Hive Introduction

Apache Hive is a data warehouse system built on top of Hadoop and is used for analyzing structured and semi-structured data. Hive abstracts the complexity of Hadoop MapReduce. Basically, it provides a mechanism to project structure onto the data and perform queries written in HQL (Hive Query Language) that are similar to SQL statements. Internally, these queries or HQL gets converted to map reduce jobs by the Hive compiler. Therefore, you don't need to worry about writing complex MapReduce programs to process your data using Hadoop. It is targeted towards users who are comfortable with SQL. Apache Hive supports Data Definition Language (DDL), Data Manipulation Language (DML) and User Defined Functions (UDF).

Features of Apache Hive

There are so many features of Apache Hive. Let's discuss them one by one-

- Hive provides data summarization, query, and analysis in much easier manner.
- Hive supports external tables which make it possible to process data without actually storing in HDFS.
- Apache Hive fits the low-level interface requirement of Hadoop perfectly.
- It also supports partitioning of data at the level of tables to improve performance.
- Hive has a rule based optimizer for optimizing logical plans.
- It is scalable, familiar, and extensible.

- Using HiveQL doesn't require any knowledge of programming language, Knowledge of basic SQL query is enough.
- We can easily process structured data in Hadoop using Hive.
- Querying in Hive is very simple as it is similar to SQL.
- We can also run Ad-hoc queries for the data analysis using Hive.

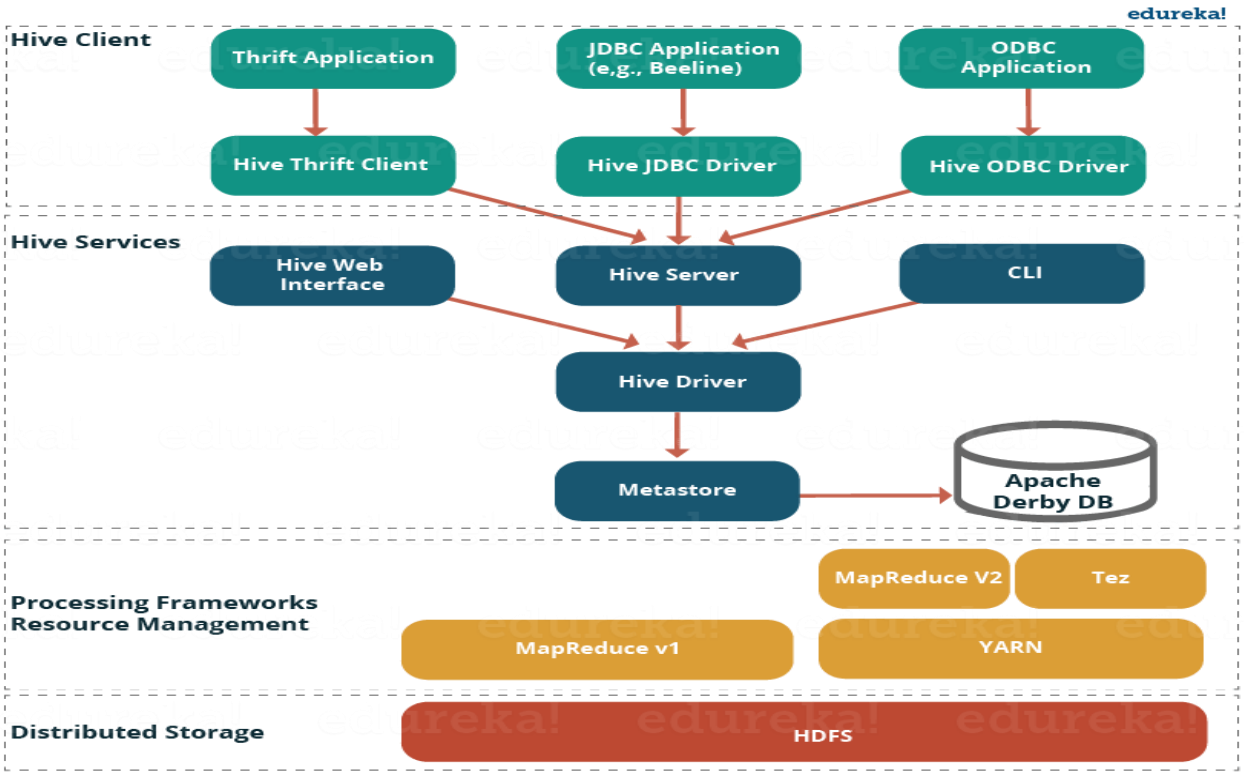
Limitation of Apache Hive

Hive has the following limitations-

- Apache does not offer real-time queries and row level updates.
- Hive also provides acceptable latency for interactive data browsing.
- It is not good for online transaction processing.
- Latency for Apache Hive queries is generally very high.

Hive Architecture

1. **Hive Clients:** Hive supports application written in many languages like Java, C++, Python etc. using JDBC, Thrift and ODBC drivers. Hence one can always write hive client application written in a language of their choice.
2. **Hive Services:** Apache Hive provides various services like CLI, Web Interface etc. to perform queries. We will explore each one of them shortly in this Hive tutorial blog.
3. **Processing framework and Resource Management:** Internally, Hive uses Hadoop MapReduce framework as de facto engine to execute the queries.
4. **Distributed Storage:** As Hive is installed on top of Hadoop, it uses the underlying HDFS for the distributed storage.



1. Hive Clients

Apache Hive supports different types of client applications for performing queries on the Hive. These clients can be categorized into three types:

- *Thrift Clients:* As Hive server is based on Apache Thrift, it can serve the request from all those programming language that supports Thrift.
- *JDBC Clients:* Hive allows Java applications to connect to it using the JDBC driver which is defined in the class org.apache.hadoop.hive.jdbc.HiveDriver.
- *ODBC Clients:* The Hive ODBC Driver allows applications that support the ODBC protocol to connect to Hive. (Like the JDBC driver, the ODBC driver uses Thrift to communicate with the Hive server.)

2. Hive Services:

Hive provides many services as shown in the image above. Let us have a look at each of them:

- Hive CLI (Command Line Interface): This is the default shell provided by the Hive where you can execute your Hive queries and commands directly.

- Apache Hive Web Interfaces: Apart from the command line interface, Hive also provides a web based GUI for executing Hive queries and commands.
- Hive Server: Hive server is built on Apache Thrift and therefore, is also referred as Thrift Server that allows different clients to submit requests to Hive and retrieve the final result.
- Apache Hive Driver: It is responsible for receiving the queries submitted through the CLI, the web UI, Thrift, ODBC or JDBC interfaces by a client. Then, the 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. In the next step, an optimized logical plan is generated in the form of a DAG (Directed Acyclic Graph) of map-reduce tasks and HDFS tasks. Finally, the execution engine executes these tasks in the order of their dependencies, using Hadoop.
- Metastore: You can think metastore as a central repository for storing all the Hive metadata information. Hive metadata includes various types of information like structure of tables and the partitions along with the column, column type, serializer and deserializer which is required for Read/Write operation on the data present in HDFS. The metastore comprises of two fundamental units:
    - A service that provides metastore access to other Hive services.
    - Disk storage for the metadata which is separate from HDFS storage.

Hive Data Model

In the order of granularity - Hive data is organized into:

- Databases: Namespaces function to avoid naming conflicts for tables, views, partitions, columns, and so on. Databases can also be used to enforce security for a user or group of users.
- Tables: Homogeneous units of data which have the same schema. An example of a table could be page_views table, where each row could comprise of the following columns (schema):
    - timestamp—which is of INT type that corresponds to a UNIX timestamp of when the page was viewed.
    - userid —which is of BIGINT type that identifies the user who viewed the page.
    - page_url—which is of STRING type that captures the location of the page.
    - referer_url—which is of STRING that captures the location of the page from where the user arrived at the current page.
    - IP—which is of STRING type that captures the IP address from where the page request was made.
- Partitions: Each Table can have one or more partition Keys which determines how the data is stored. Partitions—apart from being storage units—also allow the user to efficiently identify the rows that satisfy a specified criteria; for example, a date_partition of type STRING and country_partition of type STRING. Each unique value of the partition

keys defines a partition of the Table. For example, all "US" data from "2009-12-23" is a partition of the page_views table. Therefore, if you run analysis on only the "US" data for 2009-12-23, you can run that query only on the relevant partition of the table, thereby speeding up the analysis significantly. Note however, that just because a partition is named 2009-12-23 does not mean that it contains all or only data from that date; partitions are named after dates for convenience; it is the user's job to guarantee the relationship between partition name and data content! Partition columns are virtual columns, they are not part of the data itself but are derived on load.

- **Buckets** (or **Clusters**): Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table. For example the page_views table may be bucketed by userid, which is one of the columns, other than the partitions columns, of the page_view table. These can be used to efficiently sample the data.

Hive Types

1. Primitive Types

Integers

- TINYINT—1 byte integer
- SMALLINT—2 byte integer
- INT—4 byte integer
- BIGINT—8 byte integer

Boolean

- BOOLEAN—TRUE/FALSE

Floating point numbers

- FLOAT—single precision
- DOUBLE—Double precision

Fixed point numbers

- DECIMAL—a fixed point value of user defined scale and precision

String types

- STRING—sequence of characters in a specified character set
- VARCHAR—sequence of characters in a specified character set with a maximum length

- CHAR—sequence of characters in a specified character set with a defined length

Date and time types

- TIMESTAMP — A date and time without a timezone ("LocalDateTime" semantics)
- TIMESTAMP WITH LOCAL TIME ZONE — A point in time measured down to nanoseconds ("Instant" semantics)
- DATE—a date

Binary types

- BINARY—a sequence of bytes

2. Complex Types

Complex Types can be built up from primitive types and other composite types using:

- Structs: the elements within the type can be accessed using the DOT (.) notation. For example, for a column c of type STRUCT {a INT; b INT}, the a field is accessed by the expression c.a.
- Maps (key-value tuples): The elements are accessed using ['element name'] notation. For example in a map M comprising of a mapping from 'group' -> gid the gid value can be accessed using M['group']
- Arrays (indexable lists): The elements in the array have to be in the same type. Elements can be accessed using the [n] notation where n is an index (zero-based) into the array. For example, for an array A having the elements ['a', 'b', 'c'], A[1] retruns 'b'.

Using the primitive types and the constructs for creating complex types, types with arbitrary levels of nesting can be created. For example, a type User may comprise of the following fields:

- gender—which is a STRING.
- active—which is a BOOLEAN.

Hive Operators

1. Relational Operators

The following operators compare the passed operands and generate a TRUE or FALSE value, depending on whether the comparison between the operands holds or not.

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 A matches the SQL simple regular expression B, otherwise FALSE. The comparison is done character by character. The _ character in B matches any character in A (similar to **.** in posix regular expressions), and the % character in B matches an arbitrary number of characters in A (similar to **.*** in posix regular expressions). For example, 'foobar' LIKE 'foo' evaluates to FALSE where as 'foobar' LIKE 'foo___' evaluates to TRUE and so does 'foobar' LIKE 'foo%'. To escape % use \ (% matches one % character). If the data contains a semicolon, and you want to search for it, it needs to be escaped, columnValue LIKE 'a\;b'
A RLIKE B	strings	NULL if A or B is NULL, TRUE if any (possibly empty) substring of A matches the Java regular expression B (see Java regular expressions syntax), otherwise FALSE. For example, 'foobar' rlike 'foo' evaluates to TRUE and so does 'foobar' rlike '^f.*r$'.
A REGEXP B	strings	Same as RLIKE

2. Arithmetic Operators

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

-

A + B	all number types	Gives the result of adding A and B. The type of the result is the same as the common parent(in the type hierarchy) of

•

		the types of the operands, for example, since every integer is a float. Therefore, float is a containing type of integer so the + operator on a float and an int will result in a float.
A - B	all number types	Gives the result of subtracting B from A. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A * B	all number types	Gives the result of multiplying A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. Note that if the multiplication causing overflow, you will have to cast one of the operators to a type higher in the type hierarchy.
A / B	all number types	Gives the result of dividing B from A. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. If the operands are integer types, then the result is the quotient of the division.
A % B	all number types	Gives the reminder resulting from dividing A by B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A & B	all number types	Gives the result of bitwise AND of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A | B	all number types	Gives the result of bitwise OR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A ^ B	all number types	Gives the result of bitwise XOR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
~A	all number types	Gives the result of bitwise NOT of A. The type of the result is the same as the type of A.

3. Logical Operators

The following operators provide support for creating logical expressions. All of them return boolean TRUE or FALSE depending upon the boolean values of the operands.

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

4. Operators on Complex Types

The following operators provide mechanisms to access elements in Complex Types

A[n]	A is an Array and n is an int	returns the nth element in the array A. The first element has index 0, for example, if A is an array comprising of ['foo', 'bar'] then A[0] returns 'foo' and A[1] returns 'bar'
M[key]	M is a Map<K, V> and key has type K	returns the value corresponding to the key in the map for example, if M is a map comprising of {'f' -> 'foo', 'b' -> 'bar', 'all' -> 'foobar'} then M['all'] returns 'foobar'
S.x	S is a struct	returns the x field of S, for example, for struct foobar {int foo, int bar} foobar.foo returns the integer stored in the foo field of the struct.

Hive Functions

BIGINT	round(double a)	returns the rounded BIGINT value of the double
BIGINT	floor(double a)	returns the maximum BIGINT value that is equal or less than the double
BIGINT	ceil(double a)	returns the minimum BIGINT value that is equal or greater than the double
double	rand(), rand(int seed)	returns a random number (that changes from row to row). Specifiying the seed will make sure the generated random number sequence is deterministic.
string	concat(string A, string B,...)	returns the string resulting from concatenating B after A. For example, concat('foo', 'bar') results in 'foobar'. This function accepts arbitrary number of arguments and return the concatenation of all of them.

string	substr(string A, int start)	returns the substring of A starting from start position till the end of string A. For example, substr('foobar', 4) results in 'bar'
string	substr(string A, int start, int length)	returns the substring of A starting from start position with the given length, for example, substr('foobar', 4, 2) results in 'ba'
string	upper(string A)	returns the string resulting from converting all characters of A to upper case, for example, upper('fOoBaR') results in 'FOOBAR'
string	ucase(string A)	Same as upper
string	lower(string A)	returns the string resulting from converting all characters of B to lower case, for example, lower('fOoBaR') results in 'foobar'
string	lcase(string A)	Same as lower
string	trim(string A)	returns the string resulting from trimming spaces from both ends of A, for example, trim(' foobar ') results in 'foobar'
string	ltrim(string A)	returns the string resulting from trimming spaces from the beginning(left hand side) of A. For example, ltrim(' foobar ') results in 'foobar '
string	rtrim(string A)	returns the string resulting from trimming spaces from the end(right hand side) of A. For example, rtrim(' foobar ') results in ' foobar'
string	regexp_replace(string A, string B, string C)	returns the string resulting from replacing all substrings in B that match the Java regular expression syntax(See Java regular expressions syntax) with C. For example, regexp_replace('foobar', 'oo|ar',) returns 'fb'
int	size(Map<K.V>)	returns the number of elements in the map type
int	size(Array<T>)	returns the number of elements in the array type
value of <type>	cast(*<expr>* as *<type>*)	converts the results of the expression expr to <type>, for example, cast('1' as BIGINT) will convert 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 the 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)	Return the date part of a timestamp string: to_date("1970-01-01 00:00:00") = "1970-01-01"

int	year(string date)	Return 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)	Return 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)	Return 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)	Extract json object from a json string based on json path specified, and return json string of the extracted json object. It will return null if the input json string is invalid.

- The following built in aggregate functions are supported in Hive:

BIGINT	count(*), count(expr), count(DISTINCT expr[, expr_.])	count(*)—Returns the total number of retrieved rows, including rows containing NULL values; count(expr)—Returns the number of rows for which the supplied expression is non-NULL; count(DISTINCT expr[, expr])—Returns the number of rows for which the supplied expression(s) are unique and non-NULL.
DOUBLE	sum(col), sum(DISTINCT col)	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)	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)	returns the minimum value of the column in the group
DOUBLE	max(col)	returns the maximum value of the column in the group

HIVE Commands

1. DDL Commands on Tables in Hive

1. CREATE
2. SHOW
3. DESCRIBE
4. USE

5. DROP
6. ALTER
7. TRUNCATE

1. CREATE DATABASE in Hive

The **CREATE DATABASE** statement is used to create a database in the Hive. The DATABASE and SCHEMA are interchangeable. We can use either DATABASE or SCHEMA.

Syntax:

```
1. CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
2. [COMMENT database_comment]
3. [LOCATION hdfs_path]
4. [WITH DBPROPERTIES (property_name=property_value, ...)];
```

2. SHOW DATABASE in Hive

The **SHOW DATABASES** statement lists all the databases present in the Hive.

Syntax:

```
1. SHOW (DATABASES|SCHEMAS);
```

3. DESCRIBE DATABASE in Hive

The **DESCRIBE DATABASE** statement in Hive shows the name of Database in Hive, its comment (if set), and its location on the file system.

The **EXTENDED** can be used to get the database properties.

Syntax:

```
1. DESCRIBE DATABASE/SCHEMA [EXTENDED] db_name;
2.
```

4. USE DATABASE in Hive

The **USE** statement in Hive is used to select the specific database for a session on which all subsequent HiveQL statements would be executed.

Syntax:

1. USE database_name;

5. DROP DATABASE in Hive

The **DROP DATABASE** statement in Hive is used to Drop (delete) the database.

The default behavior is RESTRICT which means that the database is dropped only when it is empty. To drop the database with tables, we can use CASCADE.

Syntax:

1. DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT|CASCADE];

DDL Commands on Tables in Hive

1. CREATE TABLE

The **CREATE TABLE** statement in Hive is used to create a table with the given name. If a table or view already exists with the same name, then the error is thrown. We can use **IF NOT EXISTS** to skip the error.

Syntax:

1. CREATE TABLE [IF NOT EXISTS] [db_name.] table_name [(col_name data_type [COMMENT col_comment], ... [COMMENT col_comment])] [COMMENT table_comment] [ROW FORMAT row_format] [STORED AS file_format] [LOCATION hdfs_path];

2. SHOW TABLES in Hive

The **SHOW TABLES** statement in Hive lists all the base tables and views in the current database.

Syntax:

1. SHOW TABLES [IN database_name];

3. DESCRIBE TABLE in Hive

The **DESCRIBE** statement in Hive shows the lists of columns for the specified table.

Syntax:

1. DESCRIBE [EXTENDED|FORMATTED] [db_name.] table_name[.col_name ([.field_name])];

4. DROP TABLE in Hive

The **DROP TABLE** statement in Hive deletes the data for a particular table and remove all metadata associated with it from Hive metastore.

If **PURGE** is not specified then the data is actually moved to the .Trash/current directory. If **PURGE** is specified, then data is lost completely.

Syntax:

1. DROP TABLE [IF EXISTS] table_name [PURGE];

5. ALTER TABLE in Hive

The **ALTER TABLE** statement in Hive enables you to change the structure of an existing table. Using the ALTER TABLE statement we can rename the table, add columns to the table, change the table properties, etc.

Syntax to Rename a table:

1. ALTER TABLE table_name RENAME TO new_table_name;

Syntax to Add columns to a table:

1. ALTER TABLE table_name ADD COLUMNS (column1, column2) ;

6. TRUNCATE TABLE

TRUNCATE TABLE statement in Hive removes all the rows from the table or partition.

Syntax:

1. TRUNCATE TABLE table_name;

Hive DML commands

Hive DML (Data Manipulation Language) commands are used to insert, update, retrieve, and delete data from the Hive table once the table and database schema has been defined using Hive DDL commands.

The various Hive DML commands are:

1. LOAD
2. SELECT
3. INSERT
4. DELETE
5. UPDATE
6. EXPORT
7. IMPORT

1. LOAD Command

The **LOAD** statement in Hive is used to move data files into the locations corresponding to Hive tables.

- If a **LOCAL** keyword is specified, then the LOAD command will look for the file path in the local filesystem.
- If the LOCAL keyword is not specified, then the Hive will need the absolute URI of the file.
- In case the keyword OVERWRITE is specified, then the contents of the target table/partition will be deleted and replaced by the files referred by filepath.
- If the OVERWRITE keyword is not specified, then the files referred by filepath will be appended to the table.

Syntax:

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

2. SELECT COMMAND

The SELECT statement in Hive is similar to the SELECT statement in SQL used for retrieving data from the database.

Syntax:

1. SELECT col1,col2 FROM tablename;

3. INSERT Command

The **INSERT** command in Hive loads the data into a Hive table. We can do insert to both the Hive table or partition.

a. INSERT INTO

The **INSERT INTO** statement appends the data into existing data in the table or partition. INSERT INTO statement works from Hive version 0.8.

Syntax:

1. INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1 FROM from_statement;

b. INSERT OVERWRITE

The **INSERT OVERWRITE** table overwrites the existing data in the table or partition.

Syntax:

1. INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, ..) [IF NOT EXISTS]] select_statement FROM from_statement;

c. INSERT .. VALUES

INSERT ..VALUES statement in Hive inserts data into the table directly from SQL. It is available from Hive 0.14.

Syntax:

1. INSERT INTO TABLE tablename [PARTITION (partcol1[=val1], partcol2[=val2] ...)] VALUES values_row [, values_row ...];

4. DELETE command

The DELETE statement in Hive deletes the table data. If the WHERE clause is specified, then it deletes the rows that satisfy the condition in where clause.

The DELETE statement can only be used on the hive tables that support ACID.

Syntax:

1. DELETE FROM tablename [WHERE expression];

5. UPDATE Command

The update can be performed on the hive tables that support ACID.

The UPDATE statement in Hive deletes the table data. If the WHERE clause is specified, then it updates the column of the rows that satisfy the condition in WHERE clause.

Partitioning and Bucketing columns cannot be updated.

Syntax:

1. UPDATE tablename SET column = value [, column = value ...] [WHERE expression];

6. EXPORT Command

The Hive **EXPORT** statement exports the table or partition data along with the metadata to the specified output location in the HDFS.

Metadata is exported in a **_metadata** file, and data is exported in a subdirectory **'data.'**

Syntax:

1. EXPORT TABLE tablename [PARTITION (part_column="value"[, ...])]
2. TO 'export_target_path' [FOR replication('eventid')];

7. IMPORT Command

The Hive IMPORT command imports the data from a specified location to a new table or already existing table.

Syntax:

1. IMPORT [[EXTERNAL] TABLE new_or_original_tablename [PARTITION (part_column="value"[, ...])]]

2. FROM 'source_path' [LOCATION 'import_target_path'];