UNIT-IV

INTRODUCTION TO HIVE AND PIG

The term 'Big Data' is used for collections of large datasets that include huge volume, high velocity, and a variety of data that is increasing day by day. Using traditional data management systems, it is difficult to process Big Data. Therefore, the Apache Software Foundation introduced a framework called Hadoop to solve Big Data management and processing challenges.

Hadoop

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

- **MapReduce:** It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.
- **HDFS:**Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

- **Sqoop:** It is used to import and export data to and from between HDFS and RDBMS.
- **Pig:** It is a procedural language platform used to develop a script for MapReduce operations.
- **Hive:** It is a platform used to develop SQL type scripts to do MapReduce operations.

Note: There are various ways to execute MapReduce operations:

- The traditional approach using Java MapReduce program for structured, semi-structured, and unstructured data.
- The scripting approach for MapReduce to process structured and semi structured data using Pig.
- The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

What is Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

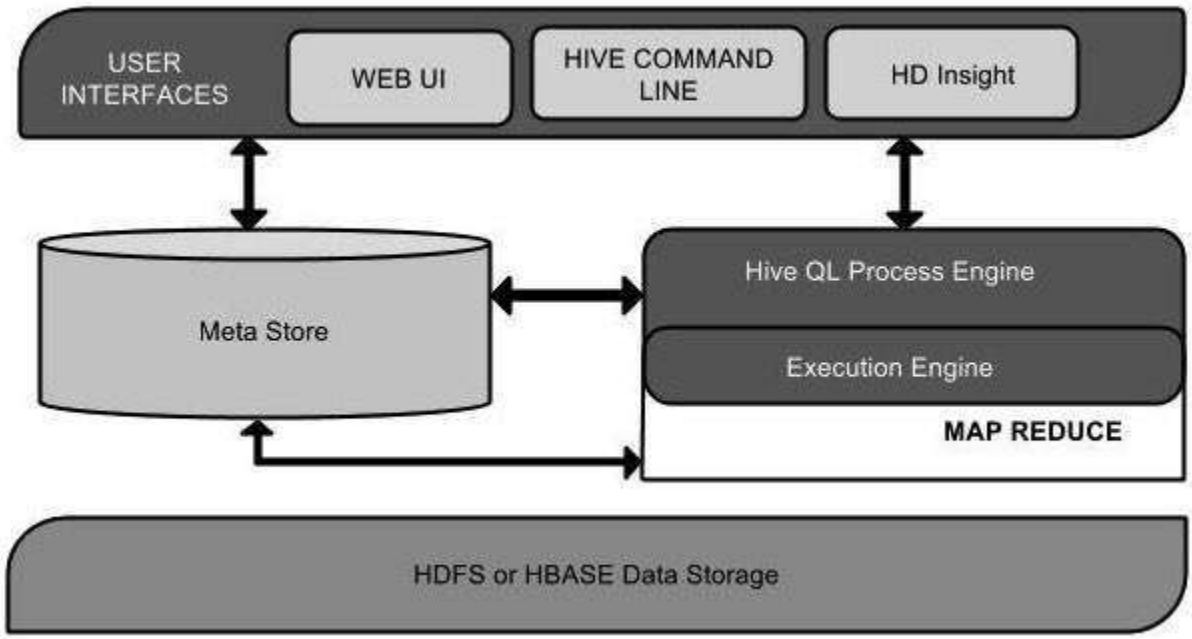
- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

Features of Hive

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP.
- It provides SQL type language for querying called HiveQL or HQL.
- It is familiar, fast, scalable, and extensible.

Architecture of Hive

The following component diagram depicts the architecture of Hive:

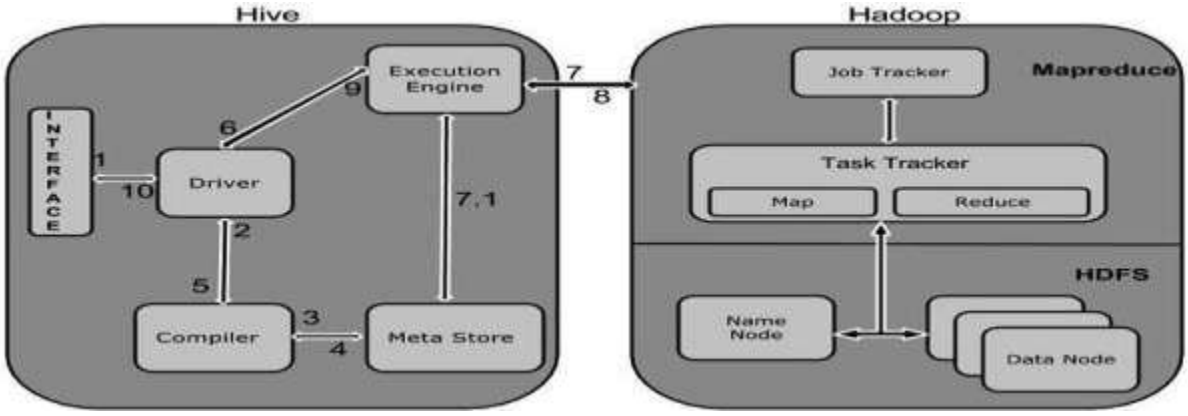


This component diagram contains different units. The following table describes each unit:

Unit Name	Operation
User Interface	Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation
1	**Execute Query** The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.
2	**Get Plan** The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	**Get Metadata** The compiler sends metadata request to Metastore (any database).
4	**Send Metadata** Metastore sends metadata as a response to the compiler.
5	**Send Plan** The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	**Execute Plan** The driver sends the execute plan to the execution engine.
7	**Execute Job** Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.
7.1	**Metadata Ops**

	Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	**Fetch Result** The execution engine receives the results from Data nodes.
9	**Send Results** The execution engine sends those resultant values to the driver.
10	**Send Results** The driver sends the results to Hive Interfaces.

File Formats in Hive

- File Format specifies how records are encoded in files
- Record Format implies how a stream of bytes for a given record are encoded
- The default file format is **TEXTFILE** – each record is a line in the file
- Hive uses different control characters as delimeters in textfiles
  - ^A (octal 001) , ^B(octal 002), ^C(octal 003), \n
- The term field is used when overriding the default delimiter
  - **FIELDS TERMINATED BY '\001'**
- Supports text files – csv, tsv
- TextFile can contain JSON or XML documents.

ommonly used File Formats –

1. **TextFile format**
   - Suitable for sharing data with other tools
   - Can be viewed/edited manually
2. **SequenceFile**
   - Flat files that stores binary key ,value pair
   - SequenceFile offers a Reader ,Writer, and Sorter classes for reading ,writing, and sorting respectively
   - Supports – Uncompressed, Record compressed (only value is compressed) and Block compressed (both key,value compressed) formats
3. **RCFile**
   - RCFile stores columns of a table in a record columnar way
4. **ORC**
5. **AVRO**

Hive Commands

Hive supports Data definition Language(DDL), Data Manipulation Language(DML) and User defined functions.

Hive DDL Commands

create database

drop database

create table

drop table

alter table

create index

create view

Hive DML Commands

Select

Where

Group By

Order By

Load Data

Join:

- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join

Hive DDL Commands

Create Database Statement

A database in Hive is a namespace or a collection of tables.

1. hive> CREATE SCHEMA userdb;
2. hive> SHOW DATABASES;

Drop database

1. ive> DROP DATABASE IF EXISTS userdb;

Creating Hive Tables

Create a table called Sonoo with two columns, the first being an integer and the other a string.

1. hive> CREATE TABLE Sonoo(foo INT, bar STRING);

Create a table called HIVE_TABLE with two columns and a partition column called ds. The partition column is a virtual column. It is not part of the data itself but is derived from the partition that a particular dataset is loaded into.By default, tables are assumed to be of text input format and the delimiters are assumed to be ^A(ctrl-a).

1. hive> CREATE TABLE HIVE_TABLE (foo INT, bar STRING) PARTITIONED BY (ds STRI NG);

Browse the table

1. hive> Show tables;

Altering and Dropping Tables

1. hive> ALTER TABLE Sonoo RENAME TO Kafka;
2. hive> ALTER TABLE Kafka ADD COLUMNS (col INT);
3. hive> ALTER TABLE HIVE_TABLE ADD COLUMNS (col1 INT COMMENT 'a comment');
4. hive> ALTER TABLE HIVE_TABLE REPLACE COLUMNS (col2 INT, weight STRING, baz INT COMMENT 'baz replaces new_col1');

Hive DML Commands

To understand the Hive DML commands, let's see the employee and employee_department table first.

Employee			Employee Department	
EMP ID	Emp Name	Address	Emp ID	Department
1	Rose	US	1	IT
2	Fred	US	2	IT
3	Jess	In	3	Eng
4	Frey	Th	4	Admin

LOAD DATA

1. hive> LOAD DATA LOCAL INPATH './usr/Desktop/kv1.txt' OVERWRITE INTO TABLE Employee;

SELECTS and FILTERS

1. hive> SELECT E.EMP_ID FROM Employee E WHERE E.Address='US';

GROUP BY

1. hive> hive> SELECT E.EMP_ID FROM Employee E GROUP BY E.Addresss;

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=’2012’)
> 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');
```

Hive Query Language

The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore. This chapter explains how to use the SELECT statement with WHERE clause.

SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. The built-in operators and functions generate an expression, which fulfils the condition.

Syntax

Given below is the syntax of the SELECT query:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
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];
```

Example

Let us take an example for SELECT…WHERE clause. Assume we have the employee table as given below, with fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 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> SELECT * FROM employee WHERE salary>30000;
```

On successful execution of the 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     |
+------+--------------+------------+-------------------+--------+
```

JDBC Program

The JDBC program to apply where clause for the given example is as follows.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class HiveQLWhere {
  private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

  public static void main(String[] args) throws SQLException {

    // Register driver and create driver instance
    Class.forName(driverName);

    // get connection
    Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "",
"");

    // create statement
    Statement stmt = con.createStatement();

    // execute statement
```

```
    Resultset res = stmt.executeQuery("SELECT * FROM employee WHERE salary>30000;");

    System.out.println("Result:");
    System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

    while (res.next()) {
      System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " +
res.getString(4) + " " + res.getString(5));
    }
    con.close();
  }
}
```

Save the program in a file named HiveQLWhere.java. Use the following commands to compile and execute this program.

```
$ javac HiveQLWhere.java
$ java HiveQLWhere
```

Output:

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

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.

```
+------+--------------+------------+------------------+--------+
| 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> SELECT Id, Name, Dept FROM employee ORDER BY DEPT;
```

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

```
+------+--------------+------------+------------------+--------+
| ID   | Name         | Salary     | Designation      | Dept   |
+------+--------------+------------+------------------+--------+
|1205  | Kranthi      | 30000      | Op Admin         | Admin  |
|1204  | Krian        | 40000      | Hr Admin         | HR     |
|1202  | Manisha      | 45000      | Proofreader      | PR     |
|1201  | Gopal        | 45000      | Technical manager | TP     |
|1203  | Masthanvali  | 40000      | Technical writer | TP     |
+------+--------------+------------+------------------+--------+
```

JDBC Program

Here is the JDBC program to apply Order By clause for the given example.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class HiveQLOrderBy {
  private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

  public static void main(String[] args) throws SQLException {

    // Register driver and create driver instance
    Class.forName(driverName);

    // get connection
```

```
    Connection  con  =  DriverManager.getConnection("jdbc:hive://localhost:10000/userdb",  "",
"");

    // create statement
    Statement stmt = con.createStatement();

    // execute statement
    Resultset res = stmt.executeQuery("SELECT * FROM employee ORDER BY DEPT;");
    System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

    while (res.next()) {
       System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " +
res.getString(4) + " " + res.getString(5));
    }

    con.close();
  }
}
```

Save the program in a file named HiveQLOrderBy.java. Use the following commands to compile and execute this program.

```
$ javac HiveQLOrderBy.java
$ java HiveQLOrderBy
```

Output:

```
ID      Name          Salary      Designation          Dept
1205    Kranthi        30000       Op Admin             Admin
1204    Krian          40000       Hr Admin             HR
1202    Manisha         45000       Proofreader          PR
1201    Gopal          45000       Technical manager    TP
1203    Masthanvali    40000       Technical writer     TP
1204    Krian          40000       Hr Admin             HR
```

The GROUP BY clause is used to group all the records in a result set using a particular collection column. It is used to query a group of records.

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.

```
+------+--------------+------------+-------------------+--------+
| ID   | Name         | Salary     | Designation       | Dept   |
+------+--------------+------------+-------------------+--------+
|1201  | Gopal        | 45000      | Technical manager | TP     |
|1202  | Manisha      | 45000      | Proofreader       | PR     |
|1203  | Masthanvali  | 40000      | Technical writer  | TP     |
|1204  | Krian        | 45000      | Proofreader       | PR     |
|1205  | Kranthi      | 30000      | Op Admin          | Admin  |
+------+--------------+------------+-------------------+--------+
```

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

```
hive> SELECT Dept,count(*) FROM employee GROUP BY DEPT;
```

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

```
+------+--------------+
| Dept | Count(*)     |
+------+--------------+
|Admin |    1         |
|PR    |    2         |
|TP    |    3         |
+------+--------------+
```

JDBC Program

Given below is the JDBC program to apply the Group By clause for the given example.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class HiveQLGroupBy {
  private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

  public static void main(String[] args) throws SQLException {

    // Register driver and create driver instance
    Class.forName(driverName);

    // get connection
```

```
    Connection con = DriverManager.
    getConnection("jdbc:hive://localhost:10000/userdb", "", "");

    // create statement
    Statement stmt = con.createStatement();

    // execute statement
    Resultset res = stmt.executeQuery("SELECT Dept,count(*) " + "FROM employee GROUP
BY DEPT; ");
    System.out.println(" Dept \t count(*)");

    while (res.next()) {
      System.out.println(res.getString(1) + " " + res.getInt(2));
    }
    con.close();
  }
}
```

Save the program in a file named HiveQLGroupBy.java. Use the following commands to compile and execute this program.

```
$ javac HiveQLGroupBy.java
$ java HiveQLGroupBy
```

Output:

```
Dept    Count(*)
Admin      1
PR         2
TP         3
```

JOIN 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.

Syntax

```
join_table:

  table_reference JOIN table_factor [join_condition]
  | table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference
  join_condition
  | table_reference LEFT SEMI JOIN table_reference join_condition
  | table_reference CROSS JOIN table_reference [join_condition]
```

Example

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 CUSTOMERS 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 |
+------+----------+--------+---------------------+
```

Bucketing

•Bucketing concept is based on (hashing function on the bucketed column) mod (by total number of buckets). The hash_function depends on the type of the bucketing column.

•Records with the same bucketed column will always be stored in the same bucket.

•We use CLUSTERED BY clause to divide the table into buckets.

•Physically, each bucket is just a file in the table directory, and Bucket numbering is 1-based.

•Bucketing can be done along with Partitioning on Hive tables and even without partitioning.

•Bucketed tables will create almost equally distributed data file parts, unless there is skew in data.

•Bucketing is enabled by setting hive.enforce.bucketing= true;

Advantages

•Bucketed tables offer efficient sampling than by non-bucketed tables. With sampling, we can try out queries on a fraction of data for testing and debugging purpose when the original data sets are very huge.

•As the data files are equal sized parts, map-side joins will be faster on bucketed tables than non-bucketed tables.

•Bucketing concept also provides the flexibility to keep the records in each bucket to be sorted by one or more columns. This makes map-side joins even more efficient, since the join of each bucket becomes an efficient merge-sort.

Bucketing Vs Partitioning

•Partitioning helps in elimination of data, if used in WHERE clause, where as bucketing helps in organizing data in each partition into multiple files, so that the same set of data is always written in same bucket.

•Bucketing helps a lot in joining of columns.

•Hive Bucket is nothing but another technique of decomposing data or decreasing the data into more manageable parts or equal parts.

Sampling

•TABLESAMPLE() gives more disordered and random records from a table as compared to LIMIT. •We can sample using the rand() function, which returns a random number.

SELECT * from users TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;

SELECT * from users TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;

•Here rand() refers to any random column. •The denominator in the bucket clause represents the number of buckets into which data will be hashed. •The numerator is the bucket number selected.

SELECT * from users TABLESAMPLE(BUCKET 2 OUT OF 4 ON name) s;

•If the columns specified in the TABLESAMPLE clause match the columns in the CLUSTERED BY clause, TABLESAMPLE queries only scan the required hash partitions of the table.

SELECT * FROM buck_users TABLESAMPLE(BUCKET 1 OUT OF 2 ON id) s LIMIT 1;

Joins and Types #

Reduce-Side Join

•If datasets are large, reduce side join takes place.

Map-Side Join

•In case one of the dataset is small, map side join takes place. •In map side join, a local job runs to create hash-table from content of HDFS file and sends it to every node.

SET hive.auto.convert.join =true;

Bucket Map Join

•The data must be bucketed on the keys used in the ON clause and the number of buckets for one table must be a multiple of the number of buckets for the other table. •When these conditions are met, Hive can join individual buckets between tables in the map phase, because it does not have to fetch the entire content of one table to match against each bucket in the other table. •set hive.optimize.bucketmapjoin =true; •SET hive.auto.convert.join =true;

SMBM Join

•Sort-Merge-Bucket (SMB) joins can be converted to SMB map joins as well.

•SMB joins are used wherever the tables are sorted and bucketed.

•The join boils down to just merging the already sorted tables, allowing this operation to be faster than an ordinary map-join.

•set hive.enforce.sortmergebucketmapjoin =false;

•set hive.auto.convert.sortmerge.join =true;

•set hive.optimize.bucketmapjoin = true;

•set hive.optimize.bucketmapjoin.sortedmerge = true;

LEFT SEMI JOIN

•A left semi-join returns records from the lefthand table if records are found in the righthand table that satisfy the ON predicates.

•It’s a special, optimized case of the more general inner join.

•Most SQL dialects support an IN … EXISTS construct to do the same thing.

•SELECT and WHERE clauses can’t reference columns from the righthand table.

•Right semi-joins are not supported in Hive.

•The reason semi-joins are more efficient than the more general inner join is as follows:

•For a given record in the lefthand table, Hive can stop looking for matching records in the righthand table as soon as any match is found.

•At that point, the selected columns from the lefthand table record can be projected

•A file format is a way in which information is stored or encoded in a computer file.

•In Hive it refers to how records are stored inside the file.

•InputFormat reads key-value pairs from files.

•As we are dealing with structured data, each record has to be its own structure.

•How records are encoded in a file defines a file format.

•These file formats mainly vary between data encoding, compression rate, usage of space and disk I/O.

•Hive does not verify whether the data that you are loading matches the schema for the table or not. •However, it verifies if the file format matches the table definition or not.

SerDe in Hive #

•The SerDe interface allows you to instruct Hive as to how a record should be processed.

•A SerDe is a combination of a Serializer and a Deserializer (hence, Ser-De).

•The Deserializer interface takes a string or binary representation of a record, and translates it into a Java object that Hive can manipulate.

•The Serializer, however, will take a Java object that Hive has been working with, and turn it into something that Hive can write to HDFS or another supported system.

•Commonly, Deserializers are used at query time to execute SELECT statements, and Serializers are used when writing data, such as through an INSERT-SELECT statement.

CSVSerDe

•Use ROW FORMAT SERDE ‘org.apache.hadoop.hive.serde2.OpenCSVSerde’

•Define following in SERDEPROPERTIES

(" separatorChar " = < value_of_separator

, " quoteChar " = < value_of_quote_character ,

" escapeChar “ = < value_of_escape_character

)

JSONSerDe

•Include hive-hcatalog-core-0.14.0.jar •Use ROW FORMAT SERDE ' org.apache.hive.hcatalog.data.JsonSerDe '

RegexSerDe

•It is used in case of pattern matching. •Use ROW FORMAT SERDE

'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'

•In SERDEPROPERTIES, define input pattern and output fields.

For Example

•input.regex = '(.)/(.)@(.*)' •output.format.string' = ' 1 s 2 s 3 s';

USE PARTITIONING AND BUCKETING

•Partitioning a table stores data in sub-directories categorized by table location, which allows Hive to exclude unnecessary data from queries without reading all the data every time a new query is made.

•Hive does support Dynamic Partitioning (DP) where column values are only known at EXECUTION TIME. To enable Dynamic Partitioning :

SET hive.exec.dynamic.partition =true;

•Another situation we want to protect against dynamic partition insert is that the user may accidentally specify all partitions to be dynamic partitions without specifying one static partition, while the original intention is to just overwrite the sub-partitions of one root partition.

SET hive.exec.dynamic.partition.mode =strict;

To enable bucketing:

SET hive.enforce.bucketing =true;

Optimizations in Hive #

•Use Denormalisation , Filtering and Projection as early as possible to reduce data before join.

•Join is a costly affair and requires extra map-reduce phase to accomplish query job. With De-normalisation, the data is present in the same table so there is no need for any joins, hence the selects are very fast.

•As join requires data to be shuffled across nodes, use filtering and projection as early as possible to reduce data before join.

TUNE CONFIGURATIONS

•To increase number of mapper, reduce split size :

SET mapred.max.split.size =1000000; (~1 MB)

•Compress map/reduce output

SET mapred.compress.map.output =true;

SET mapred.output.compress =true;

•Parallel execution

•Applies to MapReduce jobs that can run in parallel, for example jobs processing different source tables before a join.

SET hive.exec.parallel =true;

USE ORCFILE

•Hive supports ORCfile , a new table storage format that sports fantastic speed improvements through techniques like predicate push-down, compression and more.

•Using ORCFile for every HIVE table is extremely beneficial to get fast response times for your HIVE queries.

USE TEZ

•With Hadoop2 and Tez , the cost of job submission and scheduling is minimized.

•Also Tez does not restrict the job to be only Map followed by Reduce; this implies that all the query execution can be done in a single job without having to cross job boundaries.

•Let's look at an example. Consider a click-stream event table:

```
CREATE TABLE clicks (
timestamp date,
sessionID string,
url string,
source_ip string
)
STORED as ORC
tblproperties (" orc.compress " = "SNAPPY");
```

•Each record represents a click event, and we would like to find the latest URL for each sessionID

• One might consider the following approach:

SELECT clicks.sessionID, clicks.url FROM clicks inner join (select sessionID, max(timestamp) as max_ts from clicks group by sessionID) latest ON clicks.sessionID = latest.sessionID and clicks.timestamp = latest.max_ts;

•In the above query, we build a sub-query to collect the timestamp of the latest event in each session, and then use an inner join to filter out the rest.

•While the query is a reasonable solution —from a functional point of view— it turns out there's a better way to re-write this query as follows:

SELECT ranked_clicks.sessionID , ranked_clicks.url FROM (SELECT sessionID , url , RANK() over (partition by sessionID,order by timestamp desc) as rank FROM clicks) ranked_clicks WHERE ranked_clicks.rank =1;

•Here, we use Hive's OLAP functionality (OVER and RANK) to achieve the same thing, but without a Join.

•Clearly, removing an unnecessary join will almost always result in better performance, and when using big data this is more important than ever.

MAKING MULTIPLE PASS OVER SAME DATA

•Hive has a special syntax for producing multiple aggregations from a single pass through a source of data, rather than rescanning it for each aggregation.

•This change can save considerable processing time for large input data sets.

•For example, each of the following two queries creates a table from the same source table, history:

INSERT OVERWRITE TABLE sales

SELECT * FROM history WHERE action='purchased';

INSERT OVERWRITE TABLE credits

SELECT * FROM history WHERE action='returned';

Optimizations in Hive

•This syntax is correct, but inefficient.

•The following rewrite achieves the same thing, but using a single pass through the source history table:

FROM history

INSERT OVERWRITE sales SELECT * WHERE action='purchased'

INSERT OVERWRITE credits SELECT * WHERE action='returned';

What is Apache Pig

Apache Pig is a high-level data flow platform for executing MapReduce programs of Hadoop. The language used for Pig is Pig Latin.

The Pig scripts get internally converted to Map Reduce jobs and get executed on data stored in HDFS. Apart from that, Pig can also execute its job in Apache Tez or Apache Spark.

Pig can handle any type of data, i.e., structured, semi-structured or unstructured and stores the corresponding results into Hadoop Data File System. Every task which can be achieved using PIG can also be achieved using java used in MapReduce.

Features of Apache Pig

Let's see the various uses of Pig technology.

1) Ease of programming

Writing complex java programs for map reduce is quite tough for non-programmers. Pig makes this process easy. In the Pig, the queries are converted to MapReduce internally.

2) Optimization opportunities

It is how tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.

3) Extensibility

A user-defined function is written in which the user can write their logic to execute over the data set.

4) Flexible

It can easily handle structured as well as unstructured data.

5) In-built operators

It contains various type of operators such as sort, filter and joins.

Differences between Apache MapReduce and PIG

Advantages of Apache Pig

- Less code - The Pig consumes less line of code to perform any operation.
- Reusability - The Pig code is flexible enough to reuse again.

- Nested data types - The Pig provides a useful concept of nested data types like tuple, bag, and map.

Pig Latin

The Pig Latin is a data flow language used by Apache Pig to analyze the data in Hadoop. It is a textual language that abstracts the programming from the Java MapReduce idiom into a notation.

Pig Latin Statements

The Pig Latin statements are used to process the data. It is an operator that accepts a relation as an input and generates another relation as an output.

- It can span multiple lines.
- Each statement must end with a semi-colon.
- It may include expression and schemas.
- By default, these statements are processed using multi-query execution

Pig Latin Conventions

Convention	Description
( )	The parenthesis can enclose one or more items. It can also be used to indicate the tuple data type. Example - (10, xyz, (3,6,9))
[ ]	The straight brackets can enclose one or more items. It can also be used to indicate the map data type. Example - [INNER | OUTER]
{ }	The curly brackets enclose two or more items. It can also be used to indicate the bag data type Example - { block | nested_block }
...	The horizontal ellipsis points indicate that you can repeat a portion of the code. Example - cat path [path ...]

Latin Data Types

Simple Data Types

Type	Description
int	It defines the signed 32-bit integer. Example - 2
long	It defines the signed 64-bit integer. Example - 2L or 2l
float	It defines 32-bit floating point number. Example - 2.5F or 2.5f or 2.5e2f or 2.5.E2F
double	It defines 64-bit floating point number. Example - 2.5 or 2.5 or 2.5e2f or 2.5.E2F
chararray	It defines character array in Unicode UTF-8 format. Example - javatpoint
bytearray	It defines the byte array.
boolean	It defines the boolean type values. Example - true/false
datetime	It defines the values in datetime order. Example - 1970-01- 01T00:00:00.000+00:00
biginteger	It defines Java BigInteger values. Example - 5000000000000
bigdecimal	It defines Java BigDecimal values. Example - 52.232344535345

Pig Data Types

Apache Pig supports many data types. A list of Apache Pig Data Types with description and examples are given below.

Type	Description	Example
Int	Signed 32 bit integer	2
Long	Signed 64 bit integer	15L or 15l
Float	32 bit floating point	2.5f or 2.5F
Double	32 bit floating point	1.5 or 1.5e2 or 1.5E2
charArray	Character array	hello javatpoint
byteArray	BLOB(Byte array)	
tuple	Ordered set of fields	(12,43)
bag	Collection f tuples	{(12,43),(54,28)}
map	collection of tuples	[open#apache]

Apache Pig Execution Modes

You can run Apache Pig in two modes, namely, **Local Mode** and **HDFS mode**.

Local Mode

In this mode, all the files are installed and run from your local host and local file system. There is no need of Hadoop or HDFS. This mode is generally used for testing purpose.

MapReduce Mode

MapReduce mode is where we load or process the data that exists in the Hadoop File System (HDFS) using Apache Pig. In this mode, whenever we execute the Pig Latin statements to process the data, a MapReduce job is invoked in the back-end to perform a particular operation on the data that exists in the HDFS.

Apache Pig Execution Mechanisms

Apache Pig scripts can be executed in three ways, namely, interactive mode, batch mode, and embedded mode.

- **Interactive Mode** (Grunt shell) − You can run Apache Pig in interactive mode using the Grunt shell. In this shell, you can enter the Pig Latin statements and get the output (using Dump operator).
- **Batch Mode** (Script) − You can run Apache Pig in Batch mode by writing the Pig Latin script in a single file with **.pig** extension.
- **Embedded Mode** (UDF) − Apache Pig provides the provision of defining our own functions (**U**ser **D**efined **F**unctions) in programming languages such as Java, and using them in our script.
- Given below in the table are some frequently used Pig Commands.

Command	Function
load	Reads data from the system
Store	Writes data to file system
foreach	Applies expressions to each record and outputs one or more records
filter	Applies predicate and removes records that do not return true

Group/cogroup	Collects records with the same key from one or more inputs
join	Joins two or more inputs based on a key
order	Sorts records based on a key
distinct	Removes duplicate records
union	Merges data sets
split	Splits data into two or more sets based on filter conditions
stream	Sends all records through a user-provided binary
dump	Writes output to stdout

limit	Limits the number of records

Complex Types

Type	Description
tuple	It defines an ordered set of fields. Example - (15,12)
bag	It defines a collection of tuples. Example - {(15,12), (12,15)}
map	It defines a set of key-value pairs. Example - [open#apache]

Pig Latin – Relational Operations

The following table describes the relational operators of Pig Latin.

Operator	Description
Loading and Storing	
LOAD	To Load the data from the file system (local/HDFS) into a relation.
STORE	To save a relation to the file system (local/HDFS).
Filtering	

FILTER	To remove unwanted rows from a relation.
DISTINCT	To remove duplicate rows from a relation.
FOREACH, GENERATE	To generate data transformations based on columns of data.
STREAM	To transform a relation using an external program.
Grouping and Joining	
JOIN	To join two or more relations.
COGROUP	To group the data in two or more relations.
GROUP	To group the data in a single relation.
CROSS	To create the cross product of two or more relations.
Sorting	
ORDER	To arrange a relation in a sorted order based on one or more fields (ascending or descending).
LIMIT	To get a limited number of tuples from a relation.
Combining and Splitting	
UNION	To combine two or more relations into a single relation.
SPLIT	To split a single relation into two or more relations.

Diagnostic Operators	
DUMP	To print the contents of a relation on the console.
DESCRIBE	To describe the schema of a relation.
EXPLAIN	To view the logical, physical, or MapReduce execution plans to compute a relation.
ILLUSTRATE	To view the step-by-step execution of a series of statements.

Eval Functions

Given below is the list of **eval** functions provided by Apache Pig.

S.N.	Function & Description
1	AVG() To compute the average of the numerical values within a bag.
2	BagToString() To concatenate the elements of a bag into a string. While concatenating, we can place a delimiter between these values (optional).
3	CONCAT() To concatenate two or more expressions of same type.
4	COUNT() To get the number of elements in a bag, while counting the number of tuples in a bag.
5	COUNT_STAR()

	It is similar to the **COUNT()** function. It is used to get the number of elements in a bag.
6	DIFF() To compare two bags (fields) in a tuple.
7	IsEmpty() To check if a bag or map is empty.
8	MAX() To calculate the highest value for a column (numeric values or chararrays) in a single-column bag.
9	MIN() To get the minimum (lowest) value (numeric or chararray) for a certain column in a single-column bag.
10	PluckTuple() Using the Pig Latin **PluckTuple()** function, we can define a string Prefix and filter the columns in a relation that begin with the given prefix.
11	SIZE() To compute the number of elements based on any Pig data type.
12	SUBTRACT() To subtract two bags. It takes two bags as inputs and returns a bag which contains the tuples of the first bag that are not in the second bag.
13	SUM() To get the total of the numeric values of a column in a single-column bag.
14	TOKENIZE() To split a string (which contains a group of words) in a single tuple and return a bag which contains the output of the split operation.

Apache Pig provides extensive support for **U**ser **D**efined **F**unctions (UDF's). Using these UDF's, we can define our own functions and use them. The UDF support is provided in six programming languages, namely, Java, Jython, Python, JavaScript, Ruby and Groovy.

For writing UDF's, complete support is provided in Java and limited support is provided in all the remaining languages. Using Java, you can write UDF's involving all parts of the processing like data load/store, column transformation, and aggregation. Since Apache Pig has been written in Java, the UDF's written using Java language work efficiently compared to other languages.

In Apache Pig, we also have a Java repository for UDF's named **Piggybank**. Using Piggybank, we can access Java UDF's written by other users, and contribute our own UDF's.

Types of UDF's in Java

While writing UDF's using Java, we can create and use the following three types of functions −

- **Filter Functions** − The filter functions are used as conditions in filter statements. These functions accept a Pig value as input and return a Boolean value.
- **Eval Functions** − The Eval functions are used in FOREACH-GENERATE statements. These functions accept a Pig value as input and return a Pig result.
- **Algebraic Functions** − The Algebraic functions act on inner bags in a FOREACHGENERATE statement. These functions are used to perform full MapReduce operations on an inner bag.

Writing UDF's using Java

To write a UDF using Java, we have to integrate the jar file **Pig-0.15.0.jar**. In this section, we discuss how to write a sample UDF using Eclipse. Before proceeding further, make sure you have installed Eclipse and Maven in your system.

Follow the steps given below to write a UDF function −

- Open Eclipse and create a new project (say **myproject**).
- Convert the newly created project into a Maven project.
- Copy the following content in the pom.xml. This file contains the Maven dependencies for Apache Pig and Hadoop-core jar files.

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0http://maven.apache
.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>Pig_Udf</groupId>
  <artifactId>Pig_Udf</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <build>
    <sourceDirectory>src</sourceDirectory>
```

```
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>

    <dependency>
      <groupId>org.apache.pig</groupId>
      <artifactId>pig</artifactId>
      <version>0.15.0</version>
    </dependency>

    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-core</artifactId>
      <version>0.20.2</version>
    </dependency>

  </dependencies>

</project>
```

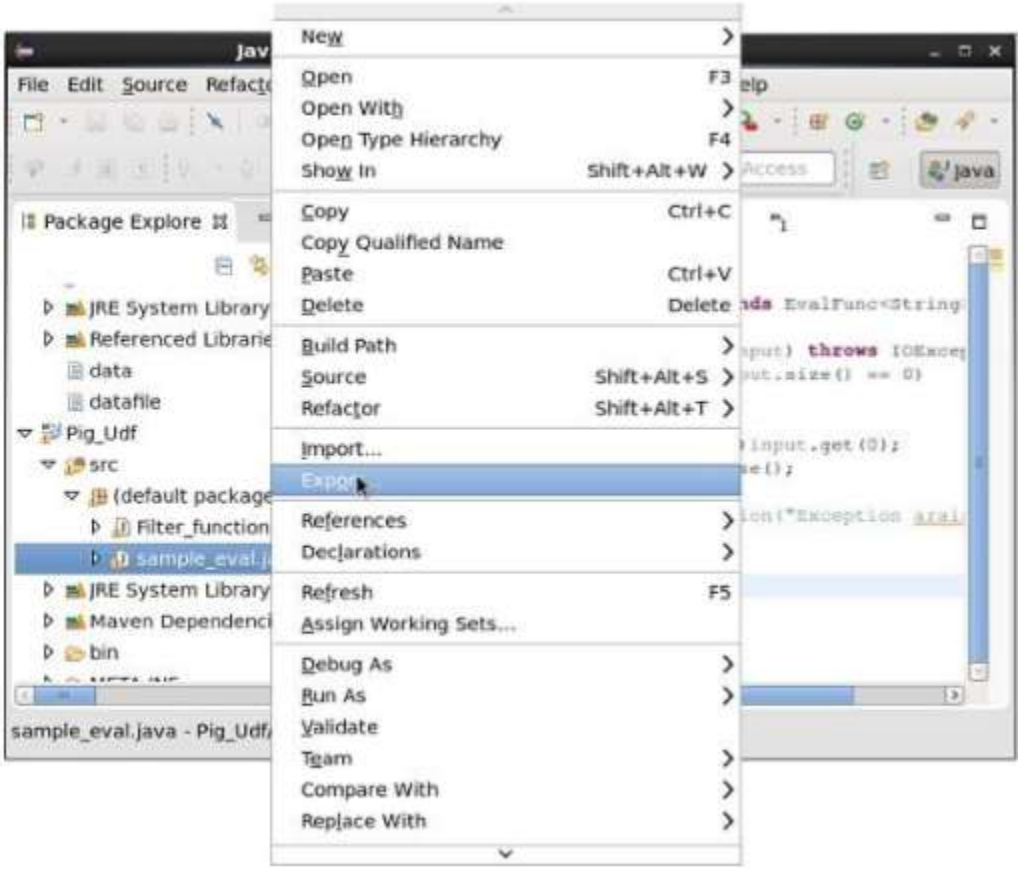
- Save the file and refresh it. In the **Maven Dependencies** section, you can find the downloaded jar files.
- Create a new class file with name **Sample_Eval** and copy the following content in it.

```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class Sample_Eval extends EvalFunc<String>{

  public String exec(Tuple input) throws IOException {
    if (input == null || input.size() == 0)
    return null;
    String str = (String)input.get(0);
    return str.toUpperCase();
```

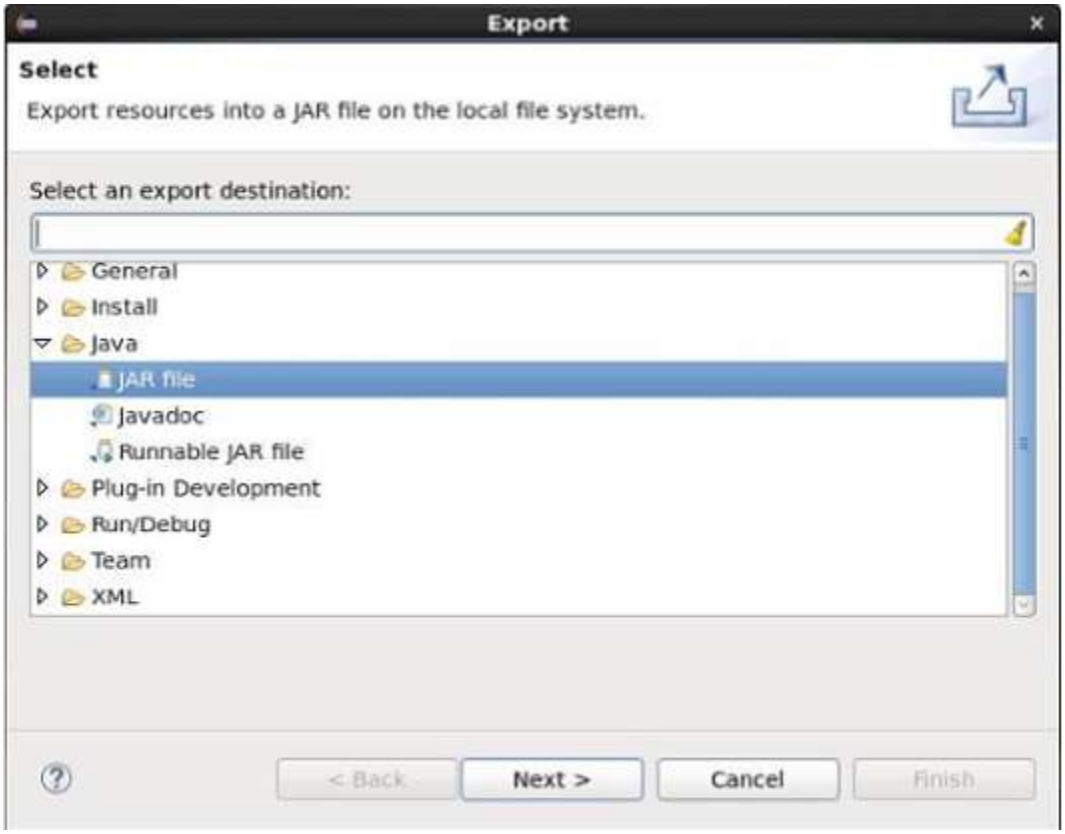
```
  }
}
```

While writing UDF's, it is mandatory to inherit the EvalFunc class and provide implementation to **exec()** function. Within this function, the code required for the UDF is written. In the above example, we have return the code to convert the contents of the given column to uppercase.

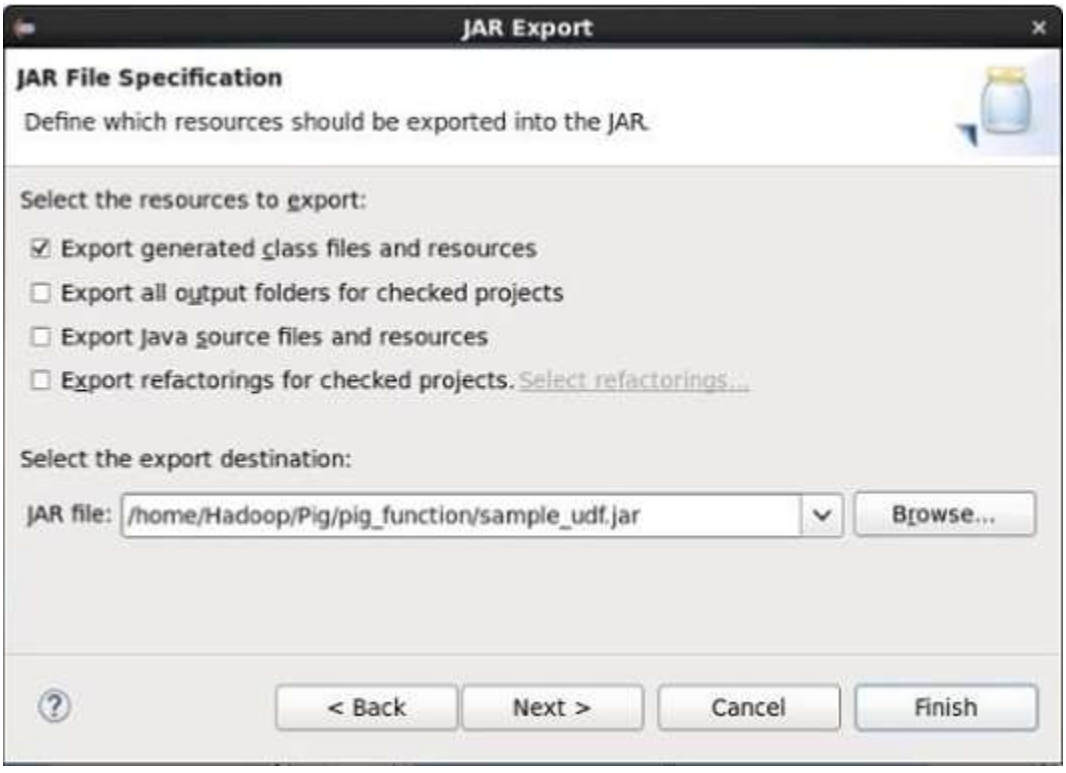
- After compiling the class without errors, right-click on the Sample_Eval.java file. It gives you a menu. Select **export** as shown in the following screenshot.



- On clicking **export**, you will get the following window. Click on **JAR file**.



- Proceed further by clicking **Next>** button. You will get another window where you need to enter the path in the local file system, where you need to store the jar file.



- Finally click the **Finish** button. In the specified folder, a Jar file **sample_udf.jar** is created. This jar file contains the UDF written in Java.

Using the UDF

After writing the UDF and generating the Jar file, follow the steps given below −

Step 1: Registering the Jar file

After writing UDF (in Java) we have to register the Jar file that contain the UDF using the Register operator. By registering the Jar file, users can intimate the location of the UDF to Apache Pig.

Syntax

Given below is the syntax of the Register operator.

```
REGISTER path;
```

Example

As an example let us register the sample_udf.jar created earlier in this chapter.

Start Apache Pig in local mode and register the jar file sample_udf.jar as shown below.

```
$cd PIG_HOME/bin
$./pig –x local

REGISTER '/$PIG_HOME/sample_udf.jar'
```

Note − assume the Jar file in the path − /$PIG_HOME/sample_udf.jar

Step 2: Defining Alias

After registering the UDF we can define an alias to it using the **Define** operator.

Syntax

Given below is the syntax of the Define operator.

```
DEFINE alias {function | [`command` [input] [output] [ship] [cache] [stderr] ] };
```

Example

Define the alias for sample_eval as shown below.

```
DEFINE sample_eval sample_eval();
```

Step 3: Using the UDF

After defining the alias you can use the UDF same as the built-in functions. Suppose there is a file named emp_data in the HDFS **/Pig_Data/** directory with the following content.

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuwaneshwar
006,Maggy,22,Chennai
```

007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuwaneshwar
012,Kelly,22,Chennai

And assume we have loaded this file into Pig as shown below.

```
grunt> emp_data = LOAD 'hdfs://localhost:9000/pig_data/emp1.txt' USING PigStorage(',')
  as (id:int, name:chararray, age:int, city:chararray);
```

Let us now convert the names of the employees in to upper case using the UDF **sample_eval**.

```
grunt> Upper_case = FOREACH emp_data GENERATE sample_eval(name);
```

Verify the contents of the relation **Upper_case** as shown below.

grunt> Dump Upper_case;

(ROBIN)
(BOB)
(MAYA)
(SARA)
(DAVID)
(MAGGY)
(ROBERT)
(SYAM)
(MARY)
(SARAN)
(STACY)
(KELLY)

Parameter substitution in Pig

Earlier I have discussed about writing reusable scripts using Apache Hive, now we see how to achieve same functionality using Pig Latin.

Pig Latin has an option called param, using this we can write dynamic scripts .

Assume ,we have a file called numbers with below data.

12

23

34

12

56

34

57

12

```
Numbers = load '/data/numbers' as (number:int);
specificNumber = filter numbers by number==12;
Dump specificNumber;
```

Usually we write above code in a file .let us assume we have written it in a file called numbers.pig

And we write code from file using

```
Pig –f /path/to/numbers.pig
```

Later if we want to see only numbers equals to 34, then we change second line to

```
specificNumber = filter numbers by number==34;
```

and we re-run the code using same command.
But Its not a good practice to touch the code in production ,so we can make this script dynamic by using –param option of Piglatin.
Whatever values we want to decide at the time of running we make them dynamic .now we want to decide number to be filtered at the time running job,we can write second line like below.

```
specificNumber = filter numbers by number==$dynanumber
```

and we run code like below.

```
Pig –param dynanumber=12  –f numbers.pig
```

Assume we even want to take path at the time of running script, now we write code like below

```
Numbers = load '$path' as (number:int);
specificNumber = filter numbers by number=='$ dynanumber';
```

```
Dump specificNumber;
```

And run like below

```
Pig –param path=/data/path –param dynanumber =34 –f numbers.pig
```

If you feel this code is missing readability, we can specify all these dynamic values in a file like below
##Dyna.params (file name)

```
Path = /data/numbers

dynanumber = 34
```

Then you can run script with param-file option like below.

```
Pig –param-file dyna.params –f numbers.pig
```

Pig Latin provides four different types of diagnostic operators −

- Dump operator
- Describe operator
- Explanation operator
- Illustration operator

Word Count Example Using Pig Script:

```
lines = LOAD '/user/hadoop/HDFS_File.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);
DUMP wordcount;
```

The above pig script, first splits each line into words using the **TOKENIZE** operator. The tokenize function creates a bag of words. Using the **FLATTEN** function, the bag is

converted into a tuple. In the third statement, the words are grouped together so that the count can be computed which is done in fourth statement.

Pig at Yahoo

Pig was initially developed by Yahoo! for its data scientists who were using Hadoop. It was incepted to focus mainly on analysis of large datasets rather than on writing mapper and reduce functions. This allowed users to focus on what they want to do rather than bothering with how its done. On top of this with Pig language you have the facility to write commands in other languages like Java, Python etc. Big applications that can be built on Pig Latin can be custom built for different companies to serve different tasks related to data management. Pig systemizes all the branches of data and relates it in a manner that when the time comes, filtering and searching data is checked efficiently and quickly.

Pig Versus Hive

Pig Vs Hive

Here are some basic difference between Hive and Pig which gives an idea of which to use depending on the type of data and purpose.

Pig	Hive
Used by Programmers and Researchers	Used by Analysts
Used for Programming	Used for Reporting
Procedural data-flow language	Declarative SQLish language
Works on the Client side of a Cluster	Works on the Server side of a Cluster
For Semi-Structured Data	For Structured Data

Why Go for Hive When Pig is There?

The tabular column below gives a comprehensive comparison between the two. The Hive can be used in places where partitions are necessary and when it is essential to define and create cross-language services for numerous languages.

Features	Hive	Pig
Language	SQL-like	PigLatin
Schemas/Types	Yes (explicit)	Yes (implicit)
Partitions	**Yes**	**No**
Server	Optional (Thrift)	No
User Defined Functions (UDF)	Yes (Java)	Yes (Java)
Custom Serializer/Deserializer	Yes	Yes
DFS Direct Access	Yes (implicit)	Yes (explicit)
Join/Order/Sort	Yes	Yes
Shell	Yes	Yes
Streaming	Yes	Yes
Web Interface	Yes	No
JDBC/ODBC	Yes (limited)	No

UNIT-V

Overview of machine learning (ML)

Machine learning is a branch in computer science that studies the design of algorithms that can learn. Typical machine learning tasks are concept learning, function learning or "predictive modeling", clustering and finding predictive patterns. These tasks are learned through available data that were observed through experiences or instructions, for example. Machine learning hopes that including the experience into its tasks will eventually improve the learning. The ultimate goal is to improve the learning in such a way that it becomes automatic, so that humans like ourselves don't need to interfere any more.

In **supervised learning** (SML), the learning algorithm is presented with labelled example inputs, where the labels indicate the desired output. SML itself is composed of **classification**, where the output is categorical, and **regression**, where the output is numerical.

In **unsupervised learning** (UML), no labels are provided, and the learning algorithm focuses solely on detecting structure in unlabelled input data.

Note that there are also **semi-supervised learning** approaches that use labelled data to inform unsupervised learning on the unlabelled data to identify and annotate new classes in the dataset (also called novelty detection).

Reinforcement learning, the learning algorithm performs a task using feedback from operating in a real or synthetic environment.

Broadly, there are 3 types of Machine Learning Algorithms

1. Supervised Learning

2. Unsupervised Learning

3. Reinforcement Learning: