



```
010010110100  
0100101101  
0100101  
0
```



Apache Pig

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

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.

Audience

This tutorial is meant for all those professionals working on Hadoop who would like to perform MapReduce operations without having to type complex codes in Java.

Prerequisites

To make the most of this tutorial, you should have a good understanding of the basics of Hadoop and HDFS commands. It will certainly help if you are good at SQL.

Copyright & Disclaimer

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial	i
Audience.....	i
Prerequisites.....	i
Copyright & Disclaimer	i
Table of Contents	ii
PART 1: INTRODUCTION.....	1
1. Apache Pig – Overview	2
What is Apache Pig?	2
Why Do We Need Apache Pig?.....	2
Features of Pig.....	2
Apache Pig Vs MapReduce	3
Apache Pig Vs SQL	3
Apache Pig Vs Hive	4
Applications of Apache Pig	4
Apache Pig – History.....	5
2. Apache Pig – Architecture	6
Apache Pig – Components.....	7
Pig Latin – Data Model	7
PART 2: ENVIRONMENT	9
3. Apache Pig – Installation.....	10
Prerequisites.....	10
Download Apache Pig.....	10
Install Apache Pig	13
Configure Apache Pig	14
4. Apache Pig – Execution	16
Apache Pig – Execution Modes	16
Apache Pig – Execution Mechanisms	16
Invoking the Grunt Shell	16
Executing Apache Pig in Batch Mode	17
5. Grunt Shell.....	18
Shell Commands	18
Utility Commands	19
PART 3: PIG LATIN	25
6. Pig Latin – Basics	26
Pig Latin – Data Model.....	26
Pig Latin – Statemets	26
Pig Latin – Data types	27
Null Values.....	27
Pig Latin – Arithmetic Operators	28

Pig Latin – Comparison Operators 28

Pig Latin – Type Construction Operators 29

Pig Latin – Relational Operations 29

PART 4: LOAD AND STORE OPERATORS..... 32

7. Apache Pig -- Reading Data 33

 Preparing HDFS 33

 The Load Operator 35

8. Storing Data 38

PART 5: DIAGNOSTIC OPERATORS..... 41

9. Diagnostic Operators 42

 Dump Operator 42

10. Describe Operator 46

11. Explain Operator 47

12. Illustrate Command 51

PART 6: GROUPING AND JOINING 52

13. Group Operator 53

 Grouping by Multiple Columns..... 54

 Group All..... 55

14. Cogroup Operator 56

 Grouping Two Relations using Cogroup 56

15. Join Operator 58

 Inner Join 58

 Self - join 59

 Outer Join 60

 Using Multiple Keys 63

16. Cross Operator..... 65

PART 7: COMBINING AND SPLITTING 68

17. Union Operator..... 69

18. Split Operator 71

PART 8: FILTERING 73

19. Filter Operator 74

20. Distinct Operator	76
21. Foreach Operator	78
PART 9: SORTING	80
22. Order By	81
23. Limit Operator	83
PART 10: PIG LATIN BUILT-IN FUNCTIONS	85
24. Eval Functions	86
Eval Functions.....	86
AVG.....	87
Max.....	88
Min	90
Count	92
COUNT_STAR.....	93
Sum.....	95
DIFF.....	97
SUBTRACT	99
IsEmpty	101
Pluck Tuple	103
Size ()	105
BagToString ()	106
Concat ()	108
Tokenize ()	110
25. Load and Store Functions	113
PigStorage ().....	113
TextLoader ().....	114
BinStorage ()	115
Handling Compression.....	117
26. Bag and Tuple Functions	118
TOBAG ()	118
TOP ()	119
TOTUPLE ()	121
TOMAP ()	122
27. String Functions	123
STARTSWITH ().....	124
ENDSWITH	126
SUBSTRING	127
EqualsIgnoreCase	128
INDEXOF ()	129
LAST_INDEX_OF ()	131
LCFIRST ()	132
UCFIRST ()	133
UPPER ()	134

LOWER ()	136
REPLACE ()	137
STRSPLIT ()	138
STRSPLITTOBAG ()	139
Trim ()	141
LTRIM ()	142
RTRIM	143
28. date-time Functions	145
ToDate ()	147
GetDay ()	148
GetHour ()	149
GetMinute ()	150
GetSecond ()	151
GetMilliSecond ()	152
GetYear	153
GetMonth ()	154
GetWeek ()	156
GetWeekYear ()	157
CurrentTime ()	158
ToString ()	159
DaysBetween ()	160
HoursBetween ()	161
MinutesBetween ()	161
SecondsBetween ()	162
MillisecondsBetween ()	163
YearsBetween ()	164
MonthsBetween ()	165
WeeksBetween ()	166
AddDuration ()	167
SubtractDuration ()	168
29. Math Functions	170
ABS ()	171
ACOS ()	172
ASIN ()	174
ATAN ()	175
CBRT ()	176
CEIL ()	177
COS ()	178
COSH ()	179
EXP ()	180
FLOOR ()	181
LOG ()	181
LOG10 ()	182
RANDOM ()	183
ROUND ()	184
SIN ()	185
SINH ()	186
SQRT ()	187
TAN ()	188

TANH () 189

PART 11: OTHER MODES OF EXECUTION 191

30. User-Defined Functions..... 192

 Types of UDF's in Java 192

 Writing UDF's using Java 192

 Using the UDF 196

31. Running Scripts 198

 Comments in Pig Script..... 198

 Executing Pig Script in Batch mode 198

 Executing a Pig Script from HDFS 199

Part 1: Introduction

1. Apache Pig – Overview

What is Apache Pig?

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 Apache 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, filter, 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.

Applications of Apache Pig

Apache Pig is generally used by data scientists for performing tasks involving ad-hoc processing and quick prototyping. Apache Pig is used;

- To process huge data sources such as web logs.

- To perform data processing for search platforms.
- To process time sensitive data loads.

Apache Pig – History

In **2006**, Apache Pig was developed as a research project at Yahoo, especially to create and execute MapReduce jobs on every dataset. In **2007**, Apache Pig was open sourced via Apache incubator. In **2008**, the first release of Apache Pig came out. In **2010**, Apache Pig graduated as an Apache top-level project.

2. Apache Pig – Architecture

The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a high-level 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.

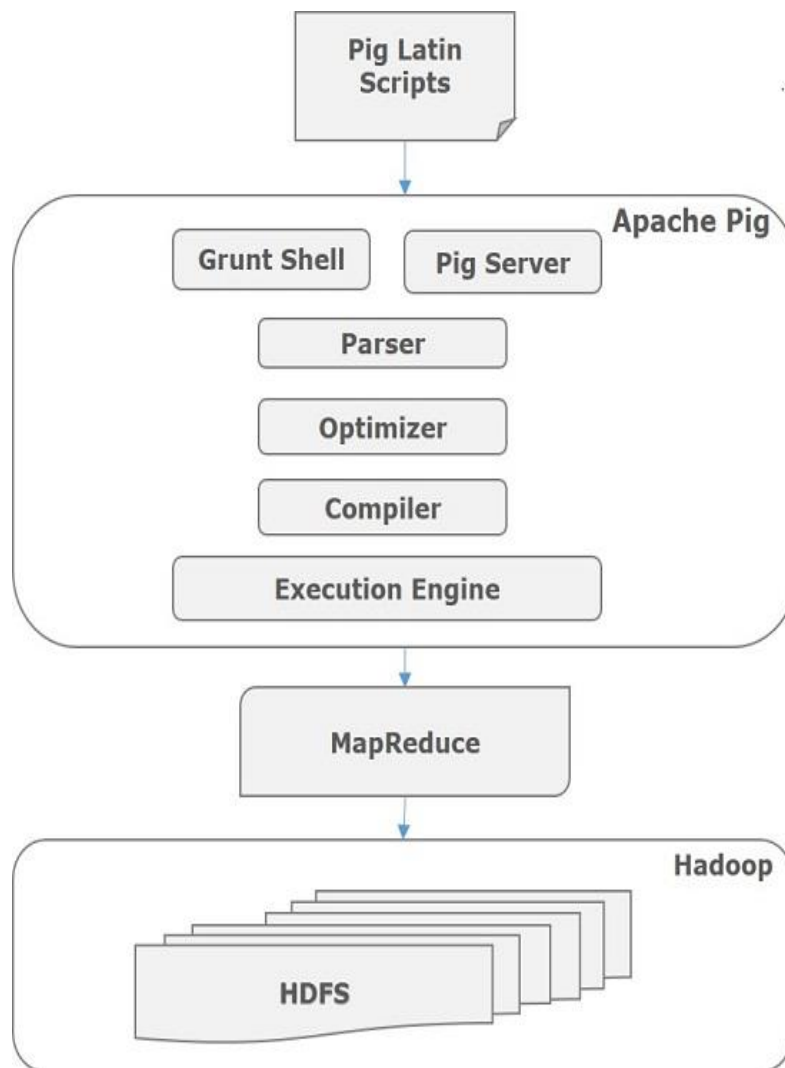


Figure: Apache Pig Architecture

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

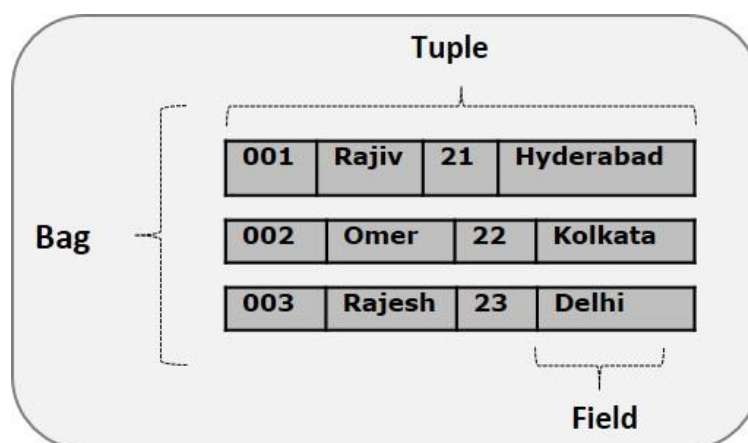
The compiler compiles the optimized logical plan into a series of MapReduce jobs.

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,}}

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).

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]

Part 2: Environment

3. Apache Pig – Installation

This chapter explains the how to download, install, and set up **Apache Pig** in your system.

Prerequisites

It is essential that you have Hadoop and Java installed on your system before you go for Apache Pig. Therefore, prior to installing Apache Pig, install Hadoop and Java by following the steps given in the following link:

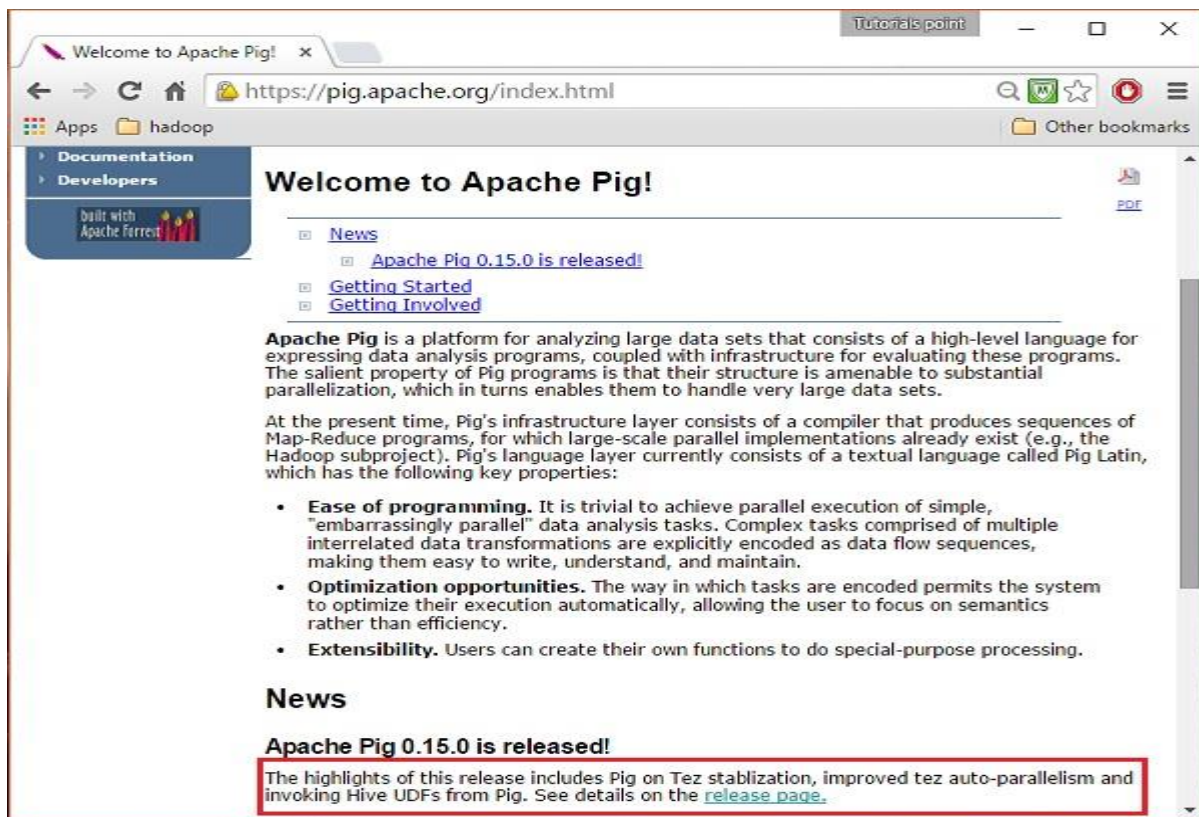
http://www.tutorialspoint.com/hadoop/hadoop_environment_setup.htm

Download Apache Pig

First of all, download the latest version of Apache Pig from the website <https://pig.apache.org/>.

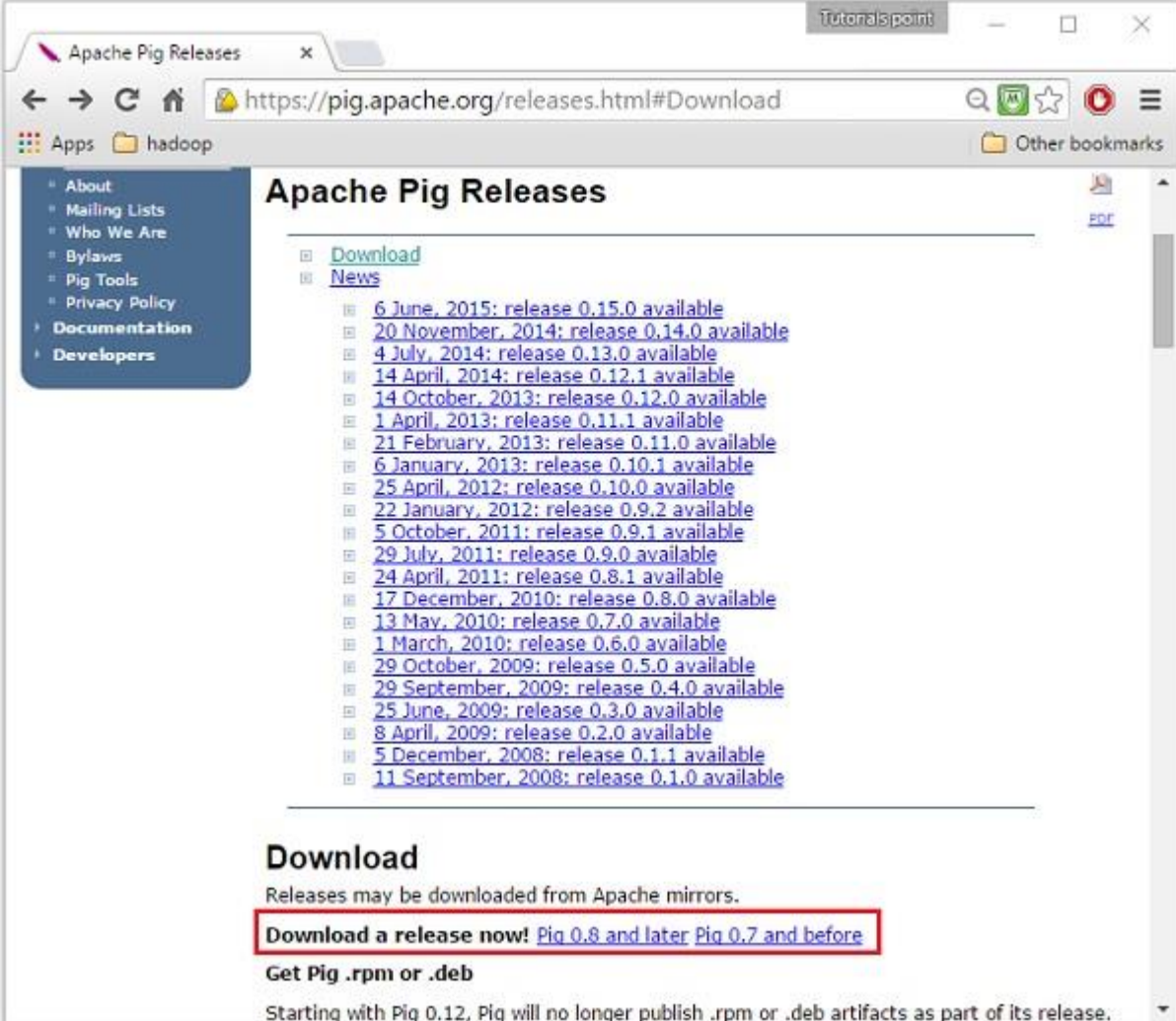
Step 1

Open the homepage of Apache Pig website. Under the section **News**, click on the link **release page** as shown in the following snapshot.



Step 2

On clicking the specified link, you will be redirected to the **Apache Pig Releases** page. On this page, under the **Download** section, you will have two links, namely, **Pig 0.8 and later** and **Pig 0.7 and before**. Click on the link **Pig 0.8 and later**, then you will be redirected to the page having a set of mirrors.



The screenshot shows a web browser window with the URL <https://pig.apache.org/releases.html#Download>. The page title is "Apache Pig Releases". On the left, there is a navigation menu with items: About, Mailing Lists, Who We Are, Bylaws, Pig Tools, Privacy Policy, Documentation, and Developers. The main content area has a "Download" section with a list of releases:

- 6 June, 2015: [release 0.15.0 available](#)
- 20 November, 2014: [release 0.14.0 available](#)
- 4 July, 2014: [release 0.13.0 available](#)
- 14 April, 2014: [release 0.12.1 available](#)
- 14 October, 2013: [release 0.12.0 available](#)
- 1 April, 2013: [release 0.11.1 available](#)
- 21 February, 2013: [release 0.11.0 available](#)
- 6 January, 2013: [release 0.10.1 available](#)
- 25 April, 2012: [release 0.10.0 available](#)
- 22 January, 2012: [release 0.9.2 available](#)
- 5 October, 2011: [release 0.9.1 available](#)
- 29 July, 2011: [release 0.9.0 available](#)
- 24 April, 2011: [release 0.8.1 available](#)
- 17 December, 2010: [release 0.8.0 available](#)
- 13 May, 2010: [release 0.7.0 available](#)
- 1 March, 2010: [release 0.6.0 available](#)
- 29 October, 2009: [release 0.5.0 available](#)
- 29 September, 2009: [release 0.4.0 available](#)
- 25 June, 2009: [release 0.3.0 available](#)
- 8 April, 2009: [release 0.2.0 available](#)
- 5 December, 2008: [release 0.1.1 available](#)
- 11 September, 2008: [release 0.1.0 available](#)

Below the list, there is a "Download" section with the text: "Releases may be downloaded from Apache mirrors." A red box highlights the link: [Download a release now! Pig 0.8 and later Pig 0.7 and before](#). Below this, it says "Get Pig .rpm or .deb" and "Starting with Pig 0.12, Pig will no longer publish .rpm or .deb artifacts as part of its release."

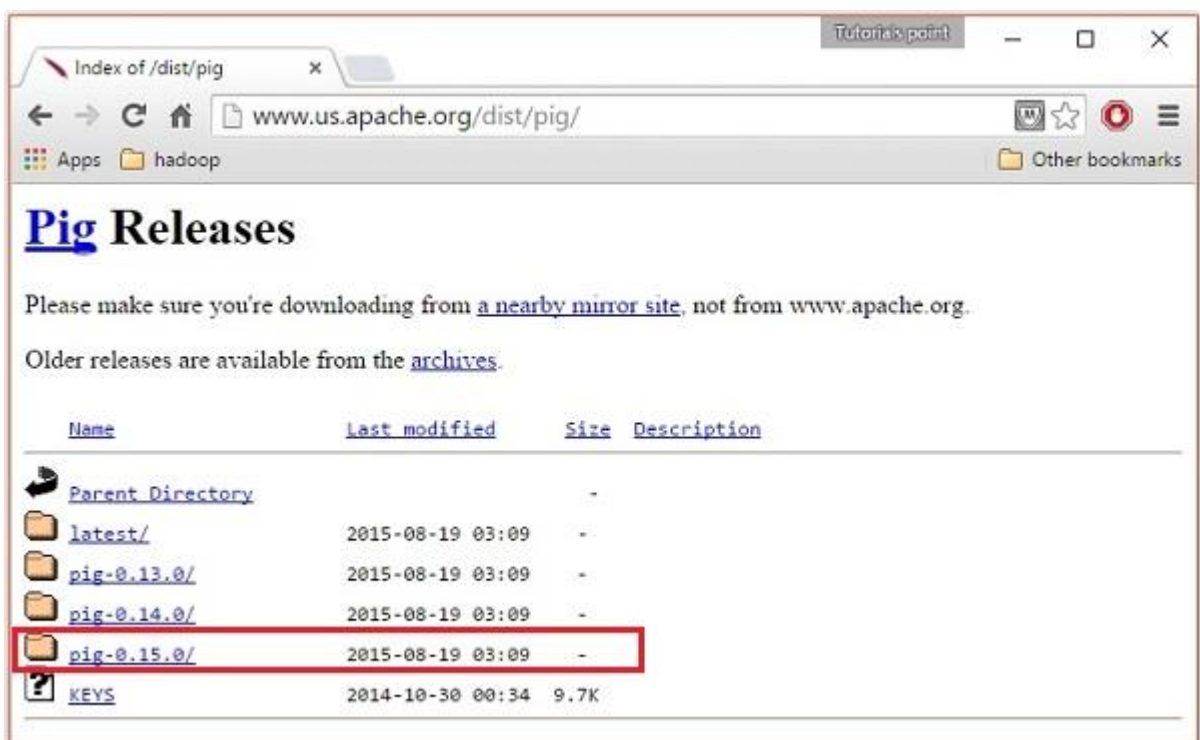
Step 3

Choose and click any one of these mirrors as shown below.



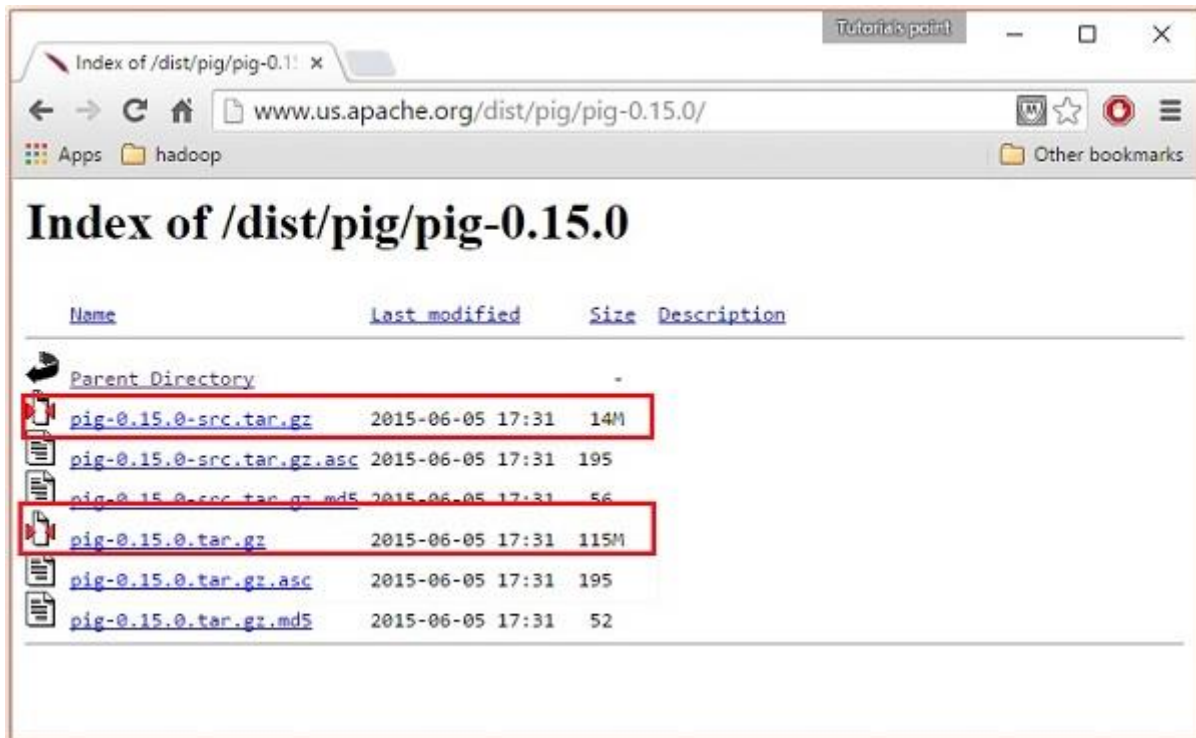
Step 4

These mirrors will take you to the **Pig Releases** page. This page contains various versions of Apache Pig. Click the latest version among them.



Step 5

Within these folders, you will have the source and binary files of Apache Pig in various distributions. Download the tar files of the source and binary files of Apache Pig 0.15, **pig-0.15.0-src.tar.gz** and **pig-0.15.0.tar.gz**.



Install Apache Pig

After downloading the Apache Pig software, install it in your Linux environment by following the steps given below.

Step 1

Create a directory with the name Pig in the same directory where the installation directories of **Hadoop**, **Java**, and other software were installed. (In our tutorial, we have created the Pig directory in the user named Hadoop).

```
$ mkdir Pig
```

Step 2

Extract the downloaded tar files as shown below.

```
$ cd Downloads/
$ tar zxvf pig-0.15.0-src.tar.gz
$ tar zxvf pig-0.15.0.tar.gz
```

Step 3

Move the content of **pig-0.15.0-src.tar.gz** file to the **Pig** directory created earlier as shown below.

```
$ mv pig-0.15.0-src.tar.gz/* /home/Hadoop/Pig/
```

Configure Apache Pig

After installing Apache Pig, we have to configure it. To configure, we need to edit two files: **bashrc** and **pig.properties**.

.bashrc file

In the **.bashrc** file, set the following variables –

- **PIG_HOME** folder to the **Apache Pig's installation folder**,
- **PATH** environment variable to the **bin** folder, and
- **PIG_CLASSPATH** environment variable to the **etc** (configuration) folder of your Hadoop installations (the directory that contains the core-site.xml, hdfs-site.xml and mapred-site.xml files).

```
export PIG_HOME = /home/Hadoop/Pig
export PATH = PATH:/home/Hadoop/pig/bin
export PIG_CLASSPATH = $HADOOP_HOME/conf
```

pig.properties file

In the **conf** folder of Pig, we have a file named **pig.properties**. In the pig.properties file, you can set various parameters as given below.

```
pig -h properties
```

The following properties are supported:

Logging:

verbose=true|false; default is false. This property is the same as -v switch

brief=true|false; default is false. This property is the same as -b switch

debug=OFF|ERROR|WARN|INFO|DEBUG; default is INFO. This property is the same as -d switch

aggregate.warning=true|false; default is true. If true, prints count of warnings of each type rather than logging each warning.

Performance tuning:

pig.cachedbag.memusage=<mem fraction>; default is 0.2 (20% of all memory).
Note that this memory is shared across all large bags used by the application.

pig.skewedjoin.reduce.memusage=<mem fraction>; default is 0.3 (30% of all memory).
Specifies the fraction of heap available for the reducer to perform the join.

```

pig.exec.nocombiner=true|false; default is false.
    Only disable combiner as a temporary workaround for problems.
opt.multiquery=true|false; multiquery is on by default.
    Only disable multiquery as a temporary workaround for problems.
opt.fetch=true|false; fetch is on by default.
    Scripts containing Filter, Foreach, Limit, Stream, and Union can be
dumped without MR jobs.
pig.tmpfilecompression=true|false; compression is off by default.
    Determines whether output of intermediate jobs is compressed.
pig.tmpfilecompression.codec=lzo|gzip; default is gzip.
    Used in conjunction with pig.tmpfilecompression. Defines
compression type.
pig.noSplitCombination=true|false. Split combination is on by default.
    Determines if multiple small files are combined into a single map.
pig.exec.mapPartAgg=true|false. Default is false.
    Determines if partial aggregation is done within map phase,
before records are sent to combiner.
pig.exec.mapPartAgg.minReduction=<min aggregation factor>. Default is 10.
    If the in-map partial aggregation does not reduce the output num records
by this factor, it gets disabled.

```

Miscellaneous:

```

exectype=mapreduce|tez|local; default is mapreduce. This property is
the same as -x switch
pig.additional.jars.uris=<comma seperated list of jars>. Used in place
of register command.
udf.import.list=<comma seperated list of imports>. Used to avoid
package names in UDF.
stop.on.failure=true|false; default is false. Set to true to terminate
on the first error.
pig.datetime.default.tz=<UTC time offset>. e.g. +08:00. Default is the
default timezone of the host.
    Determines the timezone used to handle datetime datatype and UDFs.
Additionally, any Hadoop property can be specified.

```

Verifying the Installation

Verify the installation of Apache Pig by typing the version command. If the installation is successful, you will get the version of Apache Pig as shown below.

```
$ pig -version
```

```

Apache Pig version 0.15.0 (r1682971)
compiled Jun 01 2015, 11:44:35

```

4. Apache Pig – Execution

In the previous chapter, we explained how to install Apache Pig. In this chapter, we will discuss how to execute Apache Pig.

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 (**User Defined Functions**) 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

<p>Output:</p> <pre>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></pre>	<p>Output:</p> <pre>15/09/28 10:28:46 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443416326123.log 2015-09-28 10:28:46,427 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:/// grunt></pre>
--	--

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.

5. Grunt Shell

After invoking the Grunt shell, you can run your Pig scripts in the shell. In addition to that, there are certain useful shell and utility commands provided by the Grunt shell. This chapter explains the shell and utility commands provided by the Grunt shell.

Note: In some portions of this chapter, the commands like **Load** and **Store** are used. Refer the respective chapters to get in-detail information on them.

Shell Commands

The Grunt shell of Apache Pig is mainly used to write Pig Latin scripts. Prior to that, we can invoke any shell commands using **sh** and **fs**.

sh Command

Using **sh** command, we can invoke any shell commands from the Grunt shell. Using **sh** command from the Grunt shell, we cannot execute the commands that are a part of the shell environment (**ex:** cd).

Syntax

Given below is the syntax of **sh** command.

```
grunt> sh shell command parameters
```

Example

We can invoke the **ls** command of Linux shell from the Grunt shell using the **sh** option as shown below. In this example, it lists out the files in the **/pig/bin/** directory.

```
grunt> sh ls

pig
pig_1444799121955.log
pig.cmd
pig.py
```

fs Command

Using the **fs** command, we can invoke any FsShell commands from the Grunt shell.

Syntax

Given below is the syntax of **fs** command.

```
grunt> sh File System command parameters
```

Example

We can invoke the ls command of HDFS from the Grunt shell using fs command. In the following example, it lists the files in the HDFS root directory.

```
grunt> fs -ls

Found 3 items
drwxrwxrwx - Hadoop supergroup 0 2015-09-08 14:13 Hbase
drwxr-xr-x - Hadoop supergroup 0 2015-09-09 14:52 seqgen_data
drwxr-xr-x - Hadoop supergroup 0 2015-09-08 11:30 twitter_data
```

In the same way, we can invoke all the other file system shell commands from the Grunt shell using the **fs** command.

Utility Commands

The Grunt shell provides a set of utility commands. These include utility commands such as **clear**, **help**, **history**, **quit**, and **set**; and commands such as **exec**, **kill**, and **run** to control Pig from the Grunt shell. Given below is the description of the utility commands provided by the Grunt shell.

clear Command

The **clear** command is used to clear the screen of the Grunt shell.

Syntax

You can clear the screen of the grunt shell using the **clear** command as shown below.

```
grunt> clear
```

help Command

The **help** command gives you a list of Pig commands or Pig properties.

Usage

You can get a list of Pig commands using the **help** command as shown below.

```
grunt> help
```

Commands:

`<pig latin statement>;` - See the PigLatin manual for details: <http://hadoop.apache.org/pig>

File system commands:

`fs <fs arguments>` - Equivalent to Hadoop `dfs` command:

http://hadoop.apache.org/common/docs/current/hdfs_shell.html

Diagnostic Commands:

`describe <alias>[:<alias>]` - Show the schema for the alias. Inner aliases can be described as `A::B`.

`explain [-script <pigscript>] [-out <path>] [-brief] [-dot|-xml] [-param <param_name>=<param_value>]`

`[-param_file <file_name>] [<alias>]` - Show the execution plan to compute the alias or for entire script.

`-script` - Explain the entire script.

`-out` - Store the output into directory rather than print to stdout.

`-brief` - Don't expand nested plans (presenting a smaller graph for overview).

`-dot` - Generate the output in `.dot` format. Default is text format.

`-xml` - Generate the output in `.xml` format. Default is text format.

`-param <param_name>` - See parameter substitution for details.

`-param_file <file_name>` - See parameter substitution for details.

`alias` - Alias to explain.

`dump <alias>` - Compute the alias and writes the results to stdout.

Utility Commands:

`exec [-param <param_name>=param_value] [-param_file <file_name>] <script>`

-

Execute the script with access to grunt environment including aliases.

`-param <param_name>` - See parameter substitution for details.

`-param_file <file_name>` - See parameter substitution for details.

`script` - Script to be executed.

`run [-param <param_name>=param_value] [-param_file <file_name>] <script>` -

Execute the script with access to grunt environment.

`-param <param_name>` - See parameter substitution for details.

`-param_file <file_name>` - See parameter substitution for details.

`script` - Script to be executed.

`sh <shell command>` - Invoke a shell command.

`kill <job_id>` - Kill the hadoop job specified by the hadoop job id.

`set <key> <value>` - Provide execution parameters to Pig. Keys and values are case sensitive.

The following keys are supported:

`default_parallel` - Script-level reduce parallelism. Basic input size heuristics used by default.

`debug` - Set debug on or off. Default is off.

`job.name` - Single-quoted name for jobs. Default is `PigLatin:<script name>`

`job.priority` - Priority for jobs. Values: `very_low`, `low`, `normal`, `high`, `very_high`. Default is `normal`

`stream.skippath` - String that contains the path. This is used by streaming any hadoop property.

```
help - Display this message.  
history [-n] - Display the list statements in cache.  
    -n Hide line numbers.  
quit - Quit the grunt shell.
```

history Command

This command displays a list of statements executed / used so far since the Grunt shell is invoked.

Usage

Assume we have executed two statements since opening the Grunt shell.

```
customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING  
PigStorage(',');  
  
orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING  
PigStorage(',');  
  
student = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING  
PigStorage(',');
```

Then, using the **history** command will produce the following output.

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

set Command

The **set** command is used to show/assign values to keys used in Pig.

Usage

Using this command, you can set values to the following keys.

Key	Description and values
default_parallel	You can set the number of reducers for a map job by passing any whole number as a value to this key.
debug	You can turn off or turn on the debugging feature in Pig by passing on/off to this key.
job.name	You can set the Job name to the required job by passing a string value to this key.
job.priority	You can set the job priority to a job by passing one of the following values to this key: <ul style="list-style-type: none"> • very_low • low • normal • high • very_high
stream.skippath	For streaming, you can set the path from where the data is not to be transferred, by passing the desired path in the form of a string to this key.

quit Command

You can quit from the Grunt shell using this command.

Usage

Quit from the Grunt shell as shown below.

```
grunt> quit
```

Let us now take a look at the commands using which you can control Apache Pig from the Grunt shell.

exec Command

Using the **exec** command, we can execute Pig scripts from the Grunt shell.

Syntax

Given below is the syntax of the utility command **exec**.

```
grunt> exec [-param param_name = param_value] [-param_file file_name] [script]
```

Example

Let us assume there is a file named **student.txt** in the **/pig_data/** directory of HDFS with the following content.

Student.txt

```
001,Rajiv,Hyderabad
002,siddarth,Kolkata
003,Rajesh,Delhi
```

And, assume we have a script file named **sample_script.pig** in the **/pig_data/** directory of HDFS with the following content.

Sample_script.pig

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

Dump student;
```

Now, let us execute the above script from the Grunt shell using the **exec** command as shown below.

```
grunt> exec /sample_script.pig
```

Output

The **exec** command executes the script in the **sample_script.pig**. As directed in the script, it loads the **student.txt** file into Pig and gives you the result of the Dump operator displaying the following content.

```
(1,Rajiv,Hyderabad)
(2,siddarth,Kolkata)
(3,Rajesh,Delhi)
```

kill Command

You can kill a job from the Grunt shell using this command.

Syntax

Given below is the syntax of the **kill** command.

```
grunt> kill JobId
```

Example

Suppose there is a running Pig job having id **Id_0055**, you can kill it from the Grunt shell using the **kill** command, as shown below.

```
grunt> kill Id_0055
```

run Command

You can run a Pig script from the Grunt shell using the **run** command.

Syntax

Given below is the syntax of the **run** command.

```
grunt> run [-param param_name = param_value] [-param_file file_name] script
```

Example

Let us assume there is a file named **student.txt** in the **/pig_data/** directory of HDFS with the following content.

Student.txt

```
001,Rajiv,Hyderabad
002,siddarth,Kolkata
003,Rajesh,Delhi
```

And, assume we have a script file named **sample_script.pig** in the local filesystem with the following content.

Sample_script.pig

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

Now, let us run the above script from the Grunt shell using the **run** command as shown below.

```
grunt> run /sample_script.pig
```

You can see the output of the script using the **Dump operator** as shown below.

```
grunt> Dump;

(1,Rajiv,Hyderabad)
(2,siddarth,Kolkata)
(3,Rajesh,Delhi)
```

Note: The difference between **exec** and the **run** command is that if we use **run**, the statements from the script are available in the command history.

Part 3: Pig Latin

6. Pig Latin – Basics

Pig Latin is the language used to analyze data in Hadoop using Apache Pig. In this chapter, we are going to discuss the basics of Pig Latin such as Pig Latin statements, data types, general and relational operators, and Pig Latin UDF's.

Pig Latin – Data Model

As discussed in the previous chapters, the data model of Pig is fully nested. A **Relation** is the outermost structure of the Pig Latin data model. And it is a **bag** where -

- A bag is a collection of tuples.
- A tuple is an ordered set of fields.
- A field is a piece of data.

Pig Latin – Statemets

While processing data using Pig Latin, **statements** are the basic constructs.

- These statements work with **relations**. They include **expressions** and **schemas**.
- Every statement ends with a semicolon (;).
- We will perform various operations using operators provided by Pig Latin, through statements.
- Except LOAD and STORE, while performing all other operations, Pig Latin statements take a relation as input and produce another relation as output.
- As soon as you enter a **Load** statement in the Grunt shell, its semantic checking will be carried out. To see the contents of the schema, you need to use the **Dump** operator. Only after performing the **dump** operation, the MapReduce job for loading the data into the file system will be carried out.

Example

Given below is a Pig Latin statement, which loads data to Apache Pig.

```
Student_data = LOAD 'student_data.txt' USING PigStorage(',')as ( id:int,
firstname:chararray, lastname:chararray, phone:chararray, city:chararray );
```

Pig Latin – Data types

Given below table describes the Pig Latin data types.

Data Type	Description and Example
int	Represents a signed 32-bit integer. Example: 8
long	Represents a signed 64-bit integer. Example: 5L
float	Represents a signed 32-bit floating point. Example: 5.5F
double	Represents a 64-bit floating point. Example: 10.5
chararray	Represents a character array (string) in Unicode UTF-8 format. Example: `tutorials point`
Bytearray	Represents a Byte array (blob).
Boolean	Represents a Boolean value. Example: true/ false.
Datetime	Represents a date-time. Example: 1970-01-01T00:00:00.000+00:00
Biginteger	Represents a Java BigInteger. Example: 60708090709
Bigdecimal	Represents a Java BigDecimal Example: 185.98376256272893883
Complex Types	
Tuple	A tuple is an ordered set of fields. Example: (raja, 30)
Bag	A bag is a collection of tuples. Example: {(raju,30),(Mohhammad,45)}
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.

Part 4: Load and Store Operators

7. Apache Pig – Reading Data

In general, Apache Pig works on top of Hadoop. It is an analytical tool that analyzes large datasets that exist in the **Hadoop File System**. To analyze data using Apache Pig, we have to initially load the data into Apache Pig. This chapter explains how to load data to Apache Pig from HDFS.

Preparing HDFS

In MapReduce mode, Pig reads (loads) data from HDFS and stores the results back in HDFS. Therefore, let us start HDFS and create the following sample data in HDFS.

Student ID	First Name	Last Name	Phone	City
001	Rajiv	Reddy	9848022337	Hyderabad
002	siddarth	Battacharya	9848022338	Kolkata
003	Rajesh	Khanna	9848022339	Delhi
004	Preethi	Agarwal	9848022330	Pune
005	Trupthi	Mohanthly	9848022336	Bhuwaneshwar
006	Archana	Mishra	9848022335	Chennai

The above dataset contains personal details like id, first name, last name, phone number and city, of six students.

Step 1: Verifying Hadoop

First of all, verify the installation using Hadoop version command, as shown below.

```
$ hadoop version
```

If your system contains Hadoop, and if you have set the PATH variable, then you will get the following output:

```
Hadoop 2.6.0
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r
e3496499ecb8d220fba99dc5ed4c99c8f9e33bb1
Compiled by jenkins on 2014-11-13T21:10Z
Compiled with protoc 2.5.0
From source with checksum 18e43357c8f927c0695f1e9522859d6a
This command was run using /home/Hadoop/hadoop/share/hadoop/common/hadoop-
common-2.6.0.jar
```


Step 2: Starting HDFS

Browse through the **sbin** directory of Hadoop and start **yarn** and Hadoop dfs (distributed file system) as shown below.

```
cd /$Hadoop_Home/sbin/
$ start-dfs.sh

localhost: starting namenode, logging to /home/Hadoop/hadoop/logs/hadoop-
Hadoop-namenode-localhost.localdomain.out

localhost: starting datanode, logging to /home/Hadoop/hadoop/logs/hadoop-
Hadoop-datanode-localhost.localdomain.out

Starting secondary namenodes [0.0.0.0]

starting secondarynamenode, logging to /home/Hadoop/hadoop/logs/hadoop-Hadoop-
secondarynamenode-localhost.localdomain.out

$ start-yarn.sh

starting yarn daemons

starting resourcemanager, logging to /home/Hadoop/hadoop/logs/yarn-Hadoop-
resourcemanager-localhost.localdomain.out

localhost: starting nodemanager, logging to /home/Hadoop/hadoop/logs/yarn-
Hadoop-nodemanager-localhost.localdomain.out
```

Step 3: Create a Directory in HDFS

In Hadoop DFS, you can create directories using the command **mkdir**. Create a new directory in HDFS with the name **Pig_Data** in the required path as shown below.

```
$cd /$Hadoop_Home/bin/
$ hdfs dfs -mkdir hdfs://localhost:9000/Pig_Data
```

Step 4: Placing the data in HDFS

The input file of Pig contains each tuple/record in individual lines. And the entities of the record are separated by a delimiter (In our example we used “,”).

In the local file system, create an input file **student_data.txt** containing data as shown below.

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

Now, move the file from the local file system to HDFS using **put** command as shown below. (You can use **copyFromLocal** command as well.)

```
$ cd $HADOOP_HOME/bin
$ hdfs dfs -put /home/Hadoop/Pig/Pig_Data/student_data.txt
dfs://localhost:9000/pig_data/
```

Verifying the file

You can use the **cat** command to verify whether the file has been moved into the HDFS, as shown below.

```
$ cd $HADOOP_HOME/bin
$ hdfs dfs -cat hdfs://localhost:9000/pig_data/student_data.txt
```

Output

You can see the content of the file as shown below.

```
15/10/01 12:16:55 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable

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

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**). Or, we can define our own load function.
- **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);
```

Note: We load the data without specifying the schema. In that case, the columns will be addressed as \$01, \$02, etc... (check).

Example

As an example, let us load the data in **student_data.txt** in Pig under the schema named **Student** using the **LOAD** command.

Start the Pig Grunt Shell

First of all, open the Linux terminal. Start the Pig Grunt shell in MapReduce mode as shown below.

```
$ Pig -x mapreduce
```

It will start the Pig Grunt shell as shown below.

```
15/10/01 12:33:37 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/10/01 12:33:37 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/10/01 12:33:37 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType

2015-10-01 12:33:38,080 [main] INFO org.apache.pig.Main - Apache Pig version
0.15.0 (r1682971) compiled Jun 01 2015, 11:44:35
2015-10-01 12:33:38,080 [main] INFO org.apache.pig.Main - Logging error
messages to: /home/Hadoop/pig_1443683018078.log
2015-10-01 12:33:38,242 [main] INFO org.apache.pig.impl.util.Utils - Default
bootup file /home/Hadoop/.pigbootup not found

2015-10-01 12:33:39,630 [main]
INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
Connecting to hadoop file system at: hdfs://localhost:9000

grunt>
```

Execute the Load Statement

Now load the data from the file **student_data.txt** into Pig by executing the following Pig Latin statement in the Grunt shell.

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

Following is the description of the above statement.

Relation name	We have stored the data in the schema student .					
Input file path	We are reading data from the file student_data.txt , which is in the /pig_data/ directory of HDFS.					
Storage function	We have used the PigStorage() function. It loads and stores data as structured text files. It takes a delimiter using which each entity of a tuple is separated, as a parameter. By default, it takes '\t' as a parameter.					
schema	We have stored the data using the following schema.					
	column	id	firstname	lastname	phone	city
	datatype	int	char array	char array	char array	char array

Note: The **load** statement will simply load the data into the specified relation in Pig. To verify the execution of the **Load** statement, you have to use the **Diagnostic Operators** which are discussed in the next chapters.

8. Storing Data

In the previous chapter, we learnt how to load data into Apache Pig. You can store the loaded data in the file system using the **store** operator. This chapter explains how to store data in 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

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,Mohanthi,9848022336,Bhuaneshwar
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 store the relation in the HDFS directory **"hdfs://localhost:9000/pig_Output/"** as shown below.

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

Output

After executing the **store** statement, you will get the following output. A directory is created with the specified name and the data will be stored in it.

```
2015-10-05 13:05:05,429 [main]
INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLau
ncher - 100% complete
2015-10-05 13:05:05,429 [main]
INFO org.apache.pig.tools.pigstats.mapreduce.SimplePigStats - Script
Statistics:

HadoopVersion    PigVersion      UserId          StartedAt       FinishedAt      Features
2.6.0            0.15.0         Hadoop         2015-10-05 13:03:03  2015-10-05
```

```

13:05:05    UNKNOWN

Success!

Job Stats (time in seconds):
JobId      Maps      Reduces   MaxMapTime   MinMapTime   AvgMapTime   MedianMap
Time      MaxReduceTime   MinReduceTime   AvgReduceTime   MedianReductime
Alias      Feature      Outputs
job_1443519499159_0006  1    0    n/a    n/a    n/a    n/a    0    0    0
0    student    MAP_ONLY    hdfs://localhost:9000/pig_Output,

Input(s):
Successfully read 0 records from:
"hdfs://localhost:9000/pig_data/student_data.txt"

Output(s):
Successfully stored 0 records in: "hdfs://localhost:9000/pig_Output"

Counters:
Total records written : 0
Total bytes written : 0
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG:
job_1443519499159_0006

2015-10-05 13:06:06,192 [main]
INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLau
ncher - Success!

```

Verification

You can verify the stored data as shown below.

Step 1

First of all, list out the files in the directory named **pig_output** using the **ls** command as shown below.

```

hdfs dfs -ls 'hdfs://localhost:9000/pig_Output/'
Found 2 items
rw-r--r-  1 Hadoop supergroup          0 2015-10-05 13:03
hdfs://localhost:9000/pig_Output/_SUCCESS
rw-r--r-  1 Hadoop supergroup        224 2015-10-05 13:03
hdfs://localhost:9000/pig_Output/part-m-00000

```

You can observe that two files were created after executing the **store** statement.

Step 2

Using **cat** command, list the contents of the file named **part-m-00000** as shown below.

```
$ hdfs dfs -cat 'hdfs://localhost:9000/pig_Output/part-m-00000'  
1,Rajiv,Reddy,9848022337,Hyderabad  
2,siddarth,Battacharya,9848022338,Kolkata  
3,Rajesh,Khanna,9848022339,Delhi  
4,Preethi,Agarwal,9848022330,Pune  
5,Trupthi,Mohanthy,9848022336,Bhuaneshwar  
6,Archana,Mishra,9848022335,Chennai
```

Part 5: Diagnostic Operators

9. Diagnostic Operators

The **load** statement will simply load the data into the specified relation in Apache Pig. To verify the execution of the **Load** statement, you have to use the **Diagnostic Operators**. Pig Latin provides four different types of diagnostic operators:

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

In this chapter, we will discuss the diagnostic operators of Pig Latin.

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,Mohanthi,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 print the contents of the relation using the **Dump operator** as shown below.

```
grunt> Dump student
```

Output

Once you execute the above **Pig Latin** statement, it will start a MapReduce job to read data from HDFS. It will produce the following output.

```

2015-10-01 15:05:27,642 [main]
INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLau
ncher - 100% complete
2015-10-01 15:05:27,652 [main]
INFO org.apache.pig.tools.pigstats.mapreduce.SimplePigStats - Script
Statistics:

HadoopVersion      PigVersion      UserId          StartedAt
FinishedAt                Features

2.6.0              0.15.0         Hadoop         2015-10-01
15:03:11          2015-10-01 05:27      UNKNOWN

Success!

Job Stats (time in seconds):

JobId
Maps
Reduces
MaxMapTime
MinMapTime
AvgMapTime
MedianMapTime

job_14459_0004
1
0
n/a
n/a
n/a
n/a

MaxReduceTime

MinReduceTime
AvgReduceTime
MedianReducetime
Alias

```

```

0
0
0
0
student

Feature
Outputs

MAP_ONLY
hdfs://localhost:9000/tmp/temp580182027/tmp757878456,

Input(s):
Successfully read 0 records from:
"hdfs://localhost:9000/pig_data/student_data.txt"

Output(s):
Successfully stored 0 records in:
"hdfs://localhost:9000/tmp/temp580182027/tmp757878456"

Counters:
Total records written : 0
Total bytes written : 0
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG:
job_1443519499159_0004

```

```

2015-10-01 15:06:28,403 [main]
INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLau
ncher - Success!
2015-10-01 15:06:28,441 [main] INFO org.apache.pig.data.SchemaTupleBackend -
Key [pig.schematuple] was not set... will not generate code.
2015-10-01 15:06:28,485 [main]
INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths
to process : 1
2015-10-01 15:06:28,485 [main]
INFO org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total
input paths to process : 1
(1,Rajiv,Reddy,9848022337,Hyderabad)
(2,siddarth,Battacharya,9848022338,Kolkata)
(3,Rajesh,Khanna,9848022339,Delhi)
(4,Preethi,Agarwal,9848022330,Pune)

```

(5, Trupthi, Mohanthy, 9848022336, Bhubaneswar)
(6, Archana, Mishra, 9848022335, Chennai)

10. Describe Operator

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,Mohanthi,9848022336,Bhuaneshwar
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 }
```

11. Explain Operator

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,Mohanthi,9848022336,Bhuaneshwar
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;
```

Output

It will produce the following output.

```
$ explain student;

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(', ')) - scope-
19

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
-----
```

12. Illustrate Command

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,Bhuaneshwar
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              |
-----
```

Part 6: Grouping and Joining

13. Group Operator

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.

```
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 schema 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);
```

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.

```
Dump group_data;
```

Output

Then you will get output displaying the contents of the relation named **groyp_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, Nambiyar, 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:i
nt,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 schema 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,Bhuwaneswar),{(5,Trupthi,Mohanthi,23,9848022336,Bhuwaneswar)})
((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 schema **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,Mohanthi,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)})
```

14. Cogroup Operator

The **cogroup** operator works more or less in the same way as the **group** 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,Mohanthi,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 schema names **student_details** and **employee_details** respectively, 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);

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.

```
Dump cogroup_data;
```

Output

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

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

(22, { (3, Rajesh, Khanna, 22, 9848022339, Delhi),
      (2, Siddharth, Battacharya, 22, 9848022338, Kolkata) },
 { (6, Maggy, 22, Chennai), (1, Robin, 22, newyork) })

(23, {(6, Archana, Mishra, 23, 9848022335, Chennai), (5, Trupthi, Mohanthi, 23, 9848022336,
      Bhuvaneshwar)},
 { (5, David, 23, Bhuvaneshwar), (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 schema 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 schema (**student_details** in this case) having age 21, and
- the second bag contains all the tuples from the second schema (**employee_details** in this case) having age 21.

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

15. Join Operator

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 schemas **customers** and **orders** as shown below.

```
customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING
PigStorage(',')as (id:int, name:chararray, age:int, address:chararray,
salary:int);

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 schemas.

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.

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

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.

```
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)
```

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.

```
customers1 = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING
PigStorage(',')as (id:int, name:chararray, age:int, address:chararray,
salary:int);

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.

```
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.

```
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)
```

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.

```
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.

```
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.

```
grunt> 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.

```
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 schemas **employee** and **employee_contact** as shown below.

```
employee = LOAD 'hdfs://localhost:9000/pig_data/employee.txt' USING
PigStorage(',')as (id:int, firstname:chararray, lastname:chararray, age:int,
designation:chararray, jobid:int);

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.

```
emp = JOIN employee BY (id,jobid), employee_contact BY (id,jobid);
```

Verification

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

```
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,Kolkata,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,Mohanthly,23,programmer,113,5,9848022336,Trupthi@gmail.com,Bhuwaneshwar,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)
```

16. Cross Operator

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.

```
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 schemas **customers** and **orders** as shown below.

```
customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING
PigStorage(',')as (id:int, name:chararray, age:int, address:chararray,
salary:int);

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 schemas using the **cross** operator on these two schemas as shown below.

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


Verification

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

```
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)
```

Part 7: Combining and Splitting

17. Union Operator

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,Bhuaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

Student_data2.txt

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

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

```
student1 = LOAD 'hdfs://localhost:9000/pig_data/student_data1.txt' USING
PigStorage(',') as (id:int, firstname:chararray, lastname:chararray,
phone:chararray, city:chararray);

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.

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

Verification

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

```
Dump student;
```

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,Mohanthi,9848022336,Bhuwaneswar)
(6,Archana,Mishra,9848022335,Chennai)
(7,Komal,Nayak,9848022334,trivendram)
(8,Bharathi,Nambiayar,9848022333,Chennai)
```

18. Split Operator

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,Mohanthi,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 schema 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);
```

Verification

Verify the relations **student_details1** and **student_details2** using the **DUMP** operator as shown below.

```
Dump student_details1;

Dump student_details2;
```

Output

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

```
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)

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)
```

Part 8: Filtering

19. Filter Operator

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 schema 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 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';
```

Verification

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

```
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)
```

20. Distinct Operator

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,Bhuaneshwar
006,Archana,Mishra,9848022335,Chennai
006,Archana,Mishra,9848022335,Chennai
```

And we have loaded this file into Pig with the schema 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,
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 **data** as shown below.

```
distinct_data = DISTINCT student_details;
```

Verification

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

```
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,Mohanthi,9848022336,Bhuaneshwar)
(6,Archana,Mishra,9848022335,Chennai)
```

21. Foreach Operator

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

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

Verification

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

```
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,Bhuvaneshwar)
(6,23,Chennai)
(7,24,trivendram)
(8,24,Chennai)
```

Part 9: Sorting

22. Order By

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

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

Verification

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

```
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,Mohanthi,23,9848022336,Bhuwaneswar)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(4,Preethi,Agarwal,21,9848022330,Pune)
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
```

23. Limit Operator

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 schema 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);
```

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.

```
limit_data = LIMIT student_details 4;
```

Verification

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

```
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)
```

Part 10: Pig Latin Built-in Functions

24. Eval Functions

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.

Function	Description
AVG	To compute the average of the numerical values within a bag.
BagToString	To concatenate the elements of a bag into a string. While concatenating, we can place a delimiter between these values (optional).
CONCAT	To concatenate two or more expressions of same type.
COUNT	To get the number of elements in a bag, while counting the number of tuples in a bag.
COUNT_STAR	It is similar to the COUNT() function. It is used to get the number of elements in a bag.
DIFF	To compare two bags (fields) in a tuple.
IsEmpty	To check if a bag or map is empty.
MAX	To calculate the highest value for a column (numeric values or chararrays) in a single-column bag.
MIN	To get the minimum (lowest) value (numeric or chararray) for a certain column in a single-column bag.
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.
SIZE	To compute the number of elements based on any Pig data type.

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.
SUM	To get the total of the numeric values of a column in a single-column bag.
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.

AVG

The Pig-Latin **AVG()** function is used to compute the average of the numerical values within a bag. While calculating the average value, the **AVG()** function ignores the NULL values.

Note:

- To get the global average value, we need to perform a **Group All** operation, and calculate the average value using the AVG function.
- To get the average value of a group, we need to group it using the **Group By** operator and proceed with the average function.

Syntax

Given below is the syntax of the **AVG** function.

```
grunt> AVG(expression)
```

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,89
002,siddarth,Battacharya,22,9848022338,Kolkata,78
003,Rajesh,Khanna,22,9848022339,Delhi,90
004,Preethi,Agarwal,21,9848022330,Pune,93
005,Trupthi,Mohanthi,23,9848022336,Bhuwaneshwar,75
006,Archana,Mishra,23,9848022335,Chennai,87
007,Komal,Nayak,24,9848022334,trivendram,83
008,Bharathi,Nambiayar,24,9848022333,Chennai,72
```

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

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

Calculating the Average GPA

We can use the built-in function **AVG** (case-sensitive) to calculate the average of a set of numerical values. Let's group the schema **student_details** using the Group All operator, and store the result in the schema named **student_group_all** as shown below.

```
grunt> student_group_all = Group student_details All;
```

This will produce a schema as shown below.

```
grunt> Dump student_group_all;

(all, {(8, Bharathi, Nambiayar, 24, 9848022333, Chennai, 72), (7, Komal, Nayak, 24, 9848022
334, trivendram, 83), (6, Archana, Mishra, 23, 9848022335, Chennai, 87), (5, Trupthi, Mohan
thy, 23, 9848022336, Bhuwaneshwar, 75), (4, Preethi, Agarwal, 21, 9848022330, Pune, 93), (3
, Rajesh, Khanna, 22, 9848022339, Delhi, 90), (2, siddarth, Battacharya, 22, 9848022338, Ko
lkata, 78), (1, Rajiv, Reddy, 21, 9848022337, Hyderabad, 89)})
```

Let us now calculate the global average GPA of all the students using the **AVG** function as shown below.

```
grunt> student_gpa_avg = foreach student_group_all Generate
(student_details.firstname, student_details.gpa), AVG(student_details.gpa);
```

Verification

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

```
grunt> Dump student_gpa_avg;
```

Output

It will display the contents of the relation **student_gpa_avg** as follows.

```
(({(Bharathi), (Komal), (Archana), (Trupthi), (Preethi), (Rajesh), (siddarth), (Rajiv)
}),
{ (72) , (83) , (87) , (75) , (93) , (90) , (78) ,
(89) }), 83.375)
```

Max

The Pig Latin **Max()** function is used to calculate the highest value for a column (numeric values or chararrays) in a single-column bag. While calculating the maximum value, the Max() function ignores the NULL values.

Note:

- To get the global maximum value, we need to perform a **Group All** operation, and calculate the average value using the AVG function.
- To get the maximum value of a group, we need to group it using the **Group By** operator and proceed with the average function.

Syntax

Given below is the syntax of the **Max()** function.

```
grunt> Max(expression)
```

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,89
002,siddarth,Battacharya,22,9848022338,Kolkata,78
003,Rajesh,Khanna,22,9848022339,Delhi,90
004,Preethi,Agarwal,21,9848022330,Pune,93
005,Trupthi,Mohanthi,23,9848022336,Bhuwaneshwar,75
006,Archana,Mishra,23,9848022335,Chennai,87
007,Komal,Nayak,24,9848022334,trivendram,83
008,Bharathi,Nambiayar,24,9848022333,Chennai,72
```

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

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

Calculating the Maximum GPA

We can use the built-in function **MAX** (case-sensitive) to calculate the maximum value from a set of given numerical values. Let us group the schema **student_details** using the **Group All** operator, and store the result in the schema named **student_group_all** as shown below.

```
grunt> student_group_all = Group student_details All;
```

This will produce a schema as shown below.

```
grunt> Dump student_group_all;

(all, {(8,Bharathi,Nambiayar,24,9848022333,Chennai,72),(7,Komal,Nayak,24,9848022
334,trivendram,83),(6,Archana,Mishra,23,9848022335,Chennai,87),(5,Trupthi,Mohan
thy,23,9848022336,Bhuwaneshwar,75),(4,Preethi,Agarwal,21,9848022330,Pune,93),(3
```



```
,Rajesh,Khanna,22,9848022339,Delhi,90),(2,siddarth,Battacharya,22,9848022338,Ko  
lkata,78),(1,Rajiv,Reddy,21,9848022337,Hyderabad,89))}
```

Let us now calculate the global maximum of GPA, i.e., maximum among the GPA values of all the students using the **MAX** function as shown below.

```
grunt> student_gpa_max = foreach student_group_all Generate  
(student_details.firstname, student_details.gpa), MAX(student_details.gpa);
```

Verification

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

```
grunt> Dump student_gpa_max;
```

Output

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

```
(({(Bharathi),(Komal),(Archana),(Trupthi),(Preethi),(Rajesh),(siddarth),(Rajiv)  
} ,  
 { (72) , (83) , (87) , (75) , (93) , (90) ,  
(78) , (89) }) ,93)
```

Min

The **Min()** function of Pig Latin is used to get the minimum (lowest) value (numeric or chararray) for a certain column in a single-column bag. While calculating the minimum value, the **Min()** function ignores the NULL values.

Note:

- To get the global minimum value, we need to perform a **Group All** operation, and calculate the average value using the AVG function.
- To get the minimum value of a group, we need to group it using the **Group By** operator and proceed with the average function.

Syntax

Given below is the syntax of the **Min()** function.

```
grunt> MIN(expression)
```

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,89
002,siddarth,Battacharya,22,9848022338,Kolkata,78
003,Rajesh,Khanna,22,9848022339,Delhi,90
004,Preethi,Agarwal,21,9848022330,Pune,93
005,Trupthi,Mohanthi,23,9848022336,Bhuwaneshwar,75
006,Archana,Mishra,23,9848022335,Chennai,87
007,Komal,Nayak,24,9848022334,trivendram,83
008,Bharathi,Nambiyar,24,9848022333,Chennai,72
```

And we have loaded this file into Pig with the schema named **student** as shown below.

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

Calculating the Minimum GPA

We can use the built-in function **MIN()** (case sensitive) to calculate the minimum value from a set of given numerical values. Let us group the schema **student_details** using the **Group All** operator, and store the result in the schema named **student_group_all** as shown below.

```
grunt> student_group_all = Group student_details All;
```

It will produce a schema as shown below.

```
grunt> Dump student_group_all;

(all, {(8,Bharathi,Nambiyar,24,9848022333,Chennai,72), (7,Komal,Nayak,24,9848022334,trivendram,83), (6,Archana,Mishra,23,9848022335,Chennai,87), (5,Trupthi,Mohanthi,23,9848022336,Bhuwaneshwar,75), (4,Preethi,Agarwal,21,9848022330,Pune,93), (3,Rajesh,Khanna,22,9848022339,Delhi,90), (2,siddarth,Battacharya,22,9848022338,Kolkata,78), (1,Rajiv,Reddy,21,9848022337,Hyderabad,89)})
```

Let us now calculate the global minimum of GPA, i.e., minimum among the GPA values of all the students using the **MIN** function as shown below.

```
grunt> student_gpa_min = foreach student_group_all Generate
(student_details.firstname, student_details.gpa), MIN(student_details.gpa);
```

Verification

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

```
grunt> Dump student_gpa_min;
```

Output

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

```
(({(Bharathi),(Komal),(Archana),(Trupthi),(Preethi),(Rajesh),(siddarth),(Rajiv)
} ,
{ (72) , (83) , (87) , (75) , (93) , (90) ,
(78) , (89) }) ,72)
```

Count

The **count()** function of Pig Latin is used to get the number of elements in a bag. While counting the number of tuples in a bag, the **count()** function ignores (will not count) the tuples having a NULL value in the FIRST FIELD.

Note:

- To get the global count value (total number of tuples in a bag), we need to perform a **Group All** operation, and calculate the average value using the AVG function.
- To get the count value of a group (Number of tuples in a group), we need to group it using the Group By operator and proceed with the average function.

Syntax

Given below is the syntax of the **count()** function.

```
grunt> COUNT(expression)
```

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,89
002,siddarth,Battacharya,22,9848022338,Kolkata,78
003,Rajesh,Khanna,22,9848022339,Delhi,90
004,Preethi,Agarwal,21,9848022330,Pune,93
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar,75
006,Archana,Mishra,23,9848022335,Chennai,87
007,Komal,Nayak,24,9848022334,trivendram,83
008,Bharathi,Nambiayar,24,9848022333,Chennai,72
```

And we have loaded this file into Pig with the schema named **student** as shown below.

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

Calculating the Number of Tuples

We can use the built-in function **COUNT()** (case sensitive) to calculate the number of tuples in a relation. Let us group the schema **student_details** using the **Group All** operator, and store the result in the schema named **student_group_all** as shown below.

```
grunt> student_group_all = Group student_details All;
```

It will produce a schema as shown below.

```
grunt> Dump student_group_all;

(all, {(8, Bharathi, Nambiayar, 24, 9848022333, Chennai, 72), (7, Komal, Nayak, 24, 9848022334, trivendram, 83), (6, Archana, Mishra, 23, 9848022335, Chennai, 87), (5, Trupthi, Mohanthy, 23, 9848022336, Bhuwaneshwar, 75), (4, Preethi, Agarwal, 21, 9848022330, Pune, 93), (3, Rajesh, Khanna, 22, 9848022339, Delhi, 90), (2, siddarth, Battacharya, 22, 9848022338, Kolkata, 78), (1, Rajiv, Reddy, 21, 9848022337, Hyderabad, 89)})
```

Let us now calculate number of tuples/records in the relation.

```
grunt> student_count = foreach student_group_all Generate
COUNT(student_details.gpa);
```

Verification

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

```
grunt> Dump student_count;
```

Output

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

```
8
```

COUNT_STAR

The **COUNT_STAR()** function of Pig Latin is similar to the **COUNT()** function. It is used to get the number of elements in a bag. While counting the elements, the **COUNT_STAR()** function includes the NULL values.

Note:

- To get the global count value (total number of tuples in a bag), we need to perform a **Group All** operation, and calculate the average value using the **AVG** function.
- To get the count value of a group (Number of tuples in a group), we need to group it using the **Group By** operator and proceed with the average function.

Syntax

Given below is the syntax of the COUNT_STAR function.

```
grunt> COUNT_STAR(expression)
```

Example

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

student_details.txt

```
, , , , , , ,
001,Rajiv,Reddy,21,9848022337,Hyderabad,89
002,siddarth,Battacharya,22,9848022338,Kolkata,78
003,Rajesh,Khanna,22,9848022339,Delhi,90
004,Preethi,Agarwal,21,9848022330,Pune,93
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar,75
006,Archana,Mishra,23,9848022335,Chennai,87
007,Komal,Nayak,24,9848022334,trivendram,83
008,Bharathi,Nambiayar,24,9848022333,Chennai,72
```

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

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

Calculating the Number of Tuples

We can use the built-in function **COUNT_STAR()** to calculate the number of tuples in a relation. Let us group the schema **student_details** using the **Group All** operator, and store the result in the schema named **student_group_all** as shown below.

```
grunt> student_group_all = Group student_details All;
```

It will produce a schema as shown below.

```
grunt> Dump student_group_all;

(all, {(8,Bharathi,Nambiayar,24,9848022333,Chennai,72),(7,Komal,Nayak,24,9848022
334,trivendram,83),(6,Archana,Mishra,23,9848022335,Chennai,87),(5,Trupthi,Mohan
thy,23,9848022336,Bhuwaneshwar,75),(4,Preethi,Agarwal,21,9848022330,Pune,93),(3
,Rajesh,Khanna,22,9848022339,Delhi,90),(2,siddarth,Battacharya,22,9848022338,Ko
lkata,78),(1,Rajiv,Reddy,21,9848022337,Hyderabad,89),( , , , , , )})
```

Let us now calculate the number of tuples/records in the relation.

```
grunt> student_count = foreach student_group_all Generate
COUNT_STAR(student_details.gpa);
```

Verification

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

```
grunt> Dump student_count;
```

Output

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

```
9
```

Since we have used the function COUNT_STAR, it included the null tuple and returned 9.

Sum

You can use the **Sum()** function of Pig Latin to get the total of the numeric values of a column in a single-column bag. While computing the total, the sum() function ignores the NULL values.

Note:

- To get the global sum value, we need to perform a **Group All** operation, and calculate the average value using the AVG function.
- To get the sum value of a group, we need to group it using the **Group By** operator and proceed with the average function.

Syntax

Given below is the syntax of the **sum()** function.

```
grunt> SUM(expression)
```

Example

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

employee.txt

```
1, John, 2007-01-24, 250
2, Ram, 2007-05-27, 220
3, Jack, 2007-05-06, 170
3, Jack, 2007-04-06, 100
```

```
4,Jill,2007-04-06,220
5,Zara,2007-06-06,300
5,Zara,2007-02-06,350
```

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

```
grunt> employee_data = LOAD 'hdfs://localhost:9000/pig_data/ employee.txt'
USING PigStorage(',')as (id:int, name:chararray, workdate:chararray,
daily_typing_pages:int);
```

Calculating the Sum of All GPA

To demonstrate the **SUM()** function, let's try to calculate the total number of pages typed daily of all the employees. We can use the Apache Pig's built-in function **SUM()** (case sensitive) to calculate the sum of the numerical values. Let us group the schema **employee_data** using the **Group All** operator, and store the result in the schema named **employee_group** as shown below.

```
grunt> employee_group = Group employee_data all;
```

It will produce a schema as shown below.

```
grunt> Dump employee_group;

(all, {(5,Zara,2007-02-06,350),(5,Zara,2007-06-06,300),(4,Jill,2007-04-06,220),(3,Jack,2007-04-06,100),(3,Jack,2007-05-06,170),(2,Ram,2007-05-27,220),(1,John,2007-01-24,250)})
```

Let us now calculate the global sum of the pages typed daily.

```
grunt> student_workpages_sum = foreach employee_group Generate
(employee_data.name,employee_data.daily_typing_pages),SUM(employee_data.daily_t
yping_pages);
```

Verification

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

```
grunt> Dump student_workpages_sum;
```

Output

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

```
(( { (Zara), (Zara), (Jill) ,(Jack) , (Jack) , (Ram) , (John) },
 { (350) , (300) , (220) ,(100) , (170) , (220) , (250) } ),1610)
```

DIFF

The **DIFF()** function of Pig Latin is used to compare two bags (fields) in a tuple. It takes two fields of a tuple as input and matches them. If they match, it returns an empty bag. If they do not match, it finds the elements that exist in one field (bag) and not found in the other, and returns these elements by wrapping them within a bag.

Syntax

Given below is the syntax of the **DIFF()** function.

```
grunt> DIFF (expression, expression)
```

Example

Generally the **Diff()** function compares two bags in a tuple. Given below is an example of the **DIFF()** function. Here we consider two schemas, cogroup them, and perform **DIFF()** function on them.

Assume that we have two files namely **emp_sales.txt** and **emp_bonus.txt** in the HDFS directory **/pig_data/** as shown below. The **emp_sales.txt** contains the details of the employees of the sales department and the **emp_bonus.txt** contains the employee details who got bonus.

emp_sales.txt

```
1,Robin,22,25000,sales
2,BOB,23,30000,sales
3,Maya,23,25000,sales
4,Sara,25,40000,sales
5,David,23,45000,sales
6,Maggy,22,35000,sales
```

emp_bonus.txt

```
1,Robin,22,25000,sales
2,Jaya,23,20000,admin
3,Maya,23,25000,sales
4,Alia,25,50000,admin
5,David,23,45000,sales
6,Omar,30,30000,admin
```

And we have loaded these files into Pig, with the schema names **emp_sales** and **emp_bonus** respectively.

```
emp_sales = LOAD 'hdfs://localhost:9000/pig_data/emp_sales.txt' USING
PigStorage(',')as (sno:int, name:chararray, age:int, salary:int,
dept:chararray);
```



```
emp_bonus = LOAD 'hdfs://localhost:9000/pig_data/emp_bonus.txt' USING
PigStorage(',')as (sno:int, name:chararray, age:int, salary:int,
dept:chararray);
```

Group the records/tuples of the relations **emp_sales** and **emp_bonus** with the key **sno**, using the COGROUP operator as shown below.

```
cogroup_data = COGROUP emp_sales by sno, emp_bonus by sno;
```

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

```
grunt> Dump cogroup_data;

(1,{{(1,Robin,22,25000,sales)},{(1,Robin,22,15000,sales)}})
(2,{{(2,BOB,23,30000,sales)},{(2,Jaya,23,12000,admin)}})
(3,{{(3,Maya,23,25000,sales)},{(3,Maya,23,10000,sales)}})
(4,{{(4,Sara,25,40000,sales)},{(4,Alia,25,8000,admin)}})
(5,{{(5,David,23,45000,sales)},{(5,David,23,6000,sales)}})
(6,{{(6,Maggy,22,35000,sales)},{(6,Omar,30,3000,admin)}})
```

Calculating the Difference between Two Schemas

Let us now calculate the difference between the two schemas using **DIFF()** function and store it in the schema **diff_data** as shown below.

```
diff_data = FOREACH cogroup_data GENERATE DIFF(emp_sales,emp_bonus);
```

Verification

Verify the schema **diff_data** using the DUMP operator as shown below.

```
Dump diff_data;

({})
({(2,BOB,23,30000,sales),(2,Jaya,23,20000,admin)})
({})
({(4,Sara,25,40000,sales),(4,Alia,25,50000,admin)})
({})
({(6,Maggy,22,35000,sales),(6,Omar,30,30000,admin)})
```

The **diff_data** schema will have an empty tuple if the records in **emp_bonus** and **emp_sales** match. In other cases, it will hold tuples from both the schemas (tuples that differ).

For example, if you consider the records having **sno** as **1**, then you will find them same in both the schemas (**(1,Robin,22,25000,sales)**, **(1,Robin,22,15000,sales)**). Therefore,

in the **diff_data** schema, which is the result of **DIFF()** function, you will get an empty tuple for **sno 1**.

SUBTRACT

The **subtract()** function of Pig Latin is used 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.

Syntax

Given below is the syntax of the **subtract()** function.

```
SUBTRACT(expression, expression)
```

Example

Assume that we have two files namely **emp_sales.txt** and **emp_bonus.txt** in the HDFS directory **/pig_data/** as shown below. The **emp_sales.txt** contains the details of the employees of the sales department and the **emp_bonus.txt** contains the employee details who got bonus.

emp_sales.txt

```
1,Robin,22,25000,sales
2,BOB,23,30000,sales
3,Maya,23,25000,sales
4,Sara,25,40000,sales
5,David,23,45000,sales
6,Maggy,22,35000,sales
```

emp_bonus.txt

```
1,Robin,22,25000,sales
2,Jaya,23,20000,admin
3,Maya,23,25000,sales
4,Alia,25,50000,admin
5,David,23,45000,sales
6,Omar,30,30000,admin
```

And we have loaded these files into Pig, with the schema names **emp_sales** and **emp_bonus** respectively.

```
emp_sales = LOAD 'hdfs://localhost:9000/pig_data/emp_sales.txt' USING
PigStorage(',')as (sno:int, name:chararray, age:int, salary:int,
dept:chararray);
```

```
emp_bonus = LOAD 'hdfs://localhost:9000/pig_data/emp_bonus.txt' USING
PigStorage(',')as (sno:int, name:chararray, age:int, salary:int,
dept:chararray);
```

Let us now group the records/tuples of the relations **emp_sales** and **emp_bonus** with the key **sno**, using the COGROUP operator as shown below.

```
cogroup_data = COGROUP emp_sales by sno, emp_bonus by sno;
```

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

```
grunt> Dump cogroup_data;
(1,{{(1,Robin,22,25000,sales)},{(1,Robin,22,15000,sales)}})
(2,{{(2,BOB,23,30000,sales)},{(2,Jaya,23,12000,admin)}})
(3,{{(3,Maya,23,25000,sales)},{(3,Maya,23,10000,sales)}})
(4,{{(4,Sara,25,40000,sales)},{(4,Alia,25,8000,admin)}})
(5,{{(5,David,23,45000,sales)},{(5,David,23,6000,sales)}})
(6,{{(6,Maggy,22,35000,sales)},{(6,Omar,30,3000,admin)}})
```

Subtracting One Schema from the Other

Let us now subtract the tuples of **emp_bonus** schema from **emp_sales** schema. The resulting schema holds the tuples of **emp_sales** that are not there in **emp_bonus**.

```
sub_data = FOREACH cogroup_data GENERATE SUBTRACT(emp_sales, emp_bonus);
```

Verification

Verify the schema **sub_data** using the DUMP operator as shown below. The **emp_sales** schema holds the tuples that are not there in the schema **emp_bonus**.

```
Dump sub_data;

({})
{{(2,BOB,23,30000,sales)}}
({})
{{(4,Sara,25,40000,sales)}}
({})
{{(6,Maggy,22,35000,sales)}}
({})
```

In the same way, let us subtract the **emp_sales** schema from **emp_bonus** schema as shown below.

```
sub_data = FOREACH cogroup_data GENERATE SUBTRACT(emp_bonus, emp_sales);
```

Verify the contents of the **sub_data** schema using the Dump operator as shown below.

```
({})
{{(2,Jaya,23,20000,admin)}}
({})
{{(4,Alia,25,50000,admin)}}
({})
{{(6,Omar,30,30000,admin)}}000,admin}}
```

IsEmpty

The **isEmpty()** function of Pig Latin is used to check if a bag or map is empty.

Syntax

Given below is the syntax of the IsEmpty() function.

```
IsEmpty(expression)
```

Example

Assume that we have two files namely **emp_sales.txt** and **emp_bonus.txt** in the HDFS directory **/pig_data/** as shown below. The **emp_sales.txt** contains the details of the employees of the sales department and the **emp_bonus.txt** contains the employee details who got bonus.

emp_sales.txt

```
1,Robin,22,25000,sales
2,BOB,23,30000,sales
3,Maya,23,25000,sales
4,Sara,25,40000,sales
5,David,23,45000,sales
6,Maggy,22,35000,sales
```

emp_bonus.txt

```
1,Robin,22,25000,sales
2,Jaya,23,20000,admin
3,Maya,23,25000,sales
4,Alia,25,50000,admin
5,David,23,45000,sales
6,Omar,30,30000,admin
```

And we have loaded these files into Pig, with the schema names **emp_sales** and **emp_bonus** respectively, as shown below.

```
emp_sales = LOAD 'hdfs://localhost:9000/pig_data/emp_sales.txt' USING
PigStorage(',')as (sno:int, name:chararray, age:int, salary:int,
dept:chararray);

emp_bonus = LOAD 'hdfs://localhost:9000/pig_data/emp_bonus.txt' USING
PigStorage(',')as (sno:int, name:chararray, age:int, salary:int,
dept:chararray);
```

Let us now group the records/tuples of the relations **emp_sales** and **emp_bonus** with the key **age**, using the **cogroup** operator as shown below.

```
cogroup_data = COGROUP emp_sales by sno, emp_bonus by age;
```

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

```
grunt> Dump cogroup_data;

(22, {(6, Maggy, 22, 35000, sales), (1, Robin, 22, 25000, sales)},
      {(1, Robin, 22, 25000, sales)})
(23, {(5, David, 23, 45000, sales), (3, Maya, 23, 25000, sales), (2, BOB, 23, 30000, sales)},
      {(5, David, 23, 45000, sales), (3, Maya, 23, 25000, sales), (2, Jaya, 23, 20000, admin)})

(25, {(4, Sara, 25, 40000, sales)}, {(4, Alia, 25, 50000, admin)})
(30, {}, {(6, Omar, 30, 30000, admin)})
```

The COGROUP operator groups the tuples from each schema according to age. Each group depicts a particular age value.

For example, if we consider the 1st tuple of the result, it is grouped by age 22. And it contains two bags, the first bag holds all the tuples from the first schema (student_details in this case) having age 22, and the second bag contains all the tuples from the second schema (employee_details in this case) having age 22. In case a schema doesn't have tuples having the age value 22, it returns an empty bag.

Getting the Groups having Empty Bags

Let's list such empty bags from the **emp_sales** schema in the group using the **IsEmpty()** function.

```
isempty_data = filter cogroup_data by IsEmpty(emp_sales);
```

Verification

Verify the schema **isempty_data** using the DUMP operator as shown below. The **emp_sales** schema holds the tuples that are not there in the schema **emp_bonus**.

```
Dump isempty_data;

(30, {}, {(6, Omar, 30, 30000, admin)})
```

Pluck Tuple

After performing operations like join to differentiate the columns of the two schemas, we use the function **PluckTuple()**. To use this function, first of all, we have to define a string Prefix and we have to filter for the columns in a relation that begin with that prefix.

Syntax

Given below is the syntax of the **PluckTuple()** function.

```
DEFINE pluck PluckTuple(expression1)
DEFINE pluck PluckTuple(expression1, expression3)
pluck(expression2)
```

Example

Assume that we have two files namely **emp_sales.txt** and **emp_bonus.txt** in the HDFS directory **/pig_data/**. The **emp_sales.txt** contains the details of the employees of the sales department and the **emp_bonus.txt** contains the employee details who got bonus.

emp_sales.txt

```
1,Robin,22,25000,sales
2,BOB,23,30000,sales
3,Maya,23,25000,sales
4,Sara,25,40000,sales
5,David,23,45000,sales
6,Maggy,22,35000,sales
```

emp_bonus.txt

```
1,Robin,22,25000,sales
2,Jaya,23,20000,admin
3,Maya,23,25000,sales
4,Alia,25,50000,admin
5,David,23,45000,sales
```

```
6,Omar,30,30000,admin
```

And we have loaded these files into Pig, with the schema names **emp_sales** and **emp_bonus** respectively.

```
emp_sales = LOAD 'hdfs://localhost:9000/pig_data/emp_sales.txt' USING
PigStorage(',')as (sno:int, name:chararray, age:int, salary:int,
dept:chararray);

emp_bonus = LOAD 'hdfs://localhost:9000/pig_data/emp_bonus.txt' USING
PigStorage(',')as (sno:int, name:chararray, age:int, salary:int,
dept:chararray);
```

Join these two schemas using the **join** operator as shown below.

```
join_data = join emp_sales by sno, emp_bonus by sno;
```

Verify the schema **join_data** using the **Dump** operator.

```
grunt> Dump join_data;
(1,Robin,22,25000,sales,1,Robin,22,25000,sales)
(2,BOB,23,30000,sales,2,Jaya,23,20000,admin)
(3,Maya,23,25000,sales,3,Maya,23,25000,sales)
(4,Sara,25,40000,sales,4,Alia,25,50000,admin)
(5,David,23,45000,sales,5,David,23,45000,sales)
(6,Maggy,22,35000,sales,6,Omar,30,30000,admin)
```

Using PluckTuple() Function

Now, define the required expression by which you want to differentiate the columns using **PluckTupe()** function.

```
DEFINE pluck PluckTuple('a::');
```

Filter the columns in the **join_data** relation as shown below.

```
data = foreach test generate FLATTEN(pluck(*));
```

Verify the schema of the **join_data** schema using the **describe** operator.

```
Describe test;
test: {emp_sales::sno: int,emp_sales::name: chararray,emp_sales::age:
int,emp_sales::salary: int,emp_sales::dept: chararray,emp_bonus::sno:
int,emp_bonus::name: chararray,emp_bonus::age: int,emp_bonus::salary:
int,emp_bonus::dept: chararray}
```

Since we have defined the expression as **"a::"**, the columns of the **emp_sales** schema are plucked as **emp_sales::column name** and the columns of the **emp_bonus** schema are plucked as **emp_bonus::column name**

Size ()

The **size()** function of Pig Latin is used to compute the number of elements based on any Pig data type.

Syntax

Given below is the syntax of the **size()** function.

```
SIZE(expression)
```

The return values vary according to the data types in Apache Pig.

Data type	Value
int, long, float, double	For all these types, the size function returns 1.
Char array	For a char array, the size() function returns the number of characters in the array.
Byte array	For a bytearray, the size() function returns the number of bytes in the array.
Tuple	For a tuple, the size() function returns number of fields in the tuple.
Bag	For a bag, the size() function returns number of tuples in the bag.
Map	For a map, the size() function returns the number of key/value pairs in the map.

Example

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

employee.txt

```
1, John, 2007-01-24, 250
2, Ram, 2007-05-27, 220
3, Jack, 2007-05-06, 170
3, Jack, 2007-04-06, 100
4, Jill, 2007-04-06, 220
5, Zara, 2007-06-06, 300
5, Zara, 2007-02-06, 350
```


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

```
grunt> employee_data = LOAD 'hdfs://localhost:9000/pig_data/ employee.txt'
USING PigStorage(',')as (id:int, name:chararray, workdate:chararray,
daily_typing_pages:int);
```

Calculating the Size of the Type

To calculate the size of the type of a particular column, we can use the **size()** function. Let's calculate the size of the name type as shown below.

```
grunt> size = FOREACH employee_data GENERATE SIZE(name);
```

Verification

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

```
grunt> Dump size;
```

Output

It will produce the following output, displaying the contents of the relation **size** as follows. In the example, we have calculated the size of the **name** column. Since it is of varchar type, the **size** function gives you the number of characters in the name of each employee.

```
(4)
(3)
(4)
(4)
(4)
(4)
(4)
(4)
```

BagToString ()

The Pig Latin **BagToString()** function is used to concatenate the elements of a bag into a string. While concatenating, we can place a delimiter between these values (optional).

Generally bags are disordered and we can order them by using the ORDER BY operator.

Syntax

Given below is the syntax of the **BagToString()** function.

```
BagToString(vals:bag [, delimiter:chararray])
```

Example

Assume that we have a file named **dateofbirth.txt** in the HDFS directory **/pig_data/** as shown below. This file contains the date-of-births.

employee.txt

```
22,3,1990
23,11,1989
1,3,1998
2,6,1980
26,9,1989
```

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

```
grunt> dob = LOAD 'hdfs://localhost:9000/pig_data/dob.txt' USING
PigStorage(',')as (day:int, month:int, year:int);
```

Converting Bag to String

Using the **bagtostring()** function, we can convert the data in the bag to string. Let us group the **dob** schema. The group operation will produce a bag containing all the tuples of the schema.

Group the schema **dob** using the **Group All** operator, and store the result in the schema named **group_dob** as shown below.

```
grunt> group_dob = Group dob all;
```

It will produce a schema as shown below.

```
grunt> Dump group_dob;

(all, {(26,9,1989), (2,6,1980), (1,3,1998), (23,11,1989), (22,3,1990)})
```

Here, we can observe a bag having all the date-of-births as tuples of it. Now, let's convert the bag to string using the function **BagToString()**.

```
grunt> dob_string = foreach group_dob Generate BagToString(dob);
```

Verification

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

```
grunt> Dump dob_string;
```

Output

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

```
(26_9_1989_2_6_1980_1_3_1998_23_11_1989_22_3_1990)
```

Concat ()

The **CONCAT()** function of Pig Latin is used to concatenate two or more expressions of the same type.

Syntax

```
CONCAT (expression, expression, [...expression])
```

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,89
002,siddarth,Battacharya,22,9848022338,Kolkata,78
003,Rajesh,Khanna,22,9848022339,Delhi,90
004,Preethi,Agarwal,21,9848022330,Pune,93
005,Trupthi,Mohanthi,23,9848022336,Bhuwaneshwar,75
006,Archana,Mishra,23,9848022335,Chennai,87
007,Komal,Nayak,24,9848022334,trivendram,83
008,Bharathi,Nambiayar,24,9848022333,Chennai,72
```

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

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

Concatenating Two Strings

We can use the **concat()** function to concatenate two or more expressions. First of all, verify the contents of the **student_details** schema using the Dump operator as shown below.

```
grunt> Dump student_details;
( 1,Rajiv,Reddy,21,9848022337,Hyderabad,89 )
( 2,siddarth,Battacharya,22,9848022338,Kolkata,78 )
( 3,Rajesh,Khanna,22,9848022339,Delhi,90 )
( 4,Preethi,Agarwal,21,9848022330,Pune,93 )
( 5,Trupthi,Mohanthi,23,9848022336,Bhuwaneshwar,75 )
( 6,Archana,Mishra,23,9848022335,Chennai,87 )
( 7,Komal,Nayak,24,9848022334,trivendram,83 )
( 8,Bharathi,Nambiayar,24,9848022333,Chennai,72 )
```

And, verify the schema using **describe** operator as shown below.

```
grunt> Describe student_details;

student_details: {id: int, firstname: chararray, lastname: chararray, age: int,
phone: chararray, city: chararray, gpa: int}
```

In the above schema, you can observe that the name of the student is represented using two chararray values namely **firstname** and **lastname**. Let us concatenate these two values using the **CONCAT()** function.

```
grunt> student_name_concat = foreach student_group_all Generate CONCAT
(firstname, lastname);
```

Verification

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

```
grunt> Dump student_name_concat;
```

Output

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

```
(RajivReddy)
(siddarthBattacharya)
(RajeshKhanna)
(PreethiAgarwal)
(TrupthiMohanthly)
(ArchanaMishra)
(KomalNayak)
(BharathiNambiayar)
```

We can also use an optional delimiter between the two expressions as shown below.

```
CONCAT(firstname, '_',lastname);
```

Now, let us concatenate the first name and last name of the student records in the **student_details** schema by placing **'_'** between them as shown below.

```
grunt> student_name_concat = foreach student_gpa GENERATE CONCAT(firstname,
'_',lastname);
grunt> Dump student_name_concat;
```

Verification

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

```
grunt> Dump student_name_concat;
```

Output

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

```
(Rajiv_Reddy)
(siddarth_Battacharya)
(Rajesh_Khanna)
(Preethi_Agarwal)
(Trupthi_Mohanthy)
(Archana_Mishra)
(Komal_Nayak)
(Bharathi_Nambiayar)
```

Tokenize ()

The **Tokenize** function of Pig Latin is used 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.

Syntax

Given below is the syntax of the Tokenize operation.

```
TOKENIZE(expression [, 'field_delimiter'])
```

As a delimiter to the tokenize function, we can pass space [], double quote [" "], coma [,], parenthesis [()], star [*].

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,Hyderabad
002,siddarth_Battacharya,22,Kolkata
003,Rajesh_Khanna,22,Delhi
004,Preethi_Agarwal,21,Pune
005,Trupthi_Mohanthy,23,Bhuwaneshwar
006,Archana_Mishra,23 ,Chennai
007,Komal_Nayak,24,trivendram
008,Bharathi_Nambiayar,24,Chennai
```

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

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

Tokenizing a String

We can use the **Tokenize()** function to split a string. First of all, verify the contents of the **student_details** schema using the Dump operator as shown below.

```
grunt> Dump student_details;

( 1,Rajiv,Reddy,21,9848022337,Hyderabad,89 )
( 2,siddarth,Battacharya,22,9848022338,Kolkata,78 )
( 3,Rajesh,Khanna,22,9848022339,Delhi,90 )
( 4,Preethi,Agarwal,21,9848022330,Pune,93 )
( 5,Trupthi,Mohanthi,23,9848022336,Bhuwaneshwar,75 )
( 6,Archana,Mishra,23,9848022335,Chennai,87 )
( 7,Komal,Nayak,24,9848022334,trivendram,83 )
( 8,Bharathi,Nambiayar,24,9848022333,Chennai,72 )
```

And, verify the schema using **describe** operator as shown below.

```
grunt> Describe student_details;

student_details: {id: int, firstname: chararray, lastname: chararray, age: int,
phone: chararray, city: chararray, gpa: int}
```

In the above schema, you can observe that the name of the student is represented using two chararray values namely **firstname** and **lastname**. Let us concatenate these two values using the **concat()** function as shown below.

```
grunt> student_name_concat = foreach student_group_all Generate CONCAT
(firstname, lastname);
```

Verification

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

```
grunt> Dump student_name_concat;
```

Output

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

```
(RajivReddy)
(siddarthBattacharya)
(RajeshKhanna)
```

(PreethiAgarwal)

(TrupthiMohanthu)

(ArchanaMishra)

(KomalNayak)

(BharathiNambiayar)

25. Load and Store Functions

The load/store functions in Apache Pig are used to determine how the data goes and 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.

Function	Description
PigStorage	To load and store structured files.
TextLoader	To load unstructured data into Pig.
BinStorage	To load and store data into Pig using machine readable format.
Handling Compression	In Pig Latin, we can load and store compressed data.

PigStorage ()

The PigStorage function loads and stores data as structured text files. It takes a delimiter using which each entity of a tuple is separated as a parameter. By default, it takes `'\t'` as a parameter.

Syntax

Given below is the syntax of the PigStorage() function.

```
PigStorage(field_delimiter)
```

Example

Let us suppose we have a file named **student_data.txt** in the HDFS directory named **/data/** 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,Mohanthi,9848022336,Bhuvaneshwar
006,Archana,Mishra,9848022335,Chennai.
```


We can load the data using the PigStorage function 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 );
```

In the above example, we have seen that we have used comma (',') delimiter. Therefore, we have separated the values of a record using (,).

In the same way, we can use the **PigStorage()** function to store the data in to HDFS directory as shown below.

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

This will store the data into the given directory. You can verify the data as shown below.

Verification

You can verify the stored data as shown below. First of all, list out the files in the directory named **pig_output** using **ls** command as shown below.

```
hdfs dfs -ls 'hdfs://localhost:9000/pig_Output/'
Found 2 items
-rw-r--r- 1 Hadoop supergroup 0 2015-10-05 13:03
hdfs://localhost:9000/pig_Output/_SUCCESS
-rw-r--r- 1 Hadoop supergroup 224 2015-10-05 13:03
hdfs://localhost:9000/pig_Output/part-m-00000
```

You can observe that two files were created after executing the **Store** statement.

Then, using the **cat** command, list the contents of the file named **part-m-00000** as shown below.

```
$ hdfs dfs -cat 'hdfs://localhost:9000/pig_Output/part-m-00000'

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

TextLoader ()

The Pig Latin function **TextLoader()** is a Load function which is used to load unstructured data in UTF-8 format.

Syntax

Given below is the syntax of **TextLoader()** function.

```
TextLoader()
```

Example

Let us assume there is a file with named **stu_data.txt** in the HDFS directory named **/data/** as shown below.

```
001,Rajiv_Reddy,21,Hyderabad
002,siddarth_Battacharya,22,Kolkata
003,Rajesh_Khanna,22,Delhi
004,Preethi_Agarwal,21,Pune
005,Trupthi_Mohanthy,23,Bhuwaneshwar
006,Archana_Mishra,23,Chennai
007,Komal_Nayak,24,trivendram
008,Bharathi_Nambiayar,24,Chennai
```

Now let us load the above file using the **TextLoader()** function.

```
grunt> details = LOAD 'hdfs://localhost:9000/pig_data/stu_data.txt' USING
TextLoader();
```

You can verify the loaded data using the Dump operator.

```
grunt> dump;

(001,Rajiv_Reddy,21,Hyderabad)
(002,siddarth_Battacharya,22,Kolkata)
(003,Rajesh_Khanna,22,Delhi)
(004,Preethi_Agarwal,21,Pune)
(005,Trupthi_Mohanthy,23,Bhuwaneshwar)
(006,Archana_Mishra,23,Chennai)
(007,Komal_Nayak,24,trivendram)
(008,Bharathi_Nambiayar,24,Chennai)
```

BinStorage ()

The **BinStorage()** function is used to load and store the data into Pig using machine readable format. **BinStorage()** in Pig is generally used to store temporary data generated between the MapReduce jobs. It supports multiple locations as input.

Syntax

Given below is the syntax of the BinStorage() function.

```
BinStorage();
```

Example

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

Stu_data.txt

```
001,Rajiv_Reddy,21,Hyderabad
002,siddarth_Battacharya,22,Kolkata
003,Rajesh_Khanna,22,Delhi
004,Preethi_Agarwal,21,Pune
005,Trupthi_Mohanthy,23,Bhuwaneshwar
006,Archana_Mishra,23,Chennai
007,Komal_Nayak,24,trivendram
008,Bharathi_Nambiayar,24,Chennai
```

Let us load this data into Pig into a schema as shown below.

```
student_details = LOAD 'hdfs://localhost:9000/pig_data/stu_data.txt' USING
PigStorage(',')as (id:int, firstname:chararray, age:int, city:chararray);
```

Now, we can **store** this schema into the HDFS directory named **/pig_data/** using the **BinStorage()** function.

```
STORE student_details INTO 'hdfs://localhost:9000/pig_Output/mydata' USING
BinStorage();
```

After executing the above statement, the schema is stored in the given HDFS directory. You can verify it using the HDFS **cat command** as shown below.

```
[Hadoop@localhost sbin]$ hdfs dfs -ls hdfs://localhost:9000/pig_Output/mydata/

Found 2 items
-rw-r--r--  1 Hadoop supergroup      0 2015-10-26 16:58
hdfs://localhost:9000/pig_Output/mydata/_SUCCESS
-rw-r--r--  1 Hadoop supergroup    372 2015-10-26 16:58
hdfs://localhost:9000/pig_Output/mydata/part-m-00000
```

Now, load the data from the file **part-m-00000**.

```
result = LOAD 'hdfs://localhost:9000/pig_Output/b/part-m-00000' USING
BinStorage();
```

Verify the contents of the schema as shown below.

```
Dump result;
(1,Rajiv_Reddy,21,Hyderabad)
(2,siddarth_Battacharya,22,Kolkata)
(3,Rajesh_Khanna,22,Delhi)
(4,Preethi_Agarwal,21,Pune)
(5,Trupthi_Mohanthly,23,Bhuwaneshwar)
(6,Archana_Mishra,23,Chennai)
(7,Komal_Nayak,24,trivendram)
(8,Bharathi_Nambiayar,24,Chennai)
```

Handling Compression

We can load/store compressed data in Apache Pig using the functions **BinStorage()** and **TextLoader()**.

Example

Assume we have a file named **employee.txt.zip** in the HDFS directory /pigdata/. Then, we can load the compressed file into pig as shown below.

Using PigStorage:

```
grunt > data = LOAD 'hdfs://localhost:9000/pig_data/employee.txt.zip' USING
PigStorage(',');
```

Using TextLoader:

```
grunt > data = LOAD 'hdfs://localhost:9000/pig_data/employee.txt.zip' USING
TextLoader;
```

In the same way, we can store the compressed files into pig as shown below.

Using PigStorage:

```
grunt> store data INTO 'hdfs://localhost:9000/pig_Output/data.bz' USING
PigStorage(',');
```

26. Bag and Tuple Functions

Given below is the list of Bag and Tuple functions.

Function	Description
TOBAG	To convert two or more expressions into a bag.
TOP	To get the top N tuples of a relation.
TOTUPLE	To convert one or more expressions into a tuple.
TOMAP	To convert the key-value pairs into a Map.

TOBAG ()

The **TOBAG()** function of Pig Latin converts one or more expressions to individual tuples. And these tuples are placed in a bag.

Syntax

Given below is the syntax of the **TOBAG()** function.

```
TOBAG(expression [, expression ...])
```

Example

Assume we have a file named **employee_details.txt** in the HDFS directory **/pig_data/**, with the following content.

employee_details.txt

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

We have loaded the file into the Pig schema with the name **emp_data** as shown below.

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

Let us now convert the id, name, age and city, of each student (record) into a tuple as shown below.

```
tobag = FOREACH emp_data GENERATE TOBAG (id,name,age,city);
```

Verification

You can verify the contents of the **tobag** schema using the **Dump** operator as shown below.

```
DUMP tobag;

({(1),(Robin),(22),(newyork)})
({(2),(BOB),(23),(Kolkata)})
({(3),(Maya),(23),(Tokyo)})
({(4),(Sara),(25),(London)})
({(5),(David),(23),(Bhuaneshwar)})
({(6),(Maggy),(22),(Chennai)})
```

TOP ()

The TOP() function of Pig Latin is used to get the top N tuples of a bag. To this function, as inputs, we have to pass a relation, the number of tuples we want, and the column name whose values are being compared. This function will return a bag containing the required columns.

Syntax

Given below is the syntax of the function TOP().

```
TOP(topN,column,relation)
```

Example

Assume we have a file named **employee_details.txt** in the HDFS directory **/pig_data/**, with the following content.

employee_details.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
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
```

We have loaded the file into the Pig schema with the name **emp_data** as shown below.

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

Group the schema **emp_data** by age, and store it in the schema **emp_group**.

```
emp_group = Group emp_data BY age;
```

Verify the schema **emp_group** using the Dump operator as shown below.

```
Dump emp_group;

(22, {(12, Kelly, 22, Chennai), (7, Robert, 22, newyork), (6, Maggy, 22, Chennai), (1, Robin,
22, newyork)})
(23, {(8, Syam, 23, Kolkata), (5, David, 23, Bhuwaneshwar), (3, Maya, 23, Tokyo), (2, BOB, 23,
Kolkata)})
(25, {(11, Stacy, 25, Bhuwaneshwar), (10, Saran, 25, London), (9, Mary, 25, Tokyo), (4, Sara,
25, London)})
```

Now, you can get the top two records of each group arranged in ascending order (based on id) as shown below.

```
data_top = FOREACH emp_group {
top = TOP(2, 0, emp_data);
GENERATE top;
}
```

Verification

You can verify the contents of the **data_top** schema using the **Dump** operator as shown below.

```
Dump data_top;

({(7, Robert, 22, newyork), (12, Kelly, 22, Chennai)})
({(5, David, 23, Bhuwaneshwar), (8, Syam, 23, Kolkata)})
({(10, Saran, 25, London), (11, Stacy, 25, Bhuwaneshwar)})
```

TOTUPLE ()

The **TOTUPLE()** function is used to convert one or more expressions to the data type **tuple**.

Syntax

Given below is the syntax of the **TOTUPLE()** function.

```
TOTUPLE(expression [, expression ...])
```

Example

Assume we have a file named **employee_details.txt** in the HDFS directory **/pig_data/**, with the following content.

employee_details.txt

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

We have loaded the file into the Pig schema with the name **emp_data** as shown below.

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

Let us now convert the id, name and age of each student (record) into a tuple.

```
totuple = FOREACH emp_data GENERATE TOTUPLE (id,name,age);
```

Verification

You can verify the contents of the **totuple** schema using the **Dump** operator as shown below.

```
DUMP totuple;

((1,Robin,22))
((2,BOB,23))
((3,Maya,23))
((4,Sara,25))
((5,David,23))
((6,Maggy,22))
```


TOMAP ()

The **TOMAP()** function of Pig Latin is used to convert the key-value pairs into a Map.

Syntax

Given below is the syntax of the **TOMAP()** function.

```
TOMAP(key-expression, value-expression [, key-expression, value-expression ...])
```

Example

Assume we have a file named **employee_details.txt** in the HDFS directory **/pig_data/**, with the following content.

employee_details.txt

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

We have loaded the file into the Pig schema with the name **emp_data** as shown below.

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

Let us now take the name and age of each record as key-value pairs and convert them into map as shown below.

```
tomap = FOREACH emp_data GENERATE TOMAP(name, age);
```

Verification

You can verify the contents of the **tomap** schema using the **Dump** operator as shown below.

```
DUMP tomap;

([Robin#22])
([BOB#23])
([Maya#23])
([Sara#25])
([David#23])
([Maggy#22])
```

27. String Functions

We have the following String functions in Apache Pig.

Operator	Description
ENDSWITH	ENDSWITH(string, testAgainst) To verify whether a given string ends with a particular substring.
STARTSWITH	STARTSWITH(string, substring) Accepts two string parameters and verifies whether the first string starts with the second.
SUBSTRING	SUBSTRING(string, startIndex, stopIndex) Returns a substring from a given string.
EqualsIgnoreCase	EqualsIgnoreCase(string1, string2) To compare two strings ignoring the case.
INDEXOF	INDEXOF(string, 'character', startIndex) Returns the first occurrence of a character in a string, searching forward from a start index.
LAST_INDEX_OF	LAST_INDEX_OF(expression) Returns the index of the last occurrence of a character in a string, searching backward from a start index.
LCFIRST	LCFIRST(expression) Converts the first character in a string to lower case.
UCFIRST	UCFIRST(expression) Returns a string with the first character converted to upper case.
UPPER	UPPER(expression) Returns a string converted to upper case.

LOWER	LOWER(expression) Converts all characters in a string to lower case.
REPLACE	REPLACE(string, 'oldChar', 'newChar'); To replace existing characters in a string with new characters.
STRSPLIT	STRSPLIT(string, regex, limit) To split a string around matches of a given regular expression.
SPLITTOBAG	SPLITTOBAG(string, regex, limit) Similar to the STRSPLIT() function, it splits the string by given delimiter and returns the result in a bag.
TRIM	TRIM(expression) Returns a copy of a string with leading and trailing whitespaces removed.
LTRIM	LTRIM(expression) Returns a copy of a string with leading whitespaces removed.
RTRIM	RTRIM(expression) Returns a copy of a string with trailing whitespaces removed.

STARTSWITH ()

This function accepts two string parameters. It verifies whether the first string starts with the second.

Syntax

Given below is the syntax of the **STARTSWITH()** function.

```
STARTSWITH(string, substring)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.txt

```

001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai

```

And, we have loaded this file into Pig with a schema named **emp_data** 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);

```

Example

Following is an example of the **STARTSWITH()** function. In this example, we have verified whether the names of all the employees start with the substring **"Ro"**.

```

grunt> startswith_data = FOREACH emp_data GENERATE (id,name), STARTSWITH
(name,'Ro');

```

The above statement parses the names of all the employees if any of these names starts with the substring **'Ro'**. Since the names of the employees **'Robin'** and **'Robert'** starts with the substring **'Ro'** for these two tuples the **STARTSWITH()** function returns the Boolean value **'true'** and for remaining tuples the value will be **'false'**.

The result of the statement will be stored in the schema named **startswith_data**. Verify the content of the schema **startswith_data**, using the Dump operator as shown below.

```
Dump startswith_data;
```

```

((1,Robin),true)
((2,BOB),false)
((3,Maya),false)
((4,Sara),false)
((5,David),false)
((6,maggy),false)
((7,Robert),true)
((8,Syam),false)
((9,Mary),false)
((10,Saran),false)
((11,Stacy),false)
((12,Kelly),false)

```

ENDSWITH

This function accepts two String parameters, it is used to verify whether the first string ends with the second string.

Syntax

```
ENDSWITH(string1, string2)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name age and city.

emp.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** 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);
```

Following is an example of **ENDSWITH()** function, in this example we are verifying, whether the name of every employee ends with the character **n**.

```
grunt> emp_endswith = FOREACH emp_data GENERATE (id,name),ENDSWITH ( name,
'n' );
```

The above statement verifies whether the name of the employee ends with the letter n. Since the names of the employees **Saran** and **Robin** ends with the letter n for these two tuples **ENDSWITH()** function returns the Boolean value **'true'** and for remaining tuples the value will be **'false'**.

The result of the statement will be stored in the schema named **emp_endswith**. Verify the content of the schema **emp_endswith**, using the Dump operator as shown below.

```
grunt> Dump emp_endswith;
```

```

((1,Robin),true)
((2,BOB),false)
((3,Maya),false)
((4,Sara),false)
((5,David),false)
((6,Maggy),false)
((7,Robert),false)
((8,Syam),false)
((9,Mary),false)
((10,Saran),true)
((11,Stacy),false)
((12,Kelly),false)

```

SUBSTRING

This function returns a substring from the given string.

Syntax

Given below is the syntax of the **SUBSTRING()** function. This function accepts three parameters one is the column name of the string we want. And the other two are the start and stop indexes of the substring we want from the string.

```
SUBSTRING(string, startIndex, stopIndex)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name age and city.

emp.txt

```

001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai

```

And, we have loaded this file into Pig with a schema named **emp_data** 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);

```

Following is an example of the **SUBSTRING()** function. In this example we have verified whether the names of all the employees starts with the substring **"Ro"**.

```
grunt> substring_data = FOREACH emp_data GENERATE (id,name), STARTSWITH
(name,'Ro');
```

The above statement parses the names of all the employees if any of these names starts with the substring **'Ro'**. Since the names of the employees **'Robin'** and **'Robert'** starts with the substring **'Ro'** for these two tuples the **STARTSWITH()** function returns the Boolean value **'true'** and for remaining tuples the value will be **'false'**.

The result of the statement will be stored in the schema named **startswith_data**. Verify the content of the schema **startswith_data**, using the Dump operator as shown below.

```
Dump startswith_data;
```

```
((1,Robin),true)
((2,BOB),false)
((3,Maya),false)
((4,Sara),false)
((5,David),false)
((6,maggy),false)
((7,Robert),true)
((8,Syam),false)
((9,Mary),false)
((10,Saran),false)
((11,Stacy),false)
((12,Kelly),false)
```

EqualsIgnoreCase

The **EqualsIgnoreCase()** function is used to compare two strings and verify whether they are equal. If both are equal this function returns the Boolean value **true** else it returns the value **false**.

Syntax

Given below is the syntax of the function **EqualsIgnoreCase()**

```
EqualsIgnoreCase(string1, string2)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name age and city.

emp.txt

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

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

And, we have loaded this file into Pig with a schema named **emp_data** 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);
```

Given below is an example of the **EqualsIgnoreCase()** function. In this example we are comparing the names of every employees with the string value **'Robin'**.

```
grunt> equals_data = FOREACH emp_data GENERATE (id,name),
EqualsIgnoreCase(name, 'Robin');
```

The above statement compares the string **"Robin"** (case sensitive) with the names of the employees, if the value matches it returns **true** else it returns **false**. In short, this statement searches the employee record whose name is **'Robin'**

The result of the statement will be stored in the schema named **equals_data**. Verify the content of the schema **equals_data**, using the Dump operator as shown below.

```
grunt> Dump equals_data;
```

```
((1,Robin),true)
((2,BOB),false)
((3,Maya),false)
((4,Sara),false)
((5,David),false)
((6,Maggy),false)
((7,Robert),false)
((8,Syam),false)
((9,Mary),false)
((10,Saran),false)
((11,Stacy),false)
((12,Kelly),false)
```

INDEXOF ()

The **INDEXOF()** function accepts a string value, a character and an index (integer). It returns the first occurrence of the given character in the string, searching forward from the given index.

Syntax

Given below is the syntax of the **INDEXOF()** function.

```
INDEXOF(string, 'character', startIndex)
```


Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** 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);
```

Given below is an example of the **INDEXOF()** function. In this example, we are finding the occurrence of the letter "r" in the names of every employee using this function.

```
grunt> indexof_data = FOREACH emp_data GENERATE (id,name), INDEXOF(name, 'r',0);
```

The above statement parses the name of each employee and returns the index value at which the letter 'r' occurred for the first time. If the name doesn't contain the letter 'r' it returns the value **-1**

The result of the statement will be stored in the schema named **indexof_data**. Verify the content of the schema **indexof_data**, using the Dump operator as shown below.

```
grunt> Dump indexof_data;
```

```
((1,Robin),-1)
((2,BOB),-1)
((3,Maya),-1)
((4,Sara),2)
((5,David),-1)
((6,Maggy),-1)
((7,Robert),4)
((8,Syam),-1)
((9,Mary),2)
((10,Saran),2)
```

```
((11,Stacy),-1)
((12,Kelly),-1)
```

LAST_INDEX_OF ()

The **LAST_INDEX_OF()** function accepts a string value and a character. It returns the last occurrence of the given character in the string, searching backward from the end of the string.

Syntax

Given below is the syntax of the **LAST_INDEX_OF()** function.

```
LAST_INDEX_OF(string, 'character')
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** 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);
```

Given below is an example of the **LAST_INDEX_OF()** function. In this example, we are going to find the occurrence of the letter **"g"** from the end, in the names of every employee.

```
grunt> last_index_data = FOREACH emp_data GENERATE (id,name),
LAST_INDEX_OF(name, 'g');
```

The above statement parses the name of each employee from the end and returns the index value at which the letter **'g'** occurred for the first time. If the name doesn't contain the letter **'g'** it returns the value **-1**

The result of the statement will be stored in the schema named **last_index_data**. Verify the content of the schema **last_index_data** using the Dump operator as shown below.

```
grunt> Dump last_index_data;
```

```
((1,Robin),-1)
((2,BOB),-1)
((3,Maya),-1)
((4,Sara),2)
((5,David),-1)
((6,Maggy),-1)
((7,Robert),4)
((8,Syam),-1)
((9,Mary),2)
((10,Saran),2)
((11,Stacy),-1)
((12,Kelly),-1)
```

LCFIRST ()

This function is used to convert the first character of the given string into lowercase.

Syntax

Following is the syntax of the **LCFIRST()** function.

```
LCFIRST(expression)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** 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);
```

Given below is an example of the **LCFIRST()** function. In this example, we have converted all the first letters of the names of the employees to lowercase.

```
grunt> Lcfirst_data = FOREACH emp_data GENERATE (id,name), LCFIRST(name);
```

The result of the statement will be stored in the schema named **Lcfirst_data**. Verify the content of the schema **Lcfirst_data**, using the Dump operator as shown below.

```
Dump Lcfirst_data;

((1,Robin),robin)
((2,BOB),bob)
((3,Maya),maya)
((4,Sara),sara)
((5,David),david)
((6,Maggy),maggy)
((7,Robert),robert)
((8,Syam),syam)
((9,Mary),mary)
((10,Saran),saran)
((11,Stacy),stacy)
((12,Kelly),kelly)
```

UCFIRST ()

This function accepts a string, converts the first letter of it into uppercase, and returns the result.

Syntax

Here is the syntax of the function **UCFIRST()** function.

```
UCFIRST(expression)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.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
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, we have loaded this file into Pig with a schema named **emp_data** 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);

```

Following is an example of the **UCFIRST()** function. In this example, we are trying to convert the first letters of the names of the cities, to which the employees belong to, to uppercase.

```

grunt> ucfirst_data = FOREACH emp_data GENERATE (id,city), UCFIRST();

```

The result of the statement will be stored in the schema named **ucfirst_data**. Verify the content of the schema **ucfirst_data**, using the Dump operator as shown below.

In our example, the first letter of the name of the city "newyork" is in lowercase. After applying UCFIRST() function, it turns into "NEWYORK"

```

Dump ucfirst_data;

((1,newyork),Newyork)
((2,Kolkata),Kolkata)
((3,Tokyo),Tokyo)
((4,London),London)
((5,Bhuwaneshwar),Bhuwaneshwar)
((6,Chennai),Chennai)
((7,newyork),Newyork)
((8,Kolkata),Kolkata)
((9,Tokyo),Tokyo)
((10,London),London)
((11,Bhuwaneshwar),Bhuwaneshwar)
((12,Chennai),Chennai)

```

UPPER ()

This function is used to convert all the characters in a string to uppercase.

Syntax

The syntax of the **UPPER()** function is as follows:

```
UPPER(expression)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/**. This file contains the student details such as id, name, age, and city.

emp.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** 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);
```

Given below is an example of the **UPPER()** function. In this example, we have converted the names of all the employees to upper case.

```
grunt> upper_data = FOREACH emp_data GENERATE (id,name), UPPER(name);
```

The above statement converts the names of all the employees to uppercase and returns the result.

The result of the statement will be stored in a schema named **upper_data**. Verify the content of the schema **upper_data**, using the Dump operator as shown below.

```
Dump upper_data;

((1,Robin),ROBIN)
((2,BOB),BOB)
((3,Maya),MAYA)
((4,Sara),SARA)
((5,David),DAVID)
((6,Maggy),MAGGY)
((7,Robert),ROBERT)
((8,Syam),SYAM)
((9,Mary),MARY)
((10,Saran),SARAN)
((11,Stacy),STACY)
((12,Kelly),KELLY)
```

LOWER ()

This function is used to convert all the characters in a string to lowercase.

Syntax

Following is the syntax of the **LOWER()** function.

```
LOWER(expression)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** 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);
```

Given below is an example of the **LOWER()** function. In this example, we have converted the names of all the employees to lowercase.

```
grunt> lower_data = FOREACH emp_data GENERATE (id,name), LOWER(name);
```

The above statement converts the names of all the employees to uppercase and returns the result.

The result of the statement will be stored in the schema named **lower_data**. Verify the content of the schema **lower_data**, using the Dump operator.

```
Dump upper_data;

((1,Robin),robin)
((2,BOB),bob)
((3,Maya),maya)
((4,Sara),sara)
((5,David),david)
```

```
((6,Maggy),maggy)
((7,Robert),robert)
((8,Syam),syam)
((9,Mary),mary)
((10,Saran),saran)
((11,Stacy),stacy)
((12,Kelly),kelly)
```

REPLACE ()

This function is used to replace all the characters in a given string with the new characters.

Syntax

Given below is the syntax of the **REPLACE()** function. This function accepts three parameters, namely,

- **string**: The string that is to be replaced. If we want to replace the string within a schema, we have to pass the column name the string belongs to.
- **regEXP**: Here we have to pass the string/regular expression we want to replace.
- **newChar**: Here we have to pass the new value of the string.

```
REPLACE(string, 'regExp', 'newChar');
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuaneshwar
012,Kelly,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** 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);
```


Following is an example of the **REPLACE()** function. In this example, we have replaced the name of the city **Bhubaneshwar** with a shorter form **Bhuw**.

```
grunt> replace_data = FOREACH emp_data GENERATE
(id,city),REPLACE(city,'Bhuwaneshwar','Bhuw');
```

The above statement replaces the string '**Bhuwaneshwar**' with '**Bhuw**' in the column named **city** in the **emp** schema and returns the result. This result is stored in the schema named **replace_data**. Verify the content of the schema **replace_data** using the Dump operator as shown below.

```
Dump replace_data;
((1,newyork),newyork)
((2,Kolkata),Kolkata)
((3,Tokyo),Tokyo)
((4,London),London)
((5,Bhuwaneshwar),Bhuw)
((6,Chennai),Chennai)
((7,newyork),newyork)
((8,Kolkata),Kolkata)
((9,Tokyo),Tokyo)
((10,London),London)
((11,Bhuwaneshwar),Bhuw)
((12,Chennai),Chennai)
```

STRSPLIT ()

This function is used to split a given string by a given delimiter.

Syntax

The syntax of **STRSPLIT()** is given below. This function accepts a string that is needed to be split, a regular expression, and an integer value specifying the limit (the number of substrings the string should be split). This function parses the string and when it encounters the given regular expression, it splits the string into **n** number of substrings where **n** will be the value passed to **limit**.

```
STRSPLIT(string, regex, limit)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.txt

```
001,Robin_Smith,22,newyork
002,BOB_Wilson,23,Kolkata
003,Maya_Reddy,23,Tokyo
004,Sara_Jain,25,London
005,David_Miller,23,Bhuwaneshwar
006,Maggy_Moore,22,Chennai
```

```
007,Robert_Scott,22,newyork
008,Syam_Ketavarapu,23,Kolkata
009,Mary_Carter,25,Tokyo
010,Saran_Naidu,25,London
011,Stacy_Green,25,Bhuvaneshwar
012,Kelly_Moore,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** as shown below.

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

Following is an example of the **STRSPLIT()** function. If you observe the emp.txt file, you can find that, in the **name** column, we have the names and surnames of the employees separated by the delimiter “_”.

In this example, we are trying to split the name and surname of the employees using **STRSPLIT()** function.

```
grunt> strsplit_data = FOREACH emp_data GENERATE (id,name), STRSPLIT
(name, '_',2);
```

The result of the statement will be stored in the schema named **strsplit_data**. Verify the content of the schema **strsplit_data**, using the Dump operator as shown below.

```
grunt> Dump strsplit_data;

((1,Robin_Smith),(Robin,Smith))
((2,BOB_Wilson),(BOB,Wilson))
((3,Maya_Reddy),(Maya,Reddy))
((4,Sara_Jain),(Sara,Jain))
((5,David_Miller),(David,Miller))
((6,Maggy_Moore),(Maggy,Moore))
((7,Robert_Scott),(Robert,Scott))
((8,Syam_Ketavarapu),(Syam,Ketavarapu))
((9,Mary_Carter),(Mary,Carter))
((10,Saran_Naidu),(Saran,Naidu))
((11,Stacy_Green),(Stacy,Green))
((12,Kelly_Moore),(Kelly,Moore))
```

STRSPLITTOBAG ()

This function is similar to the **STRSPLIT()** function. It splits the string by a given delimiter and returns the result in a bag.

Syntax

The syntax of **SPLITTOBAG()** is given below. This function accepts a string that is needed to be split, a regular expression, and an integer value specifying the limit (the number of substrings the string should be split). This function parses the string and when it encounters the given regular expression, it splits the string into **n** number of substrings where **n** will be the value passed to **limit**.

```
STRSPLIT(string, regex, limit)
```

Example

Assume that there is a file named **emp.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the student details such as id, name, age, and city.

emp.txt

```
001,Robin_Smith,22,newyork
002,BOB_Wilson,23,Kolkata
003,Maya_Reddy,23,Tokyo
004,Sara_Jain,25,London
005,David_Miller,23,Bhuwaneshwar
006,Maggy_Moore,22,Chennai
007,Robert_Scott,22,newyork
008,Syam_Ketavarapu,23,Kolkata
009,Mary_Carter,25,Tokyo
010,Saran_Naidu,25,London
011,Stacy_Green,25,Bhuwaneshwar
012,Kelly_Moore,22,Chennai
```

And, we have loaded this file into Pig with a schema named **emp_data** as shown below.

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

Following is an example of the **STRSPLITTOBAG()** function. If you observe the emp.txt file, you can find that, in the **name** column, we have name and surname of the employees separated by the delimiter “_”.

In this example we are trying to split the name and surname of the employee, and get the result in a bag using **STRSPLITTOBAG()** function.

```
grunt> strsplittostring_data = FOREACH emp_data GENERATE (id,name), STRSPLIT
(name, '_',2);
```

The result of the statement will be stored in the schema named **strsplittostring_data**. Verify the content of the schema **strsplittostring_data**, using the Dump operator as shown below.

```
grunt> Dump strsplittostring_data;

((1,Robin_Smith),{(Robin),(Smith)})
((2,BOB_Wilson),{(BOB),(Wilson)})
((3,Maya_Reddy),{(Maya),(Reddy)})
((4,Sara_Jain),{(Sara),(Jain)})
((5,David_Miller),{(David),(Miller)})
((6,Maggy_Moore),{(Maggy),(Moore)})
((7,Robert_Scott),{(Robert),(Scott)})
((8,Syam_Ketavarapu),{(Syam),(Ketavarapu)})
((9,Mary_Carter),{(Mary),(Carter)})
```

```
((10,Saran_Naidu),{(Saran),(Naidu)})
((11,Stacy_Green),{(Stacy),(Green)})
((12,Kelly_Moore),{(Kelly),(Moore)})
```

Trim ()

The TRIM function accepts a string and returns its copy after removing the unwanted spaces before and after it.

Syntax

Here is the syntax of the **TRIM()** function.

```
TRIM(expression)
```

Example

Assume we have some unwanted spaces before and after the names of the employees in the records of the **emp_data** schema.

```
Dump emp_data;

(1, Robin ,22,newyork)
(2,BOB,23,Kolkata)
(3, Maya ,23,Tokyo)
(4,Sara,25,London)
(5, David ,23,Bhuwaneshwar)
(6,maggy,22,Chennai)
(7,Robert,22,newyork)
(8, Syam ,23,Kolkata)
(9,Mary,25,Tokyo)
(10, Saran ,25,London)
(11, Stacy,25,Bhuwaneshwar)
(12, Kelly ,22,Chennai)
```

Using the **TRIM()** function, we can remove these heading and tailing spaces from the names, as shown below.

```
grunt> trim_data = FOREACH emp_data GENERATE (id,name),
TRIM(name);
```

The above statement returns the copy of the names by removing the heading and tailing spaces from the names of the employees. The result is stored in the schema named **trim_data**. Verify the result of the schema **trim_data** using the Dump operator as shown below.

```
grunt> Dump trim_data;

((1, Robin ),Robin)
((2,BOB),BOB)
```

```
(3, Maya ),Maya)
(4,Sara),Sara)
(5, David ),David)
(6,maggy),maggy)
(7,Robert),Robert)
(8, Syam ),Syam)
(9,Mary),Mary)
((10, Saran ),Saran)
((11, Stacy),Stacy)
((12, Kelly ),Kelly)
```

LTRIM ()

The function **LTRIM()** is same as the function **TRIM()**. It removes the unwanted spaces from the left side of the given string (heading spaces).

Syntax

Here is the syntax of the **LTRIM()** function.

```
LTRIM(expression)
```

Example

Assume we have some unwanted spaces before and after the names of the employees in the records of the **emp_data** schema.

```
Dump emp_data;

(1, Robin ,22,newyork)
(2, BOB,23,Kolkata)
(3, Maya ,23,Tokyo)
(4, Sara,25,London)
(5, David ,23,Bhuwaneswar)
(6, maggy,22,Chennai)
(7, Robert,22,newyork)
(8, Syam ,23,Kolkata)
(9, Mary,25,Tokyo)
(10, Saran ,25,London)
(11, Stacy,25,Bhuwaneswar)
(12, Kelly ,22,Chennai)
```

Using the **LTRIM()** function, we can remove the heading spaces from the names as shown below.

```
grunt> ltrim_data = FOREACH emp_data GENERATE (id,name),
LTRIM(name);
```

The above statement returns the copy of the names by removing the heading spaces from the names of the employees. The result is stored in the schema named **ltrim_data**. Verify the result of the schema **ltrim_data** using the Dump operator as shown below.

```
grunt> Dump ltrim_data;

((1, Robin ),Robin )
((2,BOB),BOB)
((3, Maya ),Maya )
((4,Sara),Sara)
((5, David ),David)
((6,maggy),maggy)
((7,Robert),Robert)
((8, Syam ),Syam)
((9,Mary),Mary)
((10, Saran),Saran)
((11, Stacy),Stacy)
((12, Kelly ),Kelly)
```

RTRIM

The function **RTRIM()** is same as the function **TRIM()**. It removes the unwanted spaces from the right side of a given string (tailing spaces).

Syntax

The syntax of the **RTRIM()** function is as follows –

```
RTRIM(expression)
```

Example

Assume we have some unwanted spaces before and after the names of the employees in the records of the **emp_data** schema as shown below.

```
Dump emp_data;

(1, Robin ,22,newyork)
(2, BOB,23,Kolkata)
(3, Maya ,23,Tokyo)
(4, Sara,25,London)
(5, David ,23,Bhuwaneshwar)
(6, maggy,22,Chennai)
(7, Robert,22,newyork)
(8, Syam ,23,Kolkata)
(9, Mary,25,Tokyo)
(10, Saran ,25,London)
(11, Stacy,25,Bhuwaneshwar)
(12, Kelly ,22,Chennai)
```

Using the **RTRIM()** function, we can remove the heading spaces from the names as shown below.

```
grunt> rtrim_data = FOREACH emp_data GENERATE (id,name),  
RTRIM(name);
```

The above statement returns the copy of the names by removing the **tailing** spaces from the names of the employees. The result is stored in the schema named **rtrim_data**. Verify the result of the schema **rtrim_data** using the Dump operator as shown below.

```
grunt> Dump rtrim_data;  
  
((1, Robin ), Robin)  
((2,BOB),BOB)  
((3, Maya ), Maya)  
((4,Sara),Sara)  
((5, David ), David)  
((6,maggy),maggy)  
((7,Robert),Robert)  
((8, Syam ), Syam)  
((9,Mary),Mary)  
((10, Saran ), Saran)  
((11, Stacy), Stacy)  
((12, Kelly ), Kelly)
```

28. date-time Functions

Apache Pig provides the following Date and Time functions –

Operator	Description
ToDate	ToDate(milliseconds), ToDate(iosstring), ToDate(userstring, format), ToDate(userstring, format, timezone) This function returns a date-time object according to the given parameters.
CurrentTime	CurrentTime() returns the date-time object of the current time.
GetDay	GetDay(datetime) Returns the day of a month from the date-time object.
GetHour	GetHour(datetime) Returns the hour of a day from the date-time object.
GetMilliSecond	GetMilliSecond(datetime) Returns the millisecond of a second from the date-time object.
GetMinute	GetMinute(datetime) Returns the minute of an hour from the date-time object.
GetMonth	GetMonth(datetime) Returns the month of a year from the date-time object.
GetSecond	GetSecond(datetime) Returns the second of a minute from the date-time object.
GetWeek	GetWeek(datetime) Returns the week of a year from the date-time object.

GetWeekYear	GetWeekYear(datetime) Returns the week year from the date-time object.
GetYear	GetYear(datetime) Returns the year from the date-time object.
ToString	ToString(datetime [, format string]) Converts the date-time object to the ISO or the customized string.
AddDuration	AddDuration(datetime, duration) Returns the result of a date-time object along with the duration object.
SubtractDuration	SubtractDuration(datetime, duration) Subtracts the Duration object from the Date-Time object and returns the result.
DaysBetween	DaysBetween(datetime1, datetime2) Returns the number of days between the two date-time objects.
HoursBetween	HoursBetween(datetime1, datetime2) Returns the number of hours between two date-time objects.
MillisecondsBetween	MillisecondsBetween(datetime1, datetime2) Returns the number of milliseconds between two date-time objects.
MinutesBetween	MinutesBetween(datetime1, datetime2) Returns the number of minutes between two date-time objects.
MonthsBetween	MonthsBetween(datetime1, datetime2) Returns the number of months between two date-time objects.
SecondsBetween	SecondsBetween(datetime1, datetime2) Returns the number of seconds between two date-time objects.
ToMilliseconds	ToMilliseconds(datetime)

	Calculates the number of milliseconds elapsed since January 1, 1970, 00:00:00.000 and returns the result.
WeeksBetween	WeeksBetween(datetime1, datetime2) Returns the number of weeks between two date-time objects.
YearsBetween	YearsBetween(datetime1, datetime2) Returns the number of years between two date-time objects.

ToDate ()

This function is used to generate a **DateTime** object according to the given parameters.

Syntax

The syntax of **ToDate()** function can be any of the following –

```
ToDate(milliseconds)

ToDate(iosstring)

ToDate(userstring, format)

ToDate(userstring, format, timezone)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth details of a particular person, id, date, and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **ToDate()** function. Here we are converting the DateTime object corresponding to the date-of-birth of every employee.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime >);
```

The result (**DateTime** object of every employee) of the statement will be stored in the schema named **todate_data**. Verify the content of this schema using the Dump operator as shown below.

```
Dump todate_data;

(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

GetDay()

This function accepts a date-time object as a parameter and returns the current day of the given date-time object.

Syntax

Here is the syntax of the **GetDay()** function.

```
GetDay(datetime)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, id, date, and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetDay()** function. The GetDay() function will retrieve the day from the given Date-Time object. Therefore, first of all, let us generate the date-time objects of all employees using **todate()** function as shown below.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );
```

```
Dump todate_data;

(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Now, let us get the day from the date-of-birth using **GetDay()** function and store it in the schema named **getday_data**.

```
getday_data = foreach todate_data generate(date_time), GetDay(date_time);
```

Verify the contents of the **getday_data** schema using the Dump operator.

```
Dump getday_data;

(1989-09-26T09:00:00.000+05:30,26)
(1980-06-20T10:22:00.000+05:30,20)
(1990-12-19T03:11:44.000+05:30,19)
```

GetHour()

This function accepts a date-time object as parameter and returns the current hour of the current day of a given date-time object.

Syntax

Here is the syntax of the **GetHour()** function.

```
GetHour(datetime)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, id, date, and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetHour()** function. The GetHour() function will retrieve the hour of the day from the given Date-Time object. Therefore, first of all, let's generate the Date-Time objects of all employees using **todate()** function.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime);

Dump todate_data;

(1989-09-26T09:00:00.000+05:30)
```

```
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Let us now get the hour from the birth time of each employee using **GetDay()** function and store it in the schema named **gethour_data**.

```
gethour_data = foreach todate_data generate (date_time), GetHour(date_time);
```

Now verify the contents of the **getday_data** schema using the Dump operator as shown below.

```
Dump gethour_data;

(1989-09-26T09:00:00.000+05:30,9)
(1980-06-20T10:22:00.000+05:30,10)
(1990-12-19T03:11:44.000+05:30,3)
```

GetMinute ()

This function accepts a date-time object as parameter and returns the minute of the current hour of a given date-time object.

Syntax

Here is the syntax of the **GetMinute()** function.

```
GetMinute(datetime)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, id, date, and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetMinute()** function. The GetMinute() function will retrieve the minute of the hour from the given date-time object. Therefore, first of all, let's generate the date-time objects of all employees using **todate()** function.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );
```

```
Dump todate_data;

(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Now, let's get the minutes from the birth time of each employee using **GetMinute()** and store it in the schema named **getminute_data** as shown below.

```
getminute_data = foreach todate_data generate (date_time),
GetMinute(date_time);
```

Now verify the contents of the **getminute_data** schema using the Dump operator as shown below.

```
Dump getminute_data;

(1989-09-26T09:00:00.000+05:30,0)
(1980-06-20T10:22:00.000+05:30,22)
(1990-12-19T03:11:44.000+05:30,11)
```

GetSecond ()

This function accepts a date-time object as a parameter and returns the seconds of the current minute of a given date-time object.

Syntax

Here is the syntax of the **GetSecond()** function.

```
GetSecond(datetime)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, id, date, and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetSecond()** function. It retrieves the seconds of a minute from the given date-time object. Therefore, let's generate the date-time objects of all employees using **todate()** function as shown below.

```
grunt todate_data = foreach raw_date generate ToDate(date, 'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );
```

```
Dump todate_data;
```

```
(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Let us now get the seconds from the birth time of each employee using **GetSecond()** function and store it in the schema named **getsecond_data** as shown below.

```
getsecond_data = foreach todate_data generate (date_time),
GetSecond(date_time);
```

Now verify the contents of the **getsecond_data** schema using the Dump operator as shown below.

```
Dump getsecond_data;
```

```
(1989-09-26T09:00:00.000+05:30,0)
(1980-06-20T10:22:00.000+05:30,0)
(1990-12-19T03:11:44.000+05:30,44)
```

GetMilliSecond ()

This function accepts a date-time object as a parameter and returns the milliseconds of the current second of a given date-time object.

Syntax

Here is the syntax of the **GetMilliSecond()** function.

```
GetMilliSecond(datetime)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, it has person id, date and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetMilliSecond()** function. The GetMilliSecond() function will retrieve the milliseconds of the current second from the given date-time object. Therefore, First of all let's generate the date-time objects of all employees using **todate()** function as shown below.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );
```

```
Dump todate_data;
```

```
(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Now, let's get the seconds from the birth time of each employee using **GetMilliSecond()** function and store it in the schema named **getmillisecond_data** as shown below.

```
getmillisecond_data = foreach todate_data generate (date_time),
GetMilliSecond(date_time);
```

Now verify the contents of the **getmillisecond_data** schema using Dump operator as shown below.

```
Dump getmillisecond_data;
```

```
(1989-09-26T09:00:00.000+05:30,0)
(1980-06-20T10:22:00.000+05:30,0)
(1990-12-19T03:11:44.000+05:30,0)
```

GetYear

This function accepts a date-time object as parameter and returns the current year from the given date-time object.

Syntax

Here is the syntax of the **GetYear()** function.

```
GetYear(datetime)
```


Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, it has person id, date and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetYear()** function. It will retrieve the current year from the given date-time object. Therefore, First of all let's generate the date-time objects of all employees using **todate()** function as shown below.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );
```

```
Dump todate_data;
```

```
(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Let us now get the year from the date-of-birth of each employee using the **GetYear()** function and store it in the schema named **getyear_data**.

```
getyear_data = foreach todate_data generate (date_time), GetYear(date_time);
```

Now verify the contents of the **getyear_data** schema using Dump operator as shown below.

```
Dump getyear_data;
```

```
(1989-09-26T09:00:00.000+05:30,1989)
(1980-06-20T10:22:00.000+05:30,1980)
(1990-12-19T03:11:44.000+05:30,1990)
```

GetMonth ()

This function accepts a date-time object as a parameter and returns the current month of the current year from the given date-time object.

Syntax

Here is the syntax of the **GetMonth()** function.

```
GetMonth(datetime)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, id, date and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date**.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetMonth()** function. It will retrieve the current month from the given date-time object. Therefore, First of all let's generate the date-time objects of all employees using **todate()** function as shown below.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );
```

```
Dump todate_data;
```

```
(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Let us now get the month from the date-of-birth of each employee using **GetMonth()** function and store it in the schema named **getmonth_data**.

```
getmonth_data = foreach todate_data generate (date_time), GetMonth(date_time);
```

Now verify the contents of the **getmonth_data** schema using Dump operator as shown below.

```
Dump getmonth_data;
```

```
(1989-09-26T09:00:00.000+05:30,9)
(1980-06-20T10:22:00.000+05:30,6)
(1990-12-19T03:11:44.000+05:30,12)
```

GetWeek ()

This function accepts a date-time object as parameter and returns the current week of the current month from the given date-time object.

Syntax

Here is the syntax of the **GetWeek()** function.

```
GetWeek(datetime)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, it has person id, date and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetWeek()** function. It will retrieve the current week from the given date-time object. Therefore, let us generate the date-time objects of all employees using **todate()** function as shown below.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );
```

```
Dump todate_data;
```

```
(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Let us now get the month from the date-of-birth of each employee using **GetWeek()** and store it in the schema named **getweek_data**.

```
getweek_data = foreach todate_data generate (date_time), GetWeek(date_time);
```

Now, verify the contents of the **getweek_data** schema using the Dump operator.

```
Dump getWeek_data;

(1989-09-26T09:00:00.000+05:30,39)
(1980-06-20T10:22:00.000+05:30,25)
(1990-12-19T03:11:44.000+05:30,51)
```

GetWeekYear ()

This function accepts a date-time object as a parameter and returns the current week year from the given date-time object.

Syntax

Here is the syntax of the **GetWeekYear()** function.

```
GetWeekYear(datetime)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/** as shown below. This file contains the date-of-birth details of a particular person, id, date, and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **GetWeekYear()** function. It will retrieve the current week year from the given date-time object. Therefore, let us generate the date-time objects of all employees using **todate()** function as shown below.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );

Dump todate_data;

(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Let us get the month from the date-of-birth of each employee using **GetWeekYear()** function and store it in the schema named **getweekyear_data** as shown below.

```
getweekyear_data = foreach todate_data generate (date_time),
GetWeekYear(date_time);
```

Now verify the contents of the **getweekyear_data** schema using the Dump operator.

```
Dump getweekyear_data;

(1989-09-26T09:00:00.000+05:30,1989)
(1980-06-20T10:22:00.000+05:30,1980)
(1990-12-19T03:11:44.000+05:30,1990)
```

CurrentTime ()

This function is used to generate **DateTime** object of the **current time**.

Syntax

Here is the syntax of **CurrentTime()** function.

```
CurrentTime()
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth details of a particular person, id, date, and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **CurrentTime()** function. Here we are generating the current time.

```
grunt> currenttime_data = foreach todate_data generate CurrentTime();
```

The result of the statement will be stored in the schema named **currenttime_data**. Verify the content of this schema using the Dump operator.

```
Dump currenttime_data;

(2015-11-06T11:31:02.013+05:30)
```

```
(2015-11-06T11:31:02.013+05:30)
(2015-11-06T11:31:02.013+05:30)
```

ToString ()

This method is used to convert the date-time object to a customized string.

Syntax

Here is the syntax of the **ToString()** function.

```
ToString(datetime [, format string])
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth details of a particular person, id, date, and time.

date.txt

```
001,1989/09/26 09:00:00
002,1980/06/20 10:22:00
003,1990/12/19 03:11:44
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
grunt > raw_date = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int,date:chararray);
```

Following is an example of the **ToString()** function. The **ToString()** function converts the given date-time objects in to String format. Therefore, let us generate the date-time objects of all employees using **todate()** function.

```
grunt todate_data = foreach raw_date generate ToDate(date,'yyyy/MM/dd
HH:mm:ss') as (date_time:DateTime );
```

```
Dump todate_data;
```

```
(1989-09-26T09:00:00.000+05:30)
(1980-06-20T10:22:00.000+05:30)
(1990-12-19T03:11:44.000+05:30)
```

Let us get the string format of the date-time objects of all the employees using **ToString()** method and store it in a schema named **tostring_data**.

```
tostring_data = foreach todate_data generate (date_time),
ToString(date_time,Text);
```

Verify the **tostring_data** schema using the Dump command as shown below.

```
Dump tostring_data;

(1989-09-26T09:00:00.000+05:30,39)
(1980-06-20T10:22:00.000+05:30,25)
(1990-12-19T03:11:44.000+05:30,51)
```

DaysBetween ()

This function accepts two date-time objects and calculates the number of days between the two given date-time objects.

Syntax

Here is the syntax of the **DaysBetween()** function.

```
DaysBetween(datetime1, datetime2)
```

Example

Assume that there is a file named **doj_dob.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth and date-of-joining details of a particular person, id, date-of-birth, and date-of-join.

doj_dob.txt

```
001,26/09/1989 09:00:00,16/01/2015 09:00:00
002,20/06/1980 10:22:00,10/08/2011 09:00:00
003,19/12/1990 03:11:44,25/10/2012 09:00:00
```

And, we have loaded this file into Pig with a schema named **doj_dob** as shown below.

```
doj_dob = LOAD 'hdfs://localhost:9000/pig_data/date1.txt' USING
PigStorage(',')as (id:int, dob:chararray, doj:chararray);
```

Let us now calculate the number of days between date-of-birth and date-of-join of the employees using the **DaysBetween()** function.

```
daysbetween_data = foreach doj_dob generate DaysBetween(ToDate(doj,'dd/MM/yyyy
HH:mm:ss'),ToDate(dob,'dd/MM/yyyy HH:mm:ss'));
```

The above statement stores the result in the schema named **daysbetween_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump daysbetween_data;

(9243)
(11372)
(7981)
```

HoursBetween ()

This function accepts two date-time objects and calculates the number of hours between the two given date-time objects.

Syntax

Here is the syntax of the **HoursBetween()** function.

```
HoursBetween(datetime1, datetime2)
```

Example

Assume that there is a file named **doj_dob.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth and date-of-joining details of a particular person, id, date-of-birth, and date-of-joining.

doj_dob.txt

```
001,26/09/1989 09:00:00,16/01/2015 09:00:00
002,20/06/1980 10:22:00,10/08/2011 09:00:00
003,19/12/1990 03:11:44,25/10/2012 09:00:00
```

And, we have loaded this file into Pig with a schema named **doj_dob** as shown below.

```
doj_dob = LOAD 'hdfs://localhost:9000/pig_data/date1.txt' USING
PigStorage(',')as (id:int, dob:chararray, doj:chararray);
```

Let us now calculate the number of hours between date-of-birth and date-of-joining of the employees using the **HoursBetween()** function as shown below.

```
hoursbetween_data = foreach doj_dob generate
HoursBetween(ToDate(doj, 'dd/MM/yyyy HH:mm:ss'),ToDate(dob, 'dd/MM/yyyy
HH:mm:ss'));
```

The above statement stores the result in the schema named **hoursbetween_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump HoursBetween;

(221832)
(272950)
(191549)
```

MinutesBetween ()

This function accepts two date-time objects and calculates the number of minutes between the two given date-time objects.

Syntax

Here is the syntax of the **MinutesBetween()** function.

```
MinutesBetween(datetime1, datetime2)
```

Example

Assume that there is a file named **doj_dob.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth and date-of-joining details of a particular person, id, date-of-birth, and date-of-joining.

doj_dob.txt

```
001,26/09/1989 09:00:00,16/01/2015 09:00:00
002,20/06/1980 10:22:00,10/08/2011 09:00:00
003,19/12/1990 03:11:44,25/10/2012 09:00:00
```

And, we have loaded this file into Pig with a schema named **doj_dob** as shown below.

```
doj_dob = LOAD 'hdfs://localhost:9000/pig_data/date1.txt' USING
PigStorage(',')as (id:int, dob:chararray, doj:chararray);
```

Now, let's calculate the number of minutes between date-of-birth and date-of-joining of the employees using the **MinutesBetween()** function as shown below.

```
minutesbetween_data = foreach doj_dob generate
MinutesBetween(ToDate(doj, 'dd/MM/yyyy HH:mm:ss'), ToDate(dob, 'dd/MM/yyyy
HH:mm:ss'));
```

The above statement stores the result in the schema named **minutesbetween_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump minutesbetween_data;

(13309920)
(16377038)
(11492988)
```

SecondsBetween ()

This function accepts two date-time objects and calculates the number of seconds between the two given date-time objects.

Syntax

Here is the syntax of the **SecondsBetween()** function.

```
SecondsBetween(datetime1, datetime2)
```

Example

Assume that there is a file named **doj_dob.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth and date-of-joining details of a particular person, id, date-of-birth and date-of-joining.

doj_dob.txt

```
001,26/09/1989 09:00:00,16/01/2015 09:00:00
002,20/06/1980 10:22:00,10/08/2011 09:00:00
003,19/12/1990 03:11:44,25/10/2012 09:00:00
```

And, we have loaded this file into Pig with a schema named **doj_dob** as shown below.

```
doj_dob = LOAD 'hdfs://localhost:9000/pig_data/date1.txt' USING
PigStorage(',')as (id:int, dob:chararray, doj:chararray);
```

Let us now calculate the number of seconds between date-of-birth and date-of-joining of the employees using the **SecondsBetween()** function as shown below.

```
secondsbetween_data = foreach doj_dob generate
SecondsBetween(ToDate(doj, 'dd/MM/yyyy HH:mm:ss'), ToDate(dob, 'dd/MM/yyyy
HH:mm:ss'));
```

The above statement stores the result in the schema named **secondsbetween_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump secondsbetween_data;

(798595200)
(982622280)
(689579296)
```

MilliSecondsBetween ()

This function accepts two date-time objects and calculates the number of milliseconds between the two given date-time objects.

Syntax

Here is the syntax of the **MilliSecondsBetween()** function.

```
MilliSecondsBetween(datetime1, datetime2)
```

Example

Assume that there is a file named **doj_dob.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth and date-of-joining details of a particular person, id, date-of-birth and date-of-joining.

doj_dob.txt

```
001,26/09/1989 09:00:00,16/01/2015 09:00:00
002,20/06/1980 10:22:00,10/08/2011 09:00:00
003,19/12/1990 03:11:44,25/10/2012 09:00:00
```

And, we have loaded this file into Pig with a schema named **doj_dob** as shown below.

```
doj_dob = LOAD 'hdfs://localhost:9000/pig_data/date1.txt' USING
PigStorage(',')as (id:int, dob:chararray, doj:chararray);
```

Let us now calculate the number of milli seconds between date-of-birth and date-of-joining of the employees using the **MilliSecondsBetween()** function as shown below.

```
millisecondsbetween_data = foreach doj_dob generate
MilliSecondsBetween(ToDate(doj, 'dd/MM/yyyy HH:mm:ss'), ToDate(dob, 'dd/MM/yyyy
HH:mm:ss'));
```

The above statement stores the result in the schema named **millisecondsbetween_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump millisecondsbetween_data;

(798595200000)
(982622280000)
(689579296000)
```

YearsBetween ()

This function accepts two date-time objects and calculates the number of years between the two given date-time objects.

Syntax

Here is the syntax of the **YearsBetween()** function.

```
YearsBetween(datetime1, datetime2)
```

Example

Assume that there is a file named **doj_dob.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth and date-of-joining details of a particular person, id, date-of-birth and date-of-joining.

doj_dob.txt

```
001,26/09/1989 09:00:00,16/01/2015 09:00:00
002,20/06/1980 10:22:00,10/08/2011 09:00:00
003,19/12/1990 03:11:44,25/10/2012 09:00:00
```

And, we have loaded this file into Pig with a schema named **doj_dob** as shown below.

```
doj_dob = LOAD 'hdfs://localhost:9000/pig_data/date1.txt' USING
PigStorage(',')as (id:int, dob:chararray, doj:chararray);
```

Let us now calculate the number of years between date-of-birth and date-of-joining of the employees using the **YearsBetween()** function as shown below.

```
yearsbetween_data = foreach doj_dob generate
YearsBetween(ToDate(doj, 'dd/MM/yyyy HH:mm:ss'),ToDate(dob, 'dd/MM/yyyy
HH:mm:ss'));
```

The above statement stores the result in the schema named **yearsbetween_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump yearsbetween_data;

(25)
(31)
(21)
```

MonthsBetween ()

This function accepts two date-time objects and calculates the number of months between the two given date-time objects.

Syntax

Here is the syntax of the **MonthsBetween()** function.

```
MonthsBetween(datetime1, datetime2)
```

Example

Assume that there is a file named **doj_dob.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth and date-of-joining details of a particular person, id, date-of-birth and date-of-joining.

doj_dob.txt

```
001,26/09/1989 09:00:00,16/01/2015 09:00:00
002,20/06/1980 10:22:00,10/08/2011 09:00:00
003,19/12/1990 03:11:44,25/10/2012 09:00:00
```

And, we have loaded this file into Pig with a schema named **doj_dob** as shown below.

```
doj_dob = LOAD 'hdfs://localhost:9000/pig_data/date1.txt' USING
PigStorage(',')as (id:int, dob:chararray, doj:chararray);
```

Let us now calculate the number of minutes between date-of-birth and date-of-joining of the employees using the **MonthsBetween()** function as shown below.

```
monthsbetween_data = foreach doj_dob generate
MinutesBetween(ToDate(doj, 'dd/MM/yyyy HH:mm:ss'), ToDate(dob, 'dd/MM/yyyy
HH:mm:ss'));
```

The above statement stores the result in the schema named **monthsbetween_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump monthsbetween;
```

```
(13309920)
(16377038)
(11492988)
```

WeeksBetween ()

This function accepts two date-time objects and calculates the number of weeks between the two given date-time objects.

Syntax

Here is the syntax of the **WeeksBetween()** function.

```
WeeksBetween(datetime1, datetime2)
```

Example

Assume that there is a file named **doj_dob.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth and date-of-joining details of a particular person, id, date-of-birth and date-of-joining.

doj_dob.txt

```
001,26/09/1989 09:00:00,16/01/2015 09:00:00
002,20/06/1980 10:22:00,10/08/2011 09:00:00
003,19/12/1990 03:11:44,25/10/2012 09:00:00
```

And, we have loaded this file into Pig with a schema named **doj_dob** as shown below.

```
doj_dob = LOAD 'hdfs://localhost:9000/pig_data/date1.txt' USING
PigStorage(',')as (id:int, dob:chararray, doj:chararray);
```

Let us now calculate the number of weeks between date-of-birth and date-of-joining of the employees using the **WeeksBetween()** function as shown below.

```
weeksbetween_data = foreach doj_dob generate
WeeksBetween(ToDate(doj, 'dd/MM/yyyy HH:mm:ss'),ToDate(dob, 'dd/MM/yyyy
HH:mm:ss'));
```

The above statement stores the result in the schema named **weeksbetween_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump weeksbetween_data;

(1320)
(1624)
(1140)
```

AddDuration ()

This function accepts a date-time object and a duration objects, and adds the given duration to the date-time object and returns a new date-time object with added duration.

Syntax

Here is the syntax of the **AddDuration()** function.

```
AddDuration(datetime, duration)
```

Note: The Duration is represented in ISO 8601 standard. According to ISO 8601 standard P is placed at the beginning, while representing the duration and it is called as duration designator. Likewise,

- **Y** is the year designator. We use this after declaring the year.
Example : P1Y represents 1 year.
- **M** is the month designator. We use this after declaring the month.
Example : P1M represents 1 month.
- **W** is the week designator. We use this after declaring the week.
Example : P1W represents 1 week.
- **D** is the day designator. We use this after declaring the day.
Example : P1D represents 1 day.
- **T** is the time designator. We use this before declaring the time.
Example : PT5H represents 5 hours.
- **H** is the hour designator. We use this after declaring the hour.
Example : PT1H represents 1 hour.
- **M** is the minute designator. We use this after declaring the minute.
Example : PT1M represents 1 minute.

- **S** is the second designator. We use this after declaring the second.
Example : PT1S represents 1 second.

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth details of a particular person, id, date and time and some duration according to ISO 8601 standard.

date.txt

```
001,1989/09/26 09:00:00,PT1M
002,1980/06/20 10:22:00,P1Y
003,1990/12/19 03:11:44,P3M
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
date_duration = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int, date:chararray, duration:chararray)
```

Following is an example of the **AddDuration()** function. You can add certain Duration to the given date-time object using this method as shown below.

```
Add_duration_data = foreach date_duration generate(date,duration),
AddDuration(ToDate(date,'yyyy/MM/dd HH:mm:ss'), duration);
```

The result of the statement will be stored in the schema named **add_duration_data**. Verify the content of this schema using the Dump operator as shown below.

```
Dump add_duration_data;

((1989/09/26 09:00:00,PT1M),1989-09-26T09:01:00.000+05:30)
((1980/06/20 10:22:00,P1Y),1981-06-20T10:22:00.000+05:30)
((1990/12/19 03:11:44,P3M),1991-03-19T03:11:44.000+05:30)
```

SubtractDuration ()

This function accepts a date-time object and a duration objects, and subtract the given duration to the date-time object and returns a new date-time object.

Syntax

Here is the syntax of the **SubtractDuration()** function.

```
SubtractDuration(datetime, duration)
```

Example

Assume that there is a file named **date.txt** in the **HDFS** directory **/pig_data/**. This file contains the date-of-birth details of a particular person, it has person id, date and time and some duration according to ISO 8601 standard.

date.txt

```
001,1989/09/26 09:00:00,PT1M
002,1980/06/20 10:22:00,P1Y
003,1990/12/19 03:11:44,P3M
```

And, we have loaded this file into Pig with a schema named **raw_date** as shown below.

```
date_duration = LOAD 'hdfs://localhost:9000/pig_data/date.txt' USING
PigStorage(',')as (id:int, date:chararray, duration:chararray)
```

Following is an example of the **SubtractDuration()** function. You can subtract certain duration from the given date-time object using this method as shown below.

```
subtractduration_data = foreach date_duration generate(date,duration),
SubtractDuration(ToDate(date,'yyyy/MM/dd HH:mm:ss'), duration);
```

The result of the statement will be stored in the schema named **subtractduration_data**. Verify the content of this schema using the Dump operator as shown below.

```
Dump subtractduration_data;

((1989/09/26 09:00:00,PT1M),1989-09-26T08:59:00.000+05:30)
((1980/06/20 10:22:00,P1Y),1979-06-20T10:22:00.000+05:30)
((1990/12/19 03:11:44,P3M),1990-09-19T03:11:44.000+05:30)
```


29. Math Functions

We have the following Math functions in Apache Pig –

Operator	Description
ABS	ABS(expression) To get the absolute value of an expression.
ACOS	ACOS(expression) To get the arc cosine of an expression.
ASIN	ASIN(expression) To get the arc sine of an expression.
ATAN	ATAN(expression) This function is used to get the arc tangent of an expression.
CBRT	CBRT(expression) This function is used to get the cube root of an expression.
CEIL	CEIL(expression) This function is used to get the value of an expression rounded up to the nearest integer.
COS	COS(expression) This function is used to get the trigonometric cosine of an expression.
COSH	COSH(expression) This function is used to get the hyperbolic cosine of an expression.
EXP	EXP(expression) This function is used to get the Euler's number e raised to the power of x.
FLOOR	FLOOR(expression) To get the value of an expression rounded down to the nearest integer.

LOG	LOG(expression) To get the natural logarithm (base e) of an expression.
LOG10	LOG10(expression) To get the base 10 logarithm of an expression.
RANDOM	RANDOM() To get a pseudo random number (type double) greater than or equal to 0.0 and less than 1.0.
ROUND	ROUND(expression) To get the value of an expression rounded to an integer (if the result type is float) or rounded to a long (if the result type is double).
SIN	SIN(expression) To get the sine of an expression.
SINH	SINH(expression) To get the hyperbolic sine of an expression.
SQRT	SQRT(expression) To get the positive square root of an expression.
TAN	TAN(expression) To get the trigonometric tangent of an angle.
TANH	TANH(expression) To get the hyperbolic tangent of an expression.

ABS ()

The **ABS()** function of Pig Latin is used to calculate the absolute value of a given expression.

Syntax

Here is the syntax of the **ABS()** function.

```
ABS(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us calculate the absolute values of the contents of the math.txt file using **ABS()** as shown below.

```
abs_data = foreach math_data generate (data), ABS(data);
```

The above statement stores the result in the schema named **abs_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump abs_data;

(5.0,5.0)
(16.0,16.0)
(9.0,9.0)
(2.5,2.5)
(5.9,5.9)
(3.1,3.1)
```

ACOS ()

The **ACOS()** function of Pig Latin is used to calculate the arc cosine value of a given expression.

Syntax

Here is the syntax of the **ACOS()** function.

```
ACOS(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the arc cosine values of the contents of the math.txt file using ACOS() function as shown below.

```
acos_data = foreach math_data generate (data), ACOS(data);
```

The above statement stores the result in the schema named **abs_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump acos_data;

(5.0,NaN)
(16.0,NaN)
(9.0,NaN)
(2.5,NaN)
(5.9,NaN)
(3.1,NaN)
```

ASIN ()

The **ASIN()** function is used to calculate the arc sine value of a given expression.

Syntax

Here is the syntax of the **ASIN()** function.

```
ASIN(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the arc sine values of the contents of the math.txt file using ASIN() function as shown below.

```
asin_data = foreach math_data generate (data), ASIN(data);
```

The above statement stores the result in the schema named **asin_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump asin_data;

(5.0,NaN)
(16.0,NaN)
(9.0,NaN)
(2.5,NaN)
(5.9,NaN)
(3.1,NaN)
```

ATAN ()

The **ATAN()** function of Pig Latin is used to calculate the arc tan value of a given expression.

Syntax

Here is the syntax of the **ATAN()** function.

```
ATAN(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the arc tan values of the contents of the math.txt file using ATAN() function as shown below.

```
atan_data = foreach math_data generate (data), ATAN(data);
```

The above statement stores the result in the schema named **asin_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump atan_data;

(5.0,1.373400766945016)
(16.0,1.5083775167989393)
(9.0,1.460139105621001)
(2.5,1.1902899496825317)
(5.9,1.4029004062076729)
(3.1,1.2587541962439153)
```

CBRT ()

The **CBRT()** function of Pig Latin is used to calculate the cube root of a given expression.

Syntax

Here is the syntax of the **CBRT()** function.

```
CBRT(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the cube root values of the contents of the math.txt file using ATAN() function as shown below.

```
cbrt_data = foreach math_data generate (data), CBRT(data);
```

The above statement stores the result in the schema named **cbrt_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump cbrt_data;
(5.0,1.709975946676697)
(16.0,2.5198420997897464)
(9.0,2.080083823051904)
(2.5,1.3572088082974532)
(5.9,1.8069688790571206)
(3.1,1.4580997208745365)
```

CEIL ()

The **CEIL()** function is used to calculate value of a given expression rounded up to the nearest integer.

Syntax

Here is the syntax of the **CEIL()** function.

```
CEIL(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the ceil values of the contents of the math.txt file using CEIL() function as shown below.

```
ceil_data = foreach math_data generate (data), CEIL(data);
```

The above statement stores the result in the schema named **ceil_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump ceil_data;
```

```
(5.0,5.0)
(16.0,16.0)
(9.0,9.0)
(2.5,3.0)
(5.9,6.0)
(3.1,4.0)
```


COS ()

The **COS()** function of Pig Latin is used to calculate the cosine value of a given expression.

Syntax

Here is the syntax of the **COS()** function.

```
COS(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Now, let's calculate the cosine values of the contents of the math.txt file using COS() function as shown below.

```
cos_data = foreach math_data generate (data), COS(data);
```

The above statement stores the result in the schema named **cos_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump cos_data;

(5.0,0.28366218546322625)
(16.0,-0.9576594803233847)
(9.0,-0.9111302618846769)
(2.5,-0.8011436155469337)
(5.9,0.9274784663996888)
(3.1,-0.999135146307834)
```

COSH()

The **COSH()** function of Pig Latin is used to calculate the hyperbolic cosine of a given expression.

Syntax

Here is the syntax of the **COSH()** function.

```
COSH(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the hyperbolic cosine values of the contents of the math.txt file using COSH() function as shown below.

```
cosh_data = foreach math_data generate (data), COSH(data);
```

The above statement stores the result in the schema named **cosh_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump cosh_data;

(5.0,74.20994852478785)
(16.0,4443055.260253992)
(9.0,4051.5420254925943)
(2.5,6.132289479663686)
(5.9,182.52012106128686)
(3.1,11.121499185584959)
```

EXP()

The **EXP()** function of Pig Latin is used to get the Euler's number e raised to the power of x (given expression).

Syntax

Here is the syntax of the **EXP()** function.

```
EXP(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the exp values of the contents of the math.txt file using EXP() function as shown below.

```
exp_data = foreach math_data generate (data), EXP(data);
```

The above statement stores the result in the schema named **exp_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump exp_data;

(5.0,148.4131591025766)
(16.0,8886110.520507872)
(9.0,8103.083927575384)
(2.5,12.182493960703473)
(5.9,365.0375026780162)
(3.1,22.197949164480132)
```

FLOOR()

The **FLOOR()** function is used to calculate the value of an expression rounded down to the nearest integer. Here is the syntax of the **FLOOR()** function.

```
FLOOR(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Now, let's calculate the floor values of the contents of the **math.txt** file using **floor()** as shown below.

```
floor_data = foreach math_data generate (data), FLOOR(data);
```

The above statement stores the result in the schema named **floor_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump floor_data;

(5.0,5.0)
(16.0,16.0)
(9.0,9.0)
(2.5,2.0)
(5.9,5.0)
(3.1,3.0)
```

LOG ()

The **LOG()** function of Pig Latin is used to calculate the natural logarithm (base **e**) value of a given expression.

```
LOG(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the log values of the contents of the math.txt file using LOG() function as shown below.

```
log_data = foreach math_data generate (data),LOG(data);
```

The above statement stores the result in the schema named **log_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump log_data;

(5.0,1.6094379124341003)
(16.0,2.772588722239781)
(9.0,2.1972245773362196)
(2.5,0.9162907318741551)
(5.9,1.774952367075645)
(3.1,1.1314020807274126)
```

LOG10 ()

The **LOG10()** function of Pig Latin is used to calculate the natural logarithm base 10 value of a given expression.

```
LOG10(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the log10 values of the contents of the math.txt file using LOG10() function as shown below.

```
log_data = foreach math_data generate (data),LOG10(data);
```

The above statement stores the result in the schema named **log_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump log10_data;

(5.0,0.6989700043360189)
(16.0,1.2041199826559248)
(9.0,0.9542425094393249)
(2.5,0.3979400086720376)
(5.9,0.7708520186620678)
(3.1,0.4913616804737727)
```

RANDOM ()

The **RANDOM()** function is used to get a pseudo random number (type double) greater than or equal to 0.0 and less than 1.0.

```
RANDOM()
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now generate random values of the contents of the math.txt file using RANDOM() function as shown below.

```
random_data = foreach math_data generate (data), RANDOM();
```

The above statement stores the result in the schema named **random_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump random_data;

(5.0,0.6842057767279982)
(16.0,0.9725172591786139)
(9.0,0.4159326414649489)
(2.5,0.30962777780713147)
(5.9,0.705213727551145)
(3.1,0.24247708413861724)
```

ROUND ()

The **ROUND()** function is used to get the value of an expression rounded to an integer (if the result type is float) or rounded to a long (if the result type is double).

```
ROUND()
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
```

```
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now generate round values of the contents of the math.txt file using ROUND() function as shown below.

```
round_data = foreach math_data generate (data), ROUND(data);
```

The above statement stores the result in the schema named **round_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump round_data;
```

```
(5.0,5)
(16.0,16)
(9.0,9)
(2.5,3)
(5.9,6)
(3.1,3)
```

SIN ()

The **SIN()** function of Pig Latin is used to calculate the sine value of a given expression.

Syntax

Here is the syntax of the **SIN()** function.

```
SIN(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt


```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Now, let's calculate the sine values of the contents of the math.txt file using SIN() function as shown below.

```
sin_data = foreach math_data generate (data), SIN(data);
```

The above statement stores the result in the schema named **sin_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump sin_data;

(5.0,-0.9589242746631385)
(16.0,-0.2879033166650653)
(9.0,0.4121184852417566)
(2.5,0.5984721441039564)
(5.9,-0.3738765763789988)
(3.1,0.04158075771824354)
```

SINH()

The **SINH()** function is used to calculate the hyperbolic sine value of a given expression.

Syntax

Here is the syntax of the **SINH()** function.

```
SINH(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the hyperbolic sine values of the contents of the math.txt file using SINH() function as shown below.

```
sinh_data = foreach math_data generate (data), SINH(data);
```

The above statement stores the result in the schema named **sinh_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump sinh_data;

(5.0,74.20321057778875)
(16.0,4443055.26025388)
(9.0,4051.54190208279)
(2.5,6.0502044810397875)
(5.9,182.51738161672935)
(3.1,11.076449978895173)
```

SQRT ()

The **SQRT()** function is used to calculate the square root of a given expression.

Syntax

Here is the syntax of the **SQRT()** function.

```
SQRT(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the square root values of the contents of the math.txt file using SQRT() function as shown below.

```
sqrt_data = foreach math_data generate (data), SQRT(data);
```

The above statement stores the result in the schema named **sqrt_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump sqrt_data;

(5.0,2.23606797749979)
(16.0,4.0)
(9.0,3.0)
(2.5,1.5811388300841898)
(5.9,2.4289915799292987)
(3.1,1.76068165908337)
```

TAN ()

The **TAN()** function is used to calculate the trigonometric tangent of a given expression (angle).

Syntax

Here is the syntax of the **TAN()** function.

```
TAN(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the tan values of the contents of the math.txt file using TAN() function as shown below.

```
tan_data = foreach math_data generate (data), TAN(data);
```

The above statement stores the result in the schema named **tan_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump tan_data;

(5.0, -3.380515006246586)
(16.0, 0.3006322420239034)
(9.0, -0.45231565944180985)
(2.5, -0.7470222972386603)
(5.9, -0.4031107890087444)
(3.1, -0.041616750118239246)
```

TANH()

The **TANH()** function is used to calculate the hyperbolic trigonometric tangent of a given expression (angle).

Syntax

Here is the syntax of the **TANH()** function.

```
TANH(expression)
```

Example

Assume that there is a file named **math.txt** in the **HDFS** directory **/pig_data/**. This file contains integer and floating point values as shown below.

math.txt

```
5
16
9
2.5
5.9
3.1
```

And, we have loaded this file into Pig with a schema named **math_data** as shown below.

```
math_data = LOAD 'hdfs://localhost:9000/pig_data/math.txt' USING
PigStorage(',')as (data:float);
```

Let us now calculate the hyperbolic tangent values for the contents of the math.txt file using TANH() function as shown below.

```
tanh_data = foreach math_data generate (data), TANH(data);
```

The above statement stores the result in the schema named **tanh_data**. Verify the contents of the schema using the Dump operator as shown below.

```
Dump tanh_data;

(5.0,0.9999092042625951)
(16.0,0.99999999999999747)
(9.0,0.999999969540041)
(2.5,0.9866142981514303)
(5.9,0.9999849909996685)
(3.1,0.9959493584508665)
```

Part 11: Other Modes of Execution

30. User-Defined Functions

In addition to the built-in functions, Apache Pig provides extensive support for **User Defined Functions (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 –

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

```

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

  <modelVersion>4.0.0</modelVersion>
  <groupId>Pig_Udf</groupId>
  <artifactId>Pig_Udf</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.apache.pig</groupId>
      <artifactId>pig</artifactId>
      <version>0.15.0</version>
    </dependency>

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

```

4. Save the file and refresh it. In the **Maven Dependencies** section, you can find the downloaded jar files.
5. 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 {

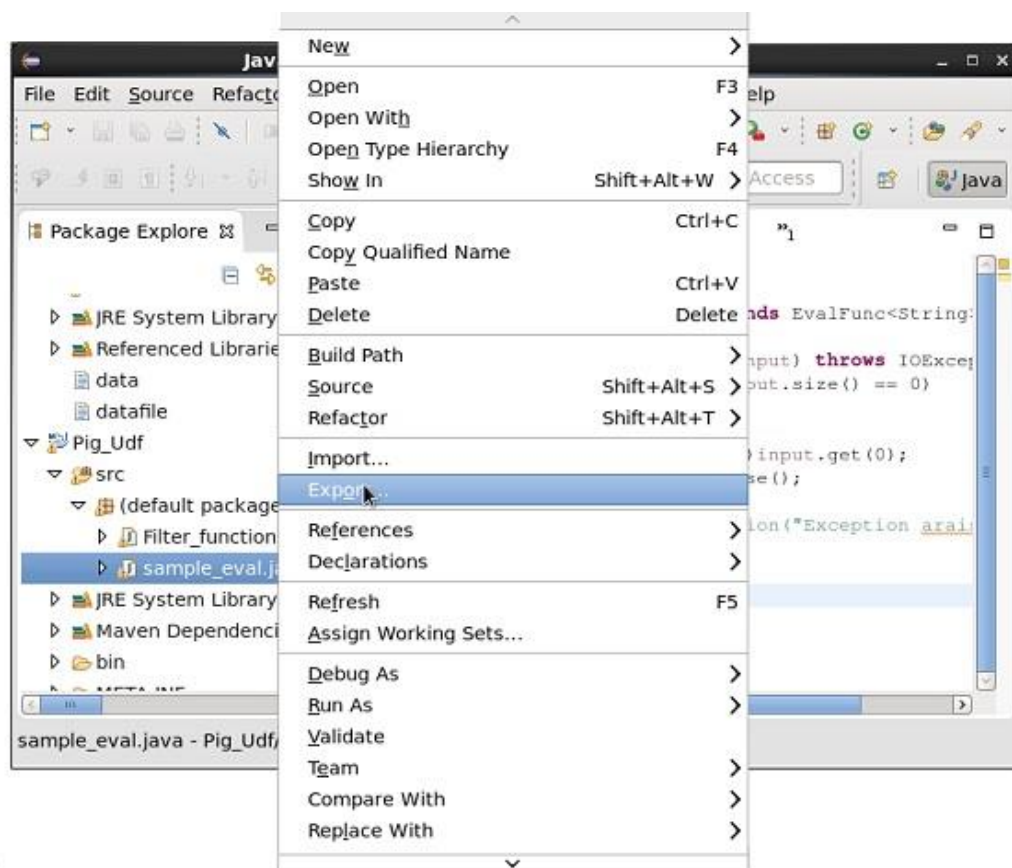
        if (input == null || input.size() == 0)
            return null;
        String str = (String)input.get(0);
        return str.toUpperCase();

    }
}

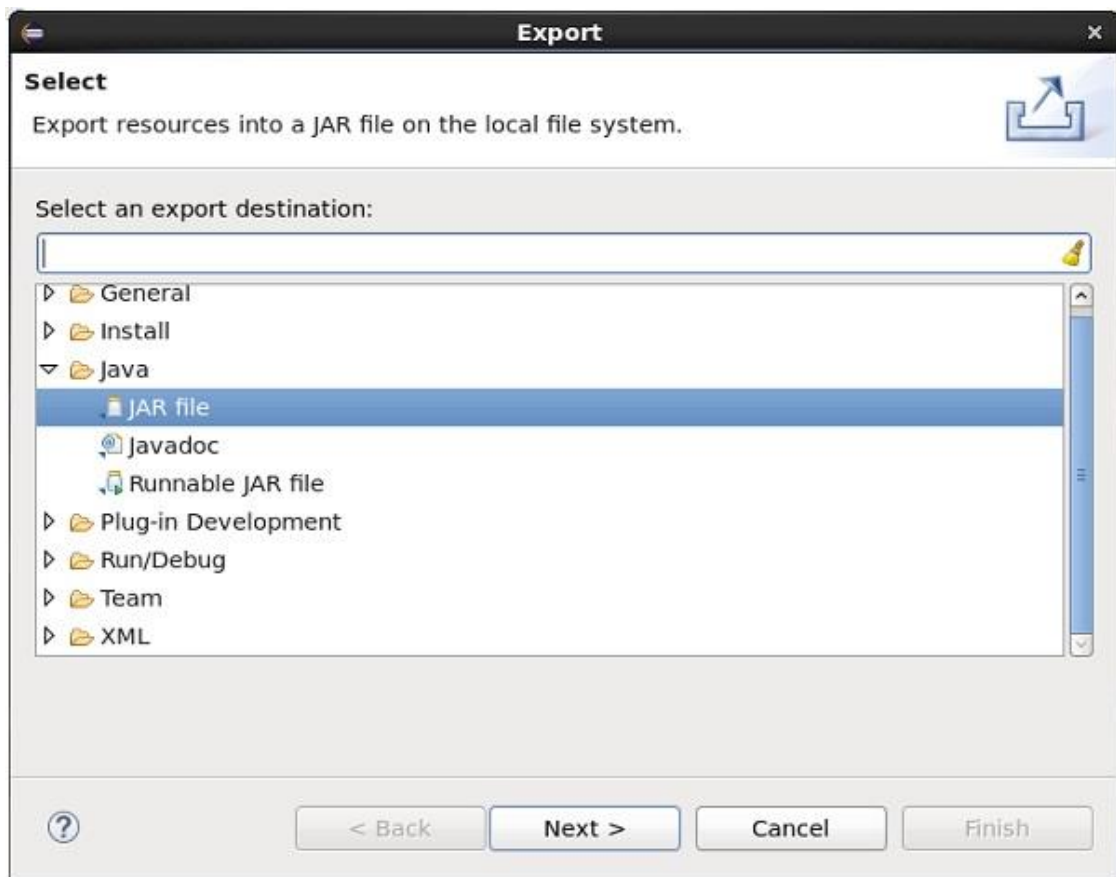
```

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.

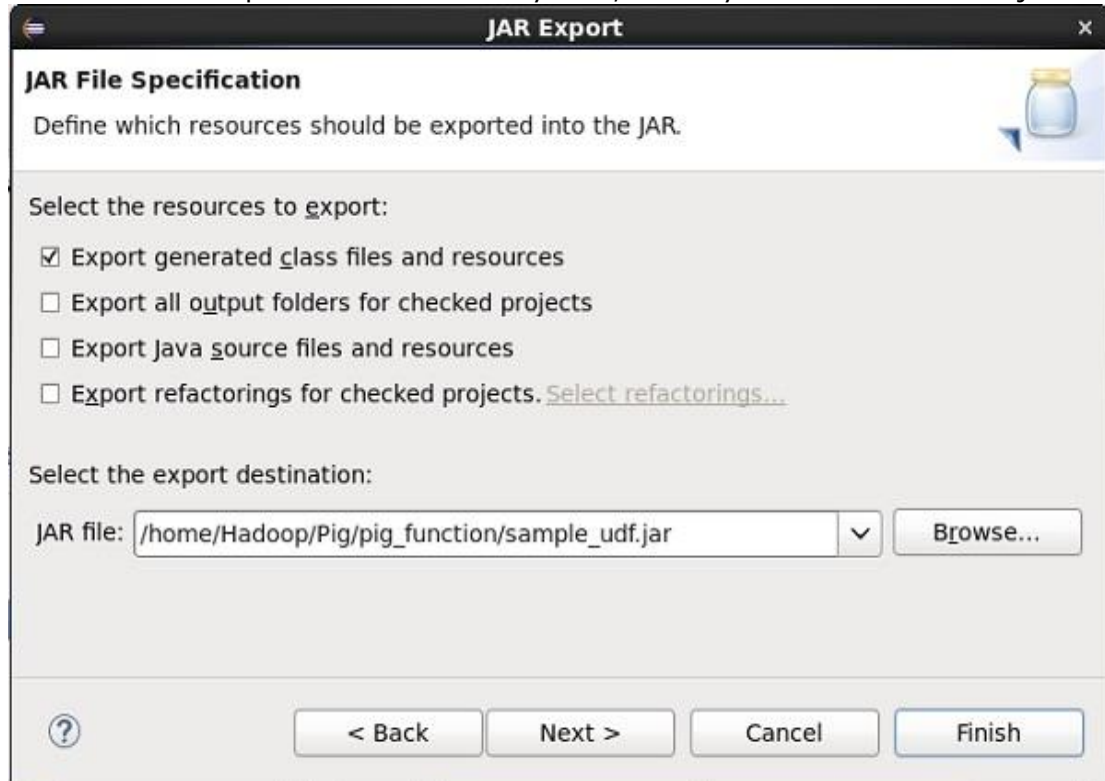
6. After compiling the class without errors, right-click on the Sample_Eval.java file. It gives you a menu. Select **export** as shown in the following screenshot.



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



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



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

Using the UDF

After writing the UDF and generating the Jar file, we have to register the Jar file using the Register operator, and define alias to the UDF using the define operator. Then you can use it in the Pig Latin statements just like any other built-in function.

Register

The **Register** operator is used to registers a JAR file which contains the UDF. By registering the Jar file, users can intimate the location of the UDF to Pig.

Syntax

Given below is the syntax of the Register operator.

```
REGISTER path;
```

Registering sample_udf.jar

Start Apache Pig in local mode as shown below.

```
$cd PIG_HOME/bin
$./pig -x local
```

Register the jar file **sample_udf.jar** which is in the path **/home/Hadoop/Pig/pig_data/sample_udf.jar**.

```
REGISTER '/home/Hadoop/Pig/pig_data/sample_udf.jar'
```

Define

The **Define** operator is used to assign an alias to a UDF or streaming command.

Syntax

Given below is the syntax of the Define operator.

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

Defining alias to the UDF

Define the alias for sample_eval as shown below.

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

Using the UDF

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

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

```
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**.

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

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

```
Dump Upper_case;
```

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

31. Running Scripts

Here in this chapter, we will see how how to run Apache Pig scripts in batch mode.

Comments in Pig Script

While writing a script in a file, we can include comments in it as shown below.

Multi-line comments

```
/* These are the multi-line comments  
   In the pig script */
```

Single –line comments

```
--we can write single line comments like this.
```

Executing Pig Script in Batch mode

While executing Apache Pig statements in batch mode, follow the steps given below.

Step 1

Write all the required Pig Latin statements in a single file. We can write all the Pig Latin statements and commands in a single file and save it as **.pig** file.

Step 2

Execute the Apache Pig script. You can execute the Pig script from the shell (Linux) as shown below.

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

You can execute it from the Grunt shell as well using the **exec** command as shown below.

```
grunt> exec /sample_script.pig
```

Executing a Pig Script from HDFS

We can also execute a Pig script that resides in the HDFS. Suppose there is a Pig script with the name **Sample_script.pig** in the HDFS directory named **/pig_data/**. We can execute it as shown below.

```
$ pig -x mapreduce hdfs://localhost:9000/pig_data/Sample_script.pig
```

Example

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

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,Mohanthi,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiyar,24,9848022333,Chennai
```

And we have read it into a relation **student** using 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 );
```

We also have a sample script with the name **sample_script.pig**, in the same HDFS directory performing operations and transformations on the **student** schema, as shown below.

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

student_order = ORDER student_details BY age DESC;

student_limit = LIMIT student_details 4;

Dump student_limit;
```

- The first statement of the script will load the data in the file named **student_data.txt** as a relation named **student**.
- The second statement of the script will arrange the tuples of the schema in descending order, based on age, and store it as **student_order**.

- The third statement of the script will store the first 4 tuples of **student_order** as **student_limit**.
- Finally the fourth statement will dump the content of the relation **student_limit**.

Let us now execute the **sample_script.pig** as shown below.

```
./pig -x mapreduce hdfs://localhost:9000/pig_data/sample_script.pig
```

Apache Pig gets executed and gives you the output with the following content.

```
(7,Komal,Nayak,24,9848022334,trivendram)
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,23,9848022335,Chennai)
2015-10-19 10:31:27,446 [main] INFO org.apache.pig.Main - Pig script completed
in 12 minutes, 32 seconds and 751 milliseconds (752751 ms)
```