UNIT-5

Applying Structure to Hadoop Data with Hive: Saying Hello to Hive, Seeing How the Hive is Put Together, Getting Started with Apache Hive, Examining the Hive Clients, Working with Hive Data Types, Creating and Managing Databases and Tables, Seeing How the Hive Data Manipulation Language Works, Querying and Analyzing Data

Apache Hive is a Data Warehousing package built on top of Hadoop and is used for data analysis. Hive is targeted towards users who are comfortable with SQL. It is similar to SQL and called HiveQL, used for managing and querying structured data. Apache Hive is used to abstract complexity of Hadoop. This language also allows traditional map/reduce programmers to plug in their custom mappers and reducers. The popular feature of Hive is that there is no need to learn Java.

Hive, an open source peta-byte scale date warehousing framework based on Hadoop, was developed by the Data Infrastructure Team at Facebook. Hive is also one of the technologies that are being used to address the requirements at Facebook. Hive is very popular with all the users internally at Facebook and is being used to run thousands of jobs on the cluster with hundreds of users, for a wide variety of applications. Hive-Hadoop cluster at Facebook stores more than 2PB of raw data and regularly loads 15 TB of data on a daily basis.

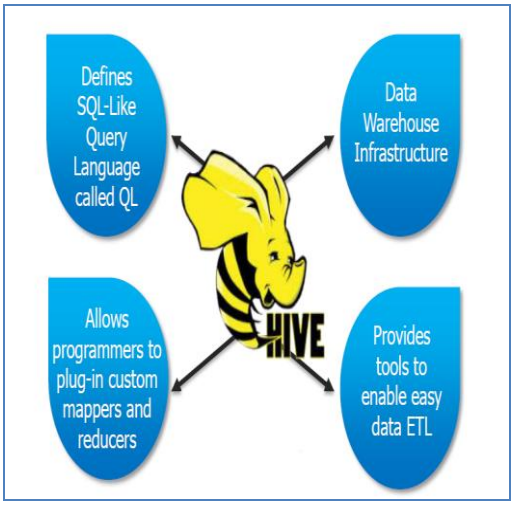

Figure 1

Let's look at some of its features that makes it popular and user friendly:

- Allows programmers to plug in custom Mappers and Reducers.
- Has Data Warehouse infrastructure.
- Provides tools to enable easy data ETL.
- Defines SQL-like query language called QL.

Before implementing Hive, Facebook faced a lot of challenges as the size of data being generated increased or rather exploded, making it really difficult to handle them. The traditional RDBMS couldn't handle the pressure and as a result Facebook was looking out for better options. To solve this impending issue, Facebook initially tried using Hadoop MapReduce, but with difficulty in programming and mandatory knowledge in SQL, made it an impractical solution. Hive allowed them to overcome the challenges they were facing.

With Hive, they are now able to perform the following:

- Tables can be portioned and bucketed
- Schema flexibility and evolution
- JDBC/ODBC drivers are available
- Hive tables can be defined directly in the HDFS
- Extensible – Types, Formats, Functions and scripts

Hive Use Case in Healthcare:

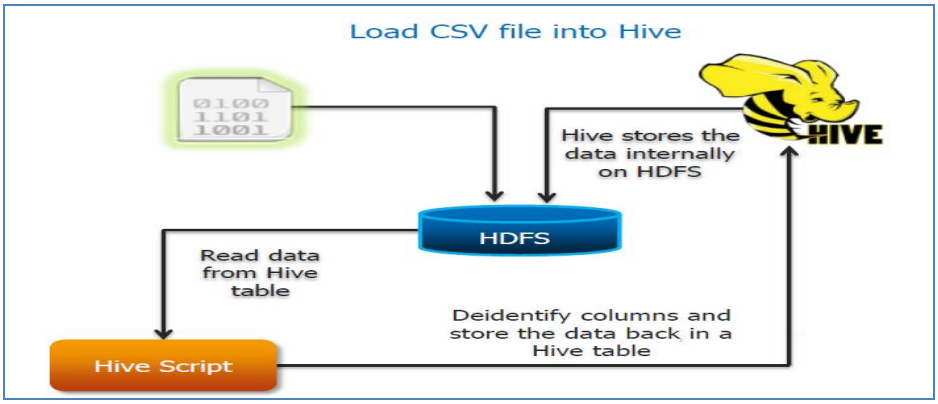

Figure 2

Where to Use Hive?

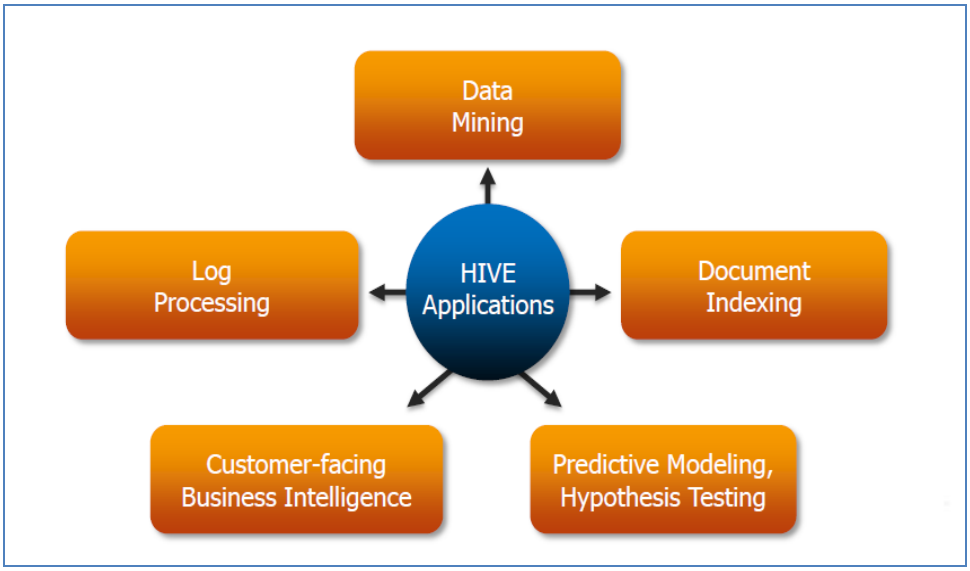

Figure 3

Apache Hive can be used in the following places:

- Data Mining
- Log Processing
- Document Indexing
- Customer Facing Business Intelligence
- Predictive Modelling
- Hypothesis Testing

Hive Architecture:

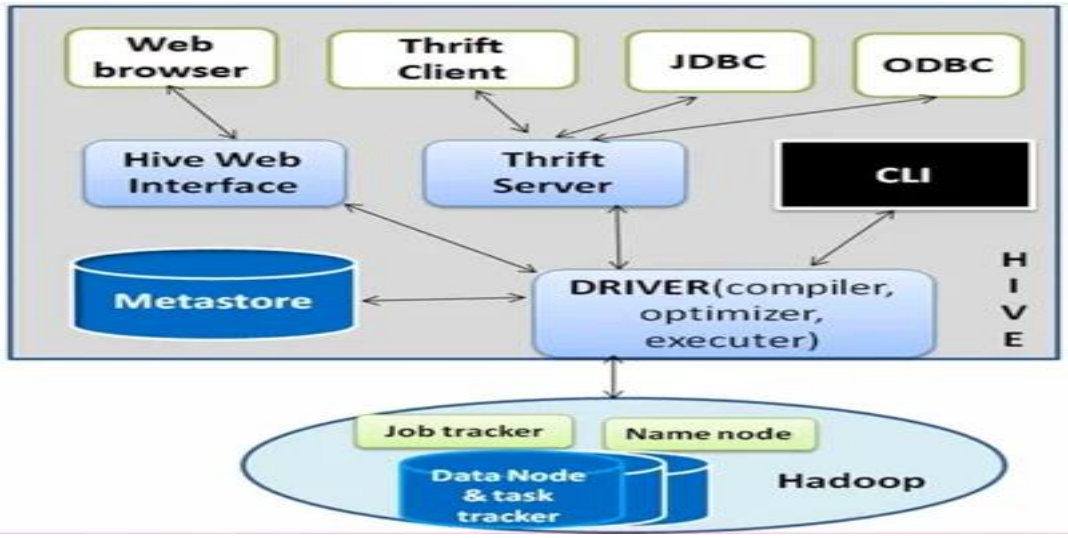

Figure 5

Hive consists of the following major components:

- **Hive Clients** – Apache Hive supports all application written in languages like C++, Java, Python etc. using JDBC, Thrift and ODBC drivers. Thus, one can easily write Hive client application written in a language of their choice.
- **Hive Services** – Hive provides various services like web Interface, CLI etc. to perform queries.
- **Processing framework and Resource Management** – Hive internally uses Hadoop Map Reduce framework to execute the queries.
- **Distributed Storage** – As seen above that Hive is built on the top of Hadoop, so it uses the underlying HDFS for the distributed storage.

Components of Hive:

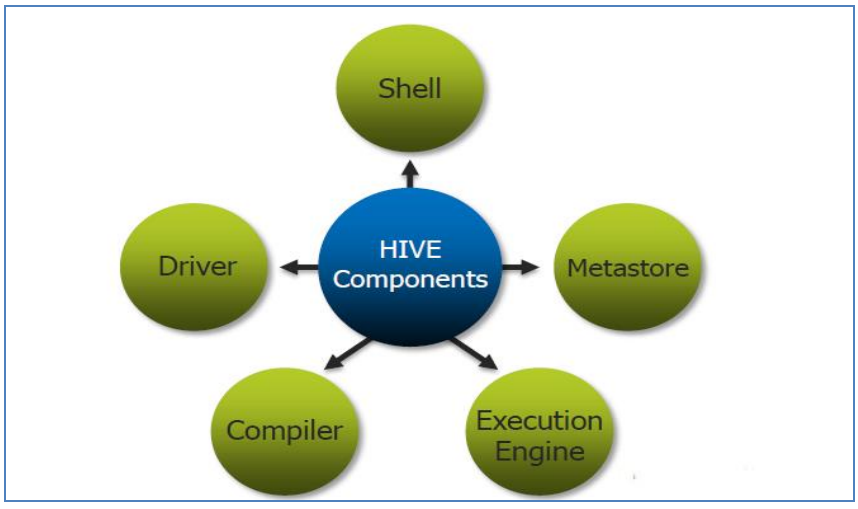

Figure5

Hive Clients

The Hive supports different types of client applications for performing queries. These clients are categorized into 3 types:

- **Thrift Clients –** As Apache Hive server is based on Thrift, so it can serve the request from all those languages that support Thrift.
- **JDBC Clients –** Apache Hive allows Java applications to connect to it using JDBC driver. It is defined in the class *apache.hadoop.hive.jdbc.HiveDriver.*
- **ODBC Clients –** ODBC Driver allows applications that support ODBC protocol to connect to Hive. For example JDBC driver, ODBC uses Thrift to communicate with the Hive server.

Hive Services

Apache Hive provides various services as shown in above diagram. Now, let us look at each in detail:

a) CLI (Command Line Interface) – This is the default shell that Hive provides, in which you can execute your Hive queries and command directly.

b) Web Interface – Hive also provides web based GUI for executing Hive queries and commands. See here different Hive Data types and operators.

c) Hive Server – It is built on Apache Thrift and thus is also called as Thrift server. It allows different clients to submit requests to Hive and retrieve the final result.

d) Hive Deriver – Driver is responsible for receiving the queries submitted Thrift, JDBC, ODBC, CLI, Web UL interface by a Hive client.

- **Complier** –After that hive driver passes the query to the compiler. Where parsing, type checking, and semantic analysis takes place with the help of schema present in the metastore.
- **Optimizer** – It generates the optimized logical plan in the form of a DAG (Directed Acyclic Graph) of MapReduce and HDFS tasks.
- **Executor** – Once compilation and optimization complete, execution engine executes these tasks in the order of their dependencies using Hadoop.

e) Metastore – Metastore is the central repository of Apache Hive metadata in the Hive Architecture. It stores metadata for Hive tables (like their schema and location) and partitions in a relational database. It provides client access to this information by using metastore service API. The Metastore stores the information about the tables, partitions, the columns within the tables. There are 3 ways of storing in Metastore: Embedded Metastore, Local Metastore and Remote Metastore. Mostly, Remote Metastore will be used in production mode.

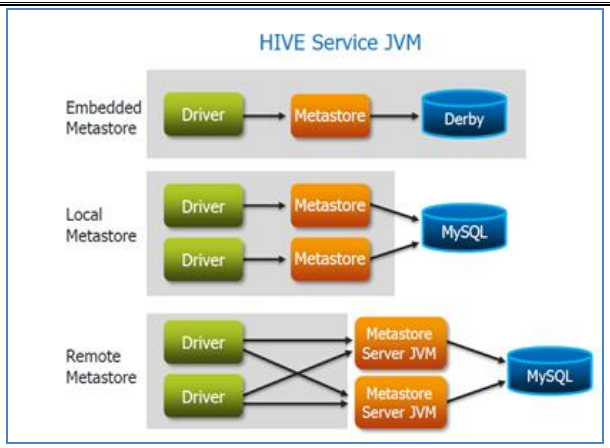

Figure 6

Limitations of Hive:

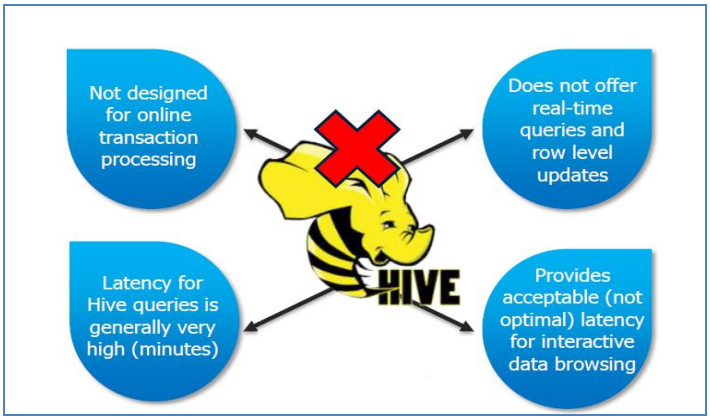

Figure 7

Hive has the following limitations and cannot be used under such circumstances:

- Not designed for online transaction processing.
- Provides acceptable latency for interactive data browsing.
- Does not offer real-time queries and row level updates.
- Latency for Hive queries is generally very high.

Working of Hive

The following Figure 9 depicts the workflow between Hive and Hadoop.

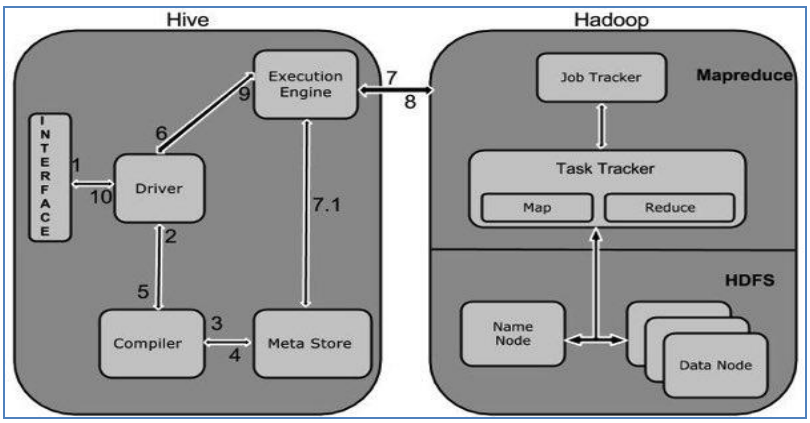

Figure 8

The following table 1 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.***

Table 1

Hive - Data Types

All the data types in Hive are classified into four types, given as follows:

- Column Types
- Literals

- Null Values
- 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 table2 depicts various INT data types:

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

Table 2

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 table3 depicts various CHAR data types:

Data Type	Length
VARCHAR	1 to 65355
CHAR	255

Table 3

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.fffffffff”.

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], ...>

Hive - 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**:

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

or

```
hive> CREATE SCHEMA userdb;
```

The following query is used to verify a databases list:

```
hive> SHOW DATABASES;
```

default

userdb

Hive - Drop Database

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**.

```
hive> DROP DATABASE IF EXISTS userdb;
```

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

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

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

```
hive> DROP SCHEMA userdb;
```

This clause was added in Hive 0.6.

Hive - Create Table

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 4 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

Table 4

The following data is a Comment, Row formatted fields such as Field terminator, Lines terminator, and Stored File type.

```
COMMENT ‘Employee details’
FIELDS TERMINATED BY ‘\t’
LINES TERMINATED BY ‘\n’
STORED IN TEXT FILE
```

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

```
hive> CREATE TABLE IF NOT EXISTS employee ( eid int, name String, salary String,
destination String)
COMMENT ‘Employee details’
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ‘\t’
LINES TERMINATED BY ‘\n’
STORED AS TEXTFILE;
```

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>
```

Load Data Statement

Generally, after creating a table in SQL, we can insert data using the Insert statement. But in Hive, we can insert data using the LOAD DATA statement.

While inserting data into Hive, it is better to use LOAD DATA to store bulk records. There are two ways to load data: one is from local file system and second is from Hadoop file system.

Syntax

The syntax for load data is as follows:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename
[PARTITION (partcol1=val1, partcol2=val2 ...)]
```

- LOCAL is identifier to specify the local path. It is optional.
- OVERWRITE is optional to overwrite the data in the table.
- PARTITION is optional.

Example

We will insert the following data into the table. It is a text file named **sample.txt** in**/home/user** directory.

1201 Gopal 45000 Technical manager
1202 Manisha 45000 Proof reader
1203 Masthanvali 40000 Technical writer
1204 Kiran 40000 Hr Admin
1205 Kranthi 30000 Op Admin

The following query loads the given text into the table.

```
hive> LOAD DATA LOCAL INPATH '/home/user/sample.txt'
OVERWRITE INTO TABLE employee;
```

On successful download, you get to see the following response:

```
OK
Time taken: 15.905 seconds
hive>
```

Hive - Alter Table

Alter table statement 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**.

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

Change Statement

The following table 5 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

Table 5

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

```
hive> ALTER TABLE employee CHANGE name ename String;
hive> ALTER TABLE employee CHANGE salary salary Double;
```

Add Columns Statement

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

```
hive> ALTER TABLE employee ADD COLUMNS (
dept STRING COMMENT 'Department name');
```

Replace Statement

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

```
hive> ALTER TABLE employee REPLACE COLUMNS ( eid INT empid Int, ename STRING
name String);
```

Hive - Drop Table

Drop Table Statement

The syntax is as follows:

DROP TABLE [IF EXISTS] table_name;

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

```
hive> DROP TABLE IF EXISTS employee;
```

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

```
OK
Time taken: 5.3 seconds
hive>
```

The following query is used to verify the list of tables:

```
hive> SHOW TABLES;
```

Hive - Partitioning

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

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

For example, a table named **Tab1** contains employee data such as id, name, dept, and yoj (i.e., year of joining). Suppose you need to retrieve the details of all employees who joined in 2012. A query searches the whole table for the required information. However, if you partition the

employee data with the year and store it in a separate file, it reduces the query processing time. The following example shows how to partition a file and its data:

The following file contains employee data table.

/tab1/employeedata/file1

id, name, dept, yoj

1, gopal, TP, 2012

2, kiran, HR, 2012

3, kaleel,SC, 2013

4, Prasanth, SC, 2013

The above data is partitioned into two files using year.

/tab1/employeedata/2012/file2

1, gopal, TP, 2012

2, kiran, HR, 2012

/tab1/employeedata/2013/file3

3, kaleel,SC, 2013

4, Prasanth, SC, 2013

Adding a Partition

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

Syntax:

ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec [LOCATION 'location1'] partition_spec [LOCATION 'location2'] ...;

partition_spec:: (p_column = p_col_value, p_column = p_col_value, ...)

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

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

Renaming a Partition

The syntax of this command is as follows.

ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION partition_spec;

The following query is used to rename a partition:

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

Dropping a Partition

The following syntax is used to drop a partition:

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

The following query is used to drop a partition:

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

Hive - Built-in Operators

There are four types of operators in Hive:

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

Relational Operators

These operators are used to compare two operands. The following table 6 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.

Table 6

Example

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

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

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

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

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

ID	Name	Salary	Designation	Dept
1205	Kranthi	30000	Op Admin	Admin

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

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

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

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

Arithmetic Operators

These operators support various common arithmetic operations on the operands. All of them return number types. The following table 7 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.

Table 7

Example

The following query adds two numbers, 20 and 30.

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

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

```
ADD
50
```

Logical Operators

The operators are logical expressions show in table 8. 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.

Table 8

Example

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

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

Complex Operators

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

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

Hive - Built-in Functions

Hive supports the following built-in functions:

Return Type	Signature	Description
BIGINT	round(double a)	It returns the rounded BIGINT value of the double.
BIGINT	floor(double a)	It returns the maximum BIGINT value that is equal or less than the double.
BIGINT	ceil(double a)	It returns the minimum BIGINT value that is equal or greater than the double.
double	rand(), rand(int seed)	It returns a random number that changes from row to row.
string	concat(string A, string B,...)	It returns the string resulting from concatenating B after A.
string	substr(string A, int start)	It returns the substring of A starting from start position till the end of string A.
string	substr(string A, int start, int length)	It returns the substring of A starting from start position with the given length.
string	upper(string A)	It returns the string resulting from converting all characters of A to upper case.
string	ucase(string A)	Same as above.
string	lower(string A)	It returns the string resulting from converting all characters of B to lower case.
string	lcase(string A)	Same as above.
string	trim(string A)	It returns the string resulting from trimming spaces from both ends of A.

string	ltrim(string A)	It returns the string resulting from trimming spaces from the beginning (left hand side) of A.
string	rtrim(string A)	rtrim(string A) It returns the string resulting from trimming spaces from the end (right hand side) of A.
string	regexp_replace(string A, string B, string C)	It returns the string resulting from replacing all substrings in B that match the Java regular expression syntax with C.
int	size(Map<K.V>)	It returns the number of elements in the map type.
int	size(Array<T>)	It returns the number of elements in the array type.
value of <type>	cast(<expr> as <type>)	It converts the results of the expression expr to <type> e.g. cast('1' as BIGINT) converts the string '1' to it integral representation. A NULL is returned if the conversion does not succeed.
string	from_unixtime(int unixtime)	convert the number of seconds from Unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00"
string	to_date(string timestamp)	It returns the date part of a timestamp string: to_date("1970-01-01 00:00:00") = "1970-01-01"
int	year(string date)	It returns the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970
int	month(string date)	It returns the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11
int	day(string date)	It returns the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1
string	get_json_object(string json_string, string path)	It extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It returns NULL if the input json string is invalid.

Example

The following queries demonstrate some built-in functions:

round() function

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

floor() function

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

ceil() function

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

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.

Hive - View and Indexes

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.**

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

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

Dropping a View

Use the following syntax to drop a view:

DROP VIEW view_name

The following query drops a view named as emp_30000:

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

Creating an Index

An Index is nothing but a pointer on a particular column of a table. Creating an index means creating a pointer on a particular column of a table. Its syntax is as follows:

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

Example

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

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.

The following query creates an index:

```
hive> CREATE INDEX inedx_salary ON TABLE employee(salary)
```

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:

```
hive> DROP INDEX index_salary ON employee;
```

HiveQL Querying and Analyzing

The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore. 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.

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

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

HiveQL - Select-Order By

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.

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

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

HiveQL - Select-Group By

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.

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

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

HiveQL - Select-Joins

JOIN 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 JOIN.

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:

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

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:

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

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.

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

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:

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