HIVE – CORE CONCEPTS

Background:

This document is a reference material collected to grasp a foundational understanding of Hive. This is for learning purpose. For a complete reference guide on syntax and features of Hive, one can get the same from Apache hive website. Based on versions leveraged, there can be some differences in the code snippets and should be referred to official website in case of any errors.

Introduction

Hive is an ETL and Datawarehouse technology built on top of Hadoop HDFS.

Main Characteristics of Hive:

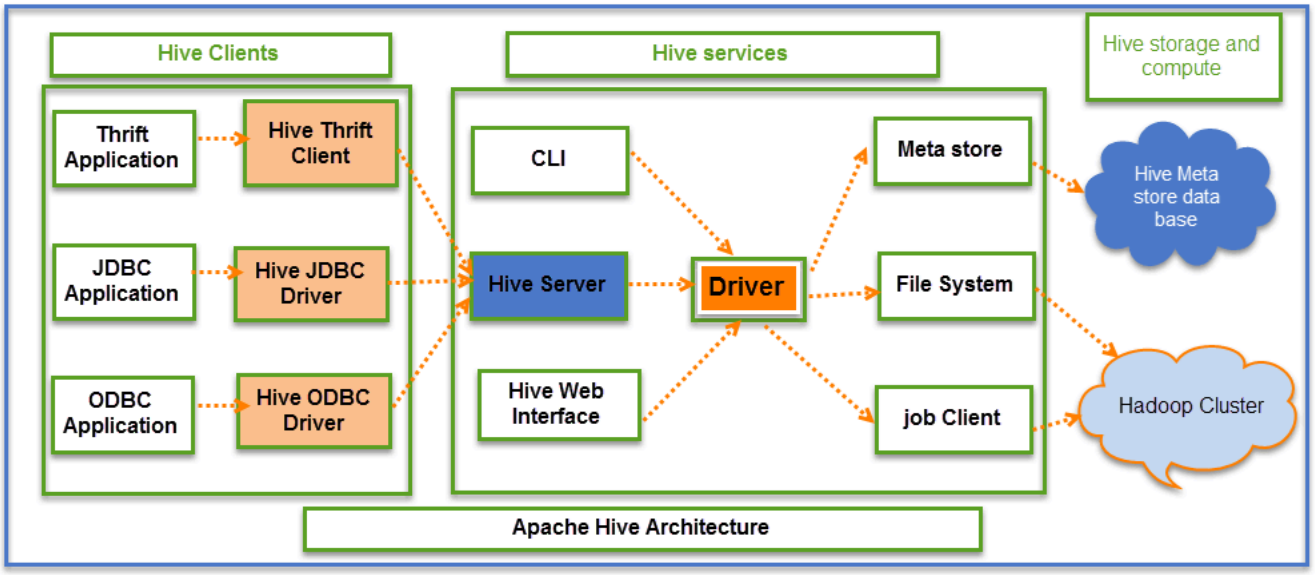
- Hive is a data warehouse to manage and extract data structured in tables.
- Hadoop's programming works on flat files. So, Hive can use directory structures to "partition" data to improve performance on certain queries.
- A new and important component of Hive i.e. Metastore used for storing schema information. This Metastore leverages RDBMS database. For single user, internal Derby database can be used or MySQL for a multi-user environment.
- Similar to SQL, Hive provides Hive Query Languages (HQL) to extract database.
- We can interact with Hive using methods like
    - Command Line Interface (CLI)
    - Web GUI
    - Java Database Connectivity (JDBC) interface

Some of the key points about Hive Query Language:

- Hive query executes on Hadoop's infrastructure primarily HDFS.
- The Hive query execution is a sequence of automatically generated map reduce Jobs.
- Hive supports partition and buckets for retrieval of data
- Hive supports custom specific UDF (User Defined Functions) that helps in writing very specific user requirements.

Hive Architecture:

HIVE – CORE CONCEPTS



Hive Consists of Mainly 3 core parts

1. **Hive Clients**
2. **Hive Services**
3. **Hive Storage and Computing**

Hive Clients:

Hive clients generally use some kind of connectors to connect to the Hive Server services.

1. Java Clients or Other Apps use Hive JDBC connectors or ODBC.
2. Thrift clients use the Hive Thrift connector

Hive Services:

All type of Client applications connects to the Hive Server inside the Hive services with help of respective jdbc/odbc drivers.

Driver is the main component that actually interfaces with data as well as the Hive Server/CLI/Hive Web Interface. It gets the request from the different sources, connects to metastore and get the request processed.

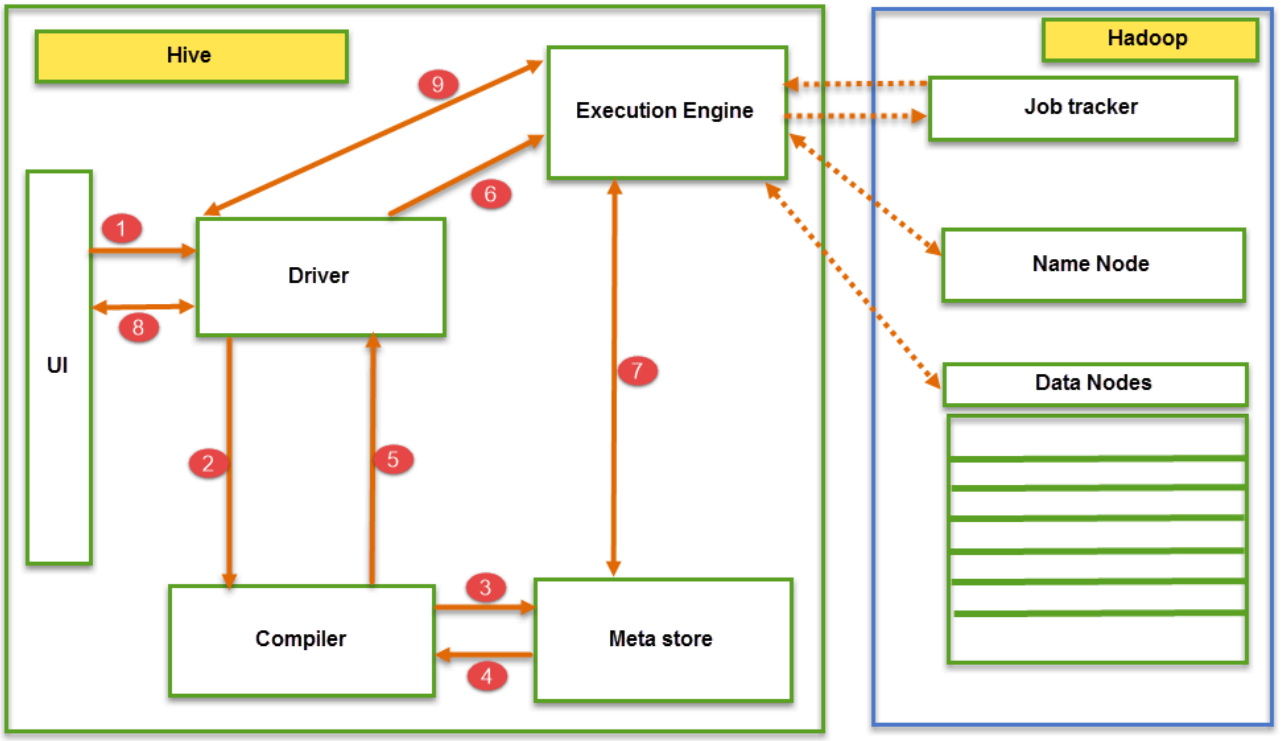
CLI / Web interface are Hive Services internal components enabling direct connection to the Hive driver for data processing.

Hive Storage and Computing:

Hive services such as Meta store, File system, and Job Client in turn communicates with Hive storage and performs the following actions

- Metadata information of tables created in Hive is stored in Hive "Meta storage database".
- Query results and data loaded in the tables are going to be stored in Hadoop cluster on HDFS.

Request Execution Flow



The data flow in Hive behaves in the following pattern;

a. Executing Query from the UI(User Interface)
b. The driver is interacting with Compiler for getting the plan. (Here plan refers to query execution) process and its related metadata information gathering
c. The compiler creates the plan for a job to be executed. Compiler communicating with Meta store for getting metadata request
d. Meta store sends metadata information back to compiler
e. Compiler communicating with Driver with the proposed plan to execute the query
f. Driver Sending execution plans to Execution engine
g. Execution Engine (EE) acts as a bridge between Hive and Hadoop to process the query. For DFS operations.

    a. EE should first contacts Name Node and then to Data nodes to get the values stored in tables.
    b. EE is going to fetch desired records from Data Nodes. The actual data of tables resides in data node only. While from Name Node it only fetches the metadata information for the query.

c. It collects actual data from data nodes related to mentioned query
d. Execution Engine (EE) communicates bi-directionally with Meta store present in Hive to perform DDL (Data Definition Language) operations. Here DDL operations like CREATE, DROP and ALTERING tables and databases are done. Meta store will store information about database name, table names and column names only. It will fetch data related to query mentioned.
e. Execution Engine (EE) in turn communicates with Hadoop daemons such as Name node, Data nodes, and job tracker to execute the query on top of Hadoop file system

8. Fetching results from driver
9. Sending results to Execution engine. Once the results fetched from data nodes to the EE, it will send results back to driver and to UI (front end)

Hive Continuously in contact with Hadoop file system and its daemons via Execution engine. The dotted arrow in the Job flow diagram shows the Execution engine communication with Hadoop daemons.

Hive Installation:

With help of the hive installation guide.txt, let's install hive on an Hadoop pseudo setup. The metastore database leveraged is the default derby database.

Basic DDL/DML in Hive:

Hive It consisits of the Data Definition Language (DDL) as well as Data Manipulation language(DML) very similar to any DBMS.
DDL/DML in Hive is very similar to SQL syntax.
Here is a sample of few commands for your practice to get the first feel.

DDL Statement	Syntax	Example
Creation of Database	create database <dbname>	`Hive> create database mytry;`
Drop database	drop database <dbname>	`Hive> drop database mytry;`
List the databases in Hive	show databases	`Hive> show databases;`

Creation of Tables	create table <tablename> (column1, column2, ….)
	
	By default, ^ is used as a delimiter.
	If you are copying pasting the code, please ensure the quotes are ascii.
		`Hive>create table myusers (userid int, username string, userage int) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';`
		
		`Hive>create table stateuser (userid int, username string, loginid string, state string) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';`

Remove tables	drop table <tablename>	`Drop table myusers;`
Change the table name	alter table <tablename> Rename to <newtablename>	`Hive> alter table myusers rename to merausers;`
Loading data into table from text file	Load data inpath <Path name> into table <tablename>	`LOAD DATA LOCAL INPATH '/home/ubuntu/mytrial/myusers.csv' INTO table merausers;`

Hive Data Types

Hive supports following data types.

Hive Numeric Data Types

Type	Memory allocation
TINY INT	Its 1-byte signed integer (-128 to 127)
SMALL INT	2-byte signed integer (-32768 to 32767)
INT	4 –byte signed integer (-2,147,484,648 to 2,147,484,647)
BIG INT	8 byte signed integer
FLOAT	4 – byte single precision floating point number
DOUBLE	8- byte double precision floating point number
DECIMAL	We can define precision and scale in this Type

Hive String Data Types

Type	Length
CHAR	255
VARCHAR	1 to 65355
STRING	We can define length here(No Limit)

Hive Date/Time Data Types

Type	Usage
Timestamp	Supports traditional Unix timestamp with optional nanosecond precision
Date	• It's in YYYY-MM-DD format. • The range of values supported for the Date type is be 0000-01-01 to 9999-12-31,

Hive Complex Data Types

Type	Usage
Arrays	**ARRAY<data_type>** Negative values and non-constant expressions not allowed
Maps	**MAP<primitive_type, data_type>** Negative values and non-constant expressions not allowed
Structs	**STRUCT<col_name :datat_type, …… >**
Union	**UNIONTYPE<data_type, datat_type, …… >**

Note: The data types support and improvement can change over different versions. It is advisable to check the Reference guide from Apache on the data types for the version currently installed in your environment.

Table Types in Hive:

There are two types of table structures like Internal and External tables depending on the loading and design of schema in Hive.

Characteristics	Internal Table	External Table
Coupling	Tightly coupled	Loosely coupled
Type	Data on schema	Schema on data
Delete table command impact	On delete table, both data as well as schema is removed.	On delete table, only schema is removed. Data still remains
Storage Place	Stored in OS filesystem in folder /user/hive/warehouse	Stored in HDFS

Table creation	`CREATE TABLE mytable_internal (id INT,Name STRING);`
	`Row format delimited`
	`Fields terminated by '\t'`
		`CREATE EXTERNAL TABLE mytable_external(id INT,Name STRING)`
		`Row format delimited`
		`Fields terminated by '\t'`
		`LOCATION '/user/myhive/mytable_external;`
		In case Location not specified, same can be done while data loading.

Load data into table	`Hive>LOAD DATA INPATH '/home/ubuntu/mytrial/myusers.txt' INTO TABLE mytable_internal;`	`Hive>LOAD DATA INPATH '/home/ubuntu/mytrial/myusers.txt' INTO TABLE mytable_external;`
When to choose?	• If the processing data available in local file system • If we want Hive to manage the complete lifecycle of data including the deletion	• If processing data available in HDFS • Useful when the files are being used outside of Hive

Major Difference between Internal Vs External tables

Feature	Internal	External
Schema	Data on Schema	Schema on Data
Storage Location	/usr/hive/warehouse	HDFS location
Data availability	Within local file system	Within HDFS

Partitions and Buckets in Hive:

Partitions in hive help store data in a split way with help of partition keys. This helps in accessing relevant data faster.

E.g. We have users database for any global platform. We could have state-wise user tables with help of partition keys. Similarly transport company might have vehicle-type wise tables for ease of access.

There are two types of Partitioning in Hive

1. Static Partitioning
2. Dynamic Partitioning

Static Partitioning	Dynamic Partitioning
In static or manual partitioning, it is required to pass the values of partitioned columns manually while loading the data into the table	In dynamic partitioning, we only mention the column name with which the partition should occur. So, it is not required to pass the values of partitioned columns manually. The partitions are automatically created on insert as per different values.
There is no table where you load all the data. Based on the different partitioning names, the parts are created.	You need to load all the data into a temporary table and then from that create the partition table.
We need to create those number of partitions manually.	Here Hive automatically generates the required partition based on distinct values of partition column.

Static Partitioning: Example of user spread across states. We need to create partition for each state. In our example we need to do it for around 8 states.

Steps	Syntax
Create the partition table	Hive> create table stateuser_static(userid int, name string, login string) PARTITIONED BY(state string) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
Load data into the partition table from local csv file as per the partition column state's values.	Hive> LOAD DATA LOCAL INPATH '/home/ubuntu/mytrial/stateuser.csv' INTO TABLE stateuser_static partition(state= "Kerala"); LOAD DATA LOCAL INPATH '/home/ubuntu/mytrial/stateuser.csv' INTO TABLE stateuser_static partition(state= "Maharashtra"); LOAD DATA LOCAL INPATH '/home/ubuntu/mytrial/stateuser.csv' INTO TABLE stateuser_static partition(state= "Meghalaya");
Browse the HDFS directory	You will be able to browse the hdfs file directory and see different state partition based hive tables have been created.

Dynamic Partitioning: Example of user spreads across states. Partition is created state-wise dynamically.

Steps	Syntax
Create table containing all the users with their states	Hive> create table stateuser (userid int, name string, login string, state string) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
Load data into the table from local csv file.	Hive> LOAD DATA LOCAL INPATH '/home/ubuntu/mytrial/stateuser.csv' INTO TABLE stateuser;
Create the partition tables	Hive> create table stateuser_part(userid int, name string, login string) PARTITIONED BY(state string);
Setting the hive property for dynamic partition mode to nonstrict.	Hive> set hive.exec.dynamic.partition=true; Hive> set hive.exec.dynamic.partition.mode = nonstrict;
Load data into partition data	Hive> INSERT OVERWRITE TABLE stateuser_part PARTITION(state) SELECT userid, name, login,state from stateuser;

Bucketing: Partitions divide the data based on different values of a particular column. Sometimes, we need to further divide or divide the data differently and not only column values. So even a partitioned table can be further bucketed.

- The data i.e. present in that partitions can be divided further into Buckets
- The division is performed based on Hash of particular columns that we selected in the table.
- Buckets use some form of Hashing algorithm at back end to read each record and place it into buckets
- In Hive, we enable buckets by using **set.hive.enforce.bucketing=true;**

Let's try an example of bucketing our stateuser table.

Case 1 : We are loading stateuser data fresh without any partitions and just bucketing.

Steps	Syntax
Create table containing all the users with their states With the number of buckets	Hive> create table stateuser_bucket (userid int, name string, login string, state string) clustered by (state) into 4 buckets ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
Load data into the table from already existing stateuser table	Hive> Insert overwrite table stateuser_bucket Select * from stateuser;

In this case, we use the insert statement. Hive convert insert queries into mapreduce whenever there are grouping, subquery,condition, join	
Check for data using hdfs dfs -cat <path of the bucket part>	`hdfs dfs -cat /user/hive/warehouse/stateuser_bucket/000000_0`

View & Index in Hive:

Views are similar to tables, which are generated based on the requirements.

```
Create VIEW <VIEWNAME> AS SELECT statement
```

e.g.

```
Hive>Create VIEW try_view AS SELECT * FROM stateuser where userid > 10014;
```

Indexes are pointers to particular column name of a table.

- The user has to manually define the index
- Wherever we are creating index, it means that we are creating pointer to particular column name of table
- Any Changes made to the column present in tables are stored using the index value created on the column name.

Syntax:

```
Create INDEX <INDEX_NAME> ON TABLE < TABLE_NAME(column names)>
```

Let's try:

```
Create INDEX sample_Index ON TABLE stateuser(userid)
```

Hive Queries: Order By, Group By, Distribute By, Cluster By

Hive provides SQL type querying language for the ETL purpose on top of Hadoop file system. Hive Query language (HiveQL) provides SQL type environment in Hive to work with tables, databases, queries.

Hive queries provides the following features:

- Data modeling such as Creation of databases, tables, etc.
- ETL functionalities such as Extraction, Transformation, and Loading data into tables
- Joins to merge different data tables
- User specific custom scripts for ease of code
- Faster querying tool on top of Hadoop

Clause	Syntax with example	Comments
Order by	Select * from <table name> order by <column name>; Hive> select * from stateuser order by name;	a. Sort the data base on the column name b. For string column, it uses lexicographical ordering

group by	Select <column name1>, aggregate fn(column name2> from <table name> group by <column name1> Where aggregate fn() can be count(), sum(), subtract() and any other aggregate functions. e.g. select state, count(userid) from stateuser group by state;	a. group by clause need to have one aggregate function to be able run the group by
Order by	Select column name1 ,... from <table name> sort by <column name> ASC/DESC Hive> select * from stateuser sort by login DESC	a. Sort works similar to order by

Cluster by	`Select <column name1>, <column name2>, ……. From <table name> cluster by <column name2>`
	
	`Hive> SELECT userid, name from stateuser CLUSTER BY userid;`
		a. Cluster BY clause used on tables present in Hive. Hive uses the columns in Cluster by to distribute the rows among reducers.
		b. Cluster BY columns will go to the multiple reducers.
		c. It ensures sorting orders of values present in multiple reducers

Distribute by	`Select <column name1>, <column name2>, ….. from <table name> distribute by <column name2>`
	
	`Hive> SELECT userid, name from stateuser DISTRIBUTE BY userid;`
		a. Hive uses the columns in Distribute by to distribute the rows among reducers.
		b. Distribute By columns will go to all reducers
		c. It ensures each of N reducers gets non-overlapping ranges of column
		d. It doesn't sort the output of each reducer

Hive Queries: Join & Subquery:

Joins are very similar to SQL joins. Below are some characterestics.

- Only Equality joins are allowed In Joins
- More than two tables can be joined in the same query
- LEFT, RIGHT, FULL OUTER joins exist in order to provide more control over ON Clause for which there is no match
- Joins are not Commutative
- Joins are left-associative irrespective of whether they are LEFT or RIGHT joins

To try the various joins, create another hive table called user address.

```
create table user_address (userid int, address string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```

Join Type	Syntax with example	Comments

Inner	Select t1.<column name1> , t2.<column name2>,..... from <table1> t1 join <table2> t2 on (t1.column name4 = t2.column name6) e.g. select t1.userid, t1.login,t2.address from stateuser t1 join user_address t2 on (t1.userid =t2.userid)	a. Returns all rows that are found in both the tables in inner join.
Left outer	Select t1.<column name1> , t1.<column name2>, t2.<column name3> from table1 t1 left outer join table2 t2 on (t1.column name4 = t2.column name5) e.g. select t1.userid, t1.login,t2.address from stateuser t1 left outer join user_address t2 on (t1.userid =t2.userid)	b. Returns all rows in left and the relevant matching columns of right. c. If there is no row in right matching, then a Null is placed in that column.
Right outer	Select t1.<column name1> , t1.<column name2>, t2.<column name3> from table1 t1 right outer join table2 t2 on (t1.column name4 = t2.column name5) e.g. select t1.userid, t1.login,t2.address from stateuser t1 right outer join user_address t2 on (t1.userid =t2.userid)	d. Returns all rows in right and the relevant matching columns of right. e. If there is no row in right matching, then a Null is placed in that column
Full Outer	Select t1.<column name1> , t1.<column name2>, t2.<column name3> from table1 t1 full outer join table2 t2 on (t1.column name4 = t2.column name5) e.g. select t1.userid, t1.login,t2.address from stateuser t1 full outer join user_address t2 on (t1.userid =t2.userid)	f. The output displaying all the records present in both the table by checking the condition mentioned in the query. Null values in output here indicates the missing values from the columns of both tables. g. It returns all the records from both tables and fills in NULL Values for the columns missing values matched on either side.

Sub queries

A Query present within a Query is known as a sub query. The main query will depend on the values returned by the subqueries.

Subqueries can be classified into two types

- Subqueries in FROM clause
- Subqueries in WHERE clause

When to use:

- To get a particular value combined from two column values from different tables
- Dependency of one table values on other tables
- Comparative checking of one column values from other tables

Syntax:

```
Subquery in FROM clause
SELECT <column names 1, 2…n> From (SubQuery) <MainTable >;

Subquery in WHERE clause
SELECT <column names 1, 2…n> From <MainTable> WHERE col1 IN (SubQuery);
```

Hive HQL & Operators:
Hive Query Language (HiveQL) is a query language in Apache Hive for processing and analyzing structured data. It separates users from the complexity of Map Reduce programming. It reuses common concepts from relational databases, such as tables, rows, columns, and schema, to ease learning. Hive provides a CLI for Hive query writing using Hive Query Language (HiveQL).

Most interactions tend to take place over a command line interface (CLI). Generally, HiveQL syntax is similar to the SQL syntax that most data analysts are familiar with.

HiveQL Built-in Operators

Hive provides Built-in operators for Data operations to be implemented on the tables present inside Hive warehouse.

These operators are used for mathematical operations on operands, and it will return specific value as per the logic applied.

Below are the main types of Built-in Operators in HiveQL:

- Relational Operators
- Arithmetic Operators
- Logical Operators
- Operators on Complex types
- Complex type Constructors

Built-in Operator	Description	Operand
X = Y	TRUE : if expression X is equivalent to expression Y Otherwise FALSE.	It takes all primitive types
X != Y	TRUE if expression X is not equivalent to expression Y Otherwise FALSE.	It takes all primitive types
X < Y	TRUE if expression X is less than expression Y Otherwise FALSE.	It takes all primitive types
X <= Y	TRUE if expression X is less than or equal to expression Y Otherwise FALSE.	It takes all primitive types
X>Y	TRUE if expression X is greater than expression Y Otherwise FALSE.	It takes all primitive types
X>= Y	TRUE if expression X is greater than or equal to expression Y Otherwise FALSE.	It takes all primitive types
X IS NULL	TRUE if expression X evaluates to NULL otherwise FALSE.	It takes all types
X IS NOT NULL	FALSE If expression X evaluates to NULL otherwise TRUE.	It takes all types
X LIKE Y	TRUE If string pattern X matches to Y otherwise FALSE.	Takes only Strings
X RLIKE Y	NULL if X or Y is NULL, TRUE if any substring of X matches the Java regular expression Y, otherwise FALSE.	Takes only Strings
X REGEXP Y	Same as RLIKE.	Takes only Strings

HiveQL Arithmetic Operators

We use Arithmetic operators for performing arithmetic operations on operands

- Arithmetic operations such as add, subtract, multiply and divide between operands we use these Operators.
- The operand types all are number types in these Operators

Built-in Operator	Description	Operand
X + Y	It will return the output of adding X and Y value.	It takes all number types
X – Y	It will return the output of subtracting Y from X value.	It takes all number types
X * Y	It will return the output of multiplying X and Y values.	It takes all number types
X / Y	It will return the output of dividing Y from X.	It takes all number types
X % Y	It will return the remainder resulting from dividing X by Y.	It takes all number types
X & Y	It will return the output of bitwise AND of X and Y.	It takes all number types
X | Y	It will return the output of bitwise OR of X and Y.	It takes all number types
X ^ Y	It will return the output of bitwise XOR of X and Y.	It takes all number types
~X	It will return the output of bitwise NOT of X.	It takes all number types

Hive QL Logical Operators

We use Logical operators for performing Logical operations on operands

- Logical operations such as AND, OR, NOT between operands we use these Operators.
- The operand types all are BOOLEAN type in these Operators

The following Table will give us details about Logical operators in HiveSQL:

Operators	Description	Operands
X AND Y	TRUE if both X and Y are TRUE, otherwise FALSE.	Boolean types only
X && Y	Same as X AND Y but here we using && symbol	Boolean types only
X OR Y	TRUE if either X or Y or both are TRUE, otherwise FALSE.	Boolean types only
X || Y	Same as X OR Y but here we using || symbol	Boolean types only
NOT X	TRUE if X is FALSE, otherwise FALSE.	Boolean types only
!X	Same as NOT X but here we using! Symbol	Boolean types only

Operators on Complex Types

The following Table will give us details about Complex Type Operators . These are operators which will provide a different mechanism to access elements in complex types.

Operators	Operands	Description
A[n]	A is an Array and n is an integer type	It will return nth element in the array A. The first element has index of 0
M[key]	M is a Map<K, V> and key has type K	It will return the values belongs to the key in the map

Complex Type Constructors

The following Table will give us details about Complex type Constructors. It will construct instances on complex data types. These are of complex data types such as Array, Map and Struct types in Hive.

In this section, we are going to see the operations performed on Complex type Constructors.

Operators	Operands	Description
array	(val1, val2, …)	It will create an array with the given elements as mentioned like val1, val2

Operators	Operands	Description
Create_ union	(tag, val1, val2, …)	It will create a union type with the values that is being mentioned to by the tag parameter
map	(key1, value1, key2, value2, …)	It will create a map with the given key/value pairs mentioned in operands
Named_struct	(name1, val1, name2, val2, …)	It will create a Struct with the given field names and values mentioned in operands
STRUCT	(val1, val2, val3, …)	Creates a Struct with the given field values. Struct field names will be col1, col2, .

Hive Functions: Built-in & UDF [User Defined Functions]

Functions are built for a specific purpose to perform operations like Mathematical, arithmetic, logical, and relational on the operands of table column names.

Built-in functions

These are functions that are already available in Hive. First, we have to check the application requirement, and then we can use these built-in functions in our applications. We can call these functions directly in our application.

The syntax and types are mentioned in the following section.

Types of Built-in Functions in HIVE

- Collection Functions
- Date Functions
- Mathematical Functions
- Conditional Functions
- String Functions
- Misc. Functions

Collection Functions

These functions are used for collections. Collections mean the grouping of elements and returning single or array of elements depends on return type mentioned in function name.

Return Type	Function Name	Description
INT	size(Map<K.V>)	It will fetch and give the components number in the map type
INT	size(Array<T>)	It will fetch and give the elements number in the array type
Array<K>	Map_keys(Map<K.V>)	It will fetch and gives an array containing the keys of the input map. Here array is in unordered
Array<V>	Map_values(Map<K.V>)	It will fetch and gives an array containing the values of the input map. Here array is in unordered
Array<t>	Sort_array(Array<T>)	sorts the input array in ascending order of array and elements and returns it

Date Functions

These are used to perform Date Manipulations and Conversion of Date types from one type to another type:

Function Name	Return Type	Description
Unix_Timestamp()	BigInt	We will get current Unix timestamp in seconds
To_date(string timestamp)	string	It will fetch and give the date part of a timestamp string:
year(string date)	INT	It will fetch and give the year part of a date or a timestamp string
quarter(date/timestamp/string)	INT	It will fetch and give the quarter of the year for a date, timestamp, or string in the range 1 to 4
month(string date)	INT	It will give the month part of a date or a timestamp string
hour(string date)	INT	It will fetch and gives the hour of the timestamp
minute(string date)	INT	It will fetch and gives the minute of the timestamp
Date_sub(string starting date, int days)	string	It will fetch and gives Subtraction of number of days to starting date
Current_date	date	It will fetch and gives the current date at the start of query evaluation
LAST _day(string date)	string	It will fetch and gives the last day of the month which the date belongs to

Function Name	Return Type	Description
trunc(string date, string format)	string	It will fetch and gives date truncated to the unit specified by the format. **Supported formats in this :** MONTH/MON/MM, YEAR/YYYY/YY.

Mathematical Functions

These functions are used for Mathematical Operations. Instead of creatingUDFs , we have some inbuilt mathematical functions in Hive.

Function Name	Return Type	Description
round(DOUBLE X)	BIGINT	It will fetch and returns the rounded BIGINT value of X
round(DOUBLE X, INT d)	DOUBLE	It will fetch and returns X rounded to d decimal places
bround(DOUBLE X)	DOUBLE	It will fetch and returns the rounded BIGINT value of X using HALF_EVEN rounding mode
floor(DOUBLE X)	BIGINT	It will fetch and returns the maximum BIGINT value that is equal to or less than X value
ceil(DOUBLE a), ceiling(DOUBLE a)	BIGINT	It will fetch and returns the minimum BIGINT value that is equal to or greater than X value
rand(), rand(INT seed)	DOUBLE	It will fetch and returns a random number that is distributed uniformly from 0 to 1
round(num)	BIGINT	It returns the BIGINT for the rounded value of DOUBLE num.
floor(num)	BIGINT	It returns the largest BIGINT that is less than or equal to num.
ceil(num), ceiling(DOUBLE num)	BIGINT	It returns the smallest BIGINT that is greater than or equal to num.
exp(num)	DOUBLE	It returns exponential of num.
ln(num)	DOUBLE	It returns the natural logarithm of num.
log10(num)	DOUBLE	It returns the base-10 logarithm of num.
sqrt(num)	DOUBLE	It returns the square root of num.
abs(num)	DOUBLE	It returns the absolute value of num.
sin(d)	DOUBLE	It returns the sin of num, in radians.
asin(d)	DOUBLE	It returns the arcsin of num, in radians.

Function Name	Return Type	Description
cos(d)	DOUBLE	It returns the cosine of num, in radians.
acos(d)	DOUBLE	It returns the arccosine of num, in radians.
tan(d)	DOUBLE	It returns the tangent of num, in radians.
atan(d)	DOUBLE	It returns the arctangent of num, in radians

Conditional Functions

These functions used for conditional values checks.

Function Name	Return Type	Description
if(Boolean testCondition, T valueTrue, T valueFalseOrNull)	T	It will fetch and gives value True when Test Condition is of true, gives value False Or Null otherwise.
ISNULL(X)	Boolean	It will fetch and gives true if X is NULL and false otherwise.
ISNOTNULL(X)	Boolean	It will fetch and gives true if X is not NULL and false otherwise.

String Functions

String manipulations and string operations these functions can be called. Some basic functions are detailed here.

Function Name	Return Type	Description
length(str)	INT	It returns the length of the string.
reverse(str)	STRING	It returns the string in reverse order.
concat(str1, str2, ...)	STRING	It returns the concatenation of two or more strings.
substr(str, start_index)	STRING	It returns the substring from the string based on the provided starting index.
substr(str, int start, int length)	STRING	It returns the substring from the string based on the provided starting index and length.
upper(str)	STRING	It returns the string in uppercase.
lower(str)	STRING	It returns the string in lowercase.
trim(str)	STRING	It returns the string by removing whitespaces from both the ends.
ltrim(str)	STRING	It returns the string by removing whitespaces from left-hand side.
reverse(string X)	string	It will give the reversed string of X
rpad(string str, int length, string pad)	string	It will fetch and gives str, which is right-padded with pad to a length of length(integer value)
rtrim(string X)	string	It will fetch and returns the string resulting from trimming spaces from the end (right hand side) of X **For example**, rtrim(‘ results ‘) results in ‘ results’

Function Name	Return Type	Description
space(INT n)	string	It will fetch and gives a string of n spaces.
split(STRING str, STRING pat)	array	Splits str around pat (pat is a regular expression).
Str_to_map(text[, delimiter1, delimiter2])	map<String ,String>	It will split text into key-value pairs using two delimiters.

Aggregate Functions in Hive

In Hive, the aggregate function returns a single value resulting from computation over many rows. Let''s see some commonly used aggregate functions: -

Return Type	Operator	Description
BIGINT	count(*)	It returns the count of the number of rows present in the file.
DOUBLE	sum(col)	It returns the sum of values.
DOUBLE	sum(DISTINCT col)	It returns the sum of distinct values.
DOUBLE	avg(col)	It returns the average of values.
DOUBLE	avg(DISTINCT col)	It returns the average of distinct values.
DOUBLE	min(col)	It compares the values and returns the minimum one form it.
DOUBLE	max(col)	It compares the values and returns the maximum one form it.

UDFs (User Defined Functions)

In Hive, the users can define own functions to meet certain client requirements. These are known as UDFs in Hive. User Defined Functions written in Java for specific modules.

Some of UDFs are specifically designed for the reusability of code in application frameworks. The developer will develop these functions in Java and integrate those UDFs with the Hive.

During the Query execution, the developer can directly use the code, and UDFs will return outputs according to the user-defined tasks. It will provide high performance in terms of coding and execution.

For example, for string stemming we don't have any predefined function in Hive. For this, we can write stem UDF in Java. Wherever we require Stem functionality, we can directly call this Stem UDF in Hive.

Here stem functionality means deriving words from its root words. It is like stemming algorithm reduces the words "loving", "loves", and "loved" to the root word "love." For performing this type of functionality, we can write UDF in Java and integrate it with Hive.

Depending on the use cases, the UDFs can be written. It will accept and produce different numbers of input and output values.

The general type of UDF will accept a single input value and produce a single output value. If the UDF is used in the query, then UDF will be called once for each row in the result data set.

In the other way, it can accept a group of values as input and return a single output value as well.