Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

This is a brief tutorial that provides an introduction on how to use Apache Hive HiveQL with Hadoop Distributed File System. This tutorial can be your first step towards becoming a successful Hadoop Developer with Hive.

Audience

This tutorial is prepared for professionals aspiring to make a career in Big Data Analytics using Hadoop Framework. ETL developers and professionals who are into analytics in general may as well use this tutorial to good effect.

Prerequisites

Before proceeding with this tutorial, you need a basic knowledge of Core Java, Database concepts of SQL, Hadoop File system, and any of Linux operating system flavors.

1. INTRODUCTION

The term 'Big Data' is used for collections of large datasets that include huge volume, high velocity, and a variety of data that is increasing day by day. Using traditional data management systems, it is difficult to process Big Data. Therefore, the Apache Software Foundation introduced a framework called Hadoop to solve Big Data management and processing challenges.

Hadoop

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

- **MapReduce:** It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.
- **HDFS:** Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

- **Sqoop**: It is used to import and export data to and fro between HDFS and RDBMS.
- **Pig**: It is a procedural language platform used to develop a script for MapReduce operations.
- **Hive**: It is a platform used to develop SQL type scripts to do MapReduce operations.

Note: There are various ways to execute MapReduce operations:

- The traditional approach using Java MapReduce program for structured, semi-structured, and unstructured data.
- The scripting approach for MapReduce to process structured and semi structured data using Pig.
- The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

What is Hive?

- Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.
- Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.
- Data analysts use Hive to explore, structure and analyze that data, then turn it into business insights.
- Hive implements a dialect of SQL (Hive QL) that focuses on analytics and presents a rich set of SQL semantics including OLAP functions, sub-queries, common table expressions and more. Hive allows SQL developers or users with SQL tools to easily query, analyze and process data stored in Hadoop.Hive also allows programmers familiar with the MapReduce framework to plug in their custom mappers and reducers to perform more sophisticated analysis that may not be supported by the built-in capabilities of the language.
- Hive users have a choice of 3 runtimes when executing SQL queries. Users can choose between Apache Hadoop MapReduce, Apache Tez or Apache Spark frameworks as their execution backend.
- The Beeline shell works in both embedded mode as well as remote mode so its mean Hive server2 supports a new command shell Beeline.
-

Hive is not

- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

Features of Hive

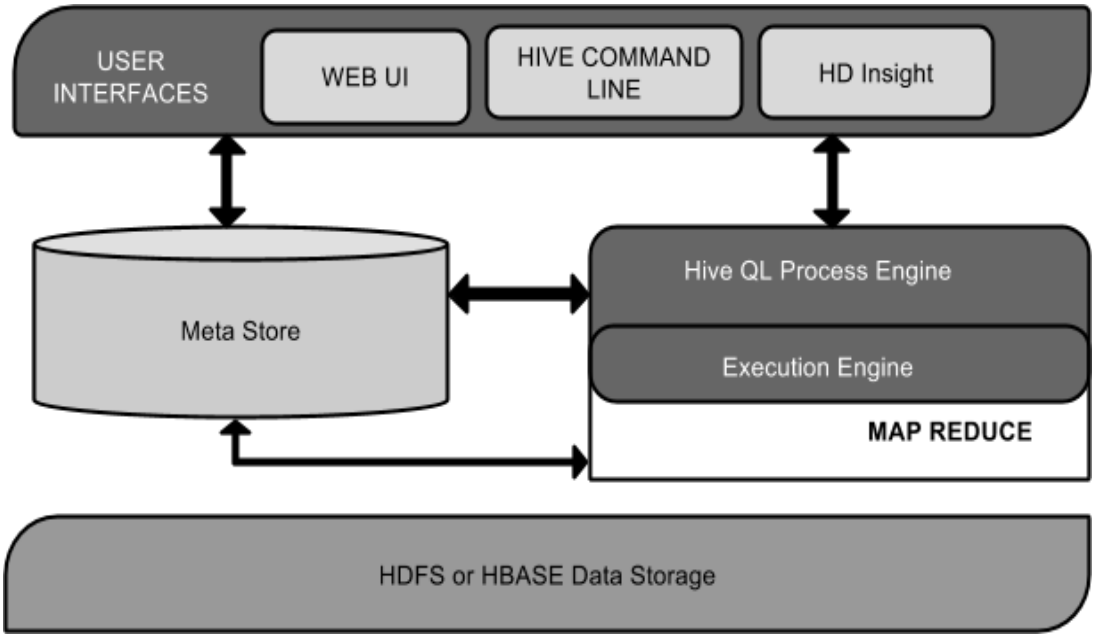
Here are the features of Hive:

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP.
- It provides SQL type language for querying called HiveQL or HQL.
- It is familiar, fast, scalable, and extensible.

Architecture of Hive

Translates SQL queries to Map reduce and TEZ jobs on cluster

The following component diagram depicts the architecture of Hive:

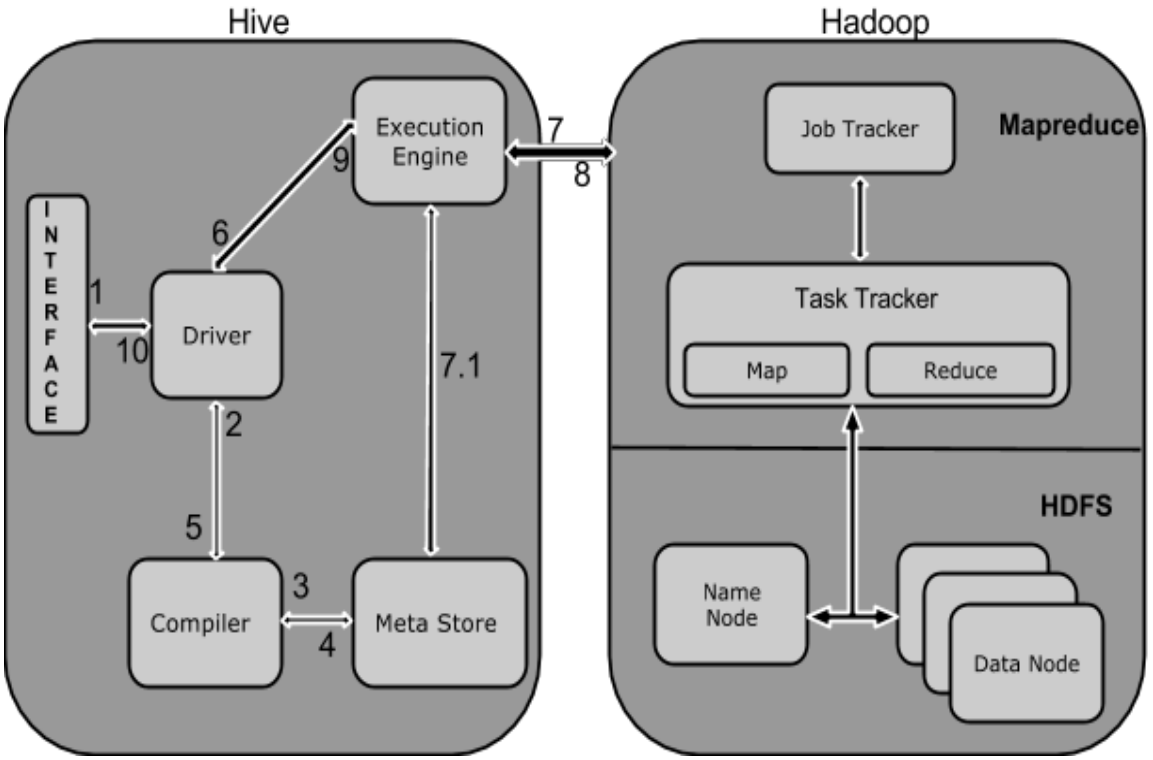


This component diagram contains different units. The following table describes each unit:

Unit Name	Operation
User Interface	Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The tables in Hive are similar to tables in a relational database, and data units are organized in a taxonomy from larger to more granular units. Databases are comprised of tables, which are made up of partitions. Data can be accessed via a simple query language and Hive supports overwriting or appending data.

Within a particular database, data in the tables is serialized and each table has a corresponding Hadoop Distributed File System (HDFS) directory. Each table can be sub-divided into partitions that determine how data is distributed within sub-directories of the table directory. Data within partitions can be further broken down into buckets.

The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation
1	**Execute Query** The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.

2	**Get Plan** The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	**Get Metadata** The compiler sends metadata request to Metastore (any database).
4	**Send Metadata** Metastore sends metadata as a response to the compiler.

5	**Send Plan** The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	**Execute Plan** The driver sends the execute plan to the execution engine.
7	**Execute Job** Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.
7.1	**Metadata Ops** Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	**Fetch Result** The execution engine receives the results from Data nodes.
9	**Send Results** The execution engine sends those resultant values to the driver.
10	**Send Results** The driver sends the results to Hive Interfaces.

3. HIVE DATA TYPES

This chapter takes you through the different data types in Hive, which are involved in the table creation. All the data types in Hive are classified into four types, given as follows:

1. Column Types
2. Literals
3. Null Values
4. Complex Types

Column Types

Column type are used as column data types of Hive. They are as follows:

Integral Types

Integer type data can be specified using integral data types, INT. When the data range exceeds the range of INT, you need to use BIGINT and if the data range is smaller than the INT, you use SMALLINT. TINYINT is smaller than SMALLINT.

The following table depicts various INT data types:

Type	Postfix	Example
TINYINT	Y	10Y
SMALLINT	S	10S
INT	-	10
BIGINT	L	10L

String Types

String type data types can be specified using single quotes (' ') or double quotes (" "). It contains two data types: VARCHAR and CHAR. Hive follows C-types escape characters.

The following table depicts various CHAR data types:

Data Type	Length
VARCHAR	1 to 65355
CHAR	255

Timestamp

It supports traditional UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff" and format "yyyy-mm-dd hh:mm:ss.ffffffffff".

Dates

DATE values are described in year/month/day format in the form {{YYYY-•MM-•DD}}.

Decimals

The DECIMAL type in Hive is as same as Big Decimal format of Java. It is used for representing immutable arbitrary precision. The syntax and example is as follows:

```
DECIMAL(precision, scale)

decimal(10,0)
```

Union Types

Union is a collection of heterogeneous data types. You can create an instance using **create union**. The syntax and example is as follows:

```
UNIONTYPE<int, double, array<string>, struct<a:int,b:string>>



{0:1}

{1:2.0}

{2:["three","four"]}

{3:{"a":5,"b":"five"}}

{2:["six","seven"]}

{3:{"a":8,"b":"eight"}}

{0:9}

{1:10.0}
```

Literals

The following literals are used in Hive:

Floating Point Types

Floating point types are nothing but numbers with decimal points. Generally, this type of data is composed of DOUBLE data type.

Decimal Type

Decimal type data is nothing but floating point value with higher range than DOUBLE data type. The range of decimal type is approximately -10^{-308} to 10^{308}.

Null Value

Missing values are represented by the special value NULL.

Complex Types

The Hive complex data types are as follows:

Arrays

Arrays in Hive are used the same way they are used in Java. Syntax:

ARRAY<data_type>

Maps

Maps in Hive are similar to Java Maps. Syntax:

MAP<primitive_type, data_type>

Structs

Structs in Hive is similar to using complex data with comment.

Syntax: STRUCT<col_name : data_type [COMMENT col_comment], ...>

4. CREATE DATABASE

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 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] <database name>;
```

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

```
CREATE DATABASE [IF NOT EXISTS] userdb;
```

or

```
CREATE SCHEMA userdb;
```

The following query is used to verify a databases list:

```
SHOW DATABASES;

default

userdb
```

Output

userdb created successfully.

5. DROP DATABASE

This chapter describes how to drop a database in Hive. The usage of SCHEMA and DATABASE are same.

Drop Database Statement

Drop Database is a statement that drops all the tables and deletes the database. Its syntax is as follows:

```
DROP DATABASE StatementDROP (DATABASE|SCHEMA) [IF EXISTS] database_name
[RESTRICT|CASCADE];
```

The following queries are used to drop a database. Let us assume that the database name is **userdb**.

```
DROP DATABASE IF EXISTS userdb;
```

The following query drops the database using **CASCADE**. It means dropping respective tables before dropping the database.

```
DROP DATABASE IF EXISTS userdb CASCADE;
```

The following query drops the database using **SCHEMA**.

```
DROP SCHEMA userdb;
```

Output:

Drop userdb database successful.

6. CREATE TABLE

This chapter explains how to create a table and how to insert data into it. The conventions of creating a table in HIVE is quite similar to creating a table using SQL.

Create Table Statement

Create Table is a statement used to create a table in Hive. The syntax and example are as follows:

Syntax

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]
table_name

[(col_name data_type [COMMENT col_comment], ...)] [COMMENT

table_comment]

[ROW FORMAT row_format]

[STORED AS file_format]
```

Example

Let us assume you need to create a table named **employee** using **CREATE TABLE** statement. The following table lists the fields and their data types in employee table:

Sr. No.	Field Name	Data Type
1	Eid	int
2	Name	String
3	Salary	Float
4	designation	String

The following query creates a table named **employee** using the above data.

```
CREATE TABLE IF NOT EXISTS employee ( eid int, name String,
        salary String, destination String);
```

If you add the option IF NOT EXISTS, Hive ignores the statement in case the table already exists.

On successful creation of table, you get to see the following response:

```
OK
Time taken: 5.905 seconds hive
```

Output:

```
Table employee created.
```

7. ALTER TABLE

This chapter explains how to alter the attributes of a table such as changing its table name, changing column names, adding columns, and deleting or replacing columns.

Alter Table Statement

It is used to alter a table in Hive.

Syntax

The statement takes any of the following syntaxes based on what attributes we wish to modify in a table.

```
ALTER TABLE name RENAME TO new_name

ALTER TABLE name ADD COLUMNS (col_spec[, col_spec ...]) ALTER TABLE

name DROP [COLUMN] column_name

ALTER TABLE name CHANGE column_name new_name new_type

ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])
```

Rename To... Statement

The following query renames the table from **employee** to **emp**.

```
ALTER TABLE employee RENAME TO emp;
```

Output:

```
Table renamed successfully.
```

Change Statement

The following table contains the fields of **employee** table and it shows the fields to be changed (in bold).

Field Name	Convert from Data Type	Change Field Name	Convert to Data Type
eid	int	eid	int
name	String	**ename**	String
salary	**Float**	salary	**Double**
designation	String	designation	String

The following queries rename the column name and column data type using the above data:

```
ALTER TABLE employee CHANGE name ename String;

ALTER TABLE employee CHANGE salary salary Double;
```

Output:

```
Change column successful.
```

Add Columns Statement

The following query adds a column named **dept** to the **employee** table.

```
ALTER TABLE table_name ADD COLUMNS (
      dept STRING COMMENT 'Department name');
```

Output:

```
Add column successful.
```

Replace Statement

The following query deletes all the columns from the **employee** table and replaces it with **emp** and **name** columns:

```
ALTER TABLE table_name CHANGE [COLUMN] col_old_name col_new_name
column_type [COMMENT col_comment] [FIRST|AFTER column_name]
example

ALTER TABLE movie_name1 CHANGE dept depart string;
```

Output:

```
Replace column successful.
```

8. DROP TABLE

This chapter describes how to drop a table in Hive. When you drop a table from Hive Metastore, it removes the table/column data and their metadata. It can be a normal table (stored in Metastore) or an external table (stored in local file system); Hive treats both in the same manner, irrespective of their types.

Drop Table Statement

The syntax is as follows:

```
DROP TABLE [IF EXISTS] table_name;
```

The following query drops a table named **employee:**

```
DROP TABLE IF EXISTS employee;
```

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

```
OK
Time taken: 5.3 seconds
```

Output:

```
Drop table successful.
```

9. 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. **The main advantage of creating table partition is faster query performance**

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.

Advantages

- Partitioning is used for distributing execution load horizontally.
- As the data is stored as slices/parts, query response time is faster to process the small part of the data instead of looking for a search in the entire data set.
- For example, In a large user table where the table is partitioned by country, then selecting users of country ‘IN’ will just scan one directory ‘country=IN’ instead of all the directories.

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.

```
ALTER TABLE employee
        ADD PARTITION (year=’2013’)
        location '/2012/part2012';

       or

       CREATE TABLE journey_v4(
       CODTF string,
       CODNRBEENF string,
       FECHAOPRCNF timestamp,
       FRECUENCIA int)
       PARTITIONED BY (year string,month string)
       ROW FORMAT DELIMITED
       FIELDS TERMINATED BY ','
       LINES TERMINATED BY '\n'
       stored as Avro
       TBLPROPERTIES ("immutable"="false","avro.compress"="zlib","immutable"="false");
```

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:

```
  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:

```
ALTER TABLE employee DROP [IF EXISTS]

    > PARTITION (year='1203');
```

External Partitioned Tables

We can create external partitioned tables as well, just by using the EXTERNAL keyword in the CREATE statement, but for creation of External Partitioned Tables, we do not need to mention LOCATION clause as we will mention locations of each partitions separately while inserting data into table.

```
CREATE TABLE partitioned_user(
        firstname VARCHAR(64),
        lastname  VARCHAR(64),
        address   STRING,
        city      VARCHAR(64),
        post      STRING,
        phone1    VARCHAR(64),
        phone2    STRING,
        email     STRING,
        web       STRING
        )
        PARTITIONED BY (country VARCHAR(64), state VARCHAR(64))
        STORED AS SEQUENCEFILE;

DESCRIBE FORMATTED partitioned_user;
```

10. BUILT-IN OPERATORS

This chapter explains the built-in operators of Hive. There are four types of operators in Hive:

1. Relational Operators
2. Arithmetic Operators
3. Logical Operators
4. 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.
A RLIKE B	Strings	NULL if A or B is NULL, TRUE if any substring of A matches the Java regular expression B , otherwise FALSE.
A REGEXP B	Strings	Same as RLIKE.

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:

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

On successful execution of 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.

```
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.

```
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.

```
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.

11. BUILT-IN FUNCTIONS

This chapter explains 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)	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

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

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

```
2.0
```

floor() function

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

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

```
2.0
```

floor() function

```
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 as 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.

12. VIEWS AND INDEXES

This chapter describes how to create and manage views. 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 scenario:

```
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:

```
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:

```
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 <index_name> ON <table_name>
```

The following query drops an index named index_salary:

```
DROP INDEX index_salary ON employee;
```

13. HIVEQL SELECT...WHERE

The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore. This chapter explains how to use the SELECT statement with WHERE clause.

SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. The built-in operators and functions generate an expression, which fulfils the condition.

Syntax

Given below is the syntax of the SELECT query:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ... FROM
table_reference
[WHERE where_condition]
[GROUP BY col_list] [HAVING
having_condition]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT number];
```

Example

Let us take an example for SELECT…WHERE clause. Assume we have the employee table as given below, with fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 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 scenario:

```
SELECT * FROM employee WHERE salary>30000;
```

On successful execution of the 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     |
+------+--------------+-------------+------------------+--------+
```

Output:

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

14. HIVEQL SELECT...ORDER BY

This chapter explains how to use the ORDER BY clause in a SELECT statement. The ORDER BY clause is used to retrieve the details based on one column and sort the result set by ascending or descending order.

Syntax

Given below is the syntax of the ORDER BY clause:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ... FROM
table_reference
[WHERE where_condition]
[GROUP BY col_list] [HAVING
having_condition] [ORDER BY
col_list]]
[LIMIT number];
```

Example

Let us take an example for SELECT...ORDER BY clause. Assume employee table as given below, with the fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details in order by using Department name.

```
+------+--------------+-------------+-------------------+--------+
| 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 scenario:

```
SELECT Id, Name, Dept FROM employee ORDER BY DEPT;
```

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

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

JDBC Program

Here is the JDBC program to apply Order By clause for the given example.

Output:

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

15. HIVEQL GROUP BY

This chapter explains the details of GROUP BY clause in a SELECT statement. The GROUP BY clause is used to group all the records in a result set using a particular collection column. It is used to query a group of records.

Syntax

The syntax of GROUP BY clause is as follows:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ... FROM
table_reference
[WHERE where_condition]
[GROUP BY col_list] [HAVING
having_condition] [ORDER BY
col_list]]
[LIMIT number];
```

Example

Let us take an example of SELECT…GROUP BY clause. Assume employee table as given below, with Id, Name, Salary, Designation, and Dept fields. Generate a query to retrieve the number of employees in each department.

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

The following query retrieves the employee details using the above scenario.

```
SELECT Dept,count(*) FROM employee GROUP BY DEPT;
```

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

```
+------+--------------+
| Dept |  Count(*)    |
+------+--------------+
|Admin |    1         |
|PR    |    2         |
|TP    |    3         |
+------+--------------+
```

Output:

Dept	Count(*)
Admin	1
PR	2
TP	3

16. HIVEQL JOINS

JOINS is a clause that is used for combining specific fields from two tables by using values common to each one. It is used to combine records from two or more tables in the database. It is more or less similar to SQL JOINS.

Syntax

```
join_table:

    table_reference JOIN table_factor [join_condition]

    | table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference join_condition

    | table_reference LEFT SEMI JOIN table_reference join_condition

    | table_reference CROSS JOIN table_reference [join_condition]
```

Example

We will use the following two tables in this chapter. Consider the following table named CUSTOMERS..

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Consider another table ORDERS as follows:

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

There are different types of joins given as follows:

- JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

JOIN

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

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

```
SELECT c.ID, c.NAME, c.AGE, o.AMOUNT
    > FROM CUSTOMERS c JOIN ORDERS o
    > ON (c.ID = o.CUSTOMER_ID);
```

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

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

LEFT OUTER JOIN

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

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

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDER tables:

```
SELECT  c.ID, c.NAME, o.AMOUNT, o.DATE
    > FROM CUSTOMERS c
    > LEFT OUTER JOIN ORDERS o
    > ON (c.ID = o.CUSTOMER_ID);
```

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

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

RIGHT OUTER JOIN

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

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

The following query demonstrates RIGHT OUTER JOIN between the CUSTOMER and ORDER tables.

```
SELECT  c.ID, c.NAME, o.AMOUNT, o.DATE
    > FROM CUSTOMERS c
    > RIGHT OUTER JOIN ORDERS o
    > ON (c.ID = o.CUSTOMER_ID);
```

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

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|  3   | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3   | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2   | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4   | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

FULL OUTER JOIN

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

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

```
SELECT  c.ID, c.NAME, o.AMOUNT, o.DATE
    > FROM CUSTOMERS c
    > FULL OUTER JOIN ORDERS o
    > ON (c.ID = o.CUSTOMER_ID);
```

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

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|  1 | Ramesh   |    NULL | NULL                |
|  2 | Khilan   |    1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |    3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |    1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |    2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |    NULL | NULL                |
|  6 | Komal    |    NULL | NULL                |
```

```
|   7 | Muffy    |    NULL | NULL                |
|   3 | kaushik  |    3000 | 2009-10-08 00:00:00 |
|   3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|   2 | Khilan   |    1560 | 2009-11-20 00:00:00 |
|   4 | Chaitali |    2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

Reference:

https://cwiki.apache.org/confluence/display/Hive/LanguageManual

https://www.qubole.com/resources/hive-function-cheat-sheet/

https://hortonworks.com/tutorial/loading-and-querying-data-with-hadoop/

http://hadooptutorial.info/partitioning-in-hive/

https://hortonworks.com/tutorial/how-to-process-data-with-apache-hive/#download-the-data