Table of contents

Hive Query Language

What is Hive

Hive is a data warehousing infrastructure based on Hadoop. Hadoop provides massive scale out and fault tolerance capabilities for data storage and processing on commodity hardware.

Hive is designed to enable easy data summarization, ad-hoc querying and analysis of large volumes of data. It provides a simple query language called HiveQL, which is based on SQL.

Hive file formats

Hive supports several file formats:

- Text file (`TEXTFILE`),
- SequenceFile (`SEQUENCEFILE`),

- RCFile (`RCFile`),
- Avro files (`AVRO`),
- ORC files (`ORC`),
- Parquet (`PARQUET`),
- Custom INPUTFORMAT and OUTPUTFORMAT.

Text file

Text file is the parameter's default value.

- Convenient format to use to exchange with other applications or scripts that produce or read delimited files.
- Human readable and parsable.
- Data stores is bulky and not as efficient to query.
- Do not support block compression.

SequenceFile

SequenceFiles are flat files consisting of binary key/value pairs. SequenceFile is basic file format which provided by Hadoop, and Hive also provides it to create a table.

- Provides a persistent data structure.
- Row based.
- Commonly used to transfer data between Map Reduce jobs.
- Can be used as an archive to pack small files in Hadoops.
- Support splitting event when the data is compressed.

RCFile

RCFile (Record Columnar File) is a data placement structure designed for MapReduce-based data warehouse systems.

RCFile stores table data in a flat file consisting of binary key/value pairs. It first partitions rows horizontally into row splits, and then it vertically partitions each row split in a columnar way. RCFile stores the metadata of a row split as the key part of a record, and all the data of a row split as the value part.

RCFile combines the advantages of both row-store and column-store to satisfy the need for fast data loading and query processing, efficient use of storage space, and adaptability to highly dynamic workload patterns.

- As row-store, RCFile guarantees that data in the same row are located in the same node.
- As column-store, RCFile can exploit column-wise data compression and skip unnecessary column reads.

Example:

C1	C2	C3	C4
11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44
51	52	53	54

To serialize the table, RCFile partitions this table first horizontally and then vertically, instead of only partitioning the table horizontally like the row-oriented DBMS (row-store). The horizontal partitioning will first partition the table into multiple row groups based on the row-group size, which is a user-specified value determining the size of each row group.

Row Group 1

C1	C2	C3	C4
11	12	13	14
21	22	23	24
31	32	33	34

Row Group 2

C1	C2	C3	C4
41	42	43	44
51	52	53	54

Then, in every row group, RCFile partitions the data vertically like column-store. Thus, the table will be serialized as:

Row Group 1

```
11, 21, 31;
12, 22, 32;
13, 23, 33;
14, 24, 34;
```

Row Group 2

```
41, 51;
42, 52;
43, 53;
44, 54;
```

Avro files

- Widely used as a serialization platform.
- Row-based, offers a compact and fast binary format.
- Schema is encoded on the file so the data can be untagged.
- Files support blocak compression and are splittable.
- Supports schema evoluation.

ORC files

The Optimized Row Columnar (ORC) file format provides a highly efficient way to store Hive data. It was designed to overcome limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data. Compared with RCFile format.

- Considered the evolution of the RCFile.
- Stores collections of rows and within the collection the row data is stored in columnar format.
- Inroduces a lightweight indexing that enabes skipping of irrelevant blocks of rows.

- Splittable: allows parallel processing of row collection.
- It comes with basic statistics on columns (min, max, sum, and count).

Parquet

Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

- Uses the record shredding and assembly algorithm described in the Dremel paper.
- Each data file contains the values for a set of rows.
- Efficient in terms of disk I/O when specific colums need to be queried.

Hive data types

All the data types in Hive are classified into four types:

- column types,
- literals,
- null values,
- complex types.

Column types

Integral types:

- `TINYINT` -- 25Y,
- `SMALLINT` -- 25S,
- `INT` -- 25,
- `BIGINT` -- 25L.

String types:

- `VARCHAR` -- length: 1 to 65355,
- `CHAR` -- length: 255.

Timestamp.

It supports traditional UNIX time stamp with optional nanosecond precision. Format: "yyyy-mm-dd hh:mm:ss.ffffffffff".

Dates.

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

Decimals:

```
DECIMAL(precision, scale)
```

Union types:

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

Literals

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 special value `NULL`.

Complex types

Arrays:

```
ARRAY<data_type>
```

Maps:

```
MAP<primitive_type, data_type>
```

Structs:

```
STRUCT<col_name : data_type [COMMENTS col_comment], ...>
```

Hive configuration properties

Set configuration property

```
SET property_var=property_value
```

Example:

```
SET hive.mapred.supports.subdirectories=TRUE
```

Where:

- **hive.mapred.supports.subdirectories** -- support sub-directories for tables/partitions,
- **TRUE** -- property value.

Hive data

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

- databases,
- tables,
- partitions,
- buckets (clusters).

Hive databases

Browse databases

To list existing databases in the warehouse:

```
SHOW DATABASES
```

Use database

```
USE database_name
```

Example:

```
USE test_database
```

Where:

- **test_database** -- database name.

Create databases

```
CREATE DATABASE|SHEMA [IF NOT EXISTS] database_name
```

Example:

```
CREATE DATABASE IF NOT EXISTS test_database
```

Where:

- **test_database** -- database name.

Hive tables

Tables. Homogeneous units of data which have the same schema. An example of a table could be **test_table**, where each row could comprise of the following columns (schema):

- **col_1** -- INT type column,
- **col_2** -- BIGINT type column,
- **col_3** -- STRING type column.

> ***Note***
>
> Not necessary for tables to be partitioned or bucketed, but these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution.

Browse tables

To list existing tables in the warehouse:

```
SHOW TABLES
```

To list columns and column types of table:

```
DESCRIBE [EXTENDED] table_reference
```

Example:

```
DESCRIBE test_table
```

Where:

- **test_table** -- table name.

Create tables

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[ROW FORMAT row_format]
[STORED AS file_format]
[LOCATION hdfs_path]
```

Example 1:

```
CREATE TABLE IF NOT EXISTS test_table (col_1 INT, col_2 BIGINT,
        col_3 STRING)
ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY "\n"
```

Where:

- **test_table** -- table name,
- **col_1** and **col_2** -- column names,
- **col_3** -- partition column name,
- **INT**, **BIGINT** and **STRING** -- data types,
- **,** -- fields terminator,
- **\n** -- lines terminator.

Example 2:

```
CREATE TABLE IF NOT EXISTS test_table (col_1 INT, col_2 BIGINT)
PARTITIONED BY (col_3 STRING)
ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY "\n"
```

Where:

- **test_table** -- table name,
- **col_1**, **col_2** and **col_3** -- column names,

- **INT**, **BIGINT** and **STRING** -- data types,
- **,** -- fields terminator,
- **\n** -- lines terminator.

Temporary tables.

A table that has been created as a temporary table will only be visible to the current session. Data will be stored in the user's scratch directory, and deleted at the end of the session.

Temporary tables have the following limitations:

- Partition columns are not supported.
- No support for creation of indexes.

External tables.

The `EXTERNAL` keyword lets you create a table and provide a `LOCATION` so that Hive does not use a default location for this table. This comes in handy if you already have data generated. When dropping an EXTERNAL table, data in the table is NOT deleted from the file system.

Alter tables

To rename existing table to a new name:

```
ALTER TABLE old_table_name RENAME TO new_table_name
```

To add the columns to an existing table:

```
ALTER TABLE table_name ADD COLUMNS (col_name data_type, ...)
```

To change the column of an existing table:

```
ALTER TABLE table_name CHANGE col_name new_col_name new_data_type
```

To rename the columns of an existing table:

```
ALTER TABLE table_name
REPLACE COLUMNS (
        old_col_name old_data_type new_col_name new_data_type, ...)
```

Add SerDe Properties:

```
ALTER TABLE table_name SET SERDEPROPERTIES serde_properties;
```

Example 1:

```
ALTER TABLE test_table SET SERDEPROPERTIES (
        "serialization.null.format"="null")
```

Where:

- **test_table** -- table name,
- **serialization.null.format** -- specify custom string for `NULL` values,
- **null** -- custom string for `NULL` values.

Example 2:

```
ALTER TABLE test_table SET SERDEPROPERTIES (
        "skip.header.line.count"='1')
```

Where:

- **test_table** -- table name,
- **skip.header.line.count** -- skip header rows,
- **1** -- number of header lines for the table file.

To delete the column of an existing table:

```
ALTER TABLE table_name DROP [COLUMN] col_name
```

Drop tables

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

Delete rows

For data sets which are not too huge, we can do something like:

```
INSERT OVERWRITE TABLE dst_table
SELECT * FROM src_table
WHERE col_name!="value"
```

or:

```
INSERT OVERWRITE TABLE dst_table
SELECT * FROM src_table
WHERE col_name NOT IN ("value_1", "value_2", ...)
```

> ***Note***
>
> In Big Data there really aren't deletes.

Hive 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 certain criteria. Partition columns are virtual columns, they are not part of the data itself but are derived on load.

Browse partitions

```
SHOW PARTITIONS table_name
```

Add partitions

```
ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec
[LOCATION "location"]
```

Example:

```
ALTER TABLE test_table
ADD IF NOT EXISTS PARTITION (col_3="2016")
LOCATION "/2016/part2016"
```

Where:

- **test_table** -- table name,
- **col_3="2016"** -- partition name (key),
- **/2016/part2016** -- location.

Rename partitions

```
ALTER TABLE table_name PARTITION old_partition_spec
RENAME TO PARTITION new_partition_spec
```

Example:

```
ALTER TABLE test_table_name PARTITION (col_3="2016")
RENAME TO PARTITION (col_1=2016)
```

Where:

- **test_table** -- table name,
- **col_3="2016"** -- current partition name (key),
- **col_1=2016** -- new partition name (key).

Drop partitions

```
ALTER TABLE DROP [IF EXISTS] PARTITION (partition_spec)
```

Example:

```
ALTER TABLE test_table DROP IF EXISTS
PARTITION (col_3="2016")
```

Where:

- **test_table** -- table name,
- **col_3="2016"** -- partition name (key).

Load data

We can insert data using the `LOAD DATA` statement:

```
LOAD DATA [LOCAL] INPATH "file_path"
[OVERWRITE] INTO TABLE table_name
[PARTITION (partcol1=val1, ...)]
```

Example 1:

```
LOAD DATA LOCAL INPATH "/home/korniichuk/sample.csv"
OVERWRITE INTO TABLE test_table
```

Where:

- **/home/korniichuk/sample.csv** -- file abs path,
- `OVERWRITE` -- overwrite the data in the table (existing data in the table is deleted),
- **test_table** -- table name.

Example 2:

```
LOAD DATA INPATH "hdfs://korniichuk.com/tmp/sample.csv"
INTO TABLE test_table
```

Where:

- **hdfs://korniichuk.com/tmp/sample.csv** -- full HDFS URI,
- **test_table** -- table name.

Note

The loaded data files are moved, not copied, into directory.

Hive SELECT statement

`SELECT` syntax:

```
SELECT [* | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
LIMIT number
```

Example:

```
SELECT *
FROM test_table
LIMIT 3
```

Where:

- `*` -- all matching rows are returned (with duplicates),
- **test_table** -- table name,
- **3** -- number of rows to be returned.

SELECT ... DISTINCT clause

`DISTINCT` clause specifies removal of duplicate rows from the result set.

```
SELECT [DISTINCT] select_expr, select_expr, ...
FROM table_reference
```

Example:

```
SELECT DISTINCT col_2, col_3
FROM test_table
```

Where:

- **col_2**, **col_3** -- duplicate rows for this columns to be removed from the result set,
- **test_table** -- table name.

SELECT ... WHERE clause

The `WHERE` condition is a boolean expression.

```
SELECT select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
```

Example:

```
SELECT *
FROM test_table
WHERE col_1 > 10 AND col_3 = "UA"
```

Where:

- `*` -- all matching rows are returned (with duplicates),
- **test_table** -- table name,
- **col_1**, **col_3** -- column names.

SELECT ... ORDER BY clause

The `ORDER BY` clause is used to retrieve the details based on one column and sort the result set by ascending or descending order.

The `ORDER BY` guarantees global ordering, but does this by pushing all data through just one reducer.

```
SELECT select_expr, select_expr, ...
FROM table_reference
[ORDER BY col_name [ASC | DESC]]
```

Example:

```
SELECT *
FROM test_table
ORDER col_1 DESC
```

Where:

- `*` -- all matching rows are returned (with duplicates),
- **test_table** -- table name,
- **col_1** -- table column,
- `DESC` -- sort the result set by descending order.

> ***Note***
>
> This is basically unacceptable for large datasets. You end up one sorted file as output.

UNION operators

`UNION` syntax:

```
SELECT select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
UNION
SELECT select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
```

> ***Note***
>
> `UNION` removes duplicate rows.

`UNION ALL` syntax:

```
SELECT select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
UNION ALL
SELECT select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
```

> ***Note***
>
> `UNION ALL` does not remove duplicate rows.

Hive GROUP BY clause

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.

```
SELECT select_expr, select_expr, ...
FROM table_reference
[GROUP BY col_list]
```

Example:

```
SELECT col_1, count(*)
FROM test_table
GROUP BY col_1
```

Where:

- **col_1**, `count(*)` -- columnes in the result set,
- **test_table** -- table name.

Hive JOINs clauses

JOINs are a clauses that are used for combining specific fields from two tables by using values common to each one. They are used to combine records from two or more tables in the database. The are more or less similar to SQL JOINs.

```
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]
```

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.

Example:

```
SELECT t.col_1, t.col_2, t_2.col_2
FROM test_table t JOIN test_2_table t_2
ON (t.col_1 = t_2.col_1)
```

LEFT OUTER JOIN

The Hive `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 OUTER 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.

Example:

```
SELECT t.col_1, t.col_2, t_2.col_2
FROM test_table t
LEFT OUTER JOIN test_2_table t_2
ON (t.col_1 = t_2.col_1)
```

RIGHT OUTER JOIN

The Hive `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 OUTER 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.

Example:

```
SELECT t.col_1, t.col_2, t_2.col_2
FROM test_table t
RIGHT OUTER JOIN test_2_table t_2
ON (t.col_1 = t_2.col_1)
```

FULL OUTER JOIN

The Hive `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.

Example:

```
SELECT t.col_1, t.col_2, t_2.col_2
FROM test_table t
FULL OUTER JOIN test_2_table t_2
ON (t.col_1 = t_2.col_1)
```

Hive build-in operators

Relational operators

```
A = B | A == B
A <=> B
A <> B | A != B
A < B
A <= B
A > B
A >= B
A [NOT] BETWEEN B AND C
A IS NULL
A IS NOT NULL
A [NOT] LIKE B
A RLIKE B | A REGEXP B
```

Arithmetic operators

```
A + B
A - B
A * B
A / B
A % B
A & B # Result of bitwise AND of A and B
A | B # Result of bitwise OR of A and B
A ^ B # Result of bitwise XOR of A and B
~A # Result of bitwise NOT of A
```

Logical operators

```
A AND B | A && B
A OR B | A || B
NOT A | ! A
A IN (val1, val2, ...)
A NOT IN (val1, val2, ...)
[NOT] EXISTS (subquery)
```

Complex operators

```
A[n] # Returns the nth element in the arr A.
     # The 1st el has index 0
M[key] # Returns the value corresponding to the key in the map.
       # {'f'->"foo",'b'->"bar"} is represented as
       # map('f',"foo",'b',"bar"). map['f'] returns "foo"
S.x # Returns the x fiel of S. Operand types: S is a struct.
    # Struct is similar to STRUCT in C language.
    # For struct foobar {int foo, int bar} foobar.foo returns
    # the int stored in the foo field of the struct
```

Hive build-in functions

- `round(double a)` -- Returns the rounded BIGINT value of the double;
- `floor(double a)` -- Returns the maximum BIGINT value that is equal or less than the double;
- `ceil(double a)` -- Return the minimum BIGINT value that is equal or greater than double;
- `rand()`, `rand(int seed)` -- Returns a random number that changes from row to row;
- `concat(string A, string B, ...)` -- Returns the string resulting from concatenating B after A;
- `substr(string A, int start)` -- Returns the substring of A starting from start position till the end of string A;
- `substr(string A, int start, int length)` -- Returns the substring of A starting from start position with the given length;
- `upper(string A) | ucase(string A)` -- Returns the string resulting from converting all characters of A to upper case;
- `lower(string A) | lcase(string A)` -- Returns the string resulting from converting all characters of B to lower case;

- `trim(string A)` -- Returns the string resulting from trimming spaces from both ends of A;
- `ltrim(string A)` -- Returns the string resulting from trimming spaces from the beginning (left hand side) of A;
- `rtrim(string A)` -- Returns the string resulting from trimming paces from the end (right hand side) of A;
- `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 with C;
- `size(Map<K.V>)` -- Returns the number of elem ents in the map type;
- `size(Array<T>)` -- Returns the number of elements in the array type;
- `cast(<expr> as <type>)` -- 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.
- `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";
- `to_date(string timestamp)` -- Returns the date part of a timestamp string: to_date("1970-01-01 00:00:00") = "1970-01-01";
- `year(string date)` -- Returns the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970;
- `month(string date)` -- Returns the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11;
- `day(string date)` -- Returns the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1;
- `get_json_object(string json_string, string path)` -- 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.

Hive build-in aggregate functions

- `count(*)`, `count(expr)`, `count(DISTINCT expr[, ...])` -- Returns the total number of retrieved rows;
- `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;
- `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;
- `min(col)` -- Returns the minimum value of the column in the group;
- `max(col)` -- Returns the maximum value of the column in the group.

Hive build-in ranking functions

Returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition:

```
ROW_NUMBER() OVER ([partition_by_clause] order_by_clause)
```

Example:

```
ROW_NUMBER() (PARTITION BY col_1 ORDER BY col_2 DESC)
```

Hive own functions

Create functions

```
CREATE FUNCTION [db_name.]function_name AS "class_name"
```

Example:

```
CREATE FUNCTION sha256 AS "org.hue.udf.SHA256String"
```

Drop functions

```
DROP FUNCTION [IF EXISTS] [db_name.]function_name
```

Example:

```
DROP FUNCTION IF EXISTS sha256
```

Hive views

Views are generated based on user requirements. We can save any result set data as a view. The usage of view in Hive is same as that of the view in SQL. It is a standard RDBMS concept. We can execute all DML operations on a view.

Create views

```
CREATE VIEW [IF NOT EXISTS] view_name
[COMMENT table_comment]
AS SELECT ...
```

Example:

```
CREATE VIEW IF NOT EXISTS test_view
AS SELECT * FROM test_table
WHERE col_1>100
```

Where:

- **test_view** -- view name,
- **test_table** -- table name,
- **col_1** -- table column.

Drop views

```
DROP VIEW view_name
```

Example:

```
DROP VIEW test_view
```

Where:

- **test_view** -- view name.

Hive indexes

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.

Browse indexes

```
SHOW INDEX ON table_name
```

Create indexes

```
CREATE INDEX index_name
ON TABLE table_name (col_name, ...)
AS "index.handler.class.name"
```

Example:

```
CREATE INDEX test_index
ON TABLE test_table (col_1)
AS "COMPACT" WITH DEFERRED REBUILD
```

Where:

- **test_index** -- index name,
- **test_table** -- table name,
- **col_name** -- column name,
- `COMPACT` -- index handler class name.

Drop indexes

```
DROP INDEX index_name
ON table_name
```