Apache Pig is an abstraction over MapReduce. It is a tool/platform which is used to analyze larger sets of data representing them as data flows. Pig is generally used with **Hadoop**; we can perform all the data manipulation operations in Hadoop using Pig.

To write data analysis programs, Pig provides a high-level language known as **Pig Latin**. This language provides various operators using which programmers can develop their own functions for reading, writing, and processing data.

To analyze data using **Apache Pig**, programmers need to write scripts using Pig Latin language. All these scripts are internally converted to Map and Reduce tasks. Apache Pig has a component known as **Pig Engine** that accepts the Pig Latin scripts as input and converts those scripts into MapReduce jobs.

Why Do We Need Apache Pig?

Programmers who are not so good at Java normally used to struggle working with Hadoop, especially while performing any MapReduce tasks. Apache Pig is a boon for all such programmers.

* Using **Pig Latin**, programmers can perform MapReduce tasks easily without having to type complex codes in Java.
* Apache Pig uses **multi-query approach**, thereby reducing the length of codes. For example, an operation that would require you to type 200 lines of code (LoC) in Java can be easily done by typing as less as just 10 LoC in Apache Pig. Ultimately Apache Pig reduces the development time by almost 16 times.
* Pig Latin is **SQL-like language** and it is easy to learn Apache Pig when you are familiar with SQL.
* Apache Pig provides many built-in operators to support data operations like joins, filters, ordering, etc. In addition, it also provides nested data types like tuples, bags, and maps that are missing from MapReduce.

Features of Pig

Apache Pig comes with the following features −

* **Rich set of operators** − It provides many operators to perform operations like join, sort, filer, etc.
* **Ease of programming** − Pig Latin is similar to SQL and it is easy to write a Pig script if you are good at SQL.
* **Optimization opportunities** − The tasks in Apache Pig optimize their execution automatically, so the programmers need to focus only on semantics of the language.
* **Extensibility** − Using the existing operators, users can develop their own functions to read, process, and write data.
* **UDF’s** − Pig provides the facility to create **User-defined Functions** in other programming languages such as Java and invoke or embed them in Pig Scripts.
* **Handles all kinds of data** − Apache Pig analyzes all kinds of data, both structured as well as unstructured. It stores the results in HDFS.

## Apache Pig Vs MapReduce

Listed below are the major differences between Apache Pig and MapReduce.

|  |  |
| --- | --- |
| **Apache Pig** | **MapReduce** |
| Apache Pig is a data flow language. | MapReduce is a data processing paradigm. |
| It is a high level language. | MapReduce is low level and rigid. |
| Performing a Join operation in Apache Pig is pretty simple. | It is quite difficult in MapReduce to perform a Join operation between datasets. |
| Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig. | Exposure to Java is must to work with MapReduce. |
| Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent. | MapReduce will require almost 20 times more the number of lines to perform the same task. |
| There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job. | MapReduce jobs have a long compilation process. |

## Apache Pig Vs SQL

Listed below are the major differences between Apache Pig and SQL.

|  |  |
| --- | --- |
| **Pig** | **SQL** |
| Pig Latin is a **procedural** language. | SQL is a **declarative** language. |
| In Apache Pig, **schema** is optional. We can store data without designing a schema (values are stored as $01, $02 etc.) | Schema is mandatory in SQL. |
| The data model in Apache Pig is **nested relational**. | The data model used in SQL **is flat relational**. |
| Apache Pig provides limited opportunity for **Query optimization**. | There is more opportunity for query optimization in SQL. |

In addition to above differences, Apache Pig Latin −

* Allows splits in the pipeline.
* Allows developers to store data anywhere in the pipeline.
* Declares execution plans.
* Provides operators to perform ETL (Extract, Transform, and Load) functions.

Apache Pig Vs Hive

Both Apache Pig and Hive are used to create MapReduce jobs. And in some cases, Hive operates on HDFS in a similar way Apache Pig does. In the following table, we have listed a few significant points that set Apache Pig apart from Hive.

|  |  |
| --- | --- |
| **Apache Pig** | **Hive** |
| Apache Pig uses a language called **Pig Latin**. It was originally created at **Yahoo**. | Hive uses a language called **HiveQL**. It was originally created at **Facebook**. |
| Pig Latin is a data flow language. | HiveQL is a query processing language. |
| Pig Latin is a procedural language and it fits in pipeline paradigm. | HiveQL is a declarative language. |
| Apache Pig can handle structured, unstructured, and semi-structured data. | Hive is mostly for structured data. |

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.



### **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**

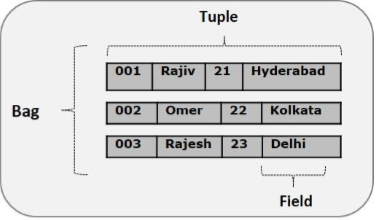
The compiler compiles the optimized logical plan into a series of MapReduce jobs.

### **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 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 |

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** |

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 – Arithmetic Operators

The following table describes the arithmetic operators of Pig Latin. Suppose a = 10 and b = 20.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | **Addition** − Adds values on either side of the operator | a + b will give 30 |
| − | **Subtraction** − Subtracts right hand operand from left hand operand | a − b will give −10 |
| \* | **Multiplication** − Multiplies values on either side of the operator | a \* b will give 200 |
| / | **Division** − Divides left hand operand by right hand operand | b / a will give 2 |
| % | **Modulus** − Divides left hand operand by right hand operand and returns remainder | b % a will give 0 |
| ? : | **Bincond** − Evaluates the Boolean operators. It has three operands as shown below.  variable **x** = (expression) ? **value1** *if true* : **value2** *if false*. | b = (a == 1)? 20: 30;  if a = 1 the value of b is 20.  if a!=1 the value of b is 30. |
| CASE  WHEN  THEN  ELSE END | **Case** − The case operator is equivalent to nested bincond operator. | CASE f2 % 2  WHEN 0 THEN 'even'  WHEN 1 THEN 'odd'  END |

Pig Latin – Comparison Operators

The following table describes the comparison operators of Pig Latin.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | **Equal** − Checks if the values of two operands are equal or not; if yes, then the condition becomes true. | (a = b) is not true |
| != | **Not Equal** − Checks if the values of two operands are equal or not. If the values are not equal, then condition becomes true. | (a != b) is true. |
| > | **Greater than** − Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true. | (a > b) is not true. |
| < | **Less than** − Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true. | (a < b) is true. |
| >= | **Greater than or equal to** − Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true. | (a >= b) is not true. |
| <= | **Less than or equal to** − Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true. | (a <= b) is true. |
| matches | **Pattern matching** − Checks whether the string in the left-hand side matches with the constant in the right-hand side. | f1 matches '.\*tutorial.\*' |

Pig Latin – Type Construction Operators

The following table describes the Type construction operators of Pig Latin.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| () | **Tuple constructor operator** − This operator is used to construct a tuple. | (Raju, 30) |
| {} | **Bag constructor operator** − This operator is used to construct a bag. | {(Raju, 30), (Mohammad, 45)} |
| [] | **Map constructor operator** − This operator is used to construct a tuple. | [name#Raja, age#30] |

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 **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.

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 describe the relation named **student** and verify the schema as shown below.

grunt> describe student;

## Output

Once you execute the above **Pig Latin** statement, it will produce the following output.

grunt> student: { id: int,firstname: chararray,lastname: chararray,phone: chararray,city: chararray }

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.

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 explain the relation named student using the **explain** operator as shown below.

grunt> explain student;

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)})

You can see the schema of the table after grouping the data using the **describe** command as shown below.

**grunt> Describe group\_data;**

group\_data: {group: int,student\_details: {(id: int,firstname: chararray,

lastname: chararray,age: int,phone: chararray,city: chararray)}}

In the same way, you can get the sample illustration of the schema using the **illustrate** command as shown below.

$ Illustrate group\_data;

It will produce the following output −

-------------------------------------------------------------------------------------------------

|group\_data| group:int | student\_details:bag{:tuple(id:int,firstname:chararray,lastname:chararray,age:int,phone:chararray,city:chararray)}|

-------------------------------------------------------------------------------------------------

| | 21 | { 4, Preethi, Agarwal, 21, 9848022330, Pune), (1, Rajiv, Reddy, 21, 9848022337, Hyderabad)}|

| | 2 | {(2,siddarth,Battacharya,22,9848022338,Kolkata),(003,Rajesh,Khanna,22,9848022339,Delhi)}|

-------------------------------------------------------------------------------------------------

## Grouping by Multiple Columns

Let us group the relation by age and city as shown below.

grunt> group\_multiple = GROUP student\_details by (age, city);

You can verify the content of the relation named **group\_multiple** using the Dump operator as shown below.

**grunt> Dump group\_multiple;**

((21,Pune),{(4,Preethi,Agarwal,21,9848022330,Pune)})

((21,Hyderabad),{(1,Rajiv,Reddy,21,9848022337,Hyderabad)})

((22,Delhi),{(3,Rajesh,Khanna,22,9848022339,Delhi)})

((22,Kolkata),{(2,siddarth,Battacharya,22,9848022338,Kolkata)})

((23,Chennai),{(6,Archana,Mishra,23,9848022335,Chennai)})

((23,Bhuwaneshwar),{(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)})

((24,Chennai),{(8,Bharathi,Nambiayar,24,9848022333,Chennai)})

(24,trivendram),{(7,Komal,Nayak,24,9848022334,trivendram)})

## Group All

You can group a relation by all the columns as shown below.

grunt> **group\_all** = GROUP **student\_details** All;

Now, verify the content of the relation **group\_all** as shown below.

**grunt> Dump group\_all;**

(all,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022334 ,trivendram),

(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,9848022336,Bhuw aneshwar),

(4,Preethi,Agarwal,21,9848022330,Pune),(3,Rajesh,Khanna,22,9848022339,Delhi),

(2,siddarth,Battacharya,22,9848022338,Kolkata),(1,Rajiv,Reddy,21,9848022337,Hyd erabad)})

The **COGROUP** operator works more or less in the same way as the [GROUP](https://www.tutorialspoint.com/apache_pig/apache_pig_group_operator.htm) operator. 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 **cogroup** operator groups the tuples from each relation according to age where each group depicts a particular age value.

For example, if we consider the 1st tuple of the result, it is grouped by age 21. And it contains two bags −

* the first bag holds all the tuples from the first relation (**student\_details** in this case) having age 21, and
* the second bag contains all the tuples from the second relation (**employee\_details** in this case) having age 21.

In case a relation doesn’t have tuples having the age value 21, it returns an empty bag.

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;

### **Verification**

Verify the relation **coustomer\_orders** using the **DUMP** operator as shown below.

grunt> Dump coustomer\_orders;

### **Output**

You will get the following output that will the contents of the relation named **coustomer\_orders**.

(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)

**Note** −

*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,,,,)

## Using Multiple Keys

We can perform JOIN operation using multiple keys.

### **Syntax**

Here is how you can perform a JOIN operation on two tables using multiple keys.

grunt> Relation3\_name = JOIN Relation2\_name BY (key1, key2), Relation3\_name BY (key1, key2);

Assume that we have two files namely **employee.txt** and **employee\_contact.txt** in the **/pig\_data/** directory of HDFS as shown below.

**employee.txt**

001,Rajiv,Reddy,21,programmer,003

002,siddarth,Battacharya,22,programmer,003

003,Rajesh,Khanna,22,programmer,003

004,Preethi,Agarwal,21,programmer,003

005,Trupthi,Mohanthy,23,programmer,003

006,Archana,Mishra,23,programmer,003

007,Komal,Nayak,24,teamlead,002

008,Bharathi,Nambiayar,24,manager,001

**employee\_contact.txt**

001,9848022337,Rajiv@gmail.com,Hyderabad,003

002,9848022338,siddarth@gmail.com,Kolkata,003

003,9848022339,Rajesh@gmail.com,Delhi,003

004,9848022330,Preethi@gmail.com,Pune,003

005,9848022336,Trupthi@gmail.com,Bhuwaneshwar,003

006,9848022335,Archana@gmail.com,Chennai,003

007,9848022334,Komal@gmail.com,trivendram,002

008,9848022333,Bharathi@gmail.com,Chennai,001

And we have loaded these two files into Pig with relations **employee** and **employee\_contact** as shown below.

grunt> employee = LOAD 'hdfs://localhost:9000/pig\_data/employee.txt' USING PigStorage(',')

as (id:int, firstname:chararray, lastname:chararray, age:int, designation:chararray, jobid:int);

grunt> employee\_contact = LOAD 'hdfs://localhost:9000/pig\_data/employee\_contact.txt' USING PigStorage(',')

as (id:int, phone:chararray, email:chararray, city:chararray, jobid:int);

Now, let us join the contents of these two relations using the **JOIN** operator as shown below.

grunt> emp = JOIN employee BY (id,jobid), employee\_contact BY (id,jobid);

### **Verification**

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

grunt> Dump emp;

### **Output**

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

(1,Rajiv,Reddy,21,programmer,113,1,9848022337,Rajiv@gmail.com,Hyderabad,113)

(2,siddarth,Battacharya,22,programmer,113,2,9848022338,siddarth@gmail.com,Kolka ta,113)

(3,Rajesh,Khanna,22,programmer,113,3,9848022339,Rajesh@gmail.com,Delhi,113)

(4,Preethi,Agarwal,21,programmer,113,4,9848022330,Preethi@gmail.com,Pune,113)

(5,Trupthi,Mohanthy,23,programmer,113,5,9848022336,Trupthi@gmail.com,Bhuwaneshw ar,113)

(6,Archana,Mishra,23,programmer,113,6,9848022335,Archana@gmail.com,Chennai,113)

(7,Komal,Nayak,24,teamlead,112,7,9848022334,Komal@gmail.com,trivendram,112)

(8,Bharathi,Nambiayar,24,manager,111,8,9848022333,Bharathi@gmail.com,Chennai,111)

For UDF’s in Java

<https://www.tutorialspoint.com/apache_pig/apache_pig_user_defined_functions.htm>

For UDF’s in python

<https://www.codementor.io/@sheena/extending-hadoop-apache-pig-with-python-udfs-du107mj6t>

<https://blog.moove-it.com/howto-create-udfs-using-python/>