates to null, otherwise false. |
| A IS NOT NULL | It returns false if A evaluates to null, otherwise true. |

### **Examples of Relational Operator in Hive**

* an example to fetch the details of the employee having salary>=25000.

1. select \* from employee where salary **>**= 25000;

# HiveQL - GROUP BY and HAVING Clause

The Hive Query Language provides GROUP BY and HAVING clauses that facilitate similar functionalities as in SQL.

## GROUP BY Clause

The **HQL Group By** clause is used to group the data from the multiple records based on one or more column. It is generally used in conjunction with the aggregate functions (like SUM, COUNT, MIN, MAX and AVG) to perform an aggregation over each group.

### **Example of GROUP BY Clause in Hive**

an example to sum the salary of employees based on department.

select department, sum(salary) from emp group by department;

## HAVING CLAUSE

The HQL **HAVING clause** is used with **GROUP BY** clause. Its purpose is to apply constraints on the group of data produced by GROUP BY clause. Thus, it always returns the data where the condition is **TRUE**.

### **Example of Having Clause in Hive**

In this example, we fetch the sum of employee's salary based on department and apply the required constraints on that sum by using HAVING clause.

* An Example the sum of employee's salary based on department having sum >= 35000 by using the following command:

select department, sum(salary) from emp group by department having sum(salary)**>**=35000;

# HiveQL - ORDER BY and SORT BY Clause

By using HiveQL ORDER BY and SORT BY clause, we can apply sort on the column. It returns the result set either in ascending or descending order.

## HiveQL - ORDER BY Clause

In HiveQL, ORDER BY clause performs a complete ordering of the query result set. Hence, the complete data is passed through a single reducer. This may take much time in the execution of large datasets. However, we can use LIMIT to minimize the sorting time.

### **Example of ORDER BY Clause in Hive**

an example to arrange the data in the sorted order by using ORDER BY clause.

* fetch the data in the descending order by using the following command:

select \* from emp order by salary desc;

## HiveQL - SORT BY Clause

The HiveQL SORT BY clause is an alternative of ORDER BY clause. It orders the data within each reducer. Hence, it performs the local ordering, where each reducer's output is sorted separately. It may also give a partially ordered result.

### **Example of SORT BY Clause in Hive**

In this example, we arrange the data in the sorted order by using SORT BY clause.

* Let's fetch the data in the descending order by using the following command:

1. hive**>** select \* from emp sort by salary desc;

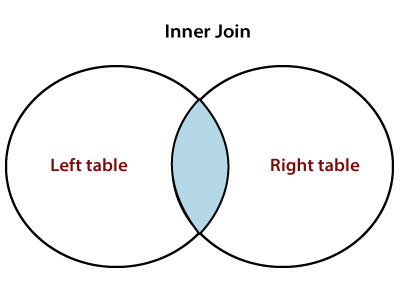
HiveQL - JOIN

The HiveQL Join clause is used to combine the data of two or more tables based on a related column between them. The various type of HiveQL joins are: -

* Inner Join
* Left Outer Join
* Right Outer Join
* Full Outer Join

## Inner Join in HiveQL

The HiveQL inner join is used to return the rows of multiple tables where the join condition satisfies. In other words, the join criteria find the match records in every table being joined.



## Example of Inner Join in Hive

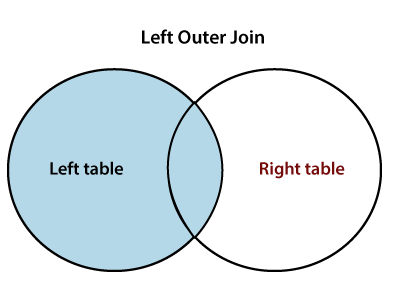
In this example, we take two table employee and employee\_department. The primary key (empid) of employee table represents the foreign key (depid) of employee\_department table.

* Now, perform the inner join operation by using the following command: -

select  e1.empname, e2.department\_name from employee e1 join employee\_department e2 on e1.empid= e2.depid;

## Left Outer Join in HiveQL

The HiveQL left outer join returns all the records from the left (first) table and only that records from the right (second) table where join criteria find the match.



### **Example of Left Outer Join in Hive**

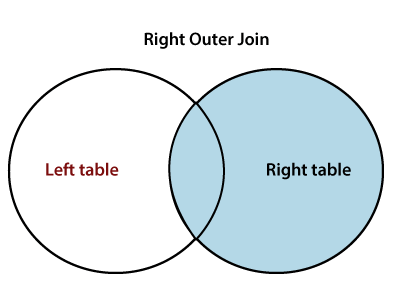
In this example, we perform the left outer join operation.

* Let's us execute the left outer join operation by using the following command: -

select  e1.empname, e2.department\_name from employee e1 left outer join employee\_department e2 on e1.empid= e2.depid;

## Right Outer Join in HiveQL

The HiveQL right outer join returns all the records from the right (second) table and only that records from the left (first) table where join criteria find the match.



### **Example of Left Outer Join in Hive**

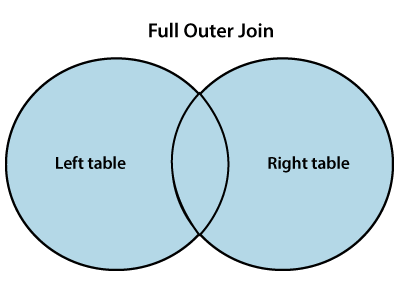
In this example, we perform the left outer join operation.

* Let's us execute the left outer join operation by using the following command: -

select  e1.empname, e2.department\_name from employee e1 right outer join employee\_department e2 on e1.empid= e2.depid;

## Full Outer Join

The HiveQL full outer join returns all the records from both the tables. It assigns Null for missing records in either table.



### **Example of Full Outer Join in Hive**

In this example, we perform the full outer join operation.

* Let's us execute the full outer join operation by using the following command: -

select  e1.empname, e2.department\_name from employee e1 full outer join employee\_department e2 on e1.empid= e2.depid;

**Hive HDBC connection Code**

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**public** **class** HiveClient {

**private** **static** String driverClass = "org.apache.hive.jdbc.HiveDriver";

**public** **static** **void** main(String args[]) **throws** SQLException {

**try** {

            Class.forName(driverClass);

        } **catch** (ClassNotFoundException exception) {

            exception.printStackTrace();

            System.exit(1);

        }

        Connection connection = DriverManager.getConnection("jdbc:hive2://", "", "");

        Statement statement = connection.createStatement();

        String table = "CUSTOMER";

**try** {

            statement.executeQuery("DROP TABLE " + table);

        } **catch** (Exception exception) {

            exception.printStackTrace();

        }

**try** {

            statement.executeQuery("CREATE TABLE " + table + " (ID INT, NAME STRING, ADDR STRING)");

        } **catch** (Exception exception) {

            exception.printStackTrace();

        }

        String sql = "SHOW TABLES '" + table + "'";

        System.out.println("Executing Show table: " + sql);

        ResultSet result = statement.executeQuery(sql);

**if** (result.next()) {

            System.out.println("Table created is :" + result.getString(1));

        }

        sql = "INSERT INTO CUSTOMER (ID,NAME,ADDR) VALUES (1, 'Ramesh', '3 NorthDrive SFO' )";

        System.out.println("Inserting table into customer: " + sql);

**try** {

            statement.executeUpdate(sql);

        } **catch** (Exception exception) {

            exception.printStackTrace();

        }

        sql = "SELECT \* FROM " + table;

        result = statement.executeQuery(sql);

        System.out.println("Running: " + sql);

        result = statement.executeQuery(sql);

**while** (result.next()) {

            System.out.println("Id=" + result.getString(1));

            System.out.println("Name=" + result.getString(2));

            System.out.println("Address=" + result.getString(3));

        }

        result.close();

        statement.close();

        connection.close();

    }

}