Hive Data Types:

This chapter takes you through the different data types in Hive, which are involved in the table creation. All the data types in Hive are classified into four types, given as follows:

1. Column Types
2. Literals
3. Null Values
4. Complex Types

**Primitive types**:

* INTEGERS
  + TINY INT 1 byte integer
  + SMALL INT 2 byte integer
  + INT 4 byte integer
  + BIGINT 8 byte integer
* BOOLEAN
  + BOOLEAN TRUE or FALSE
* FLOATING POINT numbers
  + FLOAT Single precision
  + DOUBLE Double precision
* STRING type
  + STRING Sequence of characters

**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-ddhh:mm:ss.ffffffffff”.

**Dates**

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

**Complex Types**

The Hive complex data types are as follows:

**Arrays**

Arrays in Hive are used the same way they are used in Java.

**Maps**

Maps in Hive are similar to Java Maps.

**Structs**

Structs in Hive is similar to using complex data with comment.

**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 <database name>;

hive> CREATE DATABASE userdb;

The following query is used to verify a databases list:

hive> SHOW DATABASES;

default

userdb

**Drop Database Statement**

Drop Database is a statement that drops all the tables and deletes the database. Its syntax is as follows:

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;

**Create Table Statement**

Create Table is a statement used to create a table in Hive. The syntax and example are as follows:

**Syntax**

CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db\_name.] table\_name

[(col\_namedata\_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 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 |
|  |  |  |

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

hive>CREATE TABLE IF NOT EXISTS employee ( eidint, name String, salary String, destination String) COMMENT 'Employee details'

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t'

LINES TERMINATED BY '\n'

STORED AS TEXTFILE

LOCATION ‘/user/myloc1/’;

**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 hdfs location **/user/hue/Hiveinput/** directory.

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

**LOAD** DATA INPATH '/user/hue/Hiveinput/sample.txt' INTO TABLE employee;

**Inserting data into Hive Tables from queries**

Standard syntax:

INSERT OVERWRITE TABLE tablename1 select\_statement1 FROM from\_statement;

**Example:**

INSERT OVERWRITE TABLE emplyoee1 SELECT \* FROM employee WHERE Salary>40000 &&Dept=TP;

**OVERWRITE** option will completely overwrites the table contents with the new data. Old data will be lost.

### Writing data into the filesystem from queries

Standard syntax:

INSERT OVERWRITE [LOCAL] DIRECTORY directory1

  [ROW FORMAT row\_format] [STORED AS file\_format] (Note: Only available starting with Hive 0.11.0)

  SELECT ... FROM ...

**Example**

INSERT DIRECTORY ‘/user/hue/mydr1’

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t'

LINES TERMINATED BY '\n'

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

**Alter Table Statement**

It 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\_namenew\_namenew\_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;

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\_namenew\_namenew\_type

ALTER TABLE name REPLACE COLUMNS (col\_spec[, col\_spec ...])

**Change Statement**

The following table contains the fields of **employee** table and it shows the fields to be changed (in bold).

The following table contains the fields of **employee** table and it shows the fields to be changed (in bold).

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Field Name** |  |  | **Convert from** |  |  | **Change Field** |  |  | **Convert to** |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  | **Data Type** |  |  | **Name** |  |  | **Data Type** |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | eid | |  | int | |  | eid | |  | int | |  |
|  |  | |  |  | |  |  | |  |  | |  |
|  | **name** | |  | String | |  | **ename** | |  | String | |  |
|  |  | |  |  | |  |  | |  |  | |  |
|  | salary | |  | **Float** | |  | salary | |  | **Double** | |  |
|  |  | |  |  | |  |  | |  |  | |  |
|  | designation | |  | String | |  | designation | |  | String | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

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 empidInt,

>ename STRING name String);

**Partitioning an Hive Table:**

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:

**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';

**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’);

**Operators**

There are four types of operators in Hive:

1. Relational Operators
2. Arithmetic Operators
3. Logical Operators
4. Complex Operators

**Relational Operators**

These operators are used to compare two operands. The following table describes the relational operators available in Hive:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operand** | **Description** |
|  |  |  |
|  |  |  |
| A = B | all primitive | TRUE if expression A is equivalent to expression |
|  | types | B otherwise FALSE. |
|  |  |  |
| A != B | all primitive | TRUE if expression A is *not* equivalent to |
|  | types | expression B otherwise FALSE. |
|  |  |  |
| A < B | all primitive | TRUE if expression A is less than expression B |
|  | types | otherwise FALSE. |
|  |  |  |
| A <= B | all primitive | TRUE if expression A is less than or equal to |
|  | types | expression B otherwise FALSE. |
|  |  |  |
| A > B | all primitive | TRUE if expression A is greater than expression |
|  | types | B otherwise FALSE. |
|  |  |  |
| A >= B | all primitive | TRUE if expression A is greater than or equal to |
|  | types | expression B otherwise FALSE. |
|  |  |  |
| A IS NULL | all types | TRUE if expression A evaluates to NULL |
|  |  | otherwise FALSE. |
|  |  |  |
| A IS NOT | all types | FALSE if expression A evaluates to NULL |
| 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. |
|  |  |  |

**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.

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

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

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;

**Arithmetic Operators**

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

**Example**

The following query adds two numbers, 20 and 30

hive> SELECT 20+30 ADD FROM temp;

**Logical Operators**

The operators are logical expressions. 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. | |  |
|  |  |  |  |  |  |  |  |  |  |

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

**Built-In Functions**

Hive supports the following built-in functions:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Return** |  | |  | **Signature** |  | |  | **Description** |  | |  |
|  |  | |  |  | |  |  | |  |
|  | **Type** |  | |  |  |  | |  |  |  | |  |
|  |  |  | |  |  |  | |  |  |  | |  |
|  |  |  | |  |  |  | |  |  |  | |  |
| 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 | | |  | It returns a random number that changes from row | | |  |
|  |  |  | |  | seed) | | |  | to row. | | |  |
|  | | | |  |  | | |  |  | | |  |
| string | | | |  | concat(string A, | | |  | It returns the string resulting from concatenating | | |  |
|  |  |  | |  | string B,...) | | |  | B after A. | | |  |
|  | | | |  |  | | |  |  | | |  |
| string | | | |  | substr(string A, | | |  | It returns the substring of A starting from start | | |  |
|  |  |  | |  | int start) | | |  | position till the end of string A. | | |  |
|  | | | |  |  | | |  |  | | |  |
| string | | | |  | substr(string A, | | |  | It returns the substring of A starting from start | | |  |
|  |  |  | |  | int start, int | | |  | position with the given length. | | |  |
|  |  |  | |  | 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. | | |  |
|  |  |  | |  |  |  | |  | | | |  |
|  |  |  | |  |  |  | | 55 | | | |  |
|  | | | |  | | | | Apache Hive | | | |
|  | | | |  | | | |  | | | |
| 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) | | | | It returns the string resulting from trimming | | | |
|  | | | |  | | | | spaces from the end (right hand side) of A. | | | |
|  | | | |  | | | |  | | | |
| string | | | | regexp\_replace(s | | | | It returns the string resulting from replacing all | | | |
|  | | | | tring A, string B, | | | | substrings in B that match the Java regular | | | |
|  | | | | string C) | | | | 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* | | | | cast(*<expr>* as | | | | It converts the results of the expression expr to | | | |
| *<type>* | | | | *<type>*) | | | | <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(in | | | | convert the number of seconds from Unix epoch | | | |
|  | | | | t unixtime) | | | | (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 | | | | It returns the date part of a timestamp string: | | | |
|  | | | | timestamp) | | | | 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 | | | | It returns the month part of a date or a timestamp | | | |
|  | | | | date) | | | | 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, | | | |
|  | | | |  | | | |  | | | |

**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;🡪2.0

**floor()** function

hive> SELECT ceil(2.6) from temp;🡪3.0

**Aggregate Functions**

Hive supports the following built-in **aggregate functions**. The usage of these functions is as same as the SQL aggregate functions.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | Apache Hive | | |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  | **Return** |  |  | **Signature** |  |  | **Description** |  |  |  |
|  |  |  |  |  |  |  |  |
|  | **Type** |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
| BIGINT | | |  | count(\*), | |  | count(\*) - Returns the total number of retrieved | |  |  |
|  |  |  |  | count(expr), | |  | rows. | |  |  |
|  | | |  |  | |  |  | |  |  |
| DOUBLE | | |  | sum(col), | |  | It returns the sum of the elements in the group or | |  |  |
|  |  |  |  | sum(DISTINCT | |  | the sum of the distinct values of the column in the | |  |  |
|  |  |  |  | col) | |  | group. | |  |  |
|  | | |  |  | |  |  | |  |  |
| DOUBLE | | |  | avg(col), | |  | It returns the average of the elements in the | |  |  |
|  |  |  |  | avg(DISTINCT | |  | group or the average of the distinct values of the | |  |  |
|  |  |  |  | col) | |  | 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. | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

**ORDER BY clause in a SELECT statement**

how to use the ORDER BY clause in a SELECT statement. 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;

**GROUP BY clause in a SELECT statement**

**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;

**Joins:**

JOINS 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 JOINS.

**Syntax**

table\_reference JOIN table\_factor [join\_condition]

| table\_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table\_reference

join\_condition

**Example**

We will use the following two tables in this chapter. Consider the following table named CUSTOMERS..

Create two csv files

Customers.csv

ID NAME ` AGE ADDRESS SALARY

1 Ramesh 32 Ahmedabad 2000

2 Khilan 25 Delhi 1500

3 kaushik 23 Kota 2000

4 Suma 25 Mumbai 6500

5 Hardik 27 Bhopal 8500

6 Komal 22 MP 4500

7 Muffy 24 Indore 10000

Orders.csv

|  |  |  |  |
| --- | --- | --- | --- |
| OID | DATE | CUSTOMER\_ID | AMOUNT |
| 102 | 10/8/2009 0:00 | 3 | 1000 |
| 100 | 10/8/2009 0:00 | 3 | 2000 |
| 101 | 11/20/2009 0:00 | 2 | 3000 |
| 103 | 5/20/2008 0:00 | 4 | 4000 |

Load these two files into /user/hue location

Create 2 tables CUSTOMERS AND ORDERS with this data

CREATE TABLE customers1 (address string ,agebigint,idbigint,namestring,salarybigint)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

tblproperties ("skip.header.line.count"="1");

;

LOAD DATA INPATH '/user/hue/CUSTOMERS.csv' INTO TABLE customers1;

CREATE TABLE ORDERS (oidbigint,odatestring,customer\_idbigint,amountbigint)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ',';

LOAD DATA INPATH '/user/hue/ORDERS.csv' INTO TABLE ORDERS;

**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.

SELECT c.ID, c.NAME, o.AMOUNT, o.odate FROM CUSTOMERS c RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

3 kaushik 1000 10/8/2009 0:00

3 kaushik 2000 10/8/2009 0:00

2 Khilan 3000 11/20/2009 0:00

4 Suma 4000 5/20/2008 0:00

**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:

SELECT c.ID, c.NAME, o.AMOUNT, o.oDATE FROM CUSTOMERS c LEFT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

1 Ramesh NULLNULL

2 Khilan 3000 11/20/2009 0:00

3 kaushik 1000 10/8/2009 0:00

3 kaushik 2000 10/8/2009 0:00

4 Suma 4000 5/20/2008 0:00

5 Hardik NULLNULL

6 Komal NULL NULL

7 Muffy NULL NULL

**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:

SELECT c.ID, c.NAME, o.AMOUNT, o.oDATE FROM CUSTOMERS c FULL OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

1 Ramesh NULLNULL

2 Khilan 3000 11/20/2009 0:00

3 kaushik 1000 10/8/2009 0:00

3 kaushik 2000 10/8/2009 0:00

4 Suma 4000 5/20/2008 0:00

5 Hardik NULLNULL

6 Komal NULL NULL

7 Muffy NULL NULL