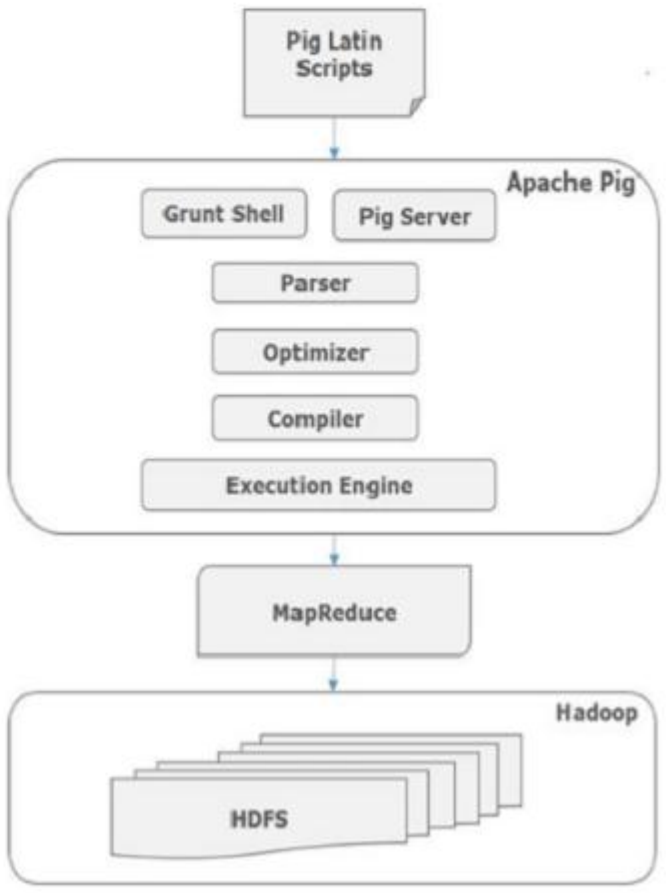
Unit -4 PIG

The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a highlevel data processing language which provides a rich set of data types and operators to perform various operations on the data.

To perform a particular task Programmers using Pig, programmers need to write a Pig script using the Pig Latin language, and execute them using any of the execution mechanisms (Grunt Shell, UDFs, Embedded). After execution, these scripts will go through a series of transformations applied by the Pig Framework, to produce the desired output.

Internally, Apache Pig converts these scripts into a series of MapReduce jobs, and thus, it makes the programmer's job easy. The architecture of Apache Pig is shown below.



Apache Pig Components

As shown in the figure, there are various components in the Apache Pig framework. Let us take a look at the major components.

Parser

Initially the Pig Scripts are handled by the Parser. It checks the syntax of the script, does type checking, and other miscellaneous checks. The output of the parser will be a DAG (directed acyclic graph), which represents the Pig Latin statements and logical operators.

In the DAG, the logical operators of the script are represented as the nodes and the data flows are represented as edges.

Optimizer

The logical plan (DAG) is passed to the logical optimizer, which carries out the logical optimizations such as projection and pushdown.

Compiler

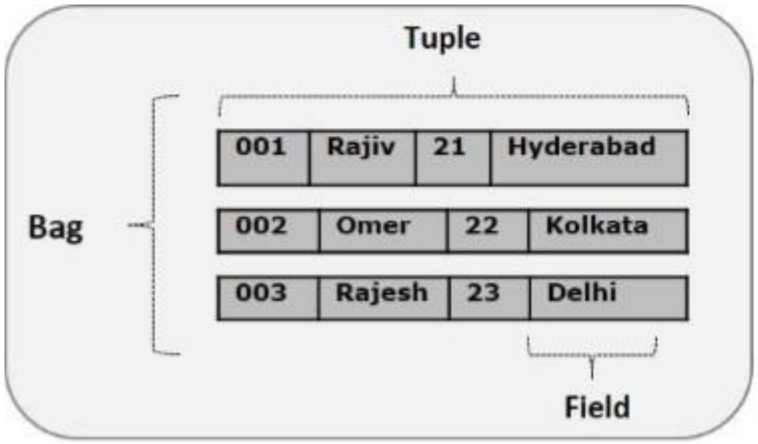
The compiler compiles the optimized logical plan into a series of MapReduce jobs.

Execution engine

Finally the MapReduce jobs are submitted to Hadoop in a sorted order. Finally, these MapReduce jobs are executed on Hadoop producing the desired results.

Pig Latin Data Model

The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as **map** and **tuple**. Given below is the diagrammatical representation of Pig Latin’s data model.



Atom

Any single value in Pig Latin, irrespective of their data, type is known as an**Atom**. It is stored as string and can be used as string and number. int, long, float, double, chararray, and bytearray are the atomic values of Pig. A piece of data or a simple atomic value is known as a **field**.

Example – 'raja' or '30'

Tuple

A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type. A tuple is similar to a row in a table of RDBMS.

Example – (Raja, 30)

Bag

A bag is an unordered set of tuples. In other words, a collection of tuples (non-unique) is known as a bag. Each tuple can have any number of fields (flexible schema). A bag is represented by '{}'. It is similar to a table in RDBMS, but unlike a table in RDBMS, it is not necessary that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

Example – {(Raja, 30), (Mohammad, 45)}

A bag can be a field in a relation; in that context, it is known as **inner bag**.

Example – {Raja, 30, **{9848022338, raja@gmail.com,}**}

Map

A map (or data map) is a set of key-value pairs. The **key** needs to be of type chararray and should be unique. The **value** might be of any type. It is represented by '[]'

Example – [name#Raja, age#30]

Relation

A relation is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

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.

Invoking the Grunt Shell

You can invoke the Grunt shell in a desired mode (local/MapReduce) using the**–x** option as shown below.

Local mode	MapReduce mode
Command – $ ./pig –x local	**Command –** $ ./pig -x mapreduce

Output –
 	
`15/09/28 10:13:03 INFO pig.Main:`
`Logging error messages to:`
`/home/Hadoop/pig_1443415383991.log`
`2015-09-28 10:13:04,838 [main]`
`INFO`
`org.apache.pig.backend.hadoop.execution`
`engine.HExecutionEngine - Connecting to`
`hadoop file system at: file:///`
 	
`grunt>`
	Output –

Either of these commands gives you the Grunt shell prompt as shown below.

```
grunt>
```

You can exit the Grunt shell using **'ctrl + d'.**

After invoking the Grunt shell, you can execute a Pig script by directly entering the Pig Latin statements in it.

```
grunt> customers = LOAD 'customers.txt' USING PigStorage(',');
```

Executing Apache Pig in Batch Mode

You can write an entire Pig Latin script in a file and execute it using the **–x command**. Let us suppose we have a Pig script in a file named**sample_script.pig** as shown below.

Sample_script.pig

```
student = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING

   PigStorage(',') as (id:int,name:chararray,city:chararray);


Dump student;
```

Now, you can execute the script in the above file as shown below.

Local mode	MapReduce mode
$ pig -x local **Sample_script.pig**	$ pig -x mapreduce **Sample_script.pig**

Note − We will discuss in detail how to run a Pig script in **Bach mode** and in**embedded mode** in subsequent chapters.

Pig Latin – Data types

Given below table describes the Pig Latin data types.

S.N.	Data Type	Description & Example
1	int	Represents a signed 32-bit integer. **Example** : 8

2	long	Represents a signed 64-bit integer. **Example** : 5L
3	float	Represents a signed 32-bit floating point. **Example** : 5.5F
4	double	Represents a 64-bit floating point. **Example** : 10.5
5	chararray	Represents a character array (string) in Unicode UTF-8 format. **Example** : ‘tutorials point’
6	Bytearray	Represents a Byte array (blob).
7	Boolean	Represents a Boolean value. **Example** : true/ false.
8	Datetime	Represents a date-time. **Example** : 1970-01-01T00:00:00.000+00:00
9	Biginteger	Represents a Java BigInteger. **Example** : 60708090709
10	Bigdecimal	Represents a Java BigDecimal **Example** : 185.98376256272893883

Complex Types		
11	Tuple	A tuple is an ordered set of fields. **Example** : (raja, 30)
12	Bag	A bag is a collection of tuples. **Example** : {(raju,30),(Mohhammad,45)}
13	Map	A Map is a set of key-value pairs. **Example** : [‘name’#’Raju’, ‘age’#30]

Null Values

Values for all the above data types can be NULL. Apache Pig treats null values in a similar way as SQL does.

A null can be an unknown value or a non-existent value. It is used as a placeholder for optional values. These nulls can occur naturally or can be the result of an operation.

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.

The Load Operator

You can load data into Apache Pig from the file system (HDFS/ Local) using**LOAD** operator of **Pig Latin**.

Syntax

The load statement consists of two parts divided by the "=" operator. On the left-hand side, we need to mention the name of the relation **where** we want to store the data, and on the right-hand side, we have to define **how** we store the data. Given below is the syntax of the **Load** operator.

```
Relation_name = LOAD 'Input file path' USING function as schema;
```

Where,

- **relation_name** – We have to mention the relation in which we want to store the data.

- **Input file path** – We have to mention the HDFS directory where the file is stored. (In MapReduce mode)
- **function** – We have to choose a function from the set of load functions provided by Apache Pig (**BinStorage, JsonLoader, PigStorage, TextLoader**).
- **Schema** – We have to define the schema of the data. We can define the required schema as follows –

```
(column1 : data type, column2 : data type, column3 : data type);
```

```
grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student_data.txt'
   USING PigStorage(',')
   as ( id:int, firstname:chararray, lastname:chararray, phone:chararray,
   city:chararray );
```

Apache Pig using the **Store** operator.

Syntax

Given below is the syntax of the Store statement.

```
STORE Relation_name INTO ' required_directory_path ' [USING function];
```

Example

```
grunt> STORE student INTO ' hdfs://localhost:9000/pig_Output/ ' USING PigStorage
(',');
```

Dump Operator

The **Dump** operator is used to run the Pig Latin statements and display the results on the screen. It is generally used for debugging Purpose.

Syntax

Given below is the syntax of the **Dump** operator.

```
grunt> Dump Relation_Name
```

Example

Assume we have a file **student_data.txt** in HDFS with the following content.

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

The **describe** operator is used to view the schema of a relation.

Syntax

The syntax of the **describe** operator is as follows −

```
grunt> Describe Relation_name
```

Example

Assume we have a file **student_data.txt** in HDFS with the following content.

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

The **explain** operator is used to display the logical, physical, and MapReduce execution plans of a relation.

Syntax

Given below is the syntax of the **explain** operator.

```
grunt> explain Relation_name;
```

Example

Assume we have a file **student_data.txt** in HDFS with the following content.

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

```
$ explain student;

2015-10-05 11:32:43,660 [main]
2015-10-05 11:32:43,660 [main] INFO  org.apache.pig.newplan.logical.optimizer
.LogicalPlanOptimizer -
{RULES_ENABLED=[AddForEach, ColumnMapKeyPrune, ConstantCalculator,
GroupByConstParallelSetter, LimitOptimizer, LoadTypeCastInserter, MergeFilter,
MergeForEach, PartitionFilterOptimizer, PredicatePushdownOptimizer,
PushDownForEachFlatten, PushUpFilter, SplitFilter, StreamTypeCastInserter]}
#-----------------------------------------------
# New Logical Plan:
#-----------------------------------------------
student: (Name: LOStore Schema:
id#31:int,firstname#32:chararray,lastname#33:chararray,phone#34:chararray,city#
35:chararray)
|
|---student: (Name: LOForEach Schema:
id#31:int,firstname#32:chararray,lastname#33:chararray,phone#34:chararray,city#
35:chararray)
    |   |
    |   (Name: LOGenerate[false,false,false,false,false] Schema:
id#31:int,firstname#32:chararray,lastname#33:chararray,phone#34:chararray,city#
35:chararray)ColumnPrune:InputUids=[34, 35, 32, 33,
31]ColumnPrune:OutputUids=[34, 35, 32, 33, 31]
    |   |   |
    |   |   (Name: Cast Type: int Uid: 31)
    |   |   |     |   |   |---id:(Name: Project Type: bytearray Uid: 31 Input: 0 Column: (*))
    |   |   |
    |   |   (Name: Cast Type: chararray Uid: 32)
    |   |   |
    |   |   |---firstname:(Name: Project Type: bytearray Uid: 32 Input: 1
Column: (*))
    |   |   |
    |   |   (Name: Cast Type: chararray Uid: 33)
    |   |   |
    |   |   |---lastname:(Name: Project Type: bytearray Uid: 33 Input: 2
          Column: (*))
    |   |   |
    |   |   (Name: Cast Type: chararray Uid: 34)
    |   |   |
    |   |   |---phone:(Name: Project Type: bytearray Uid: 34 Input: 3 Column:
(*))
    |   |   |
    |   |   (Name: Cast Type: chararray Uid: 35)
    |   |   |
    |   |   |---city:(Name: Project Type: bytearray Uid: 35 Input: 4 Column:
(*))
    |   |
    |   |---(Name: LOInnerLoad[0] Schema: id#31:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[1] Schema: firstname#32:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[2] Schema: lastname#33:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[3] Schema: phone#34:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[4] Schema: city#35:bytearray)
    |
    |---student: (Name: LOLoad Schema:
id#31:bytearray,firstname#32:bytearray,lastname#33:bytearray,phone#34:bytearray
,city#35:bytearray)RequiredFields:null
#-----------------------------------------------
# Physical Plan: #-----------------------------------------------
student: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-36
|
|---student: New For Each(false,false,false,false,false)[bag] - scope-35
    |   |
    |   Cast[int] - scope-21
    |   |
    |   |---Project[bytearray][0] - scope-20
    |   |
    |   Cast[chararray] - scope-24
    |   |
    |   |---Project[bytearray][1] - scope-23
    |   |
    |   Cast[chararray] - scope-27
    |   |
    |   |---Project[bytearray][2] - scope-26
    |   |
    |   Cast[chararray] - scope-30
    |   |
    |   |---Project[bytearray][3] - scope-29
    |   |
    |   Cast[chararray] - scope-33
    |   |
    |   |---Project[bytearray][4] - scope-32
    |
    |---student: Load(hdfs://localhost:9000/pig_data/student_data.txt:PigStorage(',')) -
scope19
2015-10-05 11:32:43,682 [main]
INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MRCompiler -
File concatenation threshold: 100 optimistic? false
2015-10-05 11:32:43,684 [main]
INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MultiQueryOp timizer -
MR plan size before optimization: 1 2015-10-05 11:32:43,685 [main]
INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.
MultiQueryOp timizer - MR plan size after optimization: 1
#--------------------------------------------------
# Map Reduce Plan
#--------------------------------------------------
MapReduce node scope-37
Map Plan
student: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-36
|
|---student: New For Each(false,false,false,false,false)[bag] - scope-35
    |   |
    |   Cast[int] - scope-21
    |   |
    |   |---Project[bytearray][0] - scope-20
    |   |
    |   Cast[chararray] - scope-24
    |   |
    |   |---Project[bytearray][1] - scope-23
    |   |
    |   Cast[chararray] - scope-27
    |   |
    |   |---Project[bytearray][2] - scope-26
    |   |
    |   Cast[chararray] - scope-30
    |   |
    |   |---Project[bytearray][3] - scope-29
    |   |
    |   Cast[chararray] - scope-33
    |   |
    |   |---Project[bytearray][4] - scope-32
    |
    |---student:
Load(hdfs://localhost:9000/pig_data/student_data.txt:PigStorage(',')) - scope
19-------- Global sort: false
 ----------------
```

The **illustrate** operator gives you the step-by-step execution of a sequence of statements.

Syntax

Given below is the syntax of the **illustrate** operator.

```
grunt> illustrate Relation_name;
```

Example

Assume we have a file **student_data.txt** in HDFS with the following content.

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

And we have read it into a relation **student** using the LOAD operator as shown below.

```
grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student_data.txt' USING
PigStorage(',')

   as ( id:int, firstname:chararray, lastname:chararray, phone:chararray,
city:chararray );
```

Now, let us illustrate the relation named student as shown below.

```
grunt> illustrate student;
```

Output

On executing the above statement, you will get the following output.

```
grunt> illustrate student;

INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.PigMapOnly$M ap - Aliases
being processed per job phase (AliasName[line,offset]): M: student[1,10] C:  R:
---------------------------------------------------------------------------------------------
|student | id:int | firstname:chararray | lastname:chararray | phone:chararray |
city:chararray |
---------------------------------------------------------------------------------------------
|        | 002    | siddarth            | Battacharya        | 9848022338      | Kolkata
|
-----------------------------------------------------------------------------------------
```

The **GROUP** operator is used to group the data in one or more relations. It collects the data having the same key.

Syntax

Given below is the syntax of the **group** operator.

```
grunt> Group_data = GROUP Relation_name BY age;
```

Example

Assume that we have a file named **student_details.txt** in the HDFS directory **/pig_data/** as shown below.

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

And we have loaded this file into Apache Pig with the relation name**student_details** as shown below.

```
grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student_details.txt'
USING PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray,
city:chararray);
```

Now, let us group the records/tuples in the relation by age as shown below.

```
grunt> group_data = GROUP student_details by age;
```

Verification

Verify the relation **group_data** using the **DUMP** operator as shown below.

```
grunt> Dump group_data;
```

Output

Then you will get output displaying the contents of the relation named**group_data** as shown below. Here you can observe that the resulting schema has two columns −

- One is **age**, by which we have grouped the relation.
- The other is a **bag**, which contains the group of tuples, student records with the respective age.

```
(21,{(4,Preethi,Agarwal,21,9848022330,Pune),(1,Rajiv,Reddy,21,9848022337,Hydera bad)})
(22,{(3,Rajesh,Khanna,22,9848022339,Delhi),(2,siddarth,Battacharya,22,984802233 8,Kolkata)})
(23,{(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,9848022336
,Bhuwaneshwar)})
(24,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022334, trivendram)
```

The **COGROUP** operator works more or less in the same way as the GROUPoperator. The only difference between the two operators is that the **group**operator is normally used with one relation, while the **cogroup** operator is used in statements involving two or more relations.

Grouping Two Relations using Cogroup

Assume that we have two files namely **student_details.txt** and**employee_details.txt** in the HDFS directory **/pig_data/** as shown below.

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
```

```
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

employee_details.txt

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

And we have loaded these files into Pig with the relation names**student_details** and **employee_details** respectively, as shown below.

```
grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student_details.txt'
USING PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray,
city:chararray);


grunt> employee_details = LOAD 'hdfs://localhost:9000/pig_data/employee_details.txt'
USING PigStorage(',')

   as (id:int, name:chararray, age:int, city:chararray);
```

Now, let us group the records/tuples of the relations **student_details** and**employee_details** with the key age, as shown below.

```
grunt> cogroup_data = COGROUP student_details by age, employee_details by age;
```

Verification

Verify the relation **cogroup_data** using the **DUMP** operator as shown below.

```
grunt> Dump cogroup_data;
```

Output

It will produce the following output, displaying the contents of the relation named **cogroup_data** as shown below.

```
(21,{(4,Preethi,Agarwal,21,9848022330,Pune), (1,Rajiv,Reddy,21,9848022337,Hyderabad)},
   {    })
(22,{ (3,Rajesh,Khanna,22,9848022339,Delhi), (2,siddarth,Battacharya,22,9848022338,Kolkata) },
   { (6,Maggy,22,Chennai),(1,Robin,22,newyork) })
(23,{(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,9848022336
,Bhuwaneshwar)},
   {(5,David,23,Bhuwaneshwar),(3,Maya,23,Tokyo),(2,BOB,23,Kolkata)})
(24,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022334, trivendram)},
   { })
(25,{   },
   {(4,Sara,25,London)})
```

The **JOIN** operator is used to combine records from two or more relations. While performing a join operation, we declare one (or a group of) tuple(s) from each relation, as keys. When these keys match, the two particular tuples are matched, else the records are dropped. Joins can be of the following types –

- Self-join
- Inner-join
- Outer-join – left join, right join, and full join

This chapter explains with examples how to use the join operator in Pig Latin. Assume that we have two files namely **customers.txt** and **orders.txt** in the**/pig_data/** directory of HDFS as shown below.

customers.txt

```
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
```

orders.txt

```
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
```

And we have loaded these two files into Pig with the relations **customers** and**orders** as shown below.

```
grunt> customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING
PigStorage(',')

   as (id:int, name:chararray, age:int, address:chararray, salary:int);


grunt> orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING
PigStorage(',')

   as (oid:int, date:chararray, customer_id:int, amount:int);
```

Let us now perform various Join operations on these two relations.

Self - join

Self-join is used to join a table with itself as if the table were two relations, temporarily renaming at least one relation.

Generally, in Apache Pig, to perform self-join, we will load the same data multiple times, under different aliases (names). Therefore let us load the contents of the file **customers.txt** as two tables as shown below.

```
grunt> customers1 = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING
PigStorage(',')

   as (id:int, name:chararray, age:int, address:chararray, salary:int);


grunt> customers2 = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING
PigStorage(',')

   as (id:int, name:chararray, age:int, address:chararray, salary:int);
```

Syntax

Given below is the syntax of performing **self-join** operation using the **JOIN**operator.

```
grunt> Relation3_name = JOIN Relation1_name BY key, Relation2_name BY key ;
```

Example

Let us perform **self-join** operation on the relation **customers**, by joining the two relations **customers1** and **customers2** as shown below.

```
grunt> customers3 = JOIN customers1 BY id, customers2 BY id;
```

Verification

Verify the relation **customers3** using the **DUMP** operator as shown below.

```
grunt> Dump customers3;
```

Output

It will produce the following output, displaying the contents of the relation**customers**.

```
(1,Ramesh,32,Ahmedabad,2000,1,Ramesh,32,Ahmedabad,2000)
(2,Khilan,25,Delhi,1500,2,Khilan,25,Delhi,1500)
(3,kaushik,23,Kota,2000,3,kaushik,23,Kota,2000)
(4,Chaitali,25,Mumbai,6500,4,Chaitali,25,Mumbai,6500)
(5,Hardik,27,Bhopal,8500,5,Hardik,27,Bhopal,8500)
(6,Komal,22,MP,4500,6,Komal,22,MP,4500)
(7,Muffy,24,Indore,10000,7,Muffy,24,Indore,10000)
```

Inner Join

Inner Join is used quite frequently; it is also referred to as **equijoin**. An inner join returns rows when there is a match in both tables.

It creates a new relation by combining column values of two relations (say A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, the column values for each matched pair of rows of A and B are combined into a result row.

Syntax

Here is the syntax of performing **inner join** operation using the **JOIN** operator.

```
grunt> result = JOIN relation1 BY columnname, relation2 BY columnname;
```

Example

Let us perform **inner join** operation on the two relations **customers** and**orders** as shown below.

```
grunt> coustomer_orders = JOIN customers BY id, orders BY customer_id;
```

Outer Join: Unlike inner join, **outer join** returns all the rows from at least one of the relations. An outer join operation is carried out in three ways –

- Left outer join
- Right outer join
- Full outer join

Left Outer Join

The **left outer Join** operation returns all rows from the left table, even if there are no matches in the right relation.

Syntax

Given below is the syntax of performing **left outer join** operation using the**JOIN** operator.

```
grunt> Relation3_name = JOIN Relation1_name BY id LEFT OUTER, Relation2_name BY customer_id;
```

Example

Let us perform left outer join operation on the two relations customers and orders as shown below.

```
grunt> outer_left = JOIN customers BY id LEFT OUTER, orders BY customer_id;
```

Verification

Verify the relation **outer_left** using the **DUMP** operator as shown below.

```
grunt> Dump outer_left;
```

Output

It will produce the following output, displaying the contents of the relation**outer_left**.

```
(1,Ramesh,32,Ahmedabad,2000,,,,)
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
(5,Hardik,27,Bhopal,8500,,,,)
(6,Komal,22,MP,4500,,,,)
(7,Muffy,24,Indore,10000,,,,)
```

Right Outer Join

The **right outer join** operation returns all rows from the right table, even if there are no matches in the left table.

Syntax

Given below is the syntax of performing **right outer join** operation using the**JOIN** operator.

```
grunt> outer_right = JOIN customers BY id RIGHT, orders BY customer_id;
```

Example

Let us perform **right outer join** operation on the two relations **customers** and**orders** as shown below.

```
grunt> outer_right = JOIN customers BY id RIGHT, orders BY customer_id;
```

Verification

Verify the relation **outer_right** using the **DUMP** operator as shown below.

```
grunt> Dump outer_right
```

Output

It will produce the following output, displaying the contents of the relation**outer_right**.

```
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
```

Full Outer Join

The **full outer join** operation returns rows when there is a match in one of the relations.

Syntax

Given below is the syntax of performing **full outer join** using the **JOIN**operator.

```
grunt> outer_full = JOIN customers BY id FULL OUTER, orders BY customer_id;
```

Example

Let us perform **full outer join** operation on the two relations **customers** and**orders** as shown below.

```
grunt> outer_full = JOIN customers BY id FULL OUTER, orders BY customer_id;
```

Verification

Verify the relation **outer_full** using the **DUMP** operator as shown below.

```
grun> Dump outer_full;
```

Output

It will produce the following output, displaying the contents of the relation**outer_full**.

```
(1,Ramesh,32,Ahmedabad,2000,,,,)
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
(5,Hardik,27,Bhopal,8500,,,,)
(6,Komal,22,MP,4500,,,,)
(7,Muffy,24,Indore,10000,,,,)
```

The **CROSS** operator computes the cross-product of two or more relations. This chapter explains with example how to use the cross operator in Pig Latin.

Syntax

Given below is the syntax of the **CROSS** operator.

```
grunt> Relation3_name = CROSS Relation1_name, Relation2_name;
```

Example

Assume that we have two files namely **customers.txt** and **orders.txt** in the**/pig_data/** directory of HDFS as shown below.

customers.txt

```
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
```

orders.txt

```
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
```

And we have loaded these two files into Pig with the relations **customers** and**orders** as shown below.

```
grunt> customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING
PigStorage(',')

   as (id:int, name:chararray, age:int, address:chararray, salary:int);


grunt> orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING
PigStorage(',')

   as (oid:int, date:chararray, customer_id:int, amount:int);
```

Let us now get the cross-product of these two relations using the **cross**operator on these two relations as shown below.

```
grunt> cross_data = CROSS customers, orders;
```

```
grunt> Dump cross_data;
```

Output

It will produce the following output, displaying the contents of the relation**cross_data**.

```
(7,Muffy,24,Indore,10000,103,2008-05-20 00:00:00,4,2060)
(7,Muffy,24,Indore,10000,101,2009-11-20 00:00:00,2,1560)
(7,Muffy,24,Indore,10000,100,2009-10-08 00:00:00,3,1500)
(7,Muffy,24,Indore,10000,102,2009-10-08 00:00:00,3,3000)
(6,Komal,22,MP,4500,103,2008-05-20 00:00:00,4,2060)
(6,Komal,22,MP,4500,101,2009-11-20 00:00:00,2,1560)
(6,Komal,22,MP,4500,100,2009-10-08 00:00:00,3,1500)
(6,Komal,22,MP,4500,102,2009-10-08 00:00:00,3,3000)
(5,Hardik,27,Bhopal,8500,103,2008-05-20 00:00:00,4,2060)
(5,Hardik,27,Bhopal,8500,101,2009-11-20 00:00:00,2,1560)
(5,Hardik,27,Bhopal,8500,100,2009-10-08 00:00:00,3,1500)
(5,Hardik,27,Bhopal,8500,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
(4,Chaitali,25,Mumbai,6500,101,2009-20 00:00:00,4,2060)
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(2,Khilan,25,Delhi,1500,100,2009-10-08 00:00:00,3,1500)
(2,Khilan,25,Delhi,1500,102,2009-10-08 00:00:00,3,3000)
```

```
(1,Ramesh,32,Ahmedabad,2000,103,2008-05-20 00:00:00,4,2060)
(1,Ramesh,32,Ahmedabad,2000,101,2009-11-20 00:00:00,2,1560)
(1,Ramesh,32,Ahmedabad,2000,100,2009-10-08 00:00:00,3,1500)
(1,Ramesh,32,Ahmedabad,2000,102,2009-10-08 00:00:00,3,3000)-11-20 00:00:00,2,1560)
(4,Chaitali,25,Mumbai,6500,100,2009-10-08 00:00:00,3,1500)
(4,Chaitali,25,Mumbai,6500,102,2009-10-08 00:00:00,3,3000)
(3,kaushik,23,Kota,2000,103,2008-05-20 00:00:00,4,2060)
(3,kaushik,23,Kota,2000,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(2,Khilan,25,Delhi,1500,103,2008-05-20 00:00:00,4,2060)
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(2,Khilan,25,Delhi,1500,100,2009-10-08 00:00:00,3,1500)
(2,Khilan,25,Delhi,1500,102,2009-10-08 00:00:00,3,3000)
(1,Ramesh,32,Ahmedabad,2000,103,2008-05-20 00:00:00,4,2060)
(1,Ramesh,32,Ahmedabad,2000,101,2009-11-20 00:00:00,2,1560)
(1,Ramesh,32,Ahmedabad,2000,100,2009-10-08 00:00:00,3,1500)
(1,Ramesh,32,Ahmedabad,2000,102,2009-10-08 00:00:00,3,3000)
```

The **UNION** operator of Pig Latin is used to merge the content of two relations. To perform UNION operation on two relations, their columns and domains must be identical.

Syntax

Given below is the syntax of the **UNION** operator.

```
grunt> Relation_name3 = UNION Relation_name1, Relation_name2;
```

Example

Assume that we have two files namely **student_data1.txt** and**student_data2.txt** in the **/pig_data/** directory of HDFS as shown below.

Student_data1.txt

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

Student_data2.txt

```
7,Komal,Nayak,9848022334,trivendram.
8,Bharathi,Nambiayar,9848022333,Chennai.
```

And we have loaded these two files into Pig with the relations **student1** and**student2** as shown below.

```
grunt> student1 = LOAD 'hdfs://localhost:9000/pig_data/student_data1.txt' USING
PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray, phone:chararray,
city:chararray);


grunt> student2 = LOAD 'hdfs://localhost:9000/pig_data/student_data2.txt' USING
PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray, phone:chararray,
city:chararray);
```

Let us now merge the contents of these two relations using the **UNION**operator as shown below.

```
grunt> student = UNION student1, student2;
```

Output

It will display the following output, displaying the contents of the relation**student**.

```
(1,Rajiv,Reddy,9848022337,Hyderabad) (2,siddarth,Battacharya,9848022338,Kolkata)
(3,Rajesh,Khanna,9848022339,Delhi)
(4,Preethi,Agarwal,9848022330,Pune)
(5,Trupthi,Mohanthy,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,9848022335,Chennai)
(7,Komal,Nayak,9848022334,trivendram)
(8,Bharathi,Nambiayar,9848022333,Chennai)
```

The **SPLIT** operator is used to split a relation into two or more relations.

Syntax

Given below is the syntax of the **SPLIT** operator.

```
grunt> SPLIT Relation1_name INTO Relation2_name IF (condition1), Relation2_name (condition2),
```

Example

Assume that we have a file named **student_details.txt** in the HDFS directory **/pig_data/** as shown below.

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

And we have loaded this file into Pig with the relation name **student_details**as shown below.

```
student_details = LOAD 'hdfs://localhost:9000/pig_data/student_details.txt' USING
PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray,
city:chararray);
```

Let us now split the relation into two, one listing the employees of age less than 23, and the other listing the employees having the age between 22 and 25.

```
SPLIT student_details into student_details1 if age<23, student_details2 if (22<age
and age>25);

grunt> Dump student_details1;


grunt> Dump student_details2;
```

Output

It will produce the following output, displaying the contents of the relations**student_details1** and **student_details2** respectively.

```
grunt> Dump student_details1;
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(4,Preethi,Agarwal,21,9848022330,Pune)

grunt> Dump student_details2;
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,23,9848022335,Chennai)
(7,Komal,Nayak,24,9848022334,trivendram)
```

```
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
```

The **FILTER** operator is used to select the required tuples from a relation based on a condition.

Syntax

Given below is the syntax of the **FILTER** operator.

```
grunt> Relation2_name = FILTER Relation1_name BY (condition);
```

Example

Assume that we have a file named **student_details.txt** in the HDFS directory **/pig_data/** as shown below.

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

And we have loaded this file into Pig with the relation name **student_details**as shown below.

```
grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student_details.txt'
USING PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray,
city:chararray);
```

Let us now use the Filter operator to get the details of the students who belong to the city Chennai.

```
filter_data = FILTER student_details BY city == 'Chennai';

grunt> Dump filter_data;
```

Output

It will produce the following output, displaying the contents of the relation**filter_data** as follows.

```
(6,Archana,Mishra,23,9848022335,Chennai)
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
```

The **DISTINCT** operator is used to remove redundant (duplicate) tuples from a relation.

Syntax

Given below is the syntax of the **DISTINCT** operator.

```
grunt> Relation_name2 = DISTINCT Relatin_name1;
```

Example

Assume that we have a file named **student_details.txt** in the HDFS directory**/pig_data/** as shown below.

student_details.txt

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai
006,Archana,Mishra,9848022335,Chennai
```

And we have loaded this file into Pig with the relation name **student_details**as shown below.

```
grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student_details.txt'
USING PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray, phone:chararray,
city:chararray);
```

Let us now remove the redundant (duplicate) tuples from the relation named**student_details** using the **DISTINCT** operator, and store it as another relation named **distinct_data** as shown below.

```
grunt> distinct_data = DISTINCT student_details;
```

```
grunt> Dump distinct_data;
```

Output

It will produce the following output, displaying the contents of the relation**distinct_data** as follows.

```
(1,Rajiv,Reddy,9848022337,Hyderabad)
(2,siddarth,Battacharya,9848022338,Kolkata)
(3,Rajesh,Khanna,9848022339,Delhi)
(4,Preethi,Agarwal,9848022330,Pune)
(5,Trupthi,Mohanthy,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,9848022335,Chennai)
```

The **FOREACH** operator is used to generate specified data transformations based on the column data.

Syntax

Given below is the syntax of **FOREACH** operator.

```
grunt> Relation_name2 = FOREACH Relatin_name1 GENERATE (required data);
```

Example

Assume that we have a file named **student_details.txt** in the HDFS directory**/pig_data/** as shown below.

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

And we have loaded this file into Pig with the relation name **student_details**as shown below.

```
grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student_details.txt'
USING PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray,age:int, phone:chararray,
city:chararray);
```

Let us now get the id, age, and city values of each student from the relation**student_details** and store it into another relation named **foreach_data** using the **foreach** operator as shown below.

```
grunt> foreach_data = FOREACH student_details GENERATE id,age,city;

grunt> Dump foreach_data;
```

Output

It will produce the following output, displaying the contents of the relation**foreach_data**.

```
(1,21,Hyderabad)
(2,22,Kolkata)
(3,22,Delhi)
(4,21,Pune)
(5,23,Bhuwaneshwar)
(6,23,Chennai)
(7,24,trivendram)
(8,24,Chennai)
```

The **ORDER BY** operator is used to display the contents of a relation in a sorted order based on one or more fields.

Syntax

Given below is the syntax of the **ORDER BY** operator.

```
grunt> Relation_name2 = ORDER Relatin_name1 BY (ASC|DESC);
```

Example

Assume that we have a file named **student_details.txt** in the HDFS directory**/pig_data/** as shown below.

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

And we have loaded this file into Pig with the relation name **student_details**as shown below.

```
grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student_details.txt'
USING PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray,age:int, phone:chararray,
city:chararray);
```

Let us now sort the relation in a descending order based on the age of the student and store it into another relation named **order_by_data** using the**ORDER BY** operator as shown below.

```
grunt> order_by_data = ORDER student_details BY age DESC;

grunt> Dump order_by_data;
```

Output

It will produce the following output, displaying the contents of the relation**order_by_data**.

```
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
(7,Komal,Nayak,24,9848022334,trivendram)
(6,Archana,Mishra,23,9848022335,Chennai)
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(4,Preethi,Agarwal,21,9848022330,Pune)
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
```

The **LIMIT** operator is used to get a limited number of tuples from a relation.

Syntax

Given below is the syntax of the **LIMIT** operator.

```
grunt> Result = LIMIT Relation_name required number of tuples;
```

Example

Assume that we have a file named **student_details.txt** in the HDFS directory**/pig_data/** as shown below.

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
```

```
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

And we have loaded this file into Pig with the relation name **student_details**as shown below.

```
grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student_details.txt'
USING PigStorage(',')

   as (id:int, firstname:chararray, lastname:chararray,age:int, phone:chararray,
city:chararray);
```

Now, let’s sort the relation in descending order based on the age of the student and store it into another relation named **limit_data** using the **ORDER BY**operator as shown below.

```
grunt> limit_data = LIMIT student_details 4;

grunt> Dump limit_data;
```

Output

It will produce the following output, displaying the contents of the relation**limit_data** as follows.

```
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(4,Preethi,Agarwal,21,9848022330,Pune)
```

Apache Pig provides various built-in functions namely **eval, load, store, math, string, bag** and **tuple** functions.

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.

The **Load** and **Store** functions in Apache Pig are used to determine how the data goes ad comes out of Pig. These functions are used with the load and store operators. Given below is the list of load and store functions available in Pig.

S.N.	Function & Description
1	**PigStorage()** To load and store structured files.
2	**TextLoader()** To load unstructured data into Pig.
3	**BinStorage()**

	To load and store data into Pig using machine readable format.
4	**Handling Compression** In Pig Latin, we can load and store compressed data.

Given below is the list of Bag and Tuple functions.

S.N.	Function & Description
1	**TOBAG()** To convert two or more expressions into a bag.
2	**TOP()** To get the top **N** tuples of a relation.
3	**TOTUPLE()** To convert one or more expressions into a tuple.
4	**TOMAP()** To convert the key-value pairs into a Map.

Apache Pig - User Defined Functions

In addition to the built-in functions, 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">
```

```xml
<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>
```

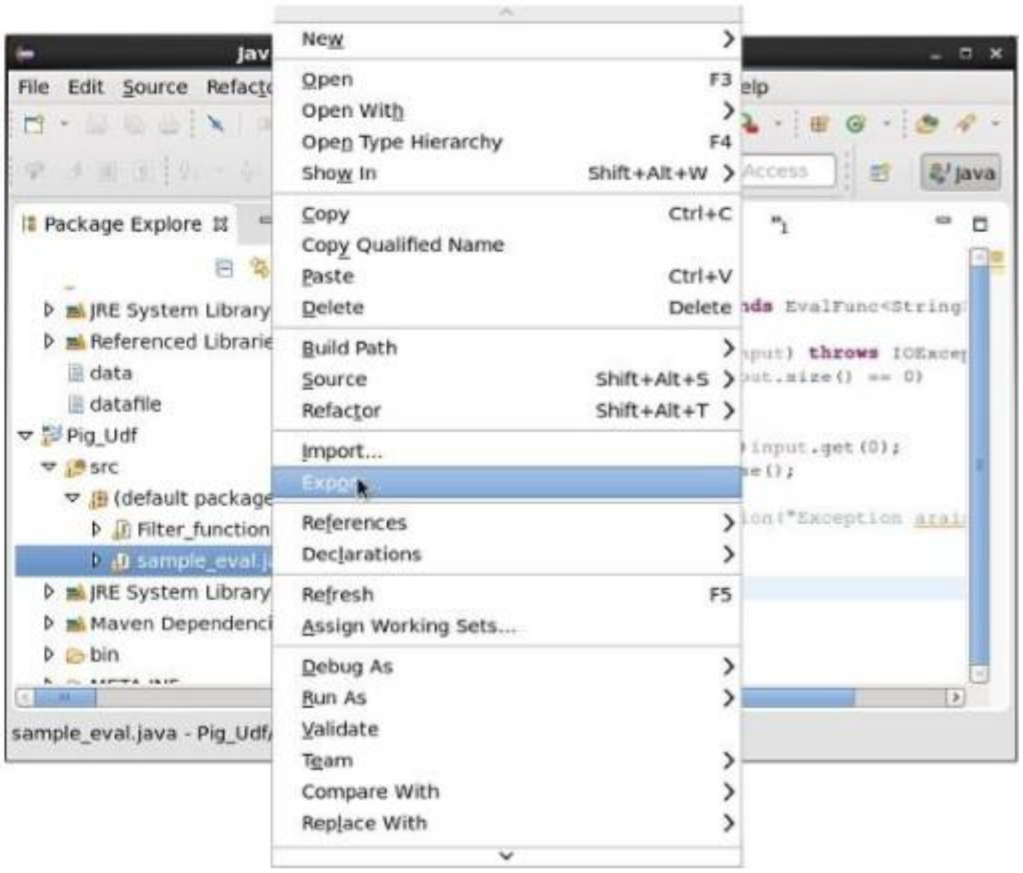
- Save the file and refresh it. In the **Maven Dependencies** section, you can find the downloaded jar files.

- Create a new class file with name **Sample_Eval** and copy the following content in it.

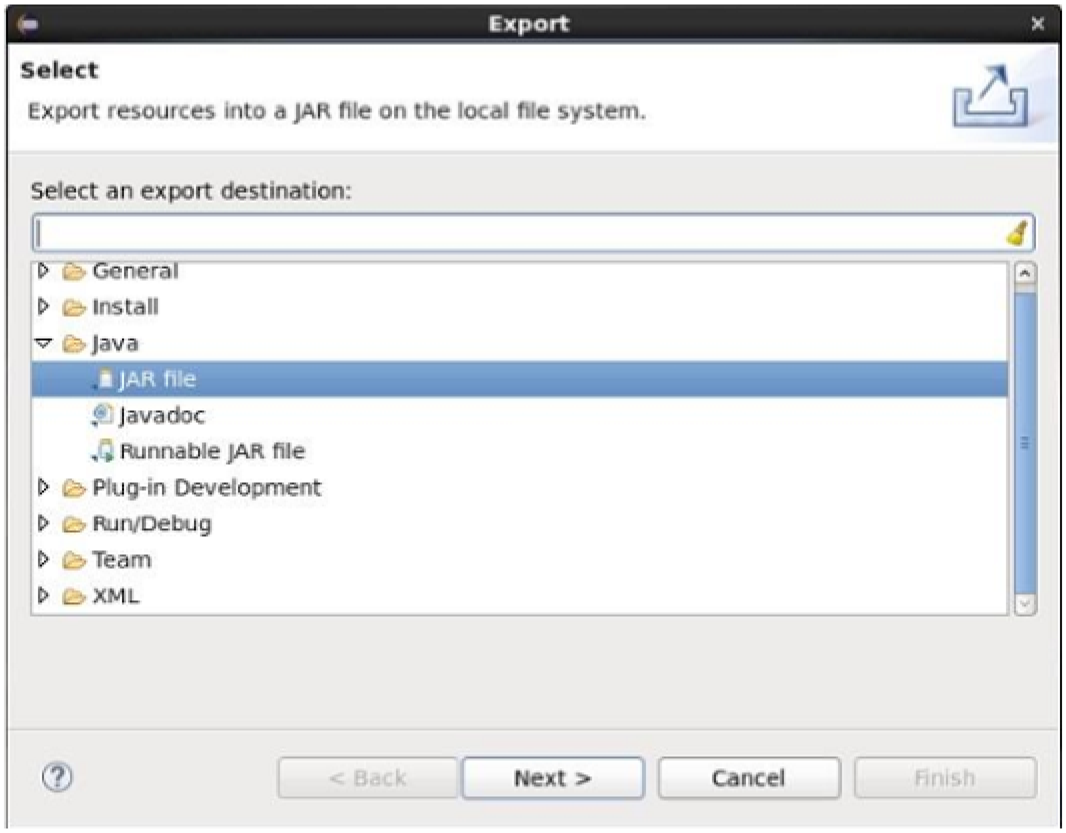
```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

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.

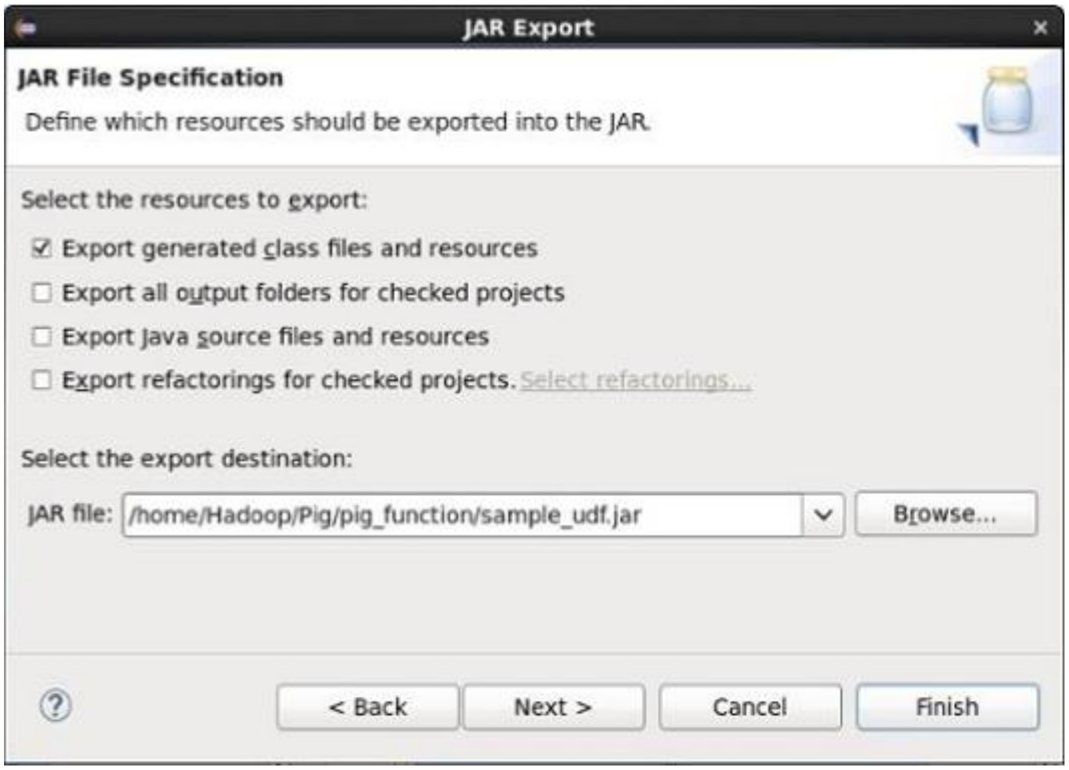
- After compiling the class without errors, right-click on the Sample_Eval.java file. It gives you a menu. Select **export** as shown in the following screenshot.



- 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.


JAR Export

JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

☑ Export generated class files and resources

☐ Export all output folders for checked projects

☐ Export Java source files and resources

☐ Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: /home/Hadoop/Pig/pig_function/sample_udf.jar

- 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)
```