**Hive Topics**

* [**HIVE DATA TYPES**](#page27)
* [**CREATE DATABASE**](#page30)
* [**DROP DATABASE**](#page33)
* [**CREATE TABLE**](#page35)
* [**ALTER TABLE**](#page41)
* [**DROP TABLE**](#page49)
* [**BUILT-IN OPERATORS**](#page55)
* [**BUILT-IN FUNCTIONS**](#page60)
* [**VIEWS AND INDEXES**](#page64)
* [**HIVEQL SELECT…WHERE**](#page67)
* [**HIVEQL SELECT…ORDER BY**](#page71)
* [**HIVEQL GROUP BY**](#page75)
* [**HIVEQL JOINS**](#page79)
* [**PARTITIONING**](#page52)
* **BUCKETING**
* **Hive Complex Data Types with Examples**
* **Analyze JSON Data using Hive**
* **Processing Log Using Hive**

[**HIVE DATA TYPES**](#page27)

Hive data types are classified by four:

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

**Column Types**

Column type are used as column data types of Hive. They are as follows:

**Integral Types:**

* TINYINT
* SMALLINT
* INT
* BIGINT

**String Types:**

* Varchar
* Char

**Timestamp:**

It supports java.sql.Timestamp format

* “YYYY-MM-DD HH:MM:SS.fffffffff”
* “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)

decimal(10,0)

**Union Types:**

Union is a collection of heterogeneous data types. You can create an instance using **create union**. The syntax and example is as follows:

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

{0:1}

{1:2.0}

{2:["three","four"]}

{3:{"a":5,"b":"five"}}

{2:["six","seven"]}

{3:{"a":8,"b":"eight"}}

{0:9}

{1:10.0}

**Literals**

The following literals are used in Hive:

**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 10308.

**Null Value**

Missing values are represented by the special value NULL.

**Complex Types**

The Hive complex data types are as follows:

**Arrays**

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

Syntax: ARRAY<data\_type>

**Maps**

Maps in Hive are similar to Java Maps.

Syntax: MAP<primitive\_type, data\_type>

**Structs**

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

Syntax: STRUCT<col\_name : data\_type [COMMENT col\_comment], ...>

**CREATE DATABASE**

**Create Database Statement**

CREATE DATABASE|SCHEMA [IF NOT EXISTS] <database name>;

hive> CREATE DATABASE IF NOT EXISTS userdb;

**or**

hive> CREATE SCHEMA userdb;

The following query is used to verify a databases list:

hive> SHOW DATABASES;

default

userdb

hive> describe database extended USERDB\_BKP\_EXT;

**DROP DATABASE**

**Drop Database Statement**

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

DROP DATABASE StatementDROP (DATABASE|SCHEMA) [IF EXISTS] database\_name

[RESTRICT|CASCADE];

The following queries are used to drop a database. Let us assume that the database name is **userdb**.

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;

The following query drops the database using **SCHEMA**.

hive> DROP SCHEMA userdb;

**CREATE TABLES**

1. **Manage Tables**
2. **External Tables**

**Create Manage Tables**

The following query to select Database.

hive> use UserDB;

The following query creates a table named **employee** using the above data.

**hive> create table patient (pid INT, pname STRING, drug STRING, gender STRING,**

**tot\_amt INT) row format delimited fields terminated by ',' stored as textfile;**

The following query load the text from local drive into the table.

**hive> load data local inpath ‘/home/dinesh/HadoopData/datagen\_10.txt’ into table patient;**

**hive> load data inpath ‘/dinesh/hadoop/data/datagen\_10.txt’ into table patient;**

The following query to view table **SCHEMA**.

hive> describe employee;

The following query to view table extended **SCHEMA**.

hive> describe extended emplyee;

**Create External Table**

The following query to select Database.

hive> use UserDB;

The following query creates a table named **employee** using the above data.

**hive> create external table** **patientext (pid INT, pname STRING, drug STRING, gender STRING, tot\_amt INT)**

**> row format delimited fields terminated by ','**

**> stored as textfile**

**> LOCATION '/dinesh/hadoop/data/patientext/';**

**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\_name new\_name new\_type

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

**hive> create table patient\_Alter (pid INT, pname STRING, drug STRING, gender STRING,**

**tot\_amt INT) row format delimited fields terminated by ',' stored as textfile;**

**Rename To… Statement**

The following query renames the table from **employee** to **emp**.

hive> ALTER TABLE **patient\_Alter** RENAME TO **patientAlter**;

The following queries rename the column name and column data type using the above data:

**Change Statement**

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

hive> ALTER TABLE **patientAlter** CHANGE **pname** **patientname** String;

hive> describe **patientalter**;

**Add Columns Statement**

The following query adds a column named **dept** to the **employee** table.

hive> ALTER TABLE **patientAlter** ADD COLUMNS (dept STRING COMMENT 'Department name');

hive> describe **patientalter**;

**Drop Table Statement**

The syntax is as follows:

DROP TABLE [IF EXISTS] table\_name;

The following query drops a table named **employee:**

hive> DROP TABLE IF EXISTS **patientalter**;

**Output:**

Drop table successful.

The following query is used to verify the list of tables:

hive> SHOW TABLES;

emp

ok

Time taken: 2.1 seconds

hive>

**BUILT – IN OPERATORS**

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. |  |
|  | |  |  |  |

hive> **select \* from patient where tot\_amt = 980;**

The following query is executed to retrieve the employee details whose salary is more than or equal to Rs 980.

hive> **select \* from patient where tot\_amt >= 833;**

**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 ;**

**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 patient where tot\_amt >= 833 AND pid > 3;**

**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**

**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 | | |  | 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. | | |
|  |  |  | |  |  |  | |  | | | |
| 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  tring 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* | | | | 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, | | | |
|  | | |  | | | | day("1970-11-01") = 1 | | | | |
|  | | |  | | | |  | | | | |
| string | | | get\_json\_object( | | | | It extracts json object from a json string based on | | | | |
|  | | | string | | | | json path specified, and returns json string of the | | | | |
|  | | | json\_string, | | | | extracted json object. It returns NULL if the input | | | | |
|  | | | string path) | | | | json string is invalid. | | | | |
|  | | |  | | | |  | | | | |

**Example**

The following queries demonstrate some built-in functions:

**round() function**

hive> **SELECT round(2.65555, 2)**;

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

3.0

**floor() function**

hive> **SELECT floor(2.6)**;

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

2.0

**Aggregate Functions**

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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  | **Return** |  |  | **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. | |  |
|  |  |  |  |  |  |  |  |  |  |

**VIEWS**

**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 ...

hive> **CREATE VIEW patient\_500 AS**

* **SELECT \* FROM patient WHERE tot\_amt >500;**

**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 patient\_500;**

**HIVEQL SELECT … WHERE**

**Syntax**

hive> **SELECT \* FROM patient WHERE tot\_amt >500;**

**HIVEQL SELECT … ORDER BY**

**Syntax**

Given below is the syntax of the ORDER BY clause:

hive> **SELECT \* FROM patient ORDER BY tot\_amt;**

**HIVEQL GROUP BY**

**Syntax**

The syntax of GROUP BY clause is as follows:

hive> **SELECT drug,sum(tot\_amt) FROM patient GROUP BY drug;**

**HIVEQL JOINS**

**Syntax**

**Load Data:**

The following query creates a table named **employee** using the above data.

**hive> create table patient\_j (pid INT, pname STRING, drug STRING, gender STRING,**

**tot\_amt INT) row format delimited fields terminated by ',' stored as textfile;**

The following query load the text from local drive into the table.

**hive> load data local inpath '/home/dinesh/HadoopData/datagen\_10.txt' into table patient\_j;**

**hive> load data inpath '/dinesh/hadoop/data/datagen\_10.txt' into table patient\_j;**

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 T1.pid,T1.pname,T2 .tot\_amt FROM patient T1 JOIN patient\_j T2 ON (T1.pid = T2.pid);**

**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 T1.pid,T1.pname,T2 .tot\_amt FROM patient T1 LEFT OUTER JOIN patient\_j T2 ON (T1.pid = T2.pid);**

**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.

**hive> SELECT T1.pid,T1.pname,T2 .tot\_amt FROM patient T1 RIGHT OUTER JOIN patient\_j T2 ON (T1.pid = T2.pid);**

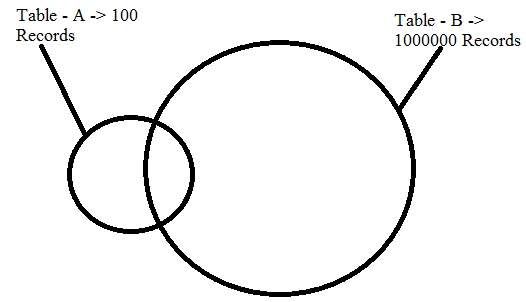
**FULL OUTER JOIN**

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 T1.pid,T1.pname,T2 .tot\_amt FROM patient T1 FULL OUTER JOIN patient\_j T2 ON (T1.pid = T2.pid);**

**Map-Side JOIN**



1. Can use this, When need join small table and large table.
2. Large table can be streamed through the mappers and small are cached in memory.
3. Hive can also do all the join in map side, since it can look up every possible match against the small tables in memory.
4. All the task performed by mapper only.

**Query:**

SELECT /\*+ MAPJOIN(dataset2) \*/ dataset1.first\_name, dataset1.eid,dataset2.eid FROM dataset1 JOIN dataset2 ON dataset1.first\_name = dataset2.first\_name;

**Hive Complex Data Types with Examples:**

**There are three complex types in hive**,

* **arrays:** It is an ordered collection of elements.The elements in the array must be of the same type.
* **map:** It is an unordered collection of key-value pairs.Keys must be of primitive types.Values can be of any type.
* **struct:** It is a collection of elements of different types.

**ARRAY:**

$ cat >arrayfile

1,abc,40000,a$b$c,hyd

2,def,3000,d$f,bang

hive> **create table arrayfile(id int,name string,sal bigint,sub array<string>,city string) row format delimited fields terminated by ',' collection items terminated by '$';**

hive> **load data local inpath '/home/dinesh/HadoopData/arrayfile.txt' overwrite into table arrayfile;**

hive>**select sub[2] from arrayfile where id=1;**

hive>**select sub[0] from arrayfile;**

**MAP:**

$ cat >mapfile

1,abc,40000,a$b$c,pf#500$epf#200,hyd

2,def,3000,d$f,pf#500,bang

hive>**create table mapfile(id int,name string,sal bigint,sub array<string>,dud map<string,int>,city string) row format delimited fields terminated by ',' collection items terminated by '$' map keys terminated by '#';**

hive> **load data local inpath '/home/dinesh/HadoopData/mapfile.txt' overwrite into table mapfile;**

hive>**select dud["pf"] from mapfile;**

hive>**select dud["pf"],dud["epf"] from mapfile;**

**STRUCT:**

cat >structfile

1,abc,40000,a$b$c,pf#500$epf#200,hyd$ap$500001

2,def,3000,d$f,pf#500,bang$kar$600038

hive> **create table structfile(id int,name string,sal bigint,sub array<string>,dud map<string,int>,addr struct<city:string,state:string,pin:bigint>) row format delimited fields terminated by ',' collection items terminated by '$' map keys terminated by '#';**

hive> **load data local inpath '/home/dinesh/HadoopData/structfile.txt' into table structfile;**

hive>**select addr.city from structfile;**

**PARTITIONING**

**Partitioning:**

1. Hive data will load into HDFS file system with below structure

/User/hive/warehouse/<DB\_Name>/<Table\_Name>/file\_1.txt Jan

/User/hive/warehouse/<DB\_Name>/<Table\_Name>/file\_1.txt Feb

/User/hive/warehouse/<DB\_Name>/<Table\_Name>/file\_1.txt Mar

1. Data storing Monthly base.
2. If we write select query against this table like

“Select \* from <Table\_Name> Where Month = “Jan”

Data will search from all the file. So query performance will decrease.

1. If we create partitioning by every month. Folder will create for each month and Files are store by month folder and query performance will be increase.

**Create Partitioning:**

hive>**create table patient\_par (pid INT, pname STRING, gender STRING, tot\_amt INT)** PARTITIONED BY (**drug** **VARCHAR(64)) row format delimited fields terminated by ','** **STORED AS SEQUENCEFILE;**

The following query load the text from local drive into the table.

**hive>** **load data local inpath '/home/dinesh/HadoopData/datagen\_10.txt' into table patient\_par;**

CREATE TABLE partitioned\_user(

firstname VARCHAR(64),

lastname VARCHAR(64),

address STRING,

city VARCHAR(64),

post STRING,

phone1 VARCHAR(64),

phone2 STRING,

email STRING,

web STRING

)

PARTITIONED BY (country VARCHAR(64), state VARCHAR(64))

STORED AS SEQUENCEFILE;

**Static Partitioning in Hive:**

Example Data

first\_name,last\_name,address,city,post,phone1,phone2,email,web

Rebbecca,Didio,171 E 24th St,Leith,7315,03-8174-9123,0458-665-290,rebbecca.didio@didio.com.au,http://www.brandtjonathanfesq.com.au

Stevie,Hallo,22222 Acoma St,Proston,4613,07-9997-3366,0497-622-620,stevie.hallo@hotmail.com,http://www.landrumtemporaryservices.com.au

Mariko,Stayer,534 Schoenborn St #51,Hamel,6215,08-5558-9019,0427-885-282,mariko\_stayer@hotmail.com,http://www.inabinetmacreesq.com.au

**Load data into partitioned table:**

hive> LOAD DATA LOCAL INPATH '${env:HOME}/staticinput.txt'

INTO TABLE partitioned\_user

PARTITION (country = 'US', state = 'CA');

**Dynamic Partitioning in Hive:**

hive>INSERT INTO TABLE partitioned\_user

PARTITION (country, state)

SELECT firstname ,

lastname ,

address ,

city ,

post ,

phone1 ,

phone2 ,

email ,

web ,

country ,

state

FROM temp\_user;

**Show Partitioning:**

hive> SHOW PARTITIONS partitioned\_user;

OK

country=AU/state=AC

country=AU/state=NS

hive> SHOW PARTITIONS partitioned\_user PARTITION(country='US');

OK

country=US/state=AK

country=US/state=AR

country=US/state=AZ

**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';

**Renaming a Partition**

The syntax of this command is as follows.

ALTER TABLE table\_name PARTITION partition\_spec RENAME TO PARTITION

partition\_spec;

The following query is used to rename a partition:

hive> ALTER TABLE employee PARTITION (year=’1203’)

> RENAME TO PARTITION (Yoj=’1203’);

**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:

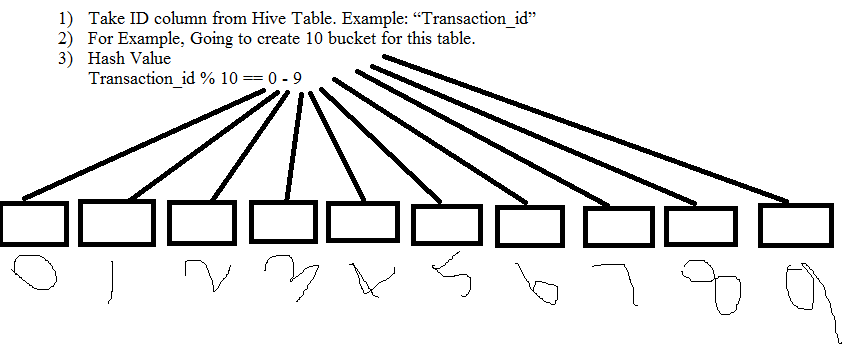
hive> ALTER TABLE employee DROP [IF EXISTS]

> PARTITION (year=’1203’);

BUCKETING IN HIVE

1. Bucket concept is based on (Hashing function) mod (By total number of bucket)
2. Take ID column from Hive Table. Example: “Transaction\_id”
3. For Example, Going to create 10 bucket for this table
4. Hash Value

Transaction\_id % 10 == 0-9



**Creating Bucket tables:**

CREATE TABLE bucketed\_user(

firstname VARCHAR(64),

lastname VARCHAR(64),

address STRING,

city VARCHAR(64),

state VARCHAR(64),

post STRING,

phone1 VARCHAR(64),

phone2 STRING,

email STRING,

web STRING

)

COMMENT 'A bucketed sorted user table'

PARTITIONED BY (country VARCHAR(64))

CLUSTERED BY (state) SORTED BY (city) INTO 32 BUCKETS

STORED AS SEQUENCEFILE;

**Inserting data into Bucket tables:**

set hive.enforce.bucketing = true;

INSERT OVERWRITE TABLE bucketed\_user PARTITION (country)

SELECT firstname ,

lastname ,

address ,

city ,

state ,

post ,

phone1 ,

phone2 ,

email ,

web ,

country

FROM temp\_user;

set hive.exec.dynamic.partition=true;

set hive.exec.dynamic.partition.mode=nonstrict;

set hive.exec.max.dynamic.partitions.pernode=1000;

set hive.enforce.bucketing = true;

DROP TABLE IF EXISTS bucketed\_user;

CREATE TEMPORARY TABLE temp\_user(

firstname VARCHAR(64),

lastname VARCHAR(64),

address STRING,

country VARCHAR(64),

city VARCHAR(64),

state VARCHAR(64),

post STRING,

phone1 VARCHAR(64),

phone2 STRING,

email STRING,

web STRING

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/home/user/user\_table.txt' INTO TABLE temp\_user;

CREATE TABLE bucketed\_user(

firstname VARCHAR(64),

lastname VARCHAR(64),

address STRING,

city VARCHAR(64),

state VARCHAR(64),

post STRING,

phone1 VARCHAR(64),

phone2 STRING,

email STRING,

web STRING

)

COMMENT 'A bucketed sorted user table'

PARTITIONED BY (country VARCHAR(64))

CLUSTERED BY (state) SORTED BY (city) INTO 32 BUCKETS

STORED AS SEQUENCEFILE;

set hive.enforce.bucketing = true;

INSERT OVERWRITE TABLE bucketed\_user PARTITION (country)

SELECT firstname ,

lastname ,

address ,

city ,

state ,

post ,

phone1 ,

phone2 ,

email ,

web ,

country

FROM temp\_user;

**Partitioned and Bucketing:**



**Analyze JSON Data using Hive**

**Json Data:**

{

  "DocId": "ABC",

  "User": {

    "Id": 1234,

    "Username": "sam1234",

    "Name": "Sam",

    "ShippingAddress": {

      "Address1": "123 Main St.",

      "Address2": null,

      "City": "Durham",

      "State": "NC"

    },

    "Orders": [

      {

        "ItemId": 6789,

        "OrderDate": "11/11/2012"

      },

      {

        "ItemId": 4352,

        "OrderDate": "12/12/2012"

      }

    ]

  }

}

Hive Schema:

CREATE TABLE complex\_json (

  DocId string,

  User struct<Id:int,

              Username:string,

              Name: string,

              ShippingAddress:struct<Address1:string,

                                     Address2:string,

                                     City:string,

                                     State:string>,

              Orders:array<struct<ItemId:int,

                                  OrderDate:string>>>

)

ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe';

Load the data:

LOAD DATA LOCAL INPATH '/tmp/complex.json'

OVERWRITE INTO TABLE complex\_json;

**Hive Query:**

SELECT DocId, User.Id, User.ShippingAddress.City as city,

       User.Orders[0].ItemId as order0id,

       User.Orders[1].ItemId as order1id

FROM complex\_json;

**Processing Log Using Hive**

Web Log Format:

This format is Apache’s common web log file format. It is contains total of 7 fields. Below is the regular expression that can parse the apache’s common web log format.

"^(\\S+)\\s+(\\S+)\\s+(\\S+)\\s+.(\\S+\\s+\\S+).\\s+.(\\S+)\\s+(\\S+)\\s+(\\S+.\\S+).\\s+(\\S+)\\s+(\\S+)$"

OR

"([^ ]\*) ([^ ]\*) ([^ ]\*) (-|\\[[^\\]]\*\\]) ([^ \"]\*|\"[^\"]\*\") (-|[0-9]\*) (-|[0-9]\*)"

HiveQL Query:

DROP TABLE IF EXISTS apache\_common\_log;

CREATE TABLE apache\_common\_log (

host STRING,

identity STRING,

user STRING,

time STRING,

request STRING,

status STRING,

size STRING

)

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'

WITH SERDEPROPERTIES (

"input.regex" = "([^ ]\*) ([^ ]\*) ([^ ]\*) (-|\\[[^\\]]\*\\]) ([^ \"]\*|\"[^\"]\*\") (-|[0-9]\*) (-|[0-9]\*)",

"output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s"

)

STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH "/home/siva/commonlog" INTO TABLE apache\_common\_log;

SELECT \* FROM apache\_common\_log ORDER BY time LIMIT 5;

**Hadoop Log Analyze:**

DROP TABLE IF EXISTS hadoop\_log;

CREATE TABLE hadoop\_log (

date1 STRING,

time1 STRING,

msgtype STRING,

classname STRING,

msgtext STRING

)

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'

WITH SERDEPROPERTIES (

"input.regex" = "^(\\d{4}-\\d{2}-\\d{2})\\s+(\\d{2}.\\d{2}.\\d{2}.\\d{3})\\s+(\\S+)\\s+(\\S+)\\s+(.\*)$",

"output.format.string" = "%1$s %2$s %3$s %4$s %5$s"

)

STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH "/home/user/hadooplog" INTO TABLE hadoop\_log;

SELECT date1, time1, msgtext FROM hadoop\_log WHERE msgtype='ERROR' OR msgtype='WARN' LIMIT 5;