There are four types of operators in Hive:

* Relational Operators
* Arithmetic Operators
* Logical Operators
* Complex Operators

**Relational Operators**

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operand** | **Description** |
| A = B | all primitive types | TRUE if expression A is equivalent to expression B otherwise FALSE. |
| A != B | all primitive types | TRUE if expression A is not equivalent to expression B otherwise FALSE. |
| A < B | all primitive types | TRUE if expression A is less than expression B otherwise FALSE. |
| A <= B | all primitive types | TRUE if expression A is less than or equal to expression B otherwise FALSE. |
| A > B | all primitive types | TRUE if expression A is greater than expression B otherwise FALSE. |
| A >= B | all primitive types | TRUE if expression A is greater than or equal to expression B otherwise FALSE. |
| A IS NULL | all types | TRUE if expression A evaluates to NULL otherwise FALSE. |
| A IS NOT NULL | all types | FALSE if expression A evaluates to NULL otherwise TRUE. |
| A LIKE B | Strings | TRUE if string pattern A matches to B otherwise FALSE. |
| A RLIKE B | Strings | NULL if A or B is NULL, TRUE if any substring of A matches the Java regular expression B , otherwise FALSE. |
| A REGEXP B | Strings | Same as RLIKE. |

**Example**

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

+-----+--------------+--------+---------------------------+------+

| Id | Name | Salary | Designation | Dept |

+-----+--------------+------------------------------------+------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

|1205 | Kranthi | 30000 | Op Admin | Admin|

+-----+--------------+--------+---------------------------+------+

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

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

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

+-----+-----------+-----------+----------------------------------+

| ID | Name | Salary | Designation | Dept |

+-----+---------------+-------+----------------------------------+

|1205 | Kranthi | 30000 | Op Admin | Admin |

+-----+-----------+-----------+----------------------------------+

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

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

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

+-----+------------+--------+----------------------------+------+

| ID | Name | Salary | Designation | Dept |

+-----+------------+--------+----------------------------+------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali| 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

+-----+------------+--------+----------------------------+------+

**Arithmetic Operators**

These operators support various common arithmetic operations on the operands. All of them return number types. The following table describes the arithmetic operators available in Hive:

|  |  |  |
| --- | --- | --- |
| **Operators** | **Operand** | **Description** |
| A + B | all number types | Gives the result of adding A and B. |
| A - B | all number types | Gives the result of subtracting B from A. |
| A \* B | all number types | Gives the result of multiplying A and B. |
| A / B | all number types | Gives the result of dividing B from A. |
| A % B | all number types | Gives the reminder resulting from dividing A by B. |
| A & B | all number types | Gives the result of bitwise AND of A and B. |
| A | B | all number types | Gives the result of bitwise OR of A and B. |
| A ^ B | all number types | Gives the result of bitwise XOR of A and B. |
| ~A | all number types | Gives the result of bitwise NOT of A. |

**Example**

The following query adds two numbers, 20 and 30.

hive> SELECT 20+30 ADD FROM temp;

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

+--------+

| ADD |

+--------+

| 50 |

+--------+

**Logical Operators**

The operators are logical expressions. 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;

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

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

+------+--------------+-------------+-------------------+--------+

**Complex Operators**

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

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

**Built-In Functions**

Hive supports the following built-in functions:

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Signature** | **Description** |
| BIGINT | round(double a) | It returns the rounded BIGINT value of the double. |
| BIGINT | floor(double a) | It returns the maximum BIGINT value that is equal or less than the double. |
| BIGINT | ceil(double a) | It returns the minimum BIGINT value that is equal or greater than the double. |
| double | rand(), rand(int seed) | It returns a random number that changes from row to row. |
| string | concat(string A, string B,...) | It returns the string resulting from concatenating B after A. |
| string | substr(string A, int start) | It returns the substring of A starting from start position till the end of string A. |
| string | substr(string A, int start, int length) | It returns the substring of A starting from start position with the given length. |
| string | upper(string A) | It returns the string resulting from converting all characters of A to upper case. |
|  |  |  |
| string | ucase(string A) | Same as above. |
| string | lower(string A) | It returns the string resulting from converting all characters of B to lower case. |
| string | lcase(string A) | Same as above. |
| string | trim(string A) | It returns the string resulting from trimming spaces from both ends of A. |
| string | ltrim(string A) | It returns the string resulting from trimming spaces from the beginning (left hand side) of A. |
| string | rtrim(string A) | rtrim(string A) It returns the string resulting from trimming spaces from the end (right hand side) of A. |
| string | regexp\_replace(string A, string B, string C) | It returns the string resulting from replacing all substrings in B that match the Java regular expression syntax with C. |
| int | size(Map<K.V>) | It returns the number of elements in the map type. |
| int | size(Array<T>) | It returns the number of elements in the array type. |
| value of <type> | cast(<expr> as <type>) | It converts the results of the expression expr to <type> e.g. cast('1' as BIGINT) converts the string '1' to it integral representation. A NULL is returned if the conversion does not succeed. |
| string | from\_unixtime(int unixtime) | convert the number of seconds from Unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00" |
| string | to\_date(string timestamp) | It returns the date part of a timestamp string: to\_date("1970-01-01 00:00:00") = "1970-01-01" |
| int | year(string date) | It returns the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970 |
| int | month(string date) | It returns the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11 |
| int | day(string date) | It returns the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1 |
| string | get\_json\_object(string json\_string, string path) | It extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It returns NULL if the input json string is invalid. |

**Example**

The following queries demonstrate some built-in functions:

**round() function**

hive> SELECT round(2.6) from temp;

select round(salary)from employee

select round(fees ) from student

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

3.0

**floor() function**

hive> SELECT floor(2.6) from temp;

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

2.0

**ceil() function**

hive> SELECT ceil(2.6) from temp;

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

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.

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

Create a new table

CREATE TABLE IF NOT EXISTS employee ( eid int, name String,

salary String, designation String , yoj int, dept string )

COMMENT ‘Employee details’

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘\t’

LINES TERMINATED BY ‘\n’

STORED AS TEXTFILE;

Insert Data

1201 Gopal 45000 Technical manager 2012 HR

1202 Manisha 45000 Proof reader 2013 Finance

1203 Masthanvali 40000 Technical writer 2012 IT

1204 Kiran 40000 Hr Admin 2020 IT

1205 Kranthi 30000 Op Admin 2021 ERP

SELECT \* FROM employee WHERE Id=1205;

SELECT \* FROM employee WHERE Salary>=40000;

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

SELECT Id, Name, Dept FROM employee ORDER BY DEPT;

SELECT Dept,count(\*) FROM employee GROUP BY DEPT;

select \* from order\_items order by order\_item\_product\_price ASC limit 1;

SELECT product\_id, producty\_name from products where product\_price (select min(product\_price) from products);

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

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Consider another table ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

There are different types of joins given as follows:

* JOIN
* LEFT OUTER JOIN
* RIGHT OUTER JOIN
* FULL OUTER JOIN

## JOIN

JOIN clause is used to combine and retrieve the records from multiple tables. JOIN is same as OUTER JOIN in SQL. A JOIN condition is to be raised using the primary keys and foreign keys of the tables.

The following query executes JOIN on the CUSTOMER and ORDER tables, and retrieves the records:

hive> SELECT c.ID, c.NAME, c.AGE, o.AMOUNT

FROM CUSTOMERS c JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

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

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

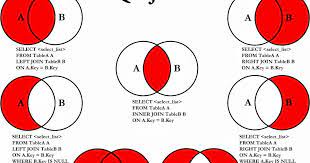
| 3 | kaushik | 23 | 3000 |

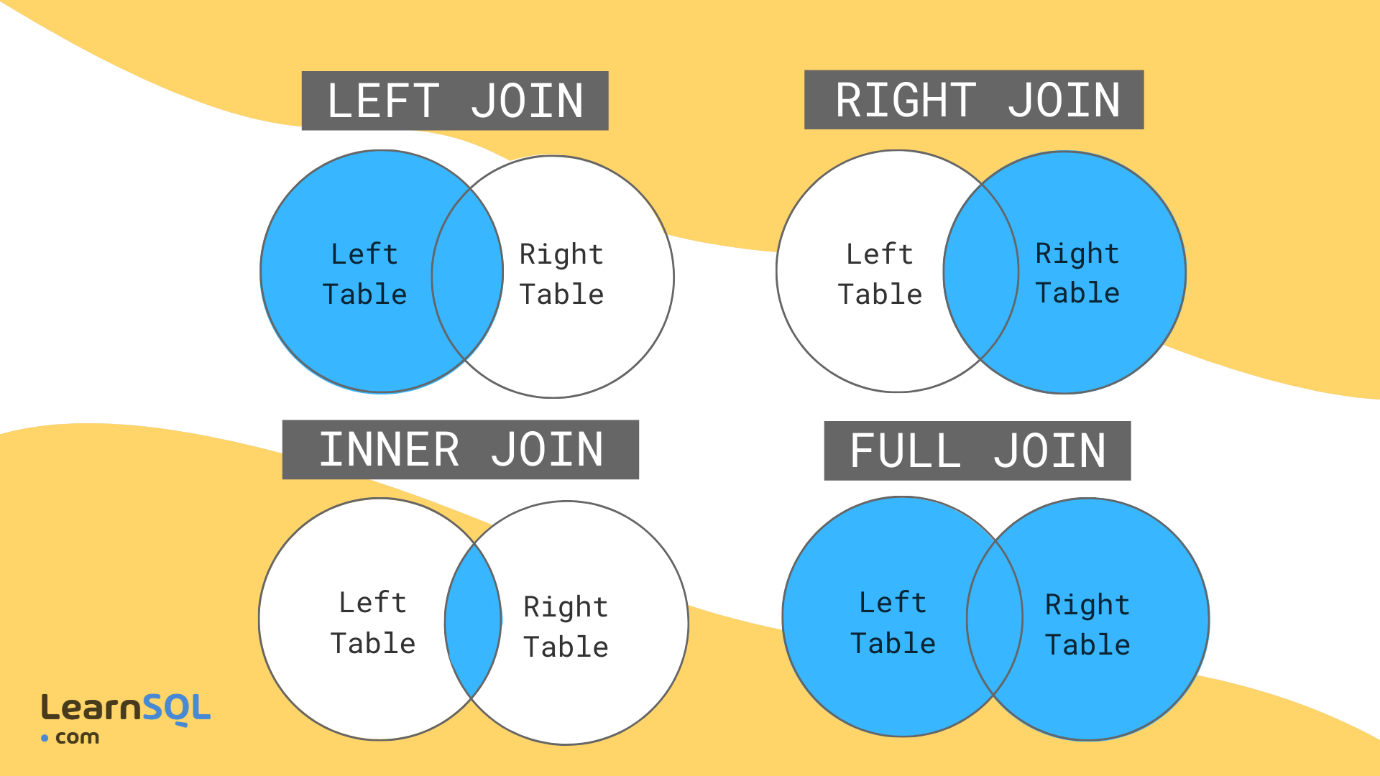
| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+





## LEFT OUTER JOIN

The HiveQL LEFT OUTER JOIN returns all the rows from the left table, even if there are no matches in the right table. This means, if the ON clause matches 0 (zero) records in the right table, the JOIN still returns a row in the result, but with NULL in each column from the right table.

A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDER tables:

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE

FROM c

LEFT OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

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

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+----------+--------+---------------------+

## 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.

notranslate"> hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

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

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

## FULL OUTER JOIN

The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.

The following query demonstrates FULL OUTER JOIN between CUSTOMER and ORDER tables:

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE

FROM CUSTOMERS c

FULL OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

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

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

## Creating a View

You can create a view at the time of executing a SELECT statement. The syntax is as follows:

CREATE VIEW [IF NOT EXISTS] view\_name [(column\_name [COMMENT column\_comment], ...) ]

[COMMENT table\_comment]

AS SELECT ...

## Example

Assume employee table as given below, with the fields Id, Name, Salary, Designation, and Dept.

Generate a query to retrieve the employee details who earn a salary of more than Rs 30000. We store the result in a view named **emp\_30000.**

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

|1205 | Kranthi | 30000 | Op Admin | Admin |

+------+--------------+-------------+-------------------+--------+

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

hive> CREATE VIEW emp\_30000 AS

SELECT \* FROM employee

WHERE salary>30000;

Select \* from emp\_30000

## Dropping a View

Use the following syntax to drop a view:

DROP VIEW view\_name

The following query drops a view named as emp\_30000:

hive> DROP VIEW emp\_30000;