# 22MCA2052 – Big Data Analytics
# Module – 5: Programming Hive

## Text Book: Jason R, Dean W, Edward C, "Programming Hive", O'reilly, 2012

## Programming Hive: Hive in the Hadoop Ecosystem, Data Types and File Formats, HiveQL: Data Definition, Databases in Hive, Alter Database, Creating Tables, External Tables, Partitioned Tables, External Partitioned Tables, Dropping Tables, Alter Tables, HiveQL: Data Manipulation, Queries (till GROUP BY Clauses).

---

### Creating Tables and Loading Them in One Query

We can also create a table and insert query results into it in one statement:

```
create table user_data
(
sno int,
usr_name string,
city string)
ROW FORMAT delimited fields terminated by ' , ' LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

```
CREATE TABLE user_data_new
AS SELECT usr_name, city
FROM user_data
```

This table contains just the name and city columns from the user_data table records. The schema for the new table is taken from the SELECT clause.

## HiveQL: Queries

### SELECT … FROM Clauses

SELECT is the *projection operator* in SQL. The FROM clause identifies from which table, view we select records.

```
SELECT * FROM employees;
```

In this case, Hive can simply read the records from employees and dump the formatted output to the console.

For a given record, SELECT specifies the columns to keep, as well as the outputs of function calls on one or more columns.

Consider the following table:

```
CREATE TABLE employees (
  name         STRING,
  salary       FLOAT,
  subordinates ARRAY<STRING>,
  deductions   MAP<STRING, FLOAT>,
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
PARTITIONED BY (country STRING, state STRING);
```

Assume 4 rows of data is inserted into this table.

Here are queries of this table and the output they produce:

```
hive> SELECT name, salary FROM employees;
        John Doe        100000.0
        Mary Smith      80000.0
        Todd Jones      70000.0
        Bill King       60000.0
```

- SELECT scans the table specified by the FROM clause
- WHERE gives the condition of what to filter
- GROUP BY gives a list of columns which specify how to aggregate the records
- CLUSTER BY, DISTRIBUTE BY, SORT BY specify the sort order and algorithm
- LIMIT specifies how many # of records to retrieve

```
SELECT [ALL | DISTINCT] select_expr1, select_expr2,
.....
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];
```

We can even use **regular expressions** to select the columns that we want.

Consider the following table.

```
CREATE EXTERNAL TABLE IF NOT EXISTS stocks (
  exchange        STRING,
  symbol          STRING,
  ymd             STRING,
  price_open      FLOAT,
  price_high      FLOAT,
  price_low       FLOAT,
  price_close     FLOAT,
  volume          INT,
  price_adj_close FLOAT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/stocks';
```

Assume data is inserted into this table.

```
hive> SELECT symbol, `price.*` FROM stocks;
        AAPL 195.69 197.88 194.0 194.12 194.12
        AAPL 192.63 196.0 190.85 195.46 195.46
        AAPL 196.73 198.37 191.57 192.05 192.05
        AAPL 195.17 200.2 194.42 199.23 199.23
        AAPL 195.91 196.32 193.38 195.86 195.86
```

## Arithmetic Operators

All the typical arithmetic operators are supported. Table 6-1 describes the specific details.

*Table 6-1. Arithmetic operators*

| Operator | Types | Description |
|---|---|---|
| A + B | Numbers | Add A and B. |
| A - B | Numbers | Subtract B from A. |
| A * B | Numbers | Multiply A and B. |
| A / B | Numbers | Divide A with B. If the operands are integer types, the quotient of the division is returned. |
| A % B | Numbers | The remainder of dividing A with B. |
| A & B | Numbers | Bitwise AND of A and B. |
| A \| B | Numbers | Bitwise OR of A and B. |
| A ^ B | Numbers | Bitwise XOR of A and B. |
| ~A | Numbers | Bitwise NOT of A. |

Arithmetic operators take any numeric type. (SMALLINT, BIGINT etc.).

If the types of data differ, then the value of the smaller of the two types is promoted to wider type of the other value. For example, for INT and BIGINT operands, the INT is promoted to BIGINT. For INT and FLOAT operands, the INT is promoted to FLOAT.

## Mathematical functions, Aggregate functions, Table generating functions

*Table 6-2. Mathematical functions*

| Return type | Signature | Description |
|---|---|---|
| BIGINT | round(d) | Return the BIGINT for the rounded value of DOUBLE d. |
| DOUBLE | round(d, N) | Return the DOUBLE for the value of d, a DOUBLE, rounded to N decimal places. |
| BIGINT | floor(d) | Return the largest BIGINT that is <= d, a DOUBLE. |
| BIGINT | ceil(d), ceiling(DOUBLE d) | Return the smallest BIGINT that is >= d. |
| DOUBLE | rand(), rand(seed) | Return a pseudorandom DOUBLE that changes for each row. Passing in an integer seed makes the return value deterministic. |
| DOUBLE | exp(d) | Return e to the d, a DOUBLE. |
| DOUBLE | ln(d) | Return the natural logarithm of d, a DOUBLE. |
| DOUBLE | log10(d) | Return the base-10 logarithm of d, a DOUBLE. |
| DOUBLE | log2(d) | Return the base-2 logarithm of d, a DOUBLE. |
| DOUBLE | log(base, d) | Return the base-base logarithm of d, where base and d are DOUBLEs. |
| DOUBLE | pow(d, p), power(d, p) | Return d raised to the power p, where d and p are DOUBLEs. |
| DOUBLE | sqrt(d) | Return the square root of d, a DOUBLE. |
| STRING | bin(i) | Return the STRING representing the binary value of i, a BIGINT. |
| STRING | hex(i) | Return the STRING representing the hexadecimal value of i, a BIGINT. |
| STRING | hex(str) | Return the STRING representing the hexadecimal value of s, where each two characters in the STRING s is converted to its hexadecimal representation. |

| Return type | Signature | Description |
|---|---|---|
| STRING | unhex(i) | The inverse of hex(str). |
| STRING | conv(i, from_base, to_base) | Return the STRING in base to_base, an INT, representing the value of i, a BIGINT, in base from_base, an INT. |
| STRING | conv(str, from_base, to_base) | Return the STRING in base to_base, an INT, representing the value of str, a STRING, in base from_base, an INT. |
| DOUBLE | abs(d) | Return the DOUBLE that is the absolute value of d, a DOUBLE. |
| INT | pmod(i1, i2) | Return the positive module INT for two INTs, i1 mod i2. |
| DOUBLE | pmod(d1, d2) | Return the positive module DOUBLE for two DOUBLEs, d1 mod d2. |
| DOUBLE | sin(d) | Return the DOUBLE that is the *sin* of d, a DOUBLE, in *radians*. |
| DOUBLE | asin(d) | Return the DOUBLE that is the *arcsin* of d, a DOUBLE, in *radians*. |
| DOUBLE | cos(d) | Return the DOUBLE that is the *cosine* of d, a DOUBLE, in *radians*. |
| DOUBLE | acos(d) | Return the DOUBLE that is the *arccosine* of d, a DOUBLE, in *radians*. |
| DOUBLE | tan(d) | Return the DOUBLE that is the *tangent* of d, a DOUBLE, in *radians*. |
| DOUBLE | atan(d) | Return the DOUBLE that is the *arctangent* of d, a DOUBLE, in *radians*. |
| DOUBLE | degrees(d) | Return the DOUBLE that is the value of d, a DOUBLE, converted from radians to degrees. |
| DOUBLE | radians(d) | Return the DOUBLE that is the value of d, a DOUBLE, converted from degrees to radians. |
| INT | positive(i) | Return the INT value of i (i.e., it's effectively the expression \ +i). |
| DOUBLE | positive(d) | Return the DOUBLE value of d (i.e., it's effectively the expression \+d). |
| INT | negative(i) | Return the negative of the INT value of i (i.e., it's effectively the expression -i). |
| DOUBLE | negative(d) | Return the negative of the DOUBLE value of d; effectively, the expression -d. |
| FLOAT | sign(d) | Return the FLOAT value 1.0 if d, a DOUBLE, is positive; return the FLOAT value -1.0 if d is negative; otherwise return 0.0. |
| DOUBLE | e() | Return the DOUBLE that is the value of the constant e, 2.718281828459045. |
| DOUBLE | pi() | Return the DOUBLE that is the value of the constant pi, 3.141592653589793. |

A special kind of function is the *aggregate* function that returns a single value resulting from some computation over many rows.

Here is a query that counts the number of our example employees and averages their salaries:

```
hive> SELECT count(*), avg(salary) FROM employees;
        4 77500.0
```

*Table 6-3. Aggregate functions*

| Return type | Signature | Description |
|---|---|---|
| BIGINT | count(*) | Return the total number of retrieved rows, including rows containing NULL values. |
| BIGINT | count(expr) | Return the number of rows for which the supplied expression is not NULL. |
| BIGINT | count(DISTINCT expr[, expr_.]) | Return the number of rows for which the supplied expression(s) are unique and not NULL. |
| DOUBLE | sum(col) | Return the sum of the values. |
| DOUBLE | sum(DISTINCT col) | Return the sum of the distinct values. |
| DOUBLE | avg(col) | Return the average of the values. |
| DOUBLE | avg(DISTINCT col) | Return the average of the distinct values. |
| DOUBLE | min(col) | Return the minimum value of the values. |
| DOUBLE | max(col) | Return the maximum value of the values. |
| DOUBLE | variance(col), var_pop(col) | Return the variance of a set of numbers in a collection: col. |
| DOUBLE | var_samp(col) | Return the sample variance of a set of numbers. |
| DOUBLE | stddev_pop(col) | Return the standard deviation of a set of numbers. |
| DOUBLE | stddev_samp(col) | Return the sample standard deviation of a set of numbers. |
| DOUBLE | covar_pop(col1, col2) | Return the covariance of a set of numbers. |
| DOUBLE | covar_samp(col1, col2) | Return the sample covariance of a set of numbers. |

**Table generating functions**

The "inverse" of aggregate functions are so-called table generating functions, which take single columns and expand them to multiple columns or rows.

Consider the table.

```
CREATE TABLE employees
(
        name STRING,
        salary FLOAT,
        subordinates ARRAY<STRING>,
        deductions MAP<STRING, FLOAT>,
        address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
);
```

The following query converts the *subordinate* array in each **employees** record into zero or more new records. If an **employees** record has an empty *subordinates* array, then no new records are generated. Otherwise, one new record per *subordinate* is generated. We used a *column alias*, sub, defined using the **AS** sub clause. When using table generating functions, column aliases are required by Hive.

```
hive> SELECT explode(subordinates) AS sub FROM employees;
Mary Smith
Todd Jones
Bill King
```

*Table 6-4. Table generating functions*

| Return type | Signature | Description |
|---|---|---|
| *N rows* | `explode(array)` | Return 0 to many rows, one row for each element from the input `array`. |
| *N rows* | `explode(map)` | (v0.8.0 and later) Return 0 to many rows, one row for each `map` key-value pair, with a field for each `map` key and a field for the `map` value. |
| *tuple* | `json_tuple(jsonStr, p1, p2, …, pn)` | Like `get_json_object`, but it takes multiple names and returns a tuple. All the input parameters and output column types are `STRING`. |
| *tuple* | `parse_url_tuple(url, partname1, partname2, …, partnameN)` where `N >= 1` | Extract N parts from a URL. It takes a URL and the `part` names to extract, returning a tuple. All the input parameters and output column types are `STRING`. The valid `partnames` are case-sensitive and should only contain a minimum of white space: `HOST`, `PATH`, `QUERY`, `REF`, `PROTOCOL`, `AUTHORITY`, `FILE`, `USERINFO`, `QUERY:<KEY_NAME>`. |
| *N rows* | `stack(n, col1, …, colM)` | Convert M columns into N rows of size M/N each. |

**CASE ... WHEN ... THEN Statements**

The CASE ... WHEN ... THEN clauses are like if statements for individual columns in query results.

For example:

```
hive> SELECT name, salary,
> CASE
> WHEN salary < 50000.0 THEN 'low'
> WHEN salary >= 50000.0 AND salary < 70000.0 THEN 'middle'
> WHEN salary >= 70000.0 AND salary < 100000.0 THEN 'high'
> ELSE 'very high'
> END AS bracket FROM employees;
John Doe 100000.0 very high
Mary Smith 80000.0 high
Todd Jones 70000.0 high
Bill King 60000.0 middle
Boss Man 200000.0 very high
Fred Finance 150000.0 very high
Stacy Accountant 60000.0 middle
...
```

**WHERE Clauses – LIKE RLIKE, Predicate Functions**

While SELECT clauses select columns, WHERE clauses are filters; they select which records to return.

WHERE clauses use predicate expressions, applying predicate operators, which we will describe in a moment, to columns. Several predicate expressions can be joined with AND and OR clauses. When the predicate expressions evaluate to true, the corresponding rows are retained in the output.

Ex: To restrict of employees in the state of California:

```
SELECT * FROM employees
WHERE country = 'US' AND state = 'CA';
```

*Table 6-6. Predicate operators*

| Operator | Types | Description |
|---|---|---|
| A = B | Primitive types | True if A equals B. False otherwise. |
| A <> B, A != B | Primitive types | NULL if A or B is NULL; true if A is not equal to B; false otherwise. |
| A < B | Primitive types | NULL if A or B is NULL; true if A is less than B; false otherwise. |
| A <= B | Primitive types | NULL if A or B is NULL; true if A is less than or equal to B; false otherwise. |
| A > B | Primitive types | NULL if A or B is NULL; true if A is greater than B; false otherwise. |
| A >= B | Primitive types | NULL if A or B is NULL; true if A is greater than or equal to B; false otherwise. |
| A IS NULL | All types | True if A evaluates to NULL; false otherwise. |
| A IS NOT NULL | All types | False if A evaluates to NULL; true otherwise. |
| A LIKE B | String | True if A matches the SQL simplified *regular expression* specification given by B; false otherwise. B is interpreted as follows: 'x%' means A must begin with the prefix 'x', '%x' means A must end with the suffix 'x', and '%x%' means A must begin with, end with, or contain the substring 'x'. Similarly, the underscore '_' matches a single character. B must match the whole string A. |
| A RLIKE B, A REGEXP B | String | True if A matches the *regular expression* given by B; false otherwise. Matching is done by the JDK regular expression library and hence it follows the rules of that library. For example, the regular expression must match the entire string A, not just a subset. See below for more information about regular expressions. |

**LIKE, RLIKE**

LIKE and RLIKE predicate operators.

LIKE lets us match on strings that begin with or end with a particular substring, or when the substring appears anywhere within the string.

RLIKE clause, lets us use Java regular expressions.

For example, the following three queries select the employee names and addresses where the street ends with Ave., the city begins with O, and the street contains Chicago:

```
hive> SELECT name, address.street FROM employees WHERE address.street LIKE '%Ave.';
John Doe 1 Michigan Ave.
Todd Jones 200 Chicago Ave.

hive> SELECT name, address.city FROM employees WHERE address.city LIKE 'O%';
Todd Jones Oak Park
Bill King Obscuria

hive> SELECT name, address.street FROM employees WHERE address.street LIKE '%Chi%';
Todd Jones 200 Chicago Ave.
```

Ex:

The following query finds all the employees whose street contains the word Chicago or Ontario:

```
hive> SELECT name, address.street
> FROM employees WHERE address.street RLIKE '.*(Chicago|Ontario).*';
Mary Smith 100 Ontario St.
Todd Jones 200 Chicago Ave.
```

The string after the RLIKE keyword has the following interpretation. A period (.) matches any character and a star (*) means repeat the "thing to the left" (period, in the two cases shown) zero to many times. The expression (x|y) means match either x or y.

**GROUP BY Clauses**

The GROUP BY statement is often used in conjunction with aggregate functions to group the result set by one or more columns and then perform an aggregation over each group.

Let's return to the stocks table. The following query groups stock records for Apple by year, then averages the closing price for each year:

```
hive> SELECT year(ymd), avg(price_close) FROM stocks
> WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
> GROUP BY year(ymd);
1984 25.578625440597534
1985 20.193676221040867
1986 32.46102808021274
1987 53.88968399108163
1988 41.540079275138766
1989 41.65976212516664
1990 37.56268799823263
1991 52.49553383386182
1992 54.80338610251119
1993 41.02671956450572
1994 34.0813495847914
...
```

The HAVING clause lets you constrain the groups produced by GROUP BY in a way that could be expressed with a subquery, using a syntax that's easier to express. Here's the previous query with an additional HAVING clause that limits the results to years where the average closing price was greater than $50.0:

```
hive> SELECT year(ymd), avg(price_close) FROM stocks
> WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
> GROUP BY year(ymd)
> HAVING avg(price_close) > 50.0;
1987 53.88968399108163
1991 52.49553383386182
1992 54.80338610251119
1999 57.77071460844979
2000 71.74892876261757
2005 52.401745992993554
...
```