

Table of Contents

CHAPTER 1: INTRODUCTION TO APACHE PIG	13
THEORY.....	13
QUICK INTRODUCTION TO HADOOP	13
<i>Why Hadoop?</i>	13
QUICK INTRODUCTION TO HADOOP DISTRIBUTED FILE SYSTEM	14
<i>Block Placement in HDFS</i>	15
<i>HDFS Architecture</i>	16
INTRODUCTION TO MAPREDUCE	17
<i>Architecture of MapReduce</i>	17
<i>Processing Data with MapReduce</i>	18
3V'S OF HADOOP	24
INTRODUCTION TO PIG	24
<i>What is Pig?</i>	24
<i>Why Pig?</i>	25
<i>Significance of the name Pig</i>	25
<i>Pig and Hadoop</i>	25
PIG ARCHITECTURE	25
<i>Modes of Execution</i>	26
<i>Interfaces for running Pig scripts</i>	27
PIG LATIN Vs SQL.....	27
PIG PHILOSOPHY	30
AIM.....	32
LAB EXERCISE 1	33
SUMMARY	34
REFERENCES.....	35
INDEX	36
CHAPTER 2: INSTALLING APACHE PIG	37
PREREQUISITES TO INSTALL APACHE PIG	37
AIM	39
LAB EXERCISE 2: INSTALLING APACHE PIG.....	40

TASK 1: DOWNLOADING APACHE PIG	41
TASK 2: INSTALLING APACHE PIG	45
TASK 3: CONFIGURING APACHE PIG.....	47
TASK 4: STARTING PIG IN LOCAL MODE AND TEZ LOCAL MODE	50
TASK 5: STARTING PIG IN MAPREDUCE MODE AND TEZ MODE.....	53
TASK 6: RUNNING PIG LATIN STATEMENTS	56
 LAB CHALLENGE.....	 60
 SUMMARY	 61
 REFERENCES.....	 62
 INDEX	 63
 CHAPTER 3: WORKING ON GRUNT.....	 64
WHAT IS GRUNT?.....	64
GRUNT FEATURES	64
PIG LATIN STATEMENTS IN GRUNT.....	65
HADOOP COMMANDS IN GRUNT	66
OTHER PIG COMMANDS IN GRUNT.....	66
AIM	67
 LAB EXERCISE 3: HANDS ON GRUNT.....	 68
TASK 1: USING HDFS COMMANDS IN GRUNT.....	69
TASK 2: USING PIG UTILITY COMMANDS IN GRUNT	75
TASK 3: USING PIG DIAGNOSTIC OPERATORS IN GRUNT	82
TASK 4: RUNNING A PIG SCRIPT	85
 LAB CHALLENGE	 88
 SUMMARY	 89
 REFERENCES.....	 90
 INDEX	 91
 CHAPTER 4: PIG LATIN DATA TYPES	 92
DATA TYPES	92

<i>Simple Types</i>	92
<i>Complex Types</i>	94
PIG LATIN NULLS.....	97
<i>Operations That Produce Nulls</i>	97
<i>Using Nulls as Constants</i>	97
PIG LATIN SCHEMAS	98
TYPE CASTING IN PIG LATIN.....	99
AIM	101
LAB EXERCISE 4: HANDS ON DATA TYPES.....	102
TASK 1: WORKING WITH SIMPLE DATA TYPES.....	103
TASK 2: WORKING WITH COMPLEX DATA TYPES	110
TASK 3: UNDERSTANDING NULL VALUES	116
TASK 4: USING CAST OPERATOR	119
LAB CHALLENGE	123
SUMMARY	124
REFERENCES.....	125
INDEX	126
CHAPTER 5: PIG LATIN – PART 1.....	127
WHAT IS PIG LATIN?.....	127
<i>Conventions used in Pig Latin</i>	128
<i>Case Sensitivity</i>	128
<i>Comments in Pig Latin</i>	129
RELATIONAL OPERATORS IN PIG LATIN - I	129
<i>LOAD</i>	129
<i>DUMP</i>	131
<i>STORE</i>	131
<i>FILTER</i>	132
<i>DISTINCT</i>	133
<i>FOREACH...GENERATE</i>	133
<i>GROUP</i>	134

<i>ORDER BY</i>	135
<i>JOIN</i>	135
<i>UNION</i>	136
<i>SAMPLE</i>	137
<i>LIMIT</i>	137
AIM	139
LAB EXERCISE 5: HANDS ON PIG LATIN PART - 1	140
TASK 1: LOADING AND STORING DATA	141
TASK 2: WORKING WITH FILTERS.....	147
TASK 3: APPLYING TRANSFORMATIONS	150
TASK 4: GROUPING AND SORTING	153
TASK 5: WORKING WITH JOINS AND UNION	159
TASK 6: USING SAMPLE AND LIMIT OPERATORS	165
LAB CHALLENGE	167
SUMMARY	168
REFERENCES.....	169
INDEX	170
CHAPTER 6: PIG LATIN – PART 2.....	171
RELATIONAL OPERATORS IN PIG LATIN - II.....	171
<i>ASSERT</i>	171
<i>COGROUP</i>	172
<i>CROSS</i>	173
<i>CUBE</i>	173
<i>ROLLUP</i>	174
<i>RANK</i>	174
<i>SPLIT</i>	176
<i>FLATTEN</i>	177
NESTED FOREACH	179
ADVANCED JOINS	181
<i>Fragment - Replicated Joins</i>	181

<i>Skewed Joins</i>	182
<i>Merge Joins</i>	183
<i>Merge-Sparse Joins</i>	184
TYPE CONSTRUCTION OPERATOR	185
DISAMBIGUATE OPERATOR	185
AIM	187
LAB EXERCISE 6: HANDS ON PIG LATIN PART - 2	188
TASK 1: USING COGROUP AND CROSS	189
TASK 2: USING CUBE AND ROLLUP	195
TASK 3: USING SPLIT OPERATOR TO SPLIT DATA	201
TASK 4: USING THE RANK OPERATOR	204
TASK 5: WORKING WITH NESTED FOREACH	206
TASK 6: OPTIMIZING JOINS	211
LAB CHALLENGE	214
SUMMARY	215
REFERENCES	216
INDEX	217
CHAPTER 7: FUNCTIONS	218
LOAD/STORE FUNCTIONS	219
<i>PigStorage</i>	219
<i>BinStorage</i>	220
<i>HBaseStorage</i>	220
<i>Other Load/Store Functions</i>	221
EVAL FUNCTIONS	222
<i>TOKENIZE</i>	222
<i>COUNT</i>	223
<i>COUNT_STAR</i>	223
<i>IsEmpty</i>	223
<i>SUM</i>	223
<i>SUBTRACT</i>	223

<i>MAX</i>	223
<i>MIN</i>	223
<i>AVG</i>	223
<i>CONCAT</i>	224
MATH FUNCTIONS.....	224
<i>RANDOM</i>	224
<i>CIEL</i>	224
<i>FLOOR</i>	224
<i>ROUND</i>	224
<i>ROUND_TO</i>	224
<i>ABS</i>	224
DATETIME FUNCTIONS	225
<i>ToDate</i>	225
<i>GetDay</i>	225
<i>DaysBetween</i>	225
STRING FUNCTIONS	225
<i>LOWER</i>	225
<i>UPPER</i>	226
<i>LTRIM</i>	226
<i>RTRIM</i>	226
<i>TRIM</i>	226
TUPLE, BAG, MAP FUNCTIONS.....	226
<i>TOTUPLE</i>	226
<i>TOBAG</i>	226
<i>TOMAP</i>	226
<i>TOP</i>	226
AIM	228
LAB EXERCISE 7: HANDS ON FUNCTIONS.....	229
TASK 1: USING LOAD/STORE FUNCTIONS	230
TASK 2: USING EVAL FUNCTIONS	233
TASK 3: USING MATH FUNCTIONS.....	237
TASK 4: USING DATETIME FUNCTIONS	239
TASK 5: USING STRING FUNCTIONS	242

TASK 6: USING TUPLE, BAG, MAP FUNCTIONS.....	245
LAB CHALLENGE	248
SUMMARY	249
REFERENCES.....	250
INDEX	251
CHAPTER 8: USER DEFINED FUNCTIONS – PART 1	252
EVAL FUNCTIONS	252
<i>Algebraic Interface.....</i>	252
<i>Accumulator Interface</i>	255
<i>Understanding Pig Types</i>	257
<i>Schemas</i>	259
<i>Error Handling</i>	259
<i>Reporting Progress</i>	259
<i>Overloading Functions</i>	260
FILTER FUNCTIONS.....	261
REGISTERING AND DEFINING UDFs.....	261
<i>Registering UDFs.....</i>	262
<i>Defining UDFs.....</i>	262
AIM	264
LAB EXERCISE 8: BUILDING EVAL AND FILTER UDFS	265
TASK 1: INSTALLING ECLIPSE IDE.....	266
TASK 2: BUILDING AN EVAL FUNCTION	274
TASK 3: IMPLEMENTING SCHEMAS IN UDFs	280
TASK 4: BUILDING A FILTER UDF	282
SUMMARY	286
REFERENCES.....	287
INDEX	288
CHAPTER 9: USER DEFINED FUNCTIONS – PART 2	289

LOAD FUNCTIONS.....	289
<i>getInputFormat()</i>	290
<i>setLocation()</i>	290
<i>prepareToRead()</i>	290
<i>getNext()</i>	290
<i>LoadMetadata</i>	291
<i>LoadPushDown</i>	291
<i>LoadCaster</i>	292
STORE FUNCTIONS.....	292
<i>getOutputFormat()</i>	293
<i>getStoreLocation()</i>	293
<i>prepareToWrite()</i>	293
<i>putNext()</i>	293
<i>StoreMetadata</i>	293
<i>StoreResources</i>	294
<i>cleanupOnFailure()</i>	294
AIM	295
LAB EXERCISE 9: BUILDING LOAD AND STORE UDFS.....	296
TASK 1: BUILDING A LOAD FUNCTION	297
TASK 2: USING THE LOAD UDF.....	302
TASK 3: BUILDING A STORE FUNCTION.....	304
TASK 4: USING THE STORE UDF.....	313
SUMMARY	316
REFERENCES.....	317
INDEX	318
CHAPTER 10: EMBEDDING PIG LATIN.....	319
EMBEDDING PIG IN PYTHON.....	319
<i>Compile</i>	320
<i>Bind</i>	321
<i>Run</i>	322
EMBEDDING PIG IN JAVA	322

<i>PigServer</i>	322
AIM	325
LAB EXERCISE 10: EMBEDDING PIG LATIN	326
TASK 1: INSTALLING JYTHON.....	327
TASK 2: EMBEDDING PIG IN PYTHON	329
TASK 3: DOWNLOADING INPUT DATA	332
TASK 4: EXECUTING PIG EMBEDDED PYTHON SCRIPT	333
TASK 5: EMBEDDING PIG IN JAVA.....	335
TASK 6: EXECUTING PIG EMBEDDED JAVA PROGRAM	338
SUMMARY	341
REFERENCES.....	342
INDEX	343
CHAPTER 11: ADMINISTERING & TESTING PIG	344
ADMINISTERING PIG	344
<i>Pig Properties</i>	344
<i>Hadoop Properties</i>	345
<i>Strict Output Location Check</i>	346
<i>Blacklist/Whitelist Pig Operators and Commands</i>	346
<i>Statistics</i>	347
TESTING PIG SCRIPTS	347
<i>PigUint</i>	347
AIM	348
LAB EXERCISE 11: HANDS ON ADMINISTERING AND TESTING PIG	349
TASK 1: USING PIG PROPERTIES.....	350
TASK 2: BLACKLIST/WHITELIST PIG COMMANDS/OPERATORS	354
TASK 3: LOOKING AT STATISTICS	356
TASK 4: USING PIGUNIT TO TEST PIG LATIN SCRIPTS	359
SUMMARY	368
REFERENCES.....	369

INDEX	370
CHAPTER 12: OPTIMIZING PIG	371
THE NEED FOR OPTIMIZATION.....	371
<i>Number of MapReduce Jobs</i>	371
<i>Key Skew at Reducer</i>	371
<i>Data at Shuffle Phase</i>	372
LIST OF OPTIMIZATION RULES.....	372
<i>ConstantCalculator</i>	373
<i>SplitFilter</i>	373
<i>PushUpFilter</i>	373
<i>MergeFilter</i>	374
<i>PushDownForEachFlatten</i>	374
<i>LimitOptimizer</i>	374
<i>ColumnMapKeyPrune</i>	375
<i>AddForEach</i>	375
<i>MergeForEach</i>	376
<i>GroupByConstParallelSetter</i>	376
<i>PartitionFilterOptimizer</i>	376
<i>PredicatePushDownOptimizer</i>	377
BEST PRACTICES TO ENHANCE PERFORMANCE	377
<i>Specifying Types Explicitly</i>	377
<i>Filtering Data Early and Frequently</i>	377
<i>Projecting Data Early and Frequently</i>	377
<i>Reducing Operators in Pipeline</i>	378
<i>Using the LIMIT operator</i>	378
<i>Using the DISTINCT operator</i>	378
<i>Implementing Algebraic Interface for UDFs</i>	379
<i>Using the Accumulator Interface</i>	379
<i>Remove Nulls in Data</i>	379
<i>Using Join Optimizations</i>	380
<i>Combining Small Files</i>	381
<i>Using Compression for Intermediate Results</i>	381
<i>Direct Fetch from HDFS</i>	381

<i>Using Automatic Local Mode</i>	382
<i>User Jar Cache</i>	382
<i>Using Parallel Features</i>	382
COMBINERS IN PIG	383
MULTI-QUERY EXECUTION.....	384
ADVANCED OPTIMIZATION USING APACHE TEZ	384
<i>DAG Generation in Tez</i>	384
<i>Container Reuse in Tez</i>	384
<i>Parallel Features</i>	385
LAB EXERCISE 12	386
SUMMARY	387
REFERENCES.....	388
INDEX	389
CHAPTER 13: THE HADOOP ECOSYSTEM & PIG	390
APACHE HIVE	390
<i>Comparing Hive and RDBMS</i>	391
<i>Comparing Hive and Pig</i>	391
APACHE HBASE	392
<i>HBase Data Model</i>	392
<i>Comparing Column Oriented and Row Oriented Databases</i>	393
<i>HBase features</i>	394
<i>Loading/Storing to HBase using Pig</i>	394
APACHE HCATALOG.....	395
<i>HCatalog and Pig</i>	395
OTHER HADOOP ECOSYSTEM PROJECTS.....	396
<i>Apache Zookeeper</i>	396
<i>Apache Sqoop</i>	397
<i>Apache Flume</i>	398
<i>Apache Oozie</i>	398
<i>Cascading</i>	399
<i>MongoDB</i>	399

LAB EXERCISE 13	400
SUMMARY	401
REFERENCES.....	402
INDEX	403

CHAPTER 1: INTRODUCTION TO APACHE PIG

Theory

This chapter is intended to provide a comprehensive introduction to the BigData technologies discussed throughout this book. We first address about what these technologies are and know how they can be applied to the challenges we face in the world of BigData. Although Apache Pig would be center of interest throughout the book but we shall discuss about related technologies whenever applicable.

Quick Introduction to Hadoop

Apache Hadoop is an open source distributed framework that allows storage and processing of large data (BigData) sets across cluster of commodity machines. Hadoop overcomes the traditional limitations of storing and computing of data by distributing the data over cluster of commodity machines making it scalable and cost-effective.

The idea of Hadoop was originated when Google released a white paper about **Google File System (GFS)** - a computing model built by Google which was designed to provide efficient, reliable access to data using large clusters of commodity hardware. The model was then adopted by Doug Cutting and Mike Cafarella for their search engine called "Nutch". Hadoop was then developed to support distribution for the Nutch search engine project by Doug Cutting and Mike Cafarella. Well, what does the name Hadoop mean? There is no significance for the name and it is not an acronym too. Hadoop is the name that Doug Cutting's son gave to his yellow stuffed elephant. The name is very unique, easy to remember and sometimes funny. Not only does Hadoop have such name with no significance but also its sub-projects tend to have such names which are based on names of animals like Pig and for the same reason. They are unique, not used anywhere else and are easy to remember.

Why Hadoop?

Companies today have been realizing that there is lot of information in unstructured documents spread across the network. A lot of data is available in the form of spreadsheets, text files, e-mails, logs, PDF's and other data formats that contains valuable information which can help discover new trends, designing new products, improving existing products, knowing customers better and what not. Data is

increasing at an alarming rate beyond limits like never before and there are no signs of slowing down, at least in the near future. To deal with such data, we need a reliable and low cost tool to meaningfully process it. Therefore, we use Hadoop. Hadoop helps us process all this BigData which is present in variety of formats reliably, in a very less time in a flexible and cost effective way.

Let us see why Hadoop is so popular and what it has in store for you.

- **Scalable:** Hadoop is scalable, meaning; you can just start from a single node server and eventually increase more nodes as you need more storage and more computing power.
- **Fault-Tolerant:** Hadoop helps prevent loss of data. All the data which is stored in Hadoop Distributed File System is broken into blocks and stored with a default replication factor of 3. While processing data, if a node goes off, the process does not stop but continues as the data still exists in other nodes.
- **Flexible:** Hadoop does not require schema. Hadoop can process unstructured, semi-structured and structured data from any kind of source or even from multiple sources.
- **Cost effective:** Hadoop does not require expensive high end computing hardware. Hadoop works well with a cluster of commodity machines by parallel computing.

Quick Introduction to Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is a File System which extends over a cluster of commodity machines rather than a single high end machine. HDFS is a distributed large scale storage component and is highly scalable. HDFS can accept node failures without losing data. HDFS is widely known for its reliability. Let us now check out why HDFS stands out of crowd when it comes to Distributed file systems.

Reliable Data Storage

HDFS is very much reliable when it comes to data storage. Whatever the data stored in HDFS is replicated by a default replication factor of 3. That means, even if a machine fails, the data will be still available in two other machines.

Cost Effective	HDFS can be deployed on cluster of commodity hardware and can save you a lot of bucks. High end expensive hardware is not required by HDFS to function.
Big Datasets	HDFS is capable of storing Petabytes of data over a cluster of machines where a file can range from Gigabytes to Terabytes of size. HDFS is not designed to store huge number of small sized files as the file system meta data is stored in memory of NameNode.
Streaming Data Access	HDFS provides streaming access to data. HDFS is best suited for batch processing of data and not suitable for interactive processing. HDFS is not designed for applications which require low latency access to data such as OnLine Transaction Processing (OLTP).
Simple Coherency Model	HDFS is designed to <i>write once and read many times</i> access model for files. Appending the content to files is supported at the end but cannot be updated at arbitrary point and it is also not possible to have multiple writers. Files can only be written by a single writer.

Block Placement in HDFS

Hadoop is designed such a way that the first block replica is placed on the same node as of client and the second replica is placed on a different rack to that of first replica. Third replica is placed on a random node on the same rack as of second replica. If the replication factor is more, random nodes in the cluster are selected to place the replicas. If a client running outside the cluster stores a file, a random node (That isn't busy) is picked to place the first replica. This way, if a node fails, the data is still available on other nodes of the cluster and if a rack fails, again, the data is still intact.

HDFS Architecture

HDFS is a Master and Slave architecture, in which the Master node controls and assigns jobs to all its slave nodes. The following terminologies are used to describe the Master and Slave nodes.

The Master Nodes in HDFS are:

- NameNode
- Secondary NameNode

The Slave Nodes in HDFS are:

- Data Nodes

These nodes are the core serving roles in HDFS architecture. Let us now look in detail about the roles of each Node and understand them better.

NameNode

NameNode is a HDFS daemon which controls all the Data Nodes and handles all the File System operations such as creating a directory, creating a file or reading and writing a file. The NameNode is responsible for managing the File System namespace image. It holds the image in memory, representing how the File System looks like. It also maintains the meta data for all the blocks of files in the File System and also tracks the replication value, so it knows the locations of blocks stored on Data Nodes within the cluster. But the meta data is not stored on to the disk and is every time recreated when it starts. NameNode stores all this information persistently on local disk in the form of namespace image and edit log. The NameNode is the *single point of failure* in the Hadoop cluster. If the NameNode fails, entire cluster fails.

Data Nodes

Data Nodes are the slave machines controlled by the NameNode, who actually do all the block operations. Data Nodes store and retrieve blocks when asked to do so by NameNode and periodically inform NameNode with the lists of blocks they store by sending heartbeats. Data Nodes replicate the data physically when instructed by the NameNode on

where and how to replicate.

Secondary NameNode

Secondary NameNode, as its name implies, is not exactly the Secondary NameNode. The secondary NameNode is not a high availability solution and does not automatically takeover the responsibilities of NameNode on failure. Its main role is to create checkpoint and take the backup of NameNode periodically. It is like a backup solution to the NameNode. The hardware specifications of secondary NameNode should be similar to that of NameNode. In case of NameNode's failure, the secondary NameNode can be manually configured to work as a primary NameNode. This is not a high availability solution.

Introduction to MapReduce

MapReduce is a programming model for processing large amounts of data stored in distributed file systems such as HDFS. MapReduce is low level programming and thus programs are written in low level languages such as Java, Python, Ruby etc. Let us look at the architecture of MapReduce.

Architecture of MapReduce

MapReduce, similar to HDFS is master-slave architecture but instead of NameNode in HDFS we have JobTracker and instead of DataNodes in HDFS as Task Trackers in MapReduce. Unlike NameNode and Data Nodes, the JobTracker and Task Trackers are not physical hardware components but Java programs running on their own JVM's inside the machines.

The detail description of MapReduce daemons is as follows:

JobTracker JobTracker is the MapReduce daemon which is master to the Task Trackers. The role of JobTracker is to receive job requests from Hadoop clients and assign work to task trackers on Data Nodes. The JobTracker queries the location of data on Data Nodes and tries to assign task trackers on Data Nodes where the data is present locally, so as to achieve data locality. If the node where the data present locally is busy, Job Tracker assigns tasks to task tracker within the same rack. If the task tracker fails for some reason, the JobTracker will assign the same task on another task tracker as the data is replicated across the cluster on other nodes too.

JobTracker is the single point of failure similar to that of NameNode. If JobTracker fails, all the task trackers fail and there will be no tasks running. So it is wiser to spend more for the machine which runs JobTracker so as to decrease the chances of failure.

Task Trackers Task Trackers are slaves to JobTracker which do the actual work. TaskTrackers accept tasks from JobTracker and performs the tasks. The task trackers send status or progress of the tasks to JobTracker in the form of heartbeats so that the JobTracker can know that the task trackers are performing as they should and they have not failed. The task trackers also send heartbeat messages to JobTracker about the free slots available with them for processing of tasks.

The failure of task tracker is not as serious as that of the JobTracker as the JobTracker can always assign the failed task to another task tracker.

Processing Data with MapReduce

MapReduce consists of two major phases through which the data is processed. The two Major MapReduce phases are:

- Map phase
- Reduce Phase

Both the phases have key-value pairs as input and output. The data types for the key-value pairs can be chosen by the developer and the developer has to specify a map function and a reducer function, which is the logic for processing the data in MapReduce. The map function serves as logic to map task and the reduce function serves as logic to reduce task.

Let us now look at the flow of data and the various stages the data is processed in MapReduce.

- The input to map task is in the form of a split. A split is a fixed chunk of data based on the inputFormat and should not be confused with HDFS block. Blocks belong to HDFS and splits belong to MapReduce. Block is the smallest size of data stored in HDFS where as a split is an input to map task. Optimal split size will be equal to the block size. A map task can process one input split at a time.
- Map tasks processes the splits using the map function in parallel and produces an output. The output from map tasks is not stored in HDFS but on local disk because the map output is intermediate result and there is no need to save it on HDFS with replications. The map output is only saved in local disk until the reducer has produced the final result. If we have specified no reducers for the job, the map output will be the ultimate result and will be stored in HDFS with replications.
- The output from all the map tasks is merged, sorted and partitioned. Merging is the process where the data from all the map tasks is merged together. Sorting is the process where the map tasks output is sorted based on the key. Partitioning is process where the data is divided based on keys so that the values from all the keys should go to one reducer. Partitioning is useful when there are multiple reducers used. This map tasks output will then be fed to reducer as input.
- The reducer then processes the map output using the reducer function and produces the desired result. Unfortunately, reducers cannot take the advantage of data locality and will be fetched across the network. Also the number of reducers is not set automatically unlike mappers as the number of mappers is automatically set based on number of splits. The output produced by reducers is the end result and will be stored in HDFS with replications.

Let us consider an example with few words and understand the concept of MapReduce in pictorial representation.

Consider a file which consists of search terms for a website with each search term in a separate line as shown below.

Input file →

```
Consultants Network
Consultants Network Website
Fusion Clouds
Fusion Clouds Website
OSP Website
Learning Voyage
Learning Voyage Website
Consultant Network
Consultant Network Website
```

For the sake of example, let us assume that each line is an input to different map tasks. Please note that, the above assumption is for better understanding of the concept and in reality each map task processes much more data. The input file is broken into splits so as to feed to the map task. The file is split line by line as each line corresponds to a search term will be sent to the mappers. The splits of the file are as shown below

Input Splits →

```
Consultants Network
Consultants Network Website
Fusion Clouds
Fusion Clouds Website
OSP Website
Learning Voyage
Learning Voyage Website
Consultant Network
Consultant Network Website
```

These input splits are further broken down into individual words and then submitted to mapper in the form of key value pairs. The key will be the word and value will be 1 as shown below.

Mapper 

Consultants	1
Network	1
Consultants	1
Network	1
Website	1
Fusion	1
Clouds	1
OSP	1
Website	1
Learning	1
Voyage	1
Learning	1
Voyage	1
Website	1
Consultant	1
Network	1
Consultant	1
Network	1
Website	1
Fusion	1
Clouds	1
Website	1

At this point, all the intermediate data from map tasks output goes through the shuffle and sort phase. All the relevant words are brought together in our example as shown below.

Shuffle & Sort →

Website	1
Network	1
Consultants	1
Consultants	1
Consultant	1
Consultant	1
Fusion	1
Fusion	1
Clouds	1
Clouds	1
Learning	1
Learning	1
Voyage	1
Voyage	1
OSP	1

When the intermediate data is processed through the shuffle and sort phase, it is time for reducer to produce the end result using the logic in reducer function. After shuffle and sort phase, the partitioner makes sure that the key value pairs with same keys go to same reducer if there are multiple reducers set.

The reducer will aggregate the input and produce the following output.

Reducer →

Clouds	2
Consultant	2
Consultants	2
Fusion	2
Learning	2
Network	4
OSP	1
Voyage	2
Website	5

The output from the reducers is concatenated to have a single file which contains the end result. The end result is as shown below.

Result →

Clouds	2
Consultant	2
Consultants	2
Fusion	2
Learning	2
Network	4
OSP	1
Voyage	2
Website	5

The result shows the occurrences of each word in a given file.

Finally let us conclude the quick introduction on HDFS by looking at 3V's of Hadoop which summarizes what Hadoop is capable of.

3V's of Hadoop

Hadoop can be better described with 3V's. The 3V's of Hadoop are as follows

- **Volume:** Hadoop is designed to process large amounts of data ranging from hundreds of Gigabytes to Petabytes. There are lots of Petabyte datasets available today and Exabyte datasets are not a distant dream. Hadoop excels while processing large amounts of data rather than small data.
- **Velocity:** Hadoop is designed to ingest data at higher speeds from multiple sources. Hadoop uses the distributed framework for parallel processing which in turn decreases the time taken to complete a job. Hadoop brings computation to data rather than bringing data to computation which makes network bandwidth a bottle neck.
- **Variety:** Hadoop can process data in structured, semi-structured and unstructured forms. There are no restrictions on schema. Unlike relational database management systems, Hadoop has the schema on read capability. There is no schema required while writing to HDFS. The schema can be parsed at read time.

The traditional computing models lack these features but this is where Hadoop excels providing you with more power to explore your data.

Now that we have had a quick introduction to Hadoop, let us shift our focus on the main topic of our discussion, Apache Pig.

Introduction to Pig

What is Pig?

Apache Pig is data flow execution platform for analyzing large datasets in parallel on Hadoop using a scripting language called *Pig Latin*. With Pig, you need not worry about the low level programming code to build mappers and reducers, testing, debugging and deploying. Pig does it all and that is why it is high level. Scripting is as simple as asking Pig to do work for you and all the low level stuff like creating MapReduce jobs under the hood is taken care by Pig. Pig script is just a few lines of statements whereas MapReduce applications require hundreds of lines of code to complete the same job. The best example for this is Joins. It takes only couple of lines to join two different datasets whereas MapReduce requires a great effort.

Pig is an open source Apache software foundation project available to download for free. You are free to download the source code, use it for your own benefit and change the source code according to your requirements if required by following the terms of Apache License.

Why Pig?

Why use Pig when we have MapReduce? Well, you might have figured out the answer from the previous section. Pig has a high level of abstraction and helps you complete your work faster. Pig helps people with weak low level programming skills to easily hook into their BigData and have their work done. Pig fits well in between low level and procedural MapReduce and declarative languages such as SQL.

Significance of the name Pig

Pig was originally developed at Yahoo Research around 2006. Pig was actually developed to have an ad-hoc way of creating and executing map-reduce jobs on very large data sets. But have you ever wondered what the significance of the name 'Hadoop' is? Well, there is no significance for the name and it is not an acronym too. A Research Engineer suggested this name and eventually was used for this project. The name is very unique, easy to remember and sometimes funny.

Pig and Hadoop

Pig is designed to run on the top of Hadoop by utilizing the Hadoop's components: Hadoop Distributed File System (HDFS) and MapReduce. By default, Pig reads input data from HDFS and also writes the output produced from jobs to HDFS. Pig also uses MapReduce as an underlying platform for executing all of its data processing. Pig itself does not have the capability to process the data but instead it uses MapReduce. Pig compiles the Pig Latin scripts and turns them into a series of one or more MapReduce jobs under the hood. The entire Map, Shuffle and Reduce phases is taken care of Pig and we need not worry about then when we are dealing with Pig. We discuss more about Pig in our next section which is Pig Architecture.

Pig Architecture

Pig's architecture is very simple and is made up of two components. They are

- Pig Latin (Language)
- Pig Latin Runtime Environment (Compiler)

Let us look at brief explanation regarding these components so that we can better understand the Pig architecture.

Pig Latin

Pig Latin is the high level language which is used to write Pig scripts for analyzing and processing large datasets. Pig Latin is

very simple to understand and does not require you to have knowledge of low level programming. A Pig Latin script is a series of operations and transformations applied to large input datasets to produce meaningful information as output.

Pig Latin Runtime Environment	The Runtime environment is the execution environment to process Pig Latin Programs. The runtime environment converts/translates the Pig Latin code into executable code such as MapReduce jobs under the hood. The developer need not bother about this process and focus on extracting information from huge datasets rather than worrying about the process of execution.
--------------------------------------	---

Modes of Execution

Pig has four modes of executing Pig scripts.

- **Local Mode:** In local mode, all the Pig scripts run on a single machine with local file system as storage and it does not require MapReduce and HDFS to function. Local Mode is useful for testing Pig logic on small datasets and going through the process of MapReduce infrastructure is not required. Local mode is best suited to run the queries on small datasets interactively, meaning, you can just write a line of statement and dump the result to the screen. You can actually view the extraction and transformation process step by step.
- **Tez Local Mode:** Tez Local mode is similar to that of local mode, except internally Pig will invoke Tez runtime engine. Apache Tez is an extensible framework for building YARN based, high performance batch and interactive data processing applications in Hadoop and its ecosystem projects such as Hive and Pig, that need to handle huge datasets. This mode is still in its experimental stage and may fail for large datasets.
- **MapReduce Mode:** Unlike local mode, Pig scripts run on cluster of machines (Hadoop cluster) unleashing the power of parallelism. The pig scripts are translated into series of MapReduce jobs and run on the large datasets to extract information required. MapReduce Mode is best suited with batch processing of large datasets. Developing a script interactively on MapReduce mode for large datasets might be time taking and hence it is recommended to use local mode with a subset of data interactively.

- **Tez Mode:** Pig can be fired up to use in Tez mode. Tez needs to have access to Hadoop cluster and HDFS installation. In MapReduce mode, the Pig script is translated to series of multiple MapReduce jobs, whereas Tez processes huge amount of data in a single Tez job in interactive and batch mode with high performance.

Interfaces for running Pig scripts

Pig scripts can be executed in three different ways.

- **Script:** A Pig script is a file which contains Pig Latin commands. The script ends with .pig extension and it is a convention but not required to successfully process the script. All you need to do is to specify the location of script from command line interface with the **pig** command. That will trigger commands inside the script and runs the job. For example, to execute a pig script with name *logs.pig* you need to run the command below:
`piglogs.pig`
- **Grunt:** Grunt is an interactive command interpreter for running Pig Latin commands. With Grunt you can run the commands interactively and see the result immediately. Pig scripts can also run within Grunt.
- **Embedded:** Pig programs can be executed within Java, Python and Javascript.

We shall look at all of them in more detail in our upcoming chapters.

Pig Latin Vs SQL

Dataflow and query languages are often described as similar but there are more than few differences which are worth mentioning. Let us look at them in detail below.

- Pig Latin is procedural data flow language. Pig Latin allows developers to build pipelines for processing the input data. Therefore there is no requirement of temporary tables or writing inside out queries. SQL on the other hand is a query language. SQL helps developers to query the data and get meaningful insights. SQL requires the developer to write queries inside-out or use temporary tables for processing the input data.

To understand this better, consider a case where data from tables, *movieRatings* and *users* have to be joined, filtered and then joined with a third table *userLocation*. The result will then be aggregated and stored in a new table *ratingsPerLocation*. The SQL code in this case can be written as follows:

```
INSERT INTO ratingsPerLocation
SELECT location, COUNT (*) FROM userLocation JOIN
(
SELECT username,ip FROM users JOIN ratings
ON(users.username=ratings.username)
WHERE rating >0;
)
USING ip
GROUPBY location;
```

While in Pig Latin, this can be written as follows:

```
Users          =LOAD 'users' AS (username, age,
ip);
Ratings        =LOAD 'ratings' AS (username,
movie, rating);
fil_ratings    =FILTER ratings BY rating >0;
user_ratings   =JOIN users BY username,
fil_ratings BY username;
user_location  =LOAD 'userLocation' AS (ip,
location);

loc_ratings    =JOIN user_ratings BY ip,
user_location BY ip;
grouped_ratings =GROUP loc_ratings BY location;
ratingsPerLocation= FOREACH grouped_ratings
GENERATE group,COUNT(loc_ratings);
STORE ratingsPerLocation INTO 'ratingsPerLocation';
```

- Pig Latin does not require schema while loading data. While SQL strictly requires that schema should be present while loading data.
- As Pig Latin is a data flow language, it involves loading, transforming and result by processing entire dataset. On the other hand SQL performs queries from inside out and does not require touching entire dataset.

- Pig Latin is much easier to work with as we can visualize the data flow in the pipeline but it is not so easy to work with complex queries in SQL.
- With Pig Latin, developers have the advantage to checkpoint data at any point of time in the pipeline but SQL has no such advantage. Having a Checkpoint for data is necessary so as to prevent the job from rerunning in case of failure. For example, consider the case above. We could have used an extra STORE statement after performing the first JOIN as below.

```

Users           =LOAD 'users' AS (username,
age,ip);
Ratings         =LOAD 'ratings' AS (username,
movie, rating);
fil_ratings     =FILTER ratings BY rating > 0;
user_ratings    =JOIN users BY username,
fil_ratings BY username;
STORE fil_ratings INTO 'filtered_ratings';
user_location   =LOAD 'userLocation' AS (ip,
location);
loc_ratings     =JOIN user_ratings BY ip,
user_location BY ip;
grouped_ratings =GROUP loc_ratings BY location;
ratingsPerLocation=FOREACH grouped_ratings GENERATE
group, COUNT(loc_ratings);
STORE ratingsPerLocation INTO 'ratingsPerLocation';

```

This would ensure that even if the job fails after joining the first datasets, you need not rerun whole script again. You can just start loading from the checkpoint.

- Pig Latin allows developers to use their own optimizations while implementing data flows but SQL does not allows developers to specify their own optimization techniques. For example, A SQL developer can specify a join operation but cannot implement joining optimization. It is up to the optimizer to decide which implementation to use. While in Pig Latin, developers can specify Joining and grouping implementations to use and Pig uses what it is asked to use.
- Pig Latin is not suitable for OLTP and low latency queries. No random reads and writes are supported. But SQL is best suited for online, low latency queries for transactions and indexes. Random reads and writes are supported.

- Pig Latin allows developers to use user code at any point of time during the pipeline. The user code is known as User Defined Function (UDF) in Pig Latin. The UDF is written in Java and can be implemented in the pipeline at any point of time. The UDF's are nothing but the functions which help in Extract, Transform and Load data. There are a bunch of built in functions in Pig but there might be cases where you would like to implement your own functions and have them used in the data flow pipeline. We shall look at the functions and UDF's in detail in upcoming chapters.

Pig Philosophy

The developers who contribute for Pig are required to follow the following founding principles to help grow Pig over time. The following presents these principles. Please check the References page to check them online at Apache website.

Pigs Eat Anything

Pig can operate on data whether it has metadata or not. It can operate on data that is relational, nested, or unstructured. And it can easily be extended to operate on data beyond files, including key/value stores, databases, etc.

Pigs Live Anywhere

Pig is intended to be a language for parallel data processing. It is not tied to one particular parallel framework. It has been implemented first on Hadoop, but we do not intend that to be only on Hadoop.

Pigs Are Domestic Animals

Pig is designed to be easily controlled and modified by its users.

Pig allows integration of user code where ever possible, so it currently supports user defined field transformation functions, user defined aggregates, and user defined conditionals. These functions can be written in Java or scripting languages that can compile down to Java (e.g. Jython). Pig supports user provided load and store functions. It supports external executables via its stream command and Map Reduce jars via its MapReduce command. It allows users to provide a custom partitioner for their jobs in some circumstances and to set the level of reduce parallelism for their jobs.

Pig has an optimizer that rearranges some operations in Pig Latin scripts to give better performance, combines Map Reduce jobs together, etc. However, users can easily turn this optimizer off to prevent it from making changes that do not make sense in their situation.

Pigs Fly

Pig processes data quickly. We want to consistently improve performance, and not implement features in ways that weigh pig down so it can't fly.

AIM

When you complete the all next chapters with lab exercises, you will be able to:

- Install and configure Apache Pig
- Write Pig Latin code and use Grunt shell
- Administer and Optimize Pig
- Write UDF's and implement them

In addition to the points above, you will have the basic knowledge of:

- Administration
- Security
- Monitoring
- Troubleshooting

LAB EXERCISE 1

"There are no activities required for this lab"

SUMMARY

Apache Pig is an ultimate tool to visualize data flow. With built in functions and user defined functions in Apache Pig, we can easily process data with fewer statements while it may take some effort with MapReduce

REFERENCES

- <http://pig.apache.org/>
- <http://pig.apache.org/philosophy.html>
- <https://developer.yahoo.com/blogs/hadoop/comparing-pig-latin-sql-constructing-data-processing-pipelines-444.html>
- <http://hadoop.apache.org/>

INDEX

Quick Introduction to Hadoop	13
Quick Introduction to Hadoop Distributed File System	14
Introduction to MapReduce	17
3V's of Hadoop	24
Introduction to Pig	24
Pig Architecture	25
Pig Latin Vs SQL	27
Pig Philosophy	30
AIM	32
LAB EXERCISE 1	33
REFERENCES	34
SUMMARY	35

CHAPTER 2: INSTALLING APACHE PIG

Theory

In the previous chapter, we have had a good Introduction to Hadoop and Pig. Now, it is time to get our hands dirty and start our journey with Apache Pig by installing it on our machines. Apache Pig is open source top level Apache project and is available for free download. The Apache Pig package is available for download along with its source code from Apache website. Apache Pig can also be downloaded as a part of Hadoop distributions such as Cloudera and HortonWorks.

In this chapter, we install Apache Pig on a Pseudo distributed Hadoop cluster. In Pseudo-Distribution mode, all the Hadoop daemons run on a single machine and all the Hadoop daemons are allocated their own JVM's. Pseudo-distributed mode can help you understand by a way of simulating how a fully distributed mode Hadoop cluster works.

Pig is also available on cloud as *Software as a Service (SAAS)* which allows you to run Pig on cloud. Either you use Hadoop and Pig in cloud or your own machines, the installation steps and instructions to use Pig remain the same. You will have to take care of all the installation as well as maintenance of the cluster. However, Amazon provides you with a service call *EMR (Elastic Map Reduce)* where Pig on cloud is pre-installed and you need not worry about installing and maintaining a cluster. All you need to do is rent a virtual Hadoop cluster and pay per number of hours you use them. Please check the *References* section for more information.

Prerequisites to Install Apache Pig

The Apache Pig 0.14.0 version requires the following mandatory dependencies to be installed before it can work as it should for both UNIX and Window users.

- **Hadoop 0.23.X, 1.X or 2.X**
Pig can run with different versions of Hadoop by setting HADOOP_HOME environment variable to point to the directory where Hadoop is installed. If HADOOP_HOME environment variable is not set, Pig will run with the embedded version by default, currently Hadoop 1.0.4.
- **Java 1.7**
Java is another prerequisite for installing Apache Pig. You will have to install Java Development Kit (JDK) and Java Runtime Environment (JRE) version 6 or later on your machine. You can obtain JDK and JRE from

Oracle's website. JAVA_HOME environment variable should point to the directory where Java is installed.)

Optionally, you can also install the following installed.

- **Python 2.7**(Python can be used while streaming Python UDF's.)
- **Ant 1.8**(Ant is used for building Pig from source.)

Windows users should have Cygwin installed.

The following components are used throughout the lab exercises.

- 64 bit Ubuntu LTS 14.10
- Hadoop 2.6.0
- Pig 0.14.0
- Java SE Development Kit 8 (JDK8)

All the above software is open source and free to use.

Let us now proceed with our lab exercise and have Pig installed up and running. Please note that the following lab exercises assume that you have Hadoop and Java installed already. If you do not have them already installed, please install them before proceeding to the lab exercises.

All measures have been taken to cover all the steps required for installing Apache Pig but if you find (unlikely) a strange error during the process, please try to resolve it by researching it over the internet.

AIM

The aim of the following lab exercise is to install Apache Pig, configure settings and run our first Pig script.

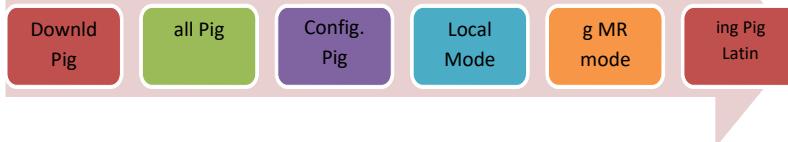
Following steps are required:

- Task 1: Downloading Apache Pig
- Task 2: Installing Apache Pig
- Task 3: Configuring Apache pig
- Task 4: Starting Apache Pig in Local Mode and Tez Local Mode
- Task 5: Starting Apache Pig in MapReduce Mode and Tez Mode
- Task 6: Running Pig Latin Statements

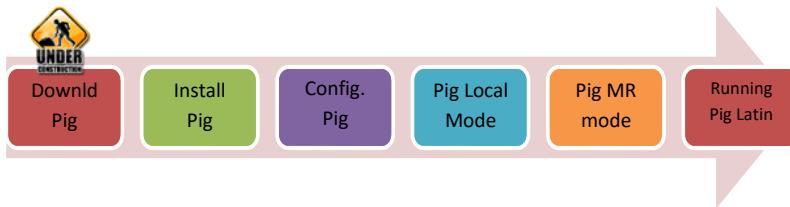
We need following packages to perform the lab exercise:

- Apache Pig
<http://apache.bytenet.in/pig/>

Lab Exercise 2: INSTALLING APACHE PIG



-
1. **Downloading Apache Pig**
 2. **Installing Apache Pig**
 3. **Configuring Apache pig**
 4. **Starting Apache Pig in Local Mode**
 5. **Starting Apache Pig in MapReduce Mode**
 6. **Running Pig Latin Statements**



Task 1: Downloading Apache Pig

Step 1: Open your internet browser and navigate to the following website:

<http://pig.apache.org/releases.html>

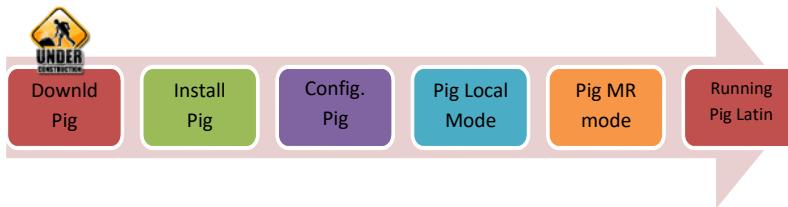
Apache Pig Releases

- Download
- News

- 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

Step 2: Scroll down a bit and you will see something like:

Download a release now! [Pig 0.8 and later](#) [Pig 0.7 and before](#)



Click on [Pig 0.8 and later](#) and you will be taken to the Apache Pig download page with bunch of mirror links.

-
- [8 April, 2009: release 0.2.0 available](#)
 - [5 December, 2008: release 0.1.1 available](#)
 - [11 September, 2008: release 0.1.0 available](#)
-

Download

Releases may be downloaded from Apache mirrors.

Download a release now! [Pig 0.8 and later](#) [Pig 0.7 and before](#)

[Get Pig .rpm or .deb](#)

Starting with Pig 0.12, Pig will no longer publish .rpm or .deb artifacts as part of its release. [Apache Bigtop](#) provides . projects. See [Bigtop's how to instal page](#) for details.

[Get Pig from Maven](#)

Pig jars, javadocs, and source code are available from [Maven Central](#).

News

20 November, 2014: release 0.14.0 available

The highlight of this release includes Pig on Tez, OrcStorage, loader predicate push down, constant calculation optim

Step 3: Click on any of the mirror link to download Apache Pig.



The Apache Software Foundation

Apache Download Mirrors

We suggest the following mirror site for your download:

<http://apache.bytenet.in/pig>

Other mirror sites are suggested below. Please use the backup mirrors only to download.

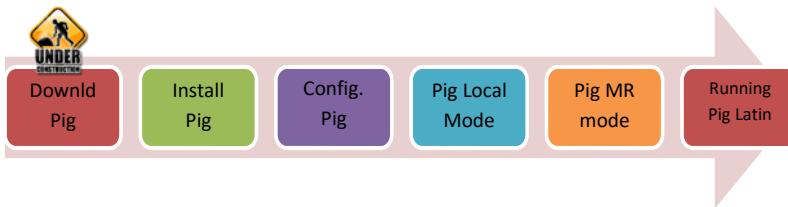
HTTP

<http://apache.bytenet.in/pig>

Backup Sites

Please use the backup mirrors only to download PGP and MD5 signatures to **verify you**

<http://www.eu.apache.org/dist/pig>



Step 4: We shall be downloading the latest release of Pig. At the point of writing this book, the latest release is 0.14.0. Click on *pig-0.14.0* to download the latest release. You may also click on the *latest* folder to download the latest Apache Pig release.

You might have an advance release at the time you read this book. It does not matter which Pig release you download but make sure the Pig release you download is compatible with version of Hadoop you have setup. All the Pig releases might not be compatible with all the Hadoop versions. You can check the compatibility in the releases page under “News” section.

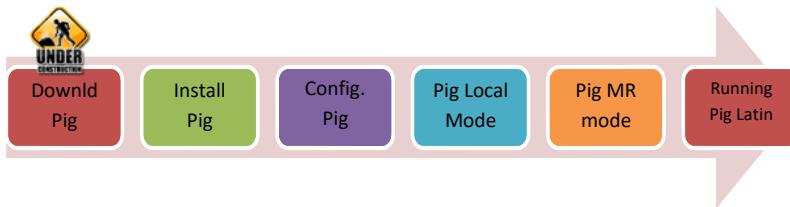
Pig Releases

Please make sure you're downloading from [a nearby mirror site](#), not from www.apache.org. Older releases are available from the [archives](#).

Name	Last modified	Size	Description
Parent Directory		-	
latest/	30-Dec-2014 11:21	-	
pig-0.12.1/	30-Dec-2014 11:21	-	
pig-0.13.0/	30-Dec-2014 11:21	-	
pig-0.14.0/	30-Dec-2014 11:21	-	
HEADER.html	13-Jan-2015 00:20	317	

Apache/2.2.22 (Debian) Server at apache.bytenet.in Port 80

Step 5: The directory consists of a Read me file along with Pig source and package. Download the package ending with **.tar.gz** as shown in the screenshot below.



Index of /pig/pig-0.14.0

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 README.txt	20-Nov-2014 14:01	1.3K	
 pig-0.14.0-src.tar.gz	20-Nov-2014 14:01	14M	
 pig-0.14.0.tar.gz	20-Nov-2014 14:01	114M	

Apache/2.2.22 (Debian) Server at apache.bytenet.in Port 80

After the download is complete, you can find the package in your *Downloads* folder.

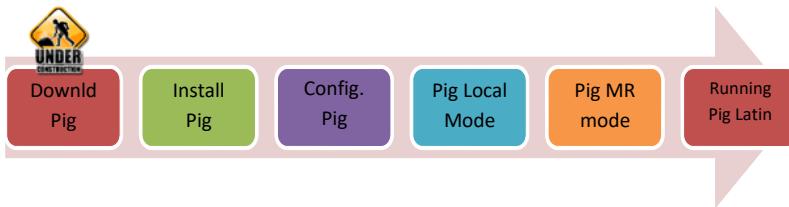
You can also download this via Terminal using the **wget** command as shown in the screenshot below. To open Terminal in Ubuntu, press Alt + Ctrl + T or use the search option and search for the terminal.

For example:

```
$ wget <Link to the package>
```

```
uzair@ubuntu:~$ wget http://apache.bytenet.in/pig/pig-0.14.0/pig-0.14.0.tar.gz
--2015-01-16 10:09:24--  http://apache.bytenet.in/pig/pig-0.14.0/pig-0.14.0.tar.
gz
Resolving apache.bytenet.in (apache.bytenet.in)... 150.107.171.242, 2400:4a80::e
Connecting to apache.bytenet.in (apache.bytenet.in)|150.107.171.242|:80... conne
cted.
HTTP request sent, awaiting response... 200 OK
Length: 119496823 (114M) [application/x-gzip]
Saving to: 'pig-0.14.0.tar.gz'

0% [                                         ] 443,528      42.3KB/s eta 39m 24s
```



After the download is complete, you can find the package in your *Home* folder.

Step 5 (Optional): You can also download the Pig source to customize Pig according to your requirements. To download Pig source, click on *pig-0.14.0-src.tar.gz* as shown in the screenshot below.

Index of /pig/pig-0.14.0

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 README.txt	20-Nov-2014 14:01	1.3K	
 pig-0.14.0-src.tar.gz	20-Nov-2014 14:01	14M	
 pig-0.14.0.tar.gz	20-Nov-2014 14:01	114M	

Apache/2.2.22 (Debian) Server at apache.bytenet.in Port 80

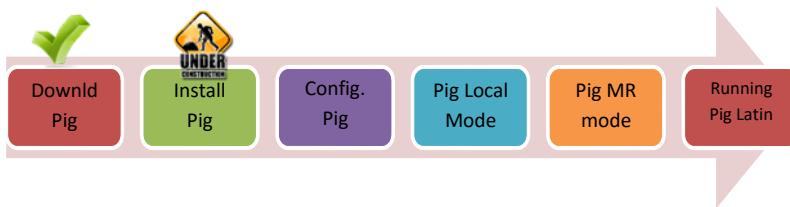
You can also download the source from Terminal. Please follow the instructions in Step 4 to do so.

Task 1 is complete!

Task 2: Installing Apache Pig

Step 1: After Pig package has been downloaded, extract the contents of the package by running the following command from the command line interface.

```
$ tar -zxvf pig-0.14.0.tar.gz
```



Please note that if you have downloaded Apache Pig via internet browser, the package might be saved in Downloads directory and not in the home directory. You will have to copy the package to home directory before you run the command above or navigate to Downloads directory and extract there.

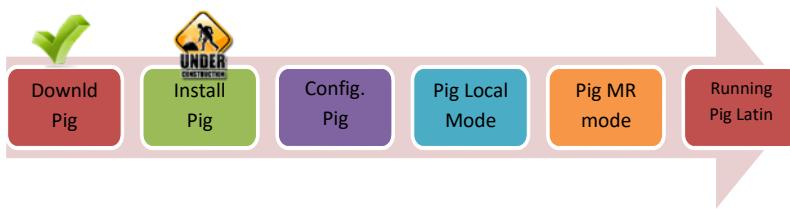
```
uzair@ubuntu: ~
uzair@ubuntu:~$ tar -zxf pig-0.14.0.tar.gz
```

Step 2: Move the extracted Pig directory to the /user/local/directory, where our Hadoop installation already exists by running the following command

```
$ sudo mv pig-0.14.0 /usr/local
```

```
uzair@ubuntu:~/Desktop/Downloads$ tar -zxf pig-0.14.0.tar.gz
pig-0.14.0/tutorial/scripts/script1-hadoop.pig
pig-0.14.0/tutorial/scripts/script1-local.pig
pig-0.14.0/tutorial/scripts/script2-hadoop.pig
pig-0.14.0/tutorial/scripts/script2-local.pig
pig-0.14.0/tutorial/src/org/apache/pig/tutorial/ExtractHour.java
pig-0.14.0/tutorial/src/org/apache/pig/tutorial/NGramGenerator.java
pig-0.14.0/tutorial/src/org/apache/pig/tutorial/NonURLDetector.java
pig-0.14.0/tutorial/src/org/apache/pig/tutorial/ScoreGenerator.java
pig-0.14.0/tutorial/src/org/apache/pig/tutorial/ToLower.java
pig-0.14.0/tutorial/src/org/apache/pig/tutorial/TutorialTest.java
pig-0.14.0/tutorial/src/org/apache/pig/tutorial/TutorialUtil.java
pig-0.14.0/bin/pig
pig-0.14.0/bin/pig.cmd
pig-0.14.0/bin/pig.py
uzair@ubuntu:~$ sudo mv pig-0.14.0 /usr/local
[sudo] password for uzair:
uzair@ubuntu:~$
```

Step 3: You can cross check if the directory has been moved by navigating to /user/local/directory so that we can proceed with configuration settings of Apache Pig.



Task 2 is complete!

Task 3: Configuring Apache Pig

Configuring Apache Pig is just setting up few environment variables so that Pig can access Hadoop daemons and run MapReduce jobs.

Step 1: The first step involves setting up the path variables in `.bashrc` file. Open the terminal and run the following command:

```
$ sudo gedit .bashrc
```

.bashrc is a shell script which is executed by bash before starting a command line shell. The .bashrc file can be used to setup environment, export variables, create functions, aliases and many more. We use .bashrc to export the Pig path automatically so that we need not export the path every time we start a terminal while working with Pig.



```

pig-0.14.0/tutorial/src/org/apache/pig/tutorial/TutorialUtil.java
pig-0.14.0/bin/pig
pig-0.14.0/bin/pig.cmd
pig-0.14.0/bin/pig.py
uzair@ubuntu:~$ sudo mv pig-0.14.0 /usr/local
[sudo] password for uzair:
uzair@ubuntu:~$ sudo gedit .bashrc
[sudo] password for uzair:

```

As soon as you run this command, you should have the `.bashrc` file opened.

Step 2: In the `.bashrc` file add the following environment variables after the Hadoop variables as shown in the screenshot.

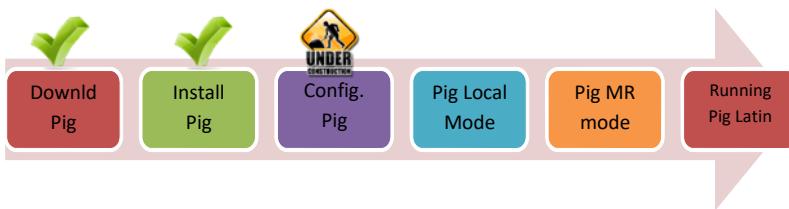
```

```
if ! shopt -oq posix; then
 if [-f /usr/share/bash-completion/bash_completion]; then
 . /usr/share/bash-completion/bash_completion
 elif [-f /etc/bash_completion]; then
 . /etc/bash_completion
 fi
fi
export JAVA_HOME=/usr/local/java/jdk1.8.0_25
export PATH=$PATH:$JAVA_HOME/bin
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH:$HADOOP_INSTALL/bin
export PATH=$PATH:$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
export PIG_INSTALL=/usr/local/pig-0.14.0
export PATH=$PATH:$PIG_INSTALL/bin
```

```

Step 3: Save and close the file. Refresh the terminal by running the following command.

```
$ . .bashrc
```



```
uzair@ubuntu:~$ sudo mv pig-0.14.0 /usr/local
[sudo] password for uzair:
uzair@ubuntu:~$ sudo gedit .bashrc
[sudo] password for uzair:

(gedit:18369): Gtk-WARNING **: Calling Inhibit failed: GDBus.Error:org.freedesktop.DBus.Error.ServiceUnknown: The name org.gnome.SessionManager was not provided by any .service files

(gedit:18369): Gtk-WARNING **: Calling Inhibit failed: GDBus.Error:org.freedesktop.DBus.Error.ServiceUnknown: The name org.gnome.SessionManager was not provided by any .service files
uzair@ubuntu:~$ .bashrc
uzair@ubuntu:~$
```

You have successfully installed and configured Apache Pig at this point.

Step 4: To check if Pig has been configured correctly, run the following command to check the version of Pig installed. You should have the output something like in the screenshot shown below.

```
$ pig -version
```

```
uzair@ubuntu:~$ .bashrc
uzair@ubuntu:~$ pig -version
Apache Pig version 0.14.0 (r1640057)
compiled Nov 16 2014, 18:02:05
uzair@ubuntu:~$
```

With this you can confirm that Pig has been installed and configured successfully and is ready to start working.

Task 3 is complete!



Task 4: Starting Pig in Local Mode and Tez Local Mode

Step 1: From the command line interface, run the following command to start Pig in local mode

```
$ pig -x local
```

```
uzair@ubuntu:~$ pig -x local
15/01/16 11:32:57 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/01/16 11:32:57 INFO pig.ExecTypeProvider: Picked LOCAL as the ExecType
2015-01-16 11:32:57,485 [main] INFO org.apache.pig.Main - Apache Pig version 0.
14.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-01-16 11:32:57,485 [main] INFO org.apache.pig.Main - Logging error messages to: /home/uzair/pig_1421436777476.log
2015-01-16 11:32:57,640 [main] INFO org.apache.pig.impl.util.Utils - Default bootstrap file /home/uzair/.pigbootup not found
2015-01-16 11:32:58,442 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-16 11:32:58,450 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2015-01-16 11:32:58,474 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: file:///
2015-01-16 11:32:58,972 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
grunt>
```

As you can see, Pig has started in local mode by connecting to the local file system. You can also observe the Grunt interface for executing Pig statements.

Step 2: You also have access to most of the file system commands in the Grunt shell so that you need not exit Grunt shell in order to access the file system operations such as listing directories, copying files etc. To try it, run `ls` command to list the directories.



```
uzair@ubuntu:~ 
2015-01-16 11:32:58,972 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
grunt> ls
file:/home/uzair/.gconf <dir>
file:/home/uzair/.Xauthority<r 1> 51
file:/home/uzair/.local <dir>
file:/home/uzair/.xsession-errors.old<r 1> 918
file:/home/uzair/.xsession-errors<r 1> 652
file:/home/uzair/.ICEauthority<r 1> 1272
file:/home/uzair/Documents <dir>
file:/home/uzair/Desktop <dir>
file:/home/uzair/.shutter <dir>
file:/home/uzair/.ssh <dir>
file:/home/uzair/.cache <dir>
file:/home/uzair/pig_1421436692499.log<r 1> 2539
file:/home/uzair/.bash_history<r 1> 1024
file:/home/uzair/.compiz <dir>
file:/home/uzair/.dmrc<r 1> 25
file:/home/uzair/.gnome2_private <dir>
file:/home/uzair/Videos <dir>
file:/home/uzair/.mozilla <dir>
file:/home/uzair/hadoop-2.6.0.tar.gz<r 1> 195257604
file:/home/uzair/.bashrc<r 1> 4065
file:/home/uzair/pig-0.14.0.tar.gz<r 1> 119496823
```

Step 3: To exit from the grunt shell press **ctrl + z** keys simultaneously or run the command **quit**.

```
file:/home/uzair/.Public <dir>
file:/home/uzair/.dbus <dir>
file:/home/uzair/Pictures <dir>
file:/home/uzair/.bashrc (~) - gedit_012.jpg<r 1> 68278
file:/home/uzair/.config <dir>
file:/home/uzair/.pig_history<r 1> 13
file:/home/uzair/Downloads <dir>
file:/home/uzair/Templates <dir>
file:/home/uzair/.thumbnails <dir>
file:/home/uzair/.bashrc~<r 1> 3989
grunt> quit
2015-01-16 11:41:23,116 [main] INFO org.apache.pig.Main - Pig script completed
in 8 minutes, 30 seconds and 438 milliseconds (510438 ms)
uzair@ubuntu:~$
```

Now that we have seen how to start Pig in local mode, let us look how to start Pig in Tez local mode.



Step 4: To start Pig in Tez Local Mode, run the following command from command line interface.

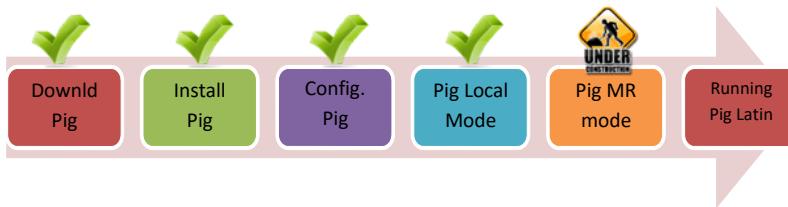
```
$ pig -x tez_local
```

```
uzair@ubuntu: ~
grunt> quit
2015-01-16 11:41:23,116 [main] INFO org.apache.pig.Main - Pig script completed
in 8 minutes, 30 seconds and 438 milliseconds (510438 ms)
uzair@ubuntu:~$ pig -x tez_local
15/01/16 11:45:20 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/01/16 11:45:20 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/01/16 11:45:20 INFO pig.ExecTypeProvider: Trying ExecType : TEZ_LOCAL
15/01/16 11:45:20 INFO pig.ExecTypeProvider: Picked TEZ_LOCAL as the ExecType
2015-01-16 11:45:20,416 [main] INFO org.apache.pig.Main - Apache Pig version 0.
14.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-01-16 11:45:20,417 [main] INFO org.apache.pig.Main - Logging error messages to: /home/uzair/pig_1421437520403.log
2015-01-16 11:45:20,564 [main] INFO org.apache.pig.impl.util.Utils - Default bootup file /home/uzair/.pigbootup not found
2015-01-16 11:45:21,359 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-16 11:45:21,362 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2015-01-16 11:45:21,386 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: file:///
2015-01-16 11:45:21,752 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
grunt>
```

When you start Pig in Tez Local mode, Pig will internally invoke the Tez runtime engine. Tez Local Mode is experimental and the queries may fail for large datasets.

Step 5: Exiting from grunt shell is similar to that of the instructions in step 3 of this task.

Task 4 is complete!



Task 5: Starting Pig in MapReduce Mode and Tez Mode

Starting Pig in MapReduce mode is very simple similar to that of local mode or Tez local mode and it just requires a single command to fire it up.

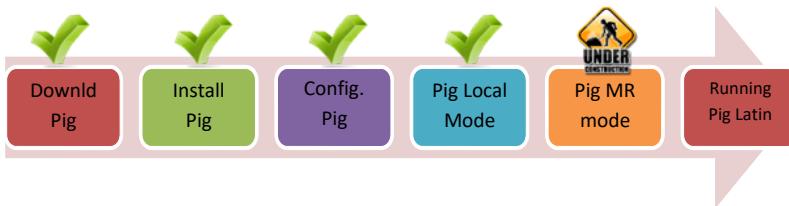
Step 1: Start all the Hadoop daemons before you start Pig in MapReduce mode. To do so, run the following commands from the terminal.

```
$ start-dfs.sh
$ start-yarn.sh
```

```
uzair@ubuntu:~$ start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/hadoop/logs/hadoop-uzair-nam
enode-ubuntu.out
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-uzair-dat
anode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-uz
air-secondarynamenode-ubuntu.out
uzair@ubuntu:~$ start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn-uzair-resourcen
anager-ubuntu.out
localhost: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-uzair-no
demanager-ubuntu.out
uzair@ubuntu:~$
```

Step 2: From the command line interface, run the following command to start Pig in MapReduce mode. Make sure all your Hadoop daemons are up and running before you run this command.

```
$ pig -x mapreduce
or
$ pig
```



You can use any of the commands above to start Pig in MapReduce mode and both do the same thing.

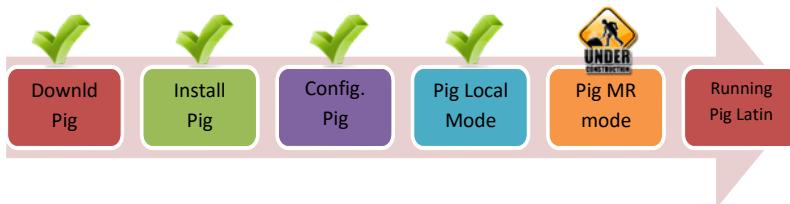
```
uzair@ubuntu:~$ pig -x mapreduce
15/01/16 12:05:28 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/01/16 12:05:28 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/01/16 12:05:28 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
2015-01-16 12:05:29,122 [main] INFO org.apache.pig.Main - Apache Pig version 0.
14.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-01-16 12:05:29,123 [main] INFO org.apache.pig.Main - Logging error messages to: /home/uzair/pig_1421438729114.log
2015-01-16 12:05:29,268 [main] INFO org.apache.pig.impl.util.Utils - Default bootstrap file /home/uzair/.pigbootup not found
2015-01-16 12:05:32,425 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2015-01-16 12:05:32,427 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-16 12:05:32,428 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:9000
2015-01-16 12:05:44,372 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> ■
```

As you can see, Pig has started in MapReduce mode by connecting to the Hadoop Distributed file system and MapReduce.

Step 3: Issue file system commands such as `ls` to explore the data inside **HDFS** as you have access to all the Hadoop shell commands from Grunt shell too.

```
uzair@ubuntu: ~
grunt> ls
hdfs://localhost:9000/user/uzair/pig_1421436692499.log<r 1> 2539
hdfs://localhost:9000/user/uzair/pig_1421438729114.log<r 1> 6047
grunt>
```

The `ls` command returns nothing, if you have not saved any files in HDFS.



Step 4: Exit from the grunt shell press ***ctrl + z*** keys simultaneously or run the command **quit**.

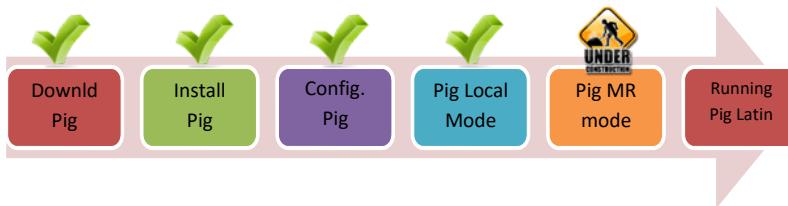
Now that we have seen how to start Pig in local mode, let us look how to start Pig in Tez local mode.

Step 5: To start Pig in Tez Mode, run the following command from command line interface. Make sure all the Hadoop daemons are up and running.

```
$ pig -x tez
```

```
uzair@ubuntu:~$ pig -x tez
15/01/16 12:40:02 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/01/16 12:40:02 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/01/16 12:40:02 INFO pig.ExecTypeProvider: Trying ExecType : TEZ_LOCAL
15/01/16 12:40:02 INFO pig.ExecTypeProvider: Trying ExecType : TEZ
15/01/16 12:40:02 INFO pig.ExecTypeProvider: Picked TEZ as the ExecType
2015-01-16 12:40:02,600 [main] INFO org.apache.pig.Main - Apache Pig version 0.
14.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-01-16 12:40:02,605 [main] INFO org.apache.pig.Main - Logging error messages to: /home/uzair/pig_1421440802599.log
2015-01-16 12:40:02,674 [main] INFO org.apache.pig.impl.util.Utils - Default bootstrap file /home/uzair/.pigbootup not found
2015-01-16 12:40:03,995 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2015-01-16 12:40:03,999 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-16 12:40:03,999 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:9000
2015-01-16 12:40:09,254 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Pig has now started in Tez mode. With Tez, you can experience increased performance for both interactive and batch queries.



Step 6: Exiting from grunt shell is similar to that of the instructions in step 4 of this task.

Task 5 is complete!

Task 6: Running Pig Latin Statements

Now that we have successfully installed and started Pig in various modes, let us write few basic Pig Latin statements before we conclude the lab. Let us begin with loading data in Pig. Do not worry about these Pig Latin statements as of now as we would learn about them in detail in the upcoming chapters.

Step 1: Download this input file (user_info.txt) from the URL below to your *Home* directory, which we will be using in this task to load into Pig.

<https://db.tt/CUBkdBhT>

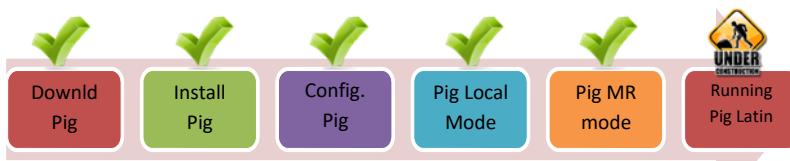
Step 2: Copy the file to HDFS by running the following command from Grunt shell

```
grunt> copyFromLocal user_info.txt .
```

```
uzair@ubuntu: ~
grunt> copyFromLocal user_info.txt .
grunt>
```

Step 3: Now that we have the input file in HDFS, let us load it into Pig. (Let's feed the Pig). Type the following Pig Latin statement to load the file.

```
grunt> users = LOAD 'user_info.txt' USING
PigStorage('|') AS (user_id:chararray, age:int,
gender:chararray, prof:chararray, zip:chararray);
```



```
grunt> users = LOAD 'user_info.txt' USING PigStorage('|') AS (user_id:chararray,
age:int, gender:chararray, prof:chararray, zip:chararray);
2015-01-16 14:22:26,660 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> ■
```

The above statement loads the file `user_info.txt` from HDFS into the relation 'users'. We use `PigStorage()` for specifying the delimiter which is '`|`' in this case. It is not necessary to specify if the fields are delimited by a tab character as `PigStorage()` default delimiter is a tab character. We also specify the schema for the data so that we can reference the fields by a name rather than the position.

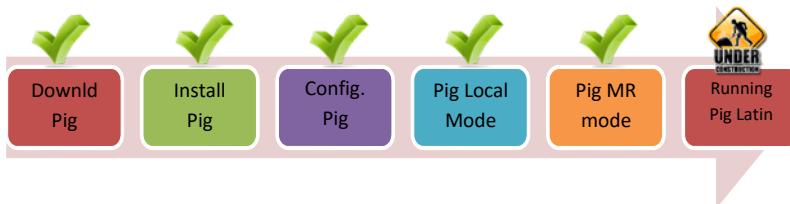
Step 4: We now have a 'users' relation and can be dumped to the screen to check out the result which should return you the entire file as is. You may dump the file to the screen or check the schema either by using the `DUMP` operator or `DESCRIBE` operator. It is recommended that you do not to dump the relation to screen as it is a big file with lots of rows, instead use `DESCRIBE` to check if the schema has worked by executing the following statement.

```
grunt> DESCRIBE users;
```

This returns the schema associated with the relation as shown in the screenshot below:

```
uzair@ubuntu:~
```

```
grunt> users = LOAD 'user_info.txt' USING PigStorage('|') AS (user_id:chararray,
age:int, gender:chararray, prof:chararray, zip:chararray);
2015-01-16 14:25:05,215 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE users;
users: {user_id: chararray, age: int, gender: chararray, prof: chararray, zip: chararray}
grunt>
```



Step 5: We can store this relation to a file using the following statement along with the schema. We can also change the delimiter from tab character to any other character. We shall use ',' (comma) as delimiter instead of the '|' (pipe) character.

```
grunt> STORE users INTO 'hdfs:/user/uzair/PigStore'
USING PigStorage(',', '-schema');
```

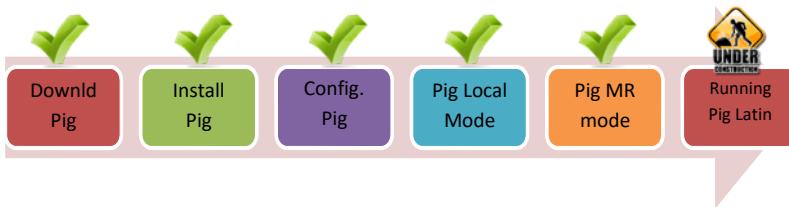
The above statement stores the relation 'users' into a file called 'PigStore' along with the schema. The fields are ',' (comma) separated instead of '|' (pipe) character. We save the schema using *-schema* because, if we were to load similar data from the same directory, we need not type the schema again as the schema gets attached automatically.

```
uzair@ubuntu:~
```

```
grunt> STORE users INTO 'hdfs:/user/uzair/PigStore' USING PigStorage(',', '-schema');
2015-01-16 13:56:34,623 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-16 13:56:34,840 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - mapred.textoutputformat.separator is deprecated. Instead, use mapreduce.output.textoutputformat.separator
2015-01-16 13:56:34,974 [main] INFO  org.apache.pig.tools.pigstats.ScriptState - Pig features used in the script: UNKNOWN
2015-01-16 13:56:35,119 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-16 13:56:35,157 [main] INFO  org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-01-16 13:56:35,252 [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]}
2015-01-16 13:56:35,621 [main] INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MRCompiler - File concatenation threshold: 100 optimistic? false
2015-01-16 13:56:35,752 [main] INFO  org.apache.pig.backend.hadoop.executionengine
```

Step 6: You can check if the file has been successfully stored by running the following `fs` command or navigating to the HDFS WebUI:

```
grunt> fs -cat PigStore/p*
```



```
uzair@ubuntu: ~
grunt> fs -cat PigStore/p*

922,29,F,administrator,21114
923,21,M,student,E2E3R
924,29,M,other,11753
925,18,F,salesman,49036
926,49,M,entertainment,01701
927,23,M,programmer,55428
928,21,M,student,55408
929,44,M,scientist,53711
930,28,F,scientist,07310
931,60,M,educator,33556
932,58,M,educator,06437
933,28,M,student,48105
934,61,M,engineer,22902
935,42,M,doctor,66221
936,24,M,other,32789
937,48,M,educator,98072
938,38,F,technician,55038
939,26,F,student,33319
940,32,M,administrator,02215
941,20,M,student,97229
942,48,F,librarian,78209
943,22,M,student,77841
grunt> █
```

Do not worry if these statements sound like some foreign language as we would look more in detail about Pig Latin in our upcoming chapters.

Task 6 is complete!

LAB CHALLENGE

Challenge

In the last task, we have loaded a very small file into Pig and dumped stored it by changing the delimiters. Now, search for a big dataset online and try loading and storing it in Pig. You can also use the DUMP statement instead of storing. Keep experimenting with many other datasets too.

SUMMARY

Apache Pig is open source top level Apache project and is available for free download. The Apache Pig package is available for download along with its source code from Apache website. Apache Pig can also be downloaded as a part of Hadoop distributions such as Cloudera and HortonWorks.

In this chapter we have straight away jumped into our labs to install Pig and seen how to start Pig in local and MapReduce and Tez modes. We have also run few Pig Latin statements to load and store data using pig.

REFERENCES

- <http://pig.apache.org/releases.html>
- <http://pig.apache.org/docs/r0.14.0/>

INDEX

Prerequisites to Install Apache Pig.....	37
AIM	39
Lab Exercise 2: INSTALLING APACHE PIG	40
Task 1: Downloading Apache Pig	41
Task 2: Installing Apache Pig	45
Task 3: Configuring Apache Pig	47
Task 4: Starting Pig in Local Mode& Tez Local Mode....	50
Task 5: Starting Pig in MapReduce Mode & Tez Mode .	53
Task 6: Running PigLatin Statements	56
LAB CHALLENGE.....	60
SUMMARY	61
REFERENCES.....	62

CHAPTER 3: WORKING ON GRUNT

Theory

In the previous chapter, we have successfully installed Apache Pig on our machines and have seen various modes to run Pig. A quick recap of various modes, which Pig can be started are Local Mode, Local Tez Mode (Experimental), MapReduce Mode and Tez mode. In this chapter we would dive deeper a step ahead into Pig and learn how to work with Pig's Grunt shell.

What is Grunt?

Grunt is Pig's interactive shell to run Pig commands and Pig scripts in any of the various modes mentioned above. Please refer to tasks 4 and 5 of Lab exercise 2 for starting Pig as desired in various modes. Grunt helps users to run Pig Latin statements interactively. With Grunt, you can design the dataflow of a job at initial development stage by interactively running Pig Latin statements and instantly dumping the result on to the screen. Grunt also helps users with what-if scenarios during the initial development stage, as it processes queries interactively and have the result displayed in no time.

Grunt Features

Grunt has almost all the similar features of a command line interface found in the standard UNIX shells but lacks few features such as background execution, pipes etc. Let us look at few important features of a Grunt shell.

- Grunt shell provides all the basic file system operations such as copy, move, and remove to name a few.
- Grunt shell makes available to reuse all the command line history and the same can be accessed using the arrow keys.
- Grunt shell supports editing the commands and also allows you to print content of files to the screen.
- Grunt shell has the tab completion feature to auto complete a command but not a file name. For example, if you type `copyF` and press tab, the command gets automatically completed to `copyFromLocal`. But if you type `copyFromLocal had` and press tab, the file name does not get auto completed to `copyFromLocal hadoop.log`. This is because of high latency in HDFS.

Pig Latin Statements in Grunt

You can start entering the Pig Latin statements in Grunt shell as soon as you have started Pig in one of the modes as shown in Task 6 of Lab Exercise 2. As mentioned earlier, Grunt is an ultimate way to interactively design a workflow pipeline in initial development stage.

You can write Pig Latin statements and visualize the dataflow by dumping the relation after each statement or dump/store the result after completing all the Pig Latin statements. For example, you can write a Pig Latin statement as shown below and dump the relation ‘users’ to check out the result or continue with next relation and dump/store at the end of Pig Latin script.

```
grunt> users = LOAD 'user_info.txt' USING  
PigStorage('|') AS (user_id:chararray, age:int,  
gender:chararray,prof:chararray,zip:chararray);  
  
grunt> DUMP users;  
...
```

Pig does not execute the Pig Latin statements unless you dump or store using the DUMP or STORE commands but it does check for syntax errors and informs you about the error. If you make an error while writing a new line of Pig Latin, you can always use the same relation and enter the line again. Pig will then take the correct last line you have entered and ignore the line which has error in it. For example,

```
grunt> users = LOAD 'user_info.txt' USING  
PigStorage('|') AS (user_id:chararray, age:int,  
gender:chararray,prof:chararray,zip:chararray);  
  
grunt> female_users = FILTER users BY geder = 'F';
```

The above line will lead to an error as we have misspelt the field *gender* as *geder*. You can then rewrite the line above with the same relation again by correctly specifying the field *gender* and Pig will accept it.

```
grunt> female_users = FILTER users BY gender = 'F';  
...
```

After you have completed entering the entire script, you can dump or store the result to screen or a file.

Writing Pig Latin commands in Grunt shell helps you get a clear visual of dataflow pipeline making your life a lot easier.

Hadoop commands in Grunt

While in Grunt shell, you can not only work with Pig Latin interactively but also use Hadoop Distributed File System (HDFS) commands too. You can perform all the basic file system commands from Grunt, which you use from the UNIX command line interface. We can access all the HDFS commands using `fs` keyword. For example,

```
grunt> fs -cat textfile.txt
```

Additionally, Pig has `sh` shell command which help you access the local shell commands from within a Pig script or Grunt shell. We can only run real programs from the `sh` command shell. Commands such as `cd` are not programs but part of the shell environment and should be explicitly invoked as "`bash cd`".

Other Pig Commands in Grunt

Apart from the HDFS commands, we also have some of the special Pig utility and diagnostic commands which help us with more power while running Pig. We shall look these commands in detail in the lab exercise.

AIM

The aim of the following lab exercise is to get familiar with HDFS commands, Pig utility and diagnostic operators in Grunt. We would also get familiar with running a Pig script from Command Line Interface.

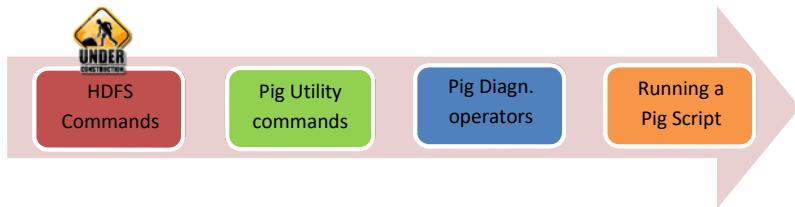
Following steps are required:

- Task 1: Using HDFS commands in Grunt
- Task 2: Using Pig Utility commands in Grunt
- Task 3: Using Pig Diagnostic operators in Grunt
- Task 4: Running a Pig Script

Lab Exercise 3: HANDS ON GRUNT



-
1. **Using HDFS commands in Grunt**
 2. **Using Pig Utility commands in Grunt**
 3. **Using Pig Diagnostic operators in Grunt**
 4. **Running a Pig Script**



Task 1: Using HDFS commands in Grunt

Step 1: Start all the Hadoop daemons as well as Pig in MapReduce mode.

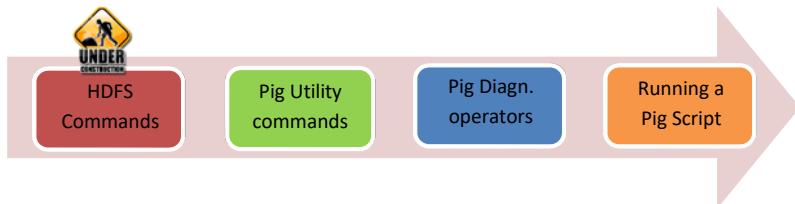
To start Hadoop daemons, run the following commands from Command Line Interface:

```
$ start-dfs.sh
$ start-yarn.sh
```

```
uzair@ubuntu:~$ start-dfs.sh
15/01/26 01:56:57 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/hadoop-2.6.0/logs/hadoop-uzair-namenode-ubuntu.out
localhost: starting datanode, logging to /usr/local/hadoop-2.6.0/logs/hadoop-uzair-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop-2.6.0/logs/hadoop-uzair-secondarynamenode-ubuntu.out
15/01/26 01:59:42 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
uzair@ubuntu:~$ start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop-2.6.0/logs/yarn-uzair-resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /usr/local/hadoop-2.6.0/logs/yarn-uzair-nodemanager-ubuntu.out
uzair@ubuntu:~$
```

To start Pig in MapReduce mode, run the following commands from Command Line Interface:

```
$ pig -x mapreduce
Or
$ pig
```



```
uzair@ubuntu:~$ pig -x mapreduce
15/01/26 02:06:49 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/01/26 02:06:49 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/01/26 02:06:49 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
2015-01-26 02:06:49,748 [main] INFO org.apache.pig.Main - Apache Pig version 0.1
4.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-01-26 02:06:49,748 [main] INFO org.apache.pig.Main - Logging error messages
to: /home/uzair/pig_1422218209746.log
2015-01-26 02:06:49,899 [main] INFO org.apache.pig.impl.util.Utils - Default bo
tup file /home/uzair/.pigbootup not found
2015-01-26 02:06:52,159 [main] INFO org.apache.hadoop.conf.Configuration.depreca
tion - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.addres
s
2015-01-26 02:06:52,162 [main] INFO org.apache.hadoop.conf.Configuration.depreca
tion - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-26 02:06:52,163 [main] INFO org.apache.pig.backend.hadoop.executionengin
e.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:9000
2015-01-26 02:07:17,228 [main] WARN org.apache.hadoop.util.NativeCodeLoader - Un
able to load native-hadoop library for your platform... using builtin-java classe
s where applicable
2015-01-26 02:07:19,517 [main] INFO org.apache.hadoop.conf.Configuration.depreca
tion - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 2: Let us start working with HDFS commands in Grunt by creating a directory in HDFS. The following command is used to create a directory in HDFS from Grunt.

```
grunt> mkdir books
```

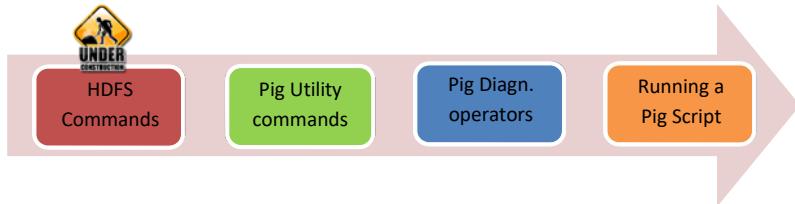
The syntax for the above command is:

```
mkdir <Directory name>
```

Confirm that the directory which we have just created has actually been created by running the following command.

```
grunt> ls
```

```
grunt> mkdir books
grunt> ls
hdfs://localhost:9000/user/uzair/books  <dir>
grunt>
```



The syntax for the above command is:

```
ls <Directory name>
or
ls
```

The `ls` command lists all the contents in the HDFS home directory. In our case we only have single directory *books* which we have just created. This also confirms that the directory has been successfully created. If `<Directory name>` is specified, the command lists the content of the specified directory. Otherwise, the content of the current working directory is listed.

Step 3: Now that we have a directory created in HDFS, let us copy a file from our local file system to HDFS. We shall be copying a book from local filesystem's home directory to the *books* directory in HDFS.

Download the book *Ancient and Modern Physics by Thomas E. Willson* to your home directory in *plain text UTF-8 format* from the link below:

```
http://goo.gl/XBolgP
```

Run the following command from Grunt to copy a file from local file system to HDFS.

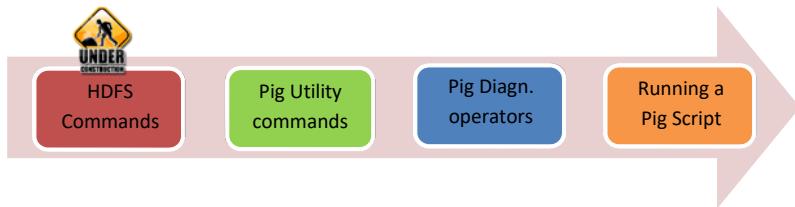
```
grunt> copyFromLocal pg10773.txt books/
```

The syntax for the above command is:

```
copyFromLocal <local source path> <HDFS destination path>
```

You can confirm that the copy operation was successful by listing the contents of the directory '*books*'.

```
grunt> ls books
```



```
grunt> copyFromLocal pg10773.txt books/
grunt> ls books
hdfs://localhost:9000/user/uzair/books/pg10773.txt<r 1> 150090
grunt>
```

Similarly, you can also copy files from HDFS to local file system. Let us copy the same file from HDFS to *Documents* directory in local file system. Run the following command from Grunt to copy a file from local file system to HDFS.

```
grunt> copyToLocal books/pg10773.txt Documents/
```

The syntax for the above command is:

```
copyToLocal <HDFS source path> <local destination path>
```

You can confirm that the copy operation was successful by checking the *Documents* directory in local file system.

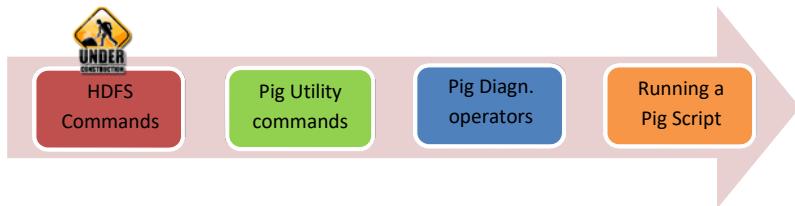
Step 4: To copy a file or directory within HDFS, run the following command from Grunt shell. Before you do that, create a new directory *books_new* in HDFS home directory, so that we can copy the file to the new directory.

```
grunt> cp books/pg10773.txt books_new/pg10773.txt
```

The syntax for the above command is:

```
cp <source path> <destination path>
```

```
uzair@ubuntu: ~
grunt> mkdir books_new
grunt> cp books/pg10773.txt books_new/pg10773.txt
grunt> ls books_new
hdfs://localhost:9000/user/uzair/books_new/pg10773.txt<r 1> 150090
grunt> ■
```



Similarly, we can also move files or directories within HDFS. The difference between move and copy is that the file/directory is removed from source as soon as it is copied to destination. Whereas, copying a file/directory does not remove it from source. The syntax for moving a file within HDFS is as follows:

```
mv <source path> <destination path>
```

Step 5: Let us now check the contents of the file we have copied to HDFS using the following command.

```
grunt> cat books/pg10773.txt
```

The syntax for the above command is:

```
cat <HDFS path to file>
```

You can also supply multiple paths to print the contents on screen. All you need to do is specify multiple paths separated by a *space* character as shown below.

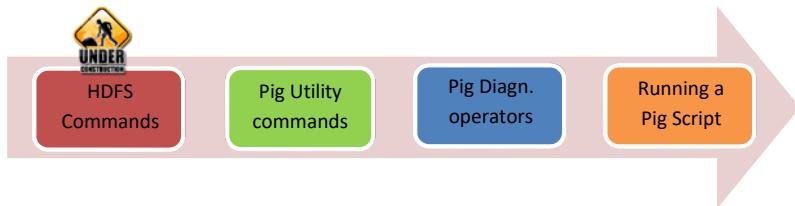
```
cat <path1> <path2> ...
```

The multiple files would be concatenated and printed out to the screen. You can also provide a path to a directory to recursively concatenate and print all the files in the directory to screen.

Step 6: Removing a file or a directory is as simple as copying or moving. Run the following command to remove a file or directory in HDFS from Grunt.

```
grunt> rm books_new/pg10773.txt
```

```
grunt> rm books_new/pg10773.txt
2015-01-26 03:54:39,501 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-26 03:54:39,774 [main] INFO  org.apache.pig.tools.grunt.GruntParser - Waited 0ms to delete file
grunt> ls books_new
Nothing is listed as the file is deleted
```



The syntax for the above command is:

```
rm <HDFS path to file>
```

You can also supply multiple paths to delete. All you need to do is specify multiple paths separated by a *space* character as shown below.

```
rm <path1> <path2> ...
```

Please note that the above command removes data permanently and recursively. It also does not prompt for confirmation.

Step 7: Finally, let us look how to know the current working directory and change the current working directory in HDFS from Grunt.

To know the current working directory, run the following command:

```
grunt> pwd
```

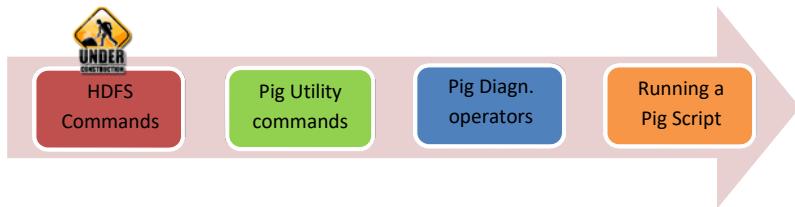
To change the current working directory to *books* directory, run the following command:

```
grunt> cd books
```

The syntax for the above command is:

```
cd <directory>
```

```
grunt> pwd
hdfs://localhost:9000/user/uzair
grunt> cd books
grunt> pwd
hdfs://localhost:9000/user/uzair/books
grunt>
```



If you would like to go back to home directory, run the command `cd` without passing any parameters.

```
grunt> cd
```

```
grunt> pwd
hdfs://localhost:9000/user/uzair
grunt> cd books
grunt> pwd
hdfs://localhost:9000/user/uzair/books
grunt> cd
grunt> pwd
hdfs://localhost:9000/user/uzair
grunt>
```

Task 1 is complete!

Task 2: Using Pig Utility Commands in Grunt

Command 1: `exec`

`exec` command is used to execute the Pig Latin script with no interaction between the script and the Grunt shell. Aliases/relations defines in the Pig script are not stored in Grunt but the output is stored and can be retrieved after the script is executed. This command is useful to test Pig Latin scripts inside from a Grunt shell. When you use `exec` command, the `store/dump` commands does not execute the script, instead, the entire script is parsed before execution starts. Let us execute a Pig script inside from a Grunt shell.

Step 1: Download the Pig script `wordcount.pig` from the URL below and save it to your home directory.

<https://db.tt/1mmPPqBB>



This Pig script counts the total number of occurrences of each word for a given input file. The input file here is the file which we have copied to HDFS in the previous task.

Step 2: Copy the `wordcount.pig` script to HDFS' home directory using the following command. Do not worry about the Pig Latin commands in the Pig script. We shall look at them in our upcoming chapters.

```
grunt> copyFromLocal wordcount.pig .
```

Observe the '.' in the place of destination. The '.' indicates current working directory of HDFS which is home directory in this case.

```
uzair@ubuntu: ~
grunt> copyFromLocal wordcount.pig .
grunt> ls
hdfs://localhost:9000/user/uzair/books <dir>
hdfs://localhost:9000/user/uzair/wordcount.pig<r 1> 365
grunt>
```

Step 3: Run the command below to execute the Pig Latin script.

```
grunt> exec wordcount.pig
```

```
grunt> exec wordcount.pig
2015-01-27 00:46:20,213 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-27 00:46:20,969 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-27 00:46:26,862 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-27 00:46:29,700 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-27 00:46:30,002 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.textoutputformat.separator is deprecated. Instead, use mapreduce.output.textoutputformat.separator
2015-01-27 00:46:30,332 [main] INFO org.apache.pig.tools.pigstats.ScriptState - Pig features used in the script: GROUP_BY,ORDER_BY
2015-01-27 00:46:30,909 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-01-27 00:46:31,120 [main] INFO org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
```



Note: If you get a strange error or the job does not finish, exit from the Grunt shell and run the following command to start the job history server.

```
$ mr-jobhistory-daemon.sh start historyserver
```

The exec command successfully executes with a *Success* message at the end similar to the screenshot below.

```
uzair@ubuntu: ~
3      13      13      wordsSorted      SAMPLER
job_1422903735037_0004 1      1      15      15      15      15      17      1
7      17      17      wordsSorted      ORDER_BY      hdfs:/user/uzair/wc_out,
Input(s):
Successfully read 2909 records (150467 bytes) from: "hdfs:/user/uzair/books/pg10
773.txt"

Output(s):
Successfully stored 4083 records (41293 bytes) in: "hdfs:/user/uzair/wc_out"

Counters:
Total records written : 4083
Total bytes written : 41293
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG:
job_1422903735037_0002 ->      job_1422903735037_0003,
job_1422903735037_0003 ->      job_1422903735037_0004,
job_1422903735037_0004
```

Step 4: Check the output using the following command. The output is stored in the `wc_out` directory.

```
grunt> cat wc_out/part-r-00000
```

Command 2: run

Run command is used to run a Pig script that can interact with the Grunt shell. The script can access the aliases defined from the Grunt shell and also the Grunt shell



has access to aliases within the script. Moreover, all the commands are remembered within the command line history. The *run* command is triggered for execution by the *STORE* operator. Running the *run* command from the Grunt shell is similar to that of typing Pig Latin statements manually.

Let us look how the *run* command actually works with an example.

Step 1: Download the Pig script *wordcount2.pig* from the URL below and save it to your home directory. This is the same Pig script which we have executed using the *exec* command, instead we remove the last couple of lines and enter them from the Grunt shell manually.

<https://db.tt/orG8jMzT>

Step 2: Copy the *wordcount2.pig* script to HDFS' home directory using the following command.

grunt> copyFromLocal wordcount2.pig .

Observe the '.' in the place of destination. The '.' indicates current working directory of HDFS which is home directory in this case.

Step 3: Run the command below to execute the Pig Latin script.

grunt> run wordcount2.pig

```
grunt> run wordcount2.pig
2015-02-02 11:50:01,034 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (lines:chararray);
2015-02-02 11:50:01,230 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> words = FOREACH file GENERATE FLATTEN(TOKENIZE(lines)) as word;
grunt> wordsGrouped = GROUP words BY word;
grunt> wordsAggregated = FOREACH wordsGrouped GENERATE group as word, COUNT(words);
grunt>
```



As you can see all the statements from the Pig script are available. You can now manually type in the rest of the statements and dump or store the result.

Step 4: Enter the following commands from the Grunt shell to complete the script and start the job.

```
grunt> wordsSorted = ORDER wordsAggregated BY $1
DESC;
grunt> DUMP wordsSorted;
```

Both the commands *exec* and *run* are extremely useful for debugging and help you save a lot of time by facilitating you to modify the Pig script right from the Grunt shell.

Command 3: kill

There might be scenarios where you would want to abort the running Pig jobs. You can achieve this using the *kill* command. All you need to enter the command *kill* followed by the job id of the job you want to kill. For example, if you want to kill a Pig job with id job_2015030200001, you would run the following command.

```
grunt> kill job_2015030200001
```

Command 4: set

Using *set* command, you can set values to certain keys in Pig. Let us look in detail the keys and the values which can be set using the *set* command.

Keys	Values	Description
job.name	Name of the job	Sets the name of the job from Grunt.
job.priority	very_low, low, normal, high, very_high	Set the priority of the job. All values are case sensitive.
debug	On/off	Turn the debugging on/off.



default_parallel	A whole number	Sets the number of reducers for all MapReduce jobs in Pig.
stream.skippath	Path	Sets the path from which data is not to be shipped for streaming.

To set 5 reducers default for a Pig job, you can issue the following statement in the Pig script:

```
grunt> SET default_parallel 5
```

To set debugging on, you can issue the following statement in the Pig script:

```
grunt> SET debug 'on'
```

Command 5: clear

The command *clear*, clears the screen in Grunt shell. You just need to run the command *clear* with no strings attached.

```
grunt> clear
```

The semi-colon (;) is not required at the end.

Command 6: history

Using the command *history*, you can display the list of Pig Latin statements used.

```
grunt> history
```

The semi-colon (;) is not required at the end.



```
grunt> history
1  file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (line
s:chararray);
2  words = FOREACH file GENERATE FLATTEN(TOKENIZE(lines)) as word;
3  wordsGrouped = GROUP words BY word;
4  wordsAggregated = FOREACH wordsGrouped GENERATE group as word, COUNT(words);
5  file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (line
s:chararray);
6  words = FOREACH file GENERATE FLATTEN(TOKENIZE(lines)) as word;
7  wordsGrouped = GROUP words BY word;
8  wordsAggregated = FOREACH wordsGrouped GENERATE group as word, COUNT(words);
9  file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (line
s:chararray);
10  words = FOREACH file GENERATE FLATTEN(TOKENIZE(lines)) as word;
```

You can also omit the line number of the statement to display using `-n` switch as shown below.

```
grunt> history -n
```

```
grunt> history -n
file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (lines:ch
ararray);
words = FOREACH file GENERATE FLATTEN(TOKENIZE(lines)) as word;
wordsGrouped = GROUP words BY word;
wordsAggregated = FOREACH wordsGrouped GENERATE group as word, COUNT(words);
file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (lines:ch
ararray);
words = FOREACH file GENERATE FLATTEN(TOKENIZE(lines)) as word;
wordsGrouped = GROUP words BY word;
wordsAggregated = FOREACH wordsGrouped GENERATE group as word, COUNT(words);
file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (lines:ch
ararray);
```

Command 7: `help`

The command `help` lists all the commands you can use from the Grunt shell. The syntax for the `help` command is as followd:

```
grunt> help
```

The semi-colon (`;`) is not required at the end.



```
uzair@ubuntu: ~
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.
```

Command 7: quit

Finally, use the command *quit* to exit from the Grunt shell.

grunt> quit

The semi-colon (;) is not required at the end.

Task 2 is complete!

Task 3: Using Pig Diagnostic Operators in Grunt

Step 1: Go back to the Grunt shell by typing *pig* from the command line interface.



Step 2: We can check out the data flow of a Pig script or relation by using the **ILLUSTRATE** operator. All you need to do is specify an alias/relation or a path to the script.

We shall check out the data flow of the wordcount script as an example using the **ILLUSTRATE** operator. You can also check the data flow of an alias/relation too. Run the following command below:

```
grunt> ILLUSTRATE -script wordcount.pig
```

```
grunt> ILLUSTRATE -script wordcount.pig
2015-02-02 13:20:28,182 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-02 13:20:29,533 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-02 13:20:29,894 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.textoutputformat.separator is deprecated. Instead, use mapreduce.output.textoutputformat.separator
2015-02-02 13:20:30,138 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-02 13:20:30,142 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:9000
2015-02-02 13:20:30,297 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
```

If you have already run this script in the previous task, make sure you delete or modify the output directory path in STORE statement. The job fails if the output directory already exists.

After the MapReduce job, you can check the data flow illustration for the given script. This is very much useful when we have very large datasets to check if your script is doing what it should do. It is recommended to use **ILLUSTRATE** when dealing with very large datasets instead of dumping the relation to the screen.

The screenshot of ILLUSTRATE result is not shown here as it does not fit in the space here.

Step 3: Similar to the **ILLUSTRATE** operator, we can also check the logical, physical and MapReduce execution plan of a Pig script or relation by using the **EXPLAIN** operator. All you need to do is specify a relation or a path to the script.



We shall check out the execution plan of the wordcount script as an example using the EXPLAIN operator. The usage is similar to that of ILLUSTRATE operator where we specify the -script switch and specify the path to script as shown below.

```
grunt> EXPLAIN -script wordcount.pig
```

```
uzair@ubuntu: ~
grunt> EXPLAIN -script wordcount.pig
2015-02-02 14:03:33,037 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-02 14:03:35,015 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-02 14:03:36,525 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-02 14:03:36,961 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-02 14:03:36,984 [main] INFO org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-02-02 14:03:37,100 [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:
#-----
wordsSorted: (Name: LOStore Schema: word#109:chararray,#112:long)
|
|---wordsSorted: (Name: LOSort Schema: word#109:chararray,#112:long)
|   |
|   (Name: Project Type: long Uid: 112 Input: 0 Column: 1)
```

Execution plans provides you with the information regarding the bottlenecks while processing.

Step 4: We can also use DESCRIBE command to check out or revive the schema of a particular alias/relation. Run the following commands. Make sure you use the input path for the text file and not the exact below path.

```
grunt> file = LOAD
'hdfs:/user/uzair/books/pg10773.txt' USING
PigStorage() AS (lines:chararray);

grunt> DESCRIBE file
```



```

uzair@ubuntu: ~
grunt> file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (lines:chararray);
2015-02-02 14:09:33,644 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE file
file: {lines: chararray}
grunt>

```

Task 3 is complete!

Task 4: Running a Pig Script

Before we conclude this chapter, let us run a Pig script which counts the total number of words in a file.

Step 1: Open the Pig script *wordcount.pig* which you have downloaded in the previous tasks in this lab exercise so that we can have a walkthrough the code.

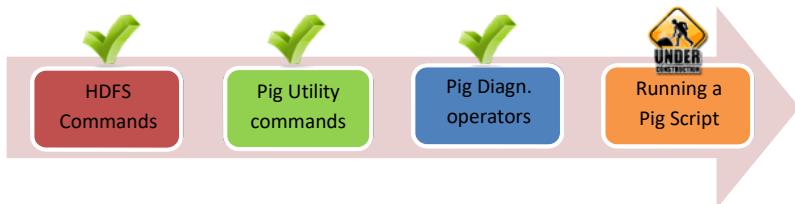
The Pig Script has the statements as shown below.

```

file = LOAD 'hdfs:/user/uzair/books/pg10773.txt' USING PigStorage() AS (lines:chararray);
words = FOREACH file GENERATE FLATTEN(TOKENIZE(lines)) as word;
wordsGrouped = GROUP words BY word;
wordsAggregated = FOREACH wordsGrouped GENERATE group as word, COUNT(words);
wordsSorted = ORDER wordsAggregated BY $1 DESC;
STORE wordsSorted INTO 'hdfs:/user/uzair/wc_out';

```

The first statement loads the contents of file in the file relation. The second statement turns all the lines into words by tokenizing and flattening (FLATTEN removes a level of nesting from bags). The next statement groups all the words and then aggregates the words by counting the words. All the words are then sorted and stored in a directory *wc_out*. Make sure this directory does not exist.

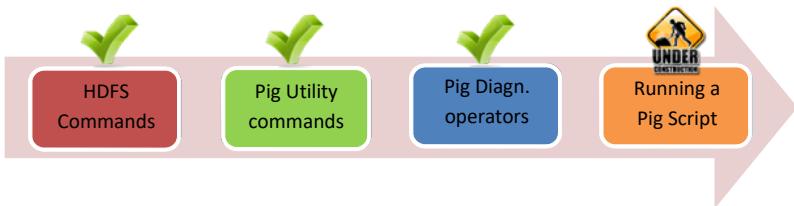


Step 2: Quit from the Grunt shell and run the following command from Terminal to execute the script:

```
$ pig wordcount.pig
```

```
uzair@ubuntu:~$ pig wordcount.pig
15/02/02 14:22:01 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/02/02 14:22:01 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/02/02 14:22:01 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
2015-02-02 14:22:01,332 [main] INFO org.apache.pig.Main - Apache Pig version 0.
14.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-02-02 14:22:01,333 [main] INFO org.apache.pig.Main - Logging error messages to: /home/uzair/pig_1422915721328.log
2015-02-02 14:22:04,061 [main] WARN org.apache.hadoop.util.NativeCodeLoader - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2015-02-02 14:22:04,915 [main] INFO org.apache.pig.impl.util.Utils - Default bootstrap file /home/uzair/.pigbootup not found
2015-02-02 14:22:05,496 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2015-02-02 14:22:05,498 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-02 14:22:05,499 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:9000
```

The job gets submitted to the cluster and produces the output by executing the Pig script. You can see the success message as shown below, after the job has been successfully completed.



```

2015-02-02 14:27:56,274 [main] INFO  org.apache.hadoop.mapred.ClientServiceDeleg
ate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirect
ing to job history server
2015-02-02 14:27:56,516 [main] INFO  org.apache.hadoop.yarn.client.RMProxy - Con
necting to ResourceManager at /0.0.0.0:8032
2015-02-02 14:27:56,560 [main] INFO  org.apache.hadoop.mapred.ClientServiceDeleg
ate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirect
ing to job history server
2015-02-02 14:27:56,768 [main] INFO  org.apache.hadoop.yarn.client.RMProxy - Con
necting to ResourceManager at /0.0.0.0:8032
2015-02-02 14:27:56,813 [main] INFO  org.apache.hadoop.mapred.ClientServiceDeleg
ate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirect
ing to job history server
2015-02-02 14:27:57,002 [main] INFO  org.apache.pig.backend.hadoop.executionengi
ne.mapReduceLayer.MapReduceLauncher - Success!
2015-02-02 14:27:57,158 [main] INFO  org.apache.pig.Main - Pig script completed
in 3 minutes, 27 seconds and 844 milliseconds (207844 ms)
uzair@ubuntu:~$
```

Step 3: Check the output of the job by running the following command:

```
$ hadoop fs -cat wc_out/p*
```

Task 4 is complete!

LAB CHALLENGE

Challenge

In the last task, we have ran a pig script and have seen the output. Similarly, run the same script on other text files and check the output. Download any text file from the internet, copy it to HDFS, change the input and output path to the Pig script and run. Also practice the HDFS, utility and diagnostic commands and get familiar with them.

SUMMARY

Grunt is Pig's interactive shell to run Pig commands and Pig scripts. Grunt helps users to run Pig Latin statements interactively. With Grunt, you can design the dataflow of a job at initial development stage by interactively running Pig Latin statements and instantly dumping the result on to the screen. Grunt also helps users with what-if scenarios during the initial development stage, as it processes queries interactively and have the result displayed in no time.

Not only can we write Pig Latin statements from Grunt shell but also access HDFS and sh shell operations.

REFERENCES

- <https://wiki.apache.org/pig/Grunt>
- <http://pig.apache.org/docs/r0.14.0/cmds.html>
- <http://pig.apache.org/docs/r0.14.0/test.html>

INDEX

What is Grunt?.....	64
Grunt Features.....	64
Pig Latin Statements in Grunt	65
Hadoop commands in Grunt.....	66
Other Pig commands in Grunt	66
AIM	67
Lab Exercise 3: Hands On Grunt.....	68
Task 1: Using HDFS Commands in Grunt.....	69
Task 2: Using Pig Utility Commands in Grunt.....	75
Task 3: Using Pig Diagnostic Operators in Grunt.....	82
Task 4: Running a Pig Script	85
LAB CHALLENGE.....	88
SUMMARY	89
REFERENCES.....	90

CHAPTER 4: PIG LATIN DATA TYPES

Theory

In the previous chapter, we have learnt about the Grunt shell and the various commands we can use from Grunt. In this chapter, we move a step ahead and learn the Pig Latin data types which are used to effectively write Pig Latin scripts.

Why does Pig Latin require data types? Similar to any other programming language, Pig Latin also uses data types to describe data to Pig. The data types can be assigned while describing schema to the dataset which is being loaded into Pig. Pig is very good at handling any kind of data ranging from plain text data to nested hierarchical data structures. Let us now look at the Pig Latin data types in detail.

Data Types

The Pig Latin data types can be classified into two types.

Simple Types	Simple Types are the data types which are available in most of the programming languages. Most of the simple types here can be found in <code>java.lang</code> classes except <code>bytearray</code> . Due to similarity of Pig Latin data types with Java, we can easily code UDF's with less effort. (UDF's are the User Defined Functions, which we will be covering in the upcoming chapters.)
Complex Types	Complex Types, as the name implies are used to assign complex data types. The data types which do not fit within Simple Types can be categorized as the complex types. An example of complex data types is a 'map'.

Simple Types

int: The `int` data type is 32-bit (4-byte) signed integer which is used to express constant integer numbers such as 25, 143, 1150 etc. `int` is represented in Pig interface by `java.lang.Integer`.

long: The `long` data type is 64-bit (8-byte) signed integer which is used to express constant long numbers such as 2500000L, 1430000L, 11500000L etc. `long` is represented in Pig interface by `java.lang.Long`. Please note that you may have

to append the letter 'L' (upper or lower case) at the end as it denotes `long` data type.

float: The `float` data type is 32-bit (4-byte) floating point which is used to express constant floating-point numbers such as `25.25F`, `14.3F`, `115.256e25F` (exponent) etc. `float` is represented in Pig interface by `java.lang.Float`. In Floating-point number calculations, the result might lose its precision. It is advised to use `int` or `long`, if you need accurate results. Please note that you may have to append the letter 'F' (upper or lower case) at the end as it denotes `float` data type.

double: The `double` data type is 64-bit (8-byte) floating point which is used to express constant floating-point numbers such as `25.253659878`, `14.3`, `115.256e-25` (exponent) etc. `double` is represented in Pig interface by `java.lang.Double`. In Floating-point number calculations, the result might lose its precision. It is advised to use `int` or `long`, if you need accurate results.

chararray: The `chararray` data type is a String or character array in Unicode UTF-8 format which is used to express constant chararrays as String literals inside single quotes such as '`hello, world`', '`consultants network`', '`user123`' etc. `chararray` is represented in Pig interface by `java.lang.String`. Moreover, if you would like to use certain characters such as tab space or new line, you will need to escape it with a backslash. For example, to express a tab space and new line as `chararray` you will have to use `\t` and `\n` respectively. You can also express Unicode characters as `\u` followed by the four-digit hexadecimal Unicode value.

bytearray: The `bytearray` data type is a Byte array (blob) which wraps a `java byte[]`. `bytearray` is represented in Pig interface by `DataByteArray`. It is not possible to specify a constant `bytearray`.

boolean: The `boolean` data type is used to express a `true` or `false` value. `boolean` is represented in Pig interface by `java.lang.Boolean`. Please note that the value `true` or `false` is not case sensitive.

datetime: The `datetime` data type used to express a timestamp. The timestamps have to be represented in ISO 8601 format. An example of such timestamp is `1996-11-05T08:15:30+05:30`.

biginteger: The `biginteger` data type is used to express constant big integer numbers such as `600000000000`. `biginteger` can handle very large integers and the size of the integer is only limited by the available memory of the JVM. `biginteger` is represented in Pig interface by `java.math.BigInteger`.

bigdecimal: The `bigdecimal` data type is used to express constant big floating-point numbers such as 786.8516549861563455556. `bigdecimal` provides operations for arithmetic, scale manipulation, rounding, comparison, hashing, and format conversion. `bigdecimal` is represented in Pig interface by `java.math.BigDecimal`.

Complex Types

Tuple: A `tuple` is a collection of ordered list of fields. A tuple can be further divided into `fields`. A `field` is a nothing but a piece of data element of any data type. While we write a Pig script, we begin loading that data from source as tuples and transform them according to the requirement. Since, tuples are ordered list of fields, we can refer them using their positional notation. When the data is loaded as tuples, any error or absence of data makes the value of the field as `NULL`. A `NULL` value in Pig means the value for that field is unknown. Tuples are represented using parentheses and commas to define the delimiters for the fields.

Let us consider an example to understand the concept of tuples. For the sake of example, consider a dataset 'Employee' for which we have a tuple as shown below.

```
('Ernesto', 35, 40000)
```

Here, in the tuple, we have three fields: name, age and salary. The collection of all these fields is known as 'Tuple'.

If any of the field was missing or has an error in a record of the data source, that field is said to be `NULL`. For the example above, if the salary field is missing for a particular record in data source, we have the tuple as shown below.

```
('Ernesto', 35, NULL)
```

Also, you can access the fields in a tuple using its position if schema is not defined while loading. You will have to use a `$` sign along with the number of the position. In Pig Latin, the position counting starts from 0. In the above example, if you want extract age of all the employees, you have to write a Pig Latin statement as shown below.

```
Employees = LOAD 'employee_data';
Age = FOREACH Employees GENERATE $1;
```

`$1` represents the second field in the tuple which returns the age of all the employees. Do not worry about the operators in the above relation as of now. We shall cover them in our upcoming chapters.

Bag: A bag is an unordered collection of tuples as it has no order of any kind and is impossible to refer the fields using positional notation as we could in tuple. If you are from SQL background, you can draw a parallel between bag and tuple as table of a database and rows in tables respectively. There are a lot of differences though, which you can understand as get deeper. Bags are represented using curly braces, with tuples inside the bag separated by a comma.

Below is an example of a bag with three tuples with three fields each.

```
{('Ernesto', 35, 40000), ('Jason', 30, 32000),  
 ('Dan', 36, 36000)}
```

A bag can have:

- Duplicate tuples.
- Tuples with different number of fields. It is not required for a bag to have same number of fields for each tuple.
- Tuple with different data type for each field. However, for effective processing of bags, it is required to have same schemas for all the tuples within the bag. To understand this better, consider a bag with half of the tuples containing int fields while other half of the tuple containing chararray fields. In this case, only half of the fields will be able to compute as the chararray fields get converted to null values.

Bags can be of two forms.

- Outer Bag
- Inner Bag

To understand this better, let us consider an example.

```
grunt> employees = LOAD 'employee_data' AS (name:  
chararray, age: int, salary: int);  
grunt> DUMP employees;  
('Ernesto', 35, 40000)  
('Jason', 30, 40000)  
('Dan', 36, 36000)
```

In the example above, the relation 'employees' is a bag of tuples or a relation. The relation can be known as outer bag. This might be confusing but you can understand this once you know what an inner bag is.

Now, to understand what inner bag is, let us continue with the example above. Let us group the relation 'employees' with 'salary' and form a new relation, say 'grouped'.

```
grunt> grouped = GROUP employees BY salary;
grunt> DUMP grouped;
(40000, {'Ernesto', 35, 40000}, ('Jason', 30,
40000))
(36000, {'Dan', 36, 36000})
```

In the example above, the relation 'grouped' is a bag of tuples or a relation. The relation 'grouped' has two types of fields: int and a bag. This can be considered as Inner bag. Therefore, an inner bag can be defined as a bag that is field within some complex type in a relation.

Map : A map is a set of key/value pairs. The key must be unique and must be of type `chararray` while the value can be of any type (`bytearray` by default). Moreover, all the values in a map may not be of same type. A value in one key-value pair might be `chararray` while the other value in other key-value pair can be `int`. The key and value pairs are separated by pound (#) symbol. The key value pair can be represented by `[key#value]` in pig.

As always, let us consider an example to understand the `map` type. For example, we have the following data in the file 'emp.txt'.

```
[Ernesto#36000, Dan#32000]
[Jason#45000, Zack#25000]
```

The Pig Latin statements to load the above data sample as map is as follows:

```
grunt> employee = LOAD 'emp.txt' AS (emp:map[int]);
grunt> salary = FOREACH employee GENERATE
emp#'Ernesto' AS sal;
grunt> DUMP salary;
36000
```

The load statement will construct two maps having two key/value pairs each. Notice that we specify the data type of values as 'int' in load statement. We can choose not to specify the type of values as below:

```
grunt> employee = LOAD 'emp.txt' AS (emp:map[]);
```

When we do so, Pig by default assigns to `bytearray` and guesses the appropriate type depending on how the Pig script handles the data. This handling occurs at runtime.

The second statement retrieves the value associated with 'Ernesto' and will return the value 36000 when dumped to the screen.

Pig Latin Nulls

In the examples above, we have come across NULL values while discussing about the data types. Null is something used to represent the data being missing or unknown. You get greeted by nulls in your data source itself or as a result of an operation during runtime. Do not confuse with the concept of null in low level programming languages such as C, Java, Python etc as the concept of null in Pig is that the value is unknown due to the data being missing or an error while processing. The value of null is simply unknown. The concept of null in Pig is similar to that of the concept of nulls in SQL.

Operations That Produce Nulls

As discussed above, nulls can be the result of an operation. These operations can produce null values:

- Division of a null by zero.
- Returns from user defined functions (UDFs).
- Dereferencing a field that does not exist.
- Dereferencing a key that does not exist in a map. For example, given a map, info, containing [name#Ernesto, salary#45000] if a user tries to use age#phone a null is returned.
- Accessing a field that does not exist in a tuple.

Using Nulls as Constants

Nulls can act as a placeholder for constant expressions in place of expressions of any type. For example, consider the following statements where we load a file to Pig and then generate a field as null.

```
grunt> employees = LOAD 'employee_data' AS (name: chararray, age: int, salary: int);
grunt> nulls = FOREACH employees GENERATE name, null;
```

```
grunt> DUMP nulls;
Ernesto, NULL
Jason, NULL
Zack, NULL
```

Pig Latin Schemas

Schemas allow you to assign names and data types to fields. It is not mandatory that you use schemas but is recommended to assign whenever possible as it provides you with more power to refer the fields. Schemas are optional as Pig automatically guesses the best possible type for the fields. The schema is actually enforced during the runtime by casting the input data to the expected data type. If the process is successful the results are returned to the user; otherwise, a warning is generated for each record that failed to convert. Pig does not know the actual types of the fields in the input data prior to the execution. Pig guesses the best possible data types for the fields and performs the right conversions during the runtime. Let us now look at known and unknown schema handling

Known Schema handling:

- Schema can be defined for both the field name and type. The fields can be accessed using the field name or positional notation of the field.
- Schema can only be defined only for the field name and the field type will be of `bytearray` by default.
- Schema may not be defined for both the field name and field type. The field name will have no name and the field type will be of `bytearray` by default. The field can be only accessed using its positional notation.

The data types assigned to fields can be changed using the `cast` operator, if you have defined schema while the data is loaded or streamed into Pig. You can also change the default type (`bytearray`) using the `cast` operator. We shall look at the `cast` operator in the next section of this chapter.

Unknown Schema handling:

- If you `JOIN`, `COGROUP` or `CROSS` multiple relations with any of the relations having unknown schema, the schema for the resulting relation will be null.
- If a bag is `FLATTEN` with empty inner schema, the schema for the resulting relation will be null.

- If two incompatible relations are performed a UNION operation, the schema for the resulting relation will be null.

Do not worry if you do not know the operators discussed above. We shall cover them in our upcoming chapters.

Schemas in more detail will be covered in the Lab exercise with examples for this chapter.

Type Casting in Pig Latin

Casting is the procedure of converting a data type to another using the operator CAST. Casting is similar to that of casting in Java. All you need to do is specify the name of the type you would like to cast to in parentheses before the value. The syntax is

(data_type) field

For example,

```
grunt> B = FOREACH A GENERATE (int)f1 +32;
```

The table below shows the legal casting for all the simple types in Pig Latin. Casting to and from the complex data types is currently not supported. Please note that the casting to bytearray is illegal as Pig does not know how to represent the various data types in binary format. This is not vice versa though, you can convert any type from bytearrays.

To/From	Int	Long	Float	Double	Chararray	Bytearray	Boolean
Int	Yes	Yes	Yes	Yes	Error	Error	Error
Long	Yes	Yes	Yes	Yes	Error	Error	Error
Float	Yes	Yes	Yes	Yes	Error	Error	Error
Double	Yes	Yes	Yes	Yes	Error	Error	Error
Chararray	Yes	Yes	Yes	Yes	Error	Error	Yes
Bytearray	Yes	Yes	Yes	Yes	Yes	Error	Yes
Boolean	Error	Error	Error	Error	Yes	Error	Error

Conversion of types is only possible as long as the conversion is legal according to the table above.

The following points are to be noted while working with casts.

- A field which is casted explicitly to a type remains of that type and does not automatically cast back. For example,

```
grunt> A = LOAD 'data' AS (f1, f2);  
grunt> B = FOREACH A GENERATE (int)f1 +32;
```
- Pig implicitly performs casts automatically depending upon the processing of data. For example, if schema is not specified while loading and a field is added to an integer, the field is automatically casted to an int as shown below.

```
grunt> B = FOREACH A GENERATE f1 + 5;
```
- Down Casting may result in loss of data. For example, down casting a float to int will cause in truncation of decimal points.
- Nonnumeric chararrays converted to any of the numeric data types will result in NULL values.

That's all for theory. Let us proceed to lab exercises and get our hands on what we have discussed.

AIM

The aim of the following lab exercise is to have your hands on simple and complex data types by assigning schemas to the fields, understanding how Null values work and casting the types.

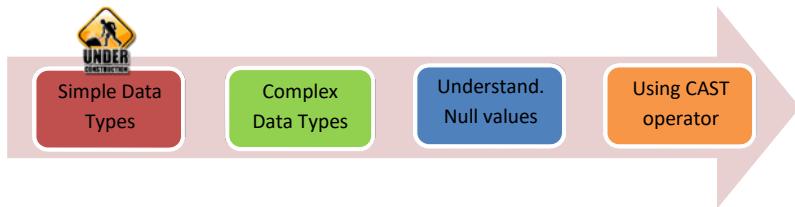
Following steps are required:

- Task 1: Working with Simple types
- Task 2: Working with Complex types
- Task 3: Understanding Null values
- Task 4: Working with Cast operator

Lab Exercise 4: HANDS ON DATA TYPES



-
1. Working with Simple Data Types
 2. Working with Complex Data Types
 3. Understanding Null Values
 4. Using Cast Operator



Task 1: Working with Simple Data Types

Let us start our lab exercise by loading a file and assigning its fields with simple data types.

Step 1: Download the file `salaries.csv` and save it to your home directory from the URL below.

<https://db.tt/3s2oghDi>

Step 2: Start all the Hadoop daemons as well as Pig in MapReduce mode using the following commands below.

```
$ start-dfs.sh
$ start-yarn.sh
$ pig
```

```
uzair@ubuntu:~$ start-dfs.sh
15/02/11 04:39:38 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/hadoop-2.6.0/logs/hadoop-namenode-ubuntu.out
localhost: starting datanode, logging to /usr/local/hadoop-2.6.0/logs/hadoop-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop-2.6.0/logs/hadoop-uzair-secondarynamenode-ubuntu.out
15/02/11 04:40:15 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
uzair@ubuntu:~$ start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop-2.6.0/logs/yarn-uzair-resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /usr/local/hadoop-2.6.0/logs/yarn-nodemanager-ubuntu.out
uzair@ubuntu:~$ pig
15/02/11 04:41:30 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/02/11 04:41:30 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/02/11 04:41:30 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
2015-02-11 04:41:30,727 [main] INFO org.apache.pig.Main - Apache Pig version
```



Simple Data Types

Complex Data Types

Understand. Null values

Using CAST operator

Step 3: Copy the file from home directory in your local file system to HDFS using the following command.

```
grunt> copyFromLocal salaries.csv .
```

In this file, we have 5 fields: year id, team id, league id, player id and salary separated with a comma as shown in the screenshot below.

```
uzair@ubuntu: ~
2010,"TEX","AL","andruel01",418420.00
2010,"TEX","AL","ariasj001",402000.00
2010,"TEX","AL","blancano01",510000.00
2010,"TEX","AL","borboju01",600000.00
2010,"TEX","AL","cruzne02",440000.00
2010,"TEX","AL","davisch02",414120.00
2010,"TEX","AL","feldmsc01",2425000.00
2010,"TEX","AL","felizne01",402000.00
2010,"TEX","AL","francfr01",3265000.00
2010,"TEX","AL","garkory01",550000.00
2010,"TEX","AL","guerryl01",5500000.00
2010,"TEX","AL","hamilj003",3250000.00
2010,"TEX","AL","harderi01",6500000.00
2010,"TEX","AL","harrima01",406090.00
2010,"TEX","AL","huntetoo2",409850.00
2010,"TEX","AL","hurleer01",401000.00
2010,"TEX","AL","kinslia01",4200000.00
2010,"TEX","AL","lewisco01",1750000.00
2010,"TEX","AL","madriwa01",404730.00
2010,"TEX","AL","mathido01",407400.00
2010,"TEX","AL","murphda07",427670.00
2010,"TEX","AL","nippedu01",665000.00
2010,"TEX","AL","odayda01",426700.00
2010,"TEX","AL","oliveda02",3000000.00
```

This is a good file to load into Pig and understand the simple data types as it contains ints, floating point numbers and chararrays.

Step 4: Let us first load this file to Pig without specifying the data types and schema to see how Pig describes them by default. Enter the following statement in the grunt shell to load the file without specifying the data types or schema associated with the file.



Simple Data Types

Complex Data Types

Understand. Null values

Using CAST operator

```
grunt> A = LOAD 'salaries.csv' USING PigStorage(',') ;
```

```
uzair@ubuntu: ~

grunt> A = LOAD 'salaries.csv' USING PigStorage(',') ;
2015-02-11 12:20:50,496 [main] INFO org.apache.hadoop.conf.Configuration.depre
cation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Please see that 'PigStorage()' function is not something we have discussed and we will be covering it in the upcoming chapters. For now, just remember it as a function which is used to specify a delimiter for fields and uses a tab character for delimiting by default. Since our data is delimited by a comma, we use a comma to delimit.

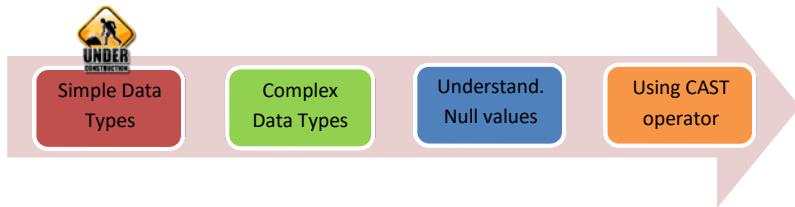
Step 5: Now, let us use the DESCRIBE command on the 'A' relation and check how Pig assigns the schema by default when no schema is specified. Please check Task 3 from previous exercise to know what DESCRIBE command does.

```
grunt> DESCRIBE A;
```

```
uzair@ubuntu: ~

grunt> A = LOAD 'salaries.csv' USING PigStorage(',') ;
2015-02-11 12:20:50,496 [main] INFO org.apache.hadoop.conf.Configuration.depre
cation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
Schema for A unknown.
grunt>
```

As you can see, Pig is unable to assign schema as we have never specified the schema. But, you cannot refer to the fields using their positional notation even though we have not specified the schema. To understand this, enter the following statement which generates first, fourth and fifth fields.



grunt> B = FOREACH A GENERATE \$0, \$3, \$4;

```
uzair@ubuntu:~$ 
grunt> A = LOAD 'salaries.csv' USING PigStorage(',');
2015-02-11 12:20:50,496 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
Schema for A unknown.
grunt> B = FOREACH A GENERATE $0, $3, $4;
```

Dump the relation 'B' to print out the result.

grunt> DUMP B;

```
uzair@ubuntu:~$ 
grunt> A = LOAD 'salaries.csv' USING PigStorage(',');
2015-02-11 12:20:50,496 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
Schema for A unknown.
grunt> B = FOREACH A GENERATE $0, $3, $4;
grunt> DUMP B;
2015-02-11 12:25:10,815 [main] INFO  org.apache.pig.tools.pigstats.ScriptState
- Pig features used in the script: UNKNOWN
2015-02-11 12:25:10,879 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-11 12:25:10,905 [main] INFO  org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-02-11 12:25:10,906 [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]}
2015-02-11 12:25:10,923 [main] INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MRCompiler - File concatenation threshold: 100 optimistic? false
2015-02-11 12:25:11,106 [main] INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MultiQueryOptimizer - MR plan size before optimization: 1
2015-02-11 12:25:11,108 [main] INFO  org.apache.pig.backend.hadoop.executioneng
```

Dumping a relation will trigger a MapReduce job and will provide you with the result after the job is complete as shown below.



Simple Data
Types

Complex
Data Types

Understand.
Null values

Using CAST
operator

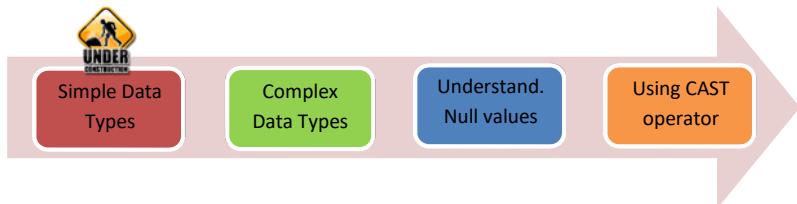
```
uzair@ubuntu: ~
(2010,desmoia01,400000)
(2010,detwiwo01,400000)
(2010,dunnado01,12000000)
(2010,englige01,400000)
(2010,floreje02,750000)
(2010,gonzaal03,415500)
(2010,guzmacr01,8000000)
(2010,harriwi01,1500000)
(2010,kennead01,1250000)
(2010,lannajo01,458000)
(2010,marqua01,7500000)
(2010,mockga01,411000)
(2010,morgany01,426500)
(2010,morsemi01,410000)
(2010,nievewi01,700000)
(2010,rodriivo1,3000000)
(2010,stammcr01,402000)
(2010,taverwi01,400000)
(2010,walkety01,650000)
(2010,wangch01,2000000)
(2010,willijo03,4600000)
(2010,zimmejo01,401000)
(2010,zimmery01,6350000)
grunt>
```

As you can see, the fields have been successfully extracted and generated as desired even though we have not specified schema explicitly while loading the file.

Note: If you have not specified the 'PigStorage()' function, Pig assumes the entire record/row as a single field and if you refer multiple fields using the positional notation, you will have nulls for every field other than the first field.

Step 6: Let us now load the same file again using the same relation names and assign names to fields. We will not be assigning data types to the fields as to check how Pig assigns the data types by default. Enter the following statement which assigns field names to the fields.

```
grunt> A = LOAD 'salaries.csv' USING
          PigStorage(',') AS (year, team_id,
          league_id, player_id, salary);
```



Tip: You can also use the upper and lower arrow keys on your keyboard to check the statements you have previously entered and edit them instead of typing the entire statement again and again.

Now, use the DESCRIBE command to check the schema associated with the relation 'salaries'.

```
grunt> DESCRIBE A;
```

```
uzair@ubuntu: ~
grunt> A = LOAD 'salaries.csv' USING PigStorage(',') AS (year, team_id, league_id, player_id, salary);
2015-02-11 12:43:29,962 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
A: {year: bytearray,team_id: bytearray,league_id: bytearray,player_id: bytearray,salary: bytearray}
grunt>
```

As you can see in the screenshot above, all the fields have been assigned the data type 'bytearray' by default. This is how Pig assigns schema when no data types to fields are assigned explicitly.

You can now refer the fields with their names or with their positional notation as we have done in the previous step. Enter the following statement which generates first, fourth and fifth fields and dump the result to screen as we have done in the previous task.

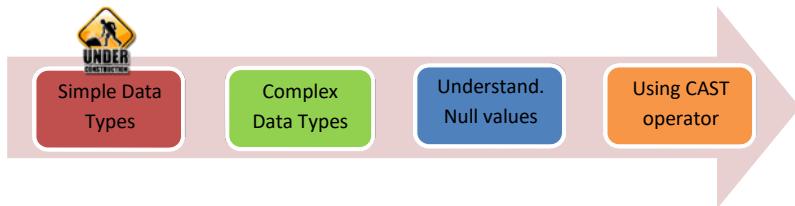
```
grunt> B = FOREACH A GENERATE $0, $3, $4;
```

OR

```
grunt> B = FOREACH A GENERATE (year, player_id, salary);
```

```
grunt> DUMP B;
```

Dumping a relation will trigger a MapReduce job and will provide you with the result after the job is complete.



Step 7: While data types are not explicitly assigned, we know that the fields default to 'bytearray' but Pig is smart enough and automatically applies implicit conversions to the fields to the context in which they are used. For example, in relation B, year is converted to integer because 15 is integer. In relation C, 'year' and 'salary' are converted to double because Pig doesn't know the type of either 'year' or 'salary'. Remember, the actual processing of data only occurs when a DUMP or STORE command is used. Entering any of the other statements in grunt does not trigger actual processing of data.

```
grunt> B = FOREACH A GENERATE year + 15;
grunt> C = FOREACH A GENERATE year + salary;
grunt> DESCRIBE B;
grunt> DESCRIBE C;
```

```
grunt> B = FOREACH A GENERATE year + 15;
2015-02-11 12:58:46,100 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Enc
ountered Warning IMPLICIT_CAST_TO_INT 1 time(s).
grunt> C = FOREACH A GENERATE year + salary;
2015-02-11 12:59:23,455 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Enc
ountered Warning IMPLICIT_CAST_TO_INT 1 time(s).
2015-02-11 12:59:23,456 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Enc
ountered Warning IMPLICIT_CAST_TO_DOUBLE 2 time(s).
grunt> DESCRIBE B;
2015-02-11 12:59:42,126 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Enc
ountered Warning IMPLICIT_CAST_TO_INT 1 time(s).
B: {int}
grunt> DESCRIBE C;
2015-02-11 12:59:48,904 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Enc
ountered Warning IMPLICIT_CAST_TO_DOUBLE 2 time(s).
C: {double}
grunt>
```

You can also see in the screenshot above that Pig warns you about the implicit conversion for each field.

Step 7: Finally, let us explicitly assign data types to the fields and check out the schema.

```
grunt> A = LOAD 'salaries.csv' USING
PigStorage(',') AS (year: int, team_id: chararray,
league_id: chararray, player_id: chararray, salary:
float);
```



Simple Data Types

Complex Data Types

Understand. Null values

Using CAST operator

Now check the schema using DESCRIBE command.

```
grunt> DESCRIBE A;
```

```
uzair@ubuntu: ~

grunt> A = LOAD 'salaries.csv' USING PigStorage(',') AS (year:int, team_id:chararray, league_id:chararray, player_id:chararray, salary:float);
2015-02-11 13:08:06,361 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
A: {year: int,team_id: chararray,league_id: chararray,player_id: chararray,salary: float}
grunt>
```

As you can see from the screenshot above, we have successfully assigned data types to the fields. Similar to this, you can assign long and double data types whenever required accordingly.

Task 1 is complete!

Task 2: Working with Complex Data Types

Tuple:

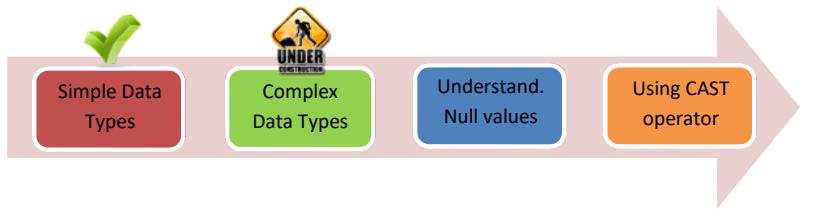
Step 1: Let us first start this task by assigning a schema to tuple type. Download the file **tuple** and save it to your home directory from the URL below.

```
https://db.tt/Hiy3Oc8M
```

Step 2: Copy the file from home directory in your local file system to HDFS using the following command.

```
grunt> copyFromLocal tuple .
```

In this file, we have 3 fields all being integers separated by a comma and enclosed with a parentheses as shown in the screenshot below. This file can be a good example for loading tuples and assigning the fields inside the tuples. For simplicity, the file contains all integer numbers as fields. Remember that you can always use other data types inside a tuple.



```
uzair@ubuntu: ~
grunt> cat tuple
(1,2,3)
(2,3,4)
(3,4,5)
(4,5,6)
(5,6,7)
(6,7,8)
(7,8,9)
(8,9,10)
(9,10,11)
(10,11,12)
(11,12,13)
(12,13,14)
(13,14,15)
(14,15,16)
(15,16,17)
(16,17,18)
(17,18,19)
(18,19,20)
(19,20,21)
(20,21,22)
(21,22,23)
(22,23,24)
(23,24,25)
```

Step 3: Let us first load this file to Pig and specify the schema for tuple using the statement as shown below.

```
grunt> A = LOAD 'tuple' AS T: tuple (f1: int, f2: int, f3: int);
```

Where T is the alias/name assigned for the tuple while f1, f2, and f3 are alias/names of the fields.

The schema to specify a tuple is:

```
alias[:tuple] (alias[:type]) [, (alias[:type]) ...] )
```

To check the schema, use the DESCRIBE command.

```
grunt> DESCRIBE A;
```



```
uzair@ubuntu: ~
grunt> A = LOAD 'tuple' AS T:tuple(f1:int, f2:int, f3:int);
2015-02-11 14:26:08,235 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
A: {T: (f1: int,f2: int,f3: int)}
grunt>
```

Not only can you assign a single tuple using the tuple type but you can also assign multiple tuples. Refer to the example below.

Consider a file 'example' having data as shown below.

(1, 2, 3)	(Ernesto, 30)
(5, 6, 7)	(Jason, 35)
(4, 5, 6)	(Zack, 40)

We can load the file using the following statement.

```
A = LOAD example AS (T1: tuple(f1: int,f2: int,f3: int),T2: tuple(t1: chararray,t2: int));
```

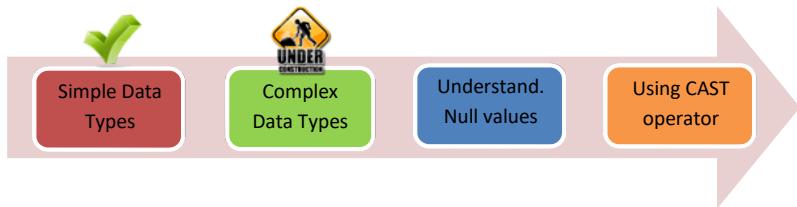
Bag:

Step 4: Let us now assign schema for bag. Download the file `bag` and save it to your home directory from the URL below.

<https://db.tt/kX7fkksV>

Step 5: Copy the file from home directory in your local file system to HDFS using the following command.

```
grunt> copyFromLocal bag .
```



In this file, we have 3 fields all being integers separated by a comma and enclosed with a parentheses and braces as shown in the screenshot below. This file can be a good example for loading bags and assigning the fields inside the tuples in bags. For simplicity, the file contains all integer numbers as fields. Remember that you can always use other data types inside a bag.

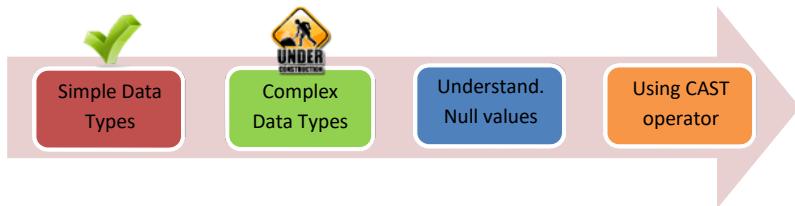
```
uzair@ubuntu: ~
grunt> cat bag
{{1,2,3}}
{{2,3,4}}
{{3,4,5}}
{{4,5,6}}
{{5,6,7}}
{{6,7,8}}
{{7,8,9}}
{{8,9,10}}
{{9,10,11}}
{{10,11,12}}
{{11,12,13}}
{{12,13,14}}
{{13,14,15}}
{{14,15,16}}
{{15,16,17}}
{{16,17,18}}
{{17,18,19}}
{{18,19,20}}
{{19,20,21}}
{{20,21,22}}
{{21,22,23}}
{{22,23,24}}
{{23,24,25}}
```

Step 6: Let us first load this file to Pig and specify the schema for bag using the statement as shown below.

```
grunt> A = LOAD 'bag' AS (B: bag{T: tuple (f1: int, f2: int, f3: int)});
```

Where B is the alias/name assigned for bag, T is the alias/name assigned for the tuple while f1, f2, and f3 are alias/names of the fields.

The schema to specify a bag is:



```
alias[:bag] {tuple}
```

To check the schema, use the DESCRIBE command.

```
grunt> DESCRIBE A;
```

```
uzair@ubuntu: ~
grunt> A = LOAD 'bag' AS (B: bag{T: tuple (f1: int, f2: int, f3: int)});;
2015-02-11 15:00:08,824 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-11 15:00:08,943 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
A: {B: {T: (f1: int,f2: int,f3: int)}}
grunt>
```

Map :

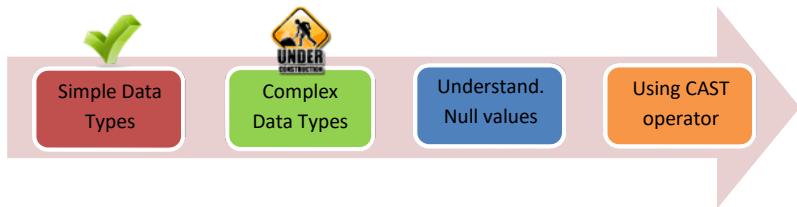
Step 7: Let us now assign schema for map. Download the file map and save it to your home directory from the URL below.

```
https://db.tt/Gs79ibD8
```

Step 8: Copy the file from home directory in your local file system to HDFS using the following command.

```
grunt> copyFromLocal map .
```

In this file, we have 2 fields name and county separated by a pound (#) symbol as shown in the screenshot below. This file can be a good example for loading maps and assigning the fields inside the map.



```
uzair@ubuntu: ~
grunt> cat map
[Cairo#Costa Rica]
[Velma#Jordan]
[Wing#Sao Tome and Principe]
[Zenaida#Cyprus]
[Liberty#Montenegro]
[Fritz#Anguilla]
[Oleg#Pakistan]
[Dieter#Maldives]
[Gary#Myanmar]
[Kylie#Italy]
[Elizabeth#Canada]
[Ralph#Venezuela]
[Rafael#Georgia]
[Brandon#South Africa]
[Alden#Central African Republic]
[Chelsea#Finland]
[Martha#Serbia]
[Macey#Paraguay]
[Iola#Pitcairn Islands]
[Martina#Saint Martin]
[Warren#Guinea]
[Heidi#Ukraine]
[Eve#Côte D'Ivoire (Ivory Coast)]
```

Step 9: Let us first load this file to Pig and specify the schema for map using the statement as shown below.

```
grunt> A = LOAD 'map' AS (M: map[chararray]);
```

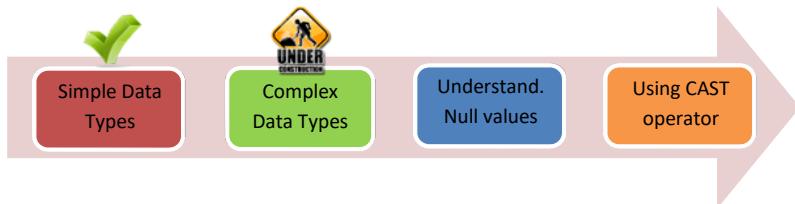
Where M is the alias/name assigned for map.

The schema to specify a bag is:

```
alias: map[type]
```

To check the schema, use the DESCRIBE command.

```
grunt> DESCRIBE A;
```



```
grunt> A = LOAD 'map' AS (M: map[chararray]);
2015-02-11 16:44:38,034 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
A: {M: map[chararray]}
grunt>
```

Task 2 is complete!

Task 3: Understanding Null Values

As we have discussed previously in this chapter, nulls are produced in many ways. Let us look at few examples how nulls are produced.

Step 1: Let us load the file `salaries.csv` into Pig and assign additional schema to a field which does not exist at all.

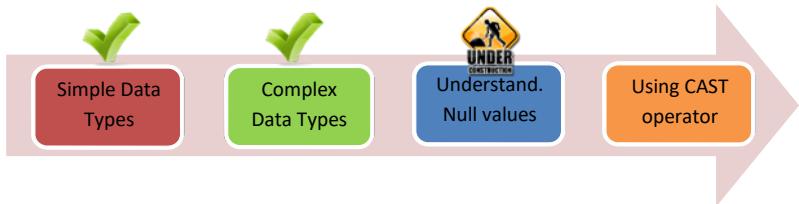
```
grunt> A = LOAD 'salaries.csv' USING
PigStorage(',') AS (year: int, team_id: chararray,
league_id: chararray, player_id: chararray, salary:
float, location: chararray);
```

The schema returns as we have assigned when we use the `DESCRIBE` command.

```
grunt> DESCRIBE A;
```

```
uzair@ubuntu: ~

grunt> A = LOAD 'salaries.csv' USING PigStorage(',') AS (year: int, team_id: cha
rarray, league_id: chararray, player_id: chararray, salary: float, location: cha
rarray);
2015-02-12 04:42:08,839 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> DESCRIBE A;
A: {year: int,team_id: chararray,league_id: chararray,player_id: chararray,salar
y: float,location: chararray}
grunt>
```



But when you dump the relation, you will notice that the sixth field returns empty for each record as the file we have loaded contains only five fields. These empty fields are the null values.

```
grunt> DUMP A;
```

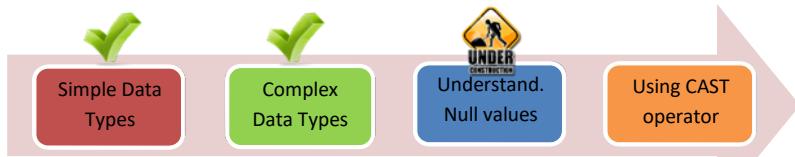
```
uzair@ubuntu: ~
(2010,MIL,NL,stettmi01,420000.0,)
(2010,MIL,NL,suppaje01,1.275E7,)
(2010,MIL,NL,vargacl01,900000.0,)
(2010,MIL,NL,villac01,950000.0,)
(2010,MIL,NL,weeksri01,2750000.0,)
(2010,MIL,NL,wolfra02,8800276.0,)
(2010,MIL,NL,zaungr01,1900000.0,)
(2010,MIN,AL,bakersc02,3000000.0,)
(2010,MIN,AL,blackni01,750000.0,)
(2010,MIN,AL,burneal01,4000000.0,)
(2010,MIN,AL,byterdr01,4000000.0,)
(2010,MIN,AL,casillal01,437500.0,)
(2010,MIN,AL,condrc101,900000.0,)
(2010,MIN,AL,crainje01,2000000.0,)
(2010,MIN,AL,cuddymi01,9416666.0,)
(2010,MIN,AL,duensbr01,417500.0,)
(2010,MIN,AL,guerrma02,3150000.0,)
(2010,MIN,AL,hardyjj01,5100000.0,)
(2010,MIN,AL,harribr01,1450000.0,)
(2010,MIN,AL,hudsoor01,5000000.0,)
(2010,MIN,AL,kubelja01,4100000.0,)
(2010,MIN,AL,liriafr01,1600000.0,)
(2010,MIN,AL,mauerjo01,1.25E7,)
(2010,MIN,AL,mijarjo01,430000.0,)
```

Step 2: Similarly, nulls can be used as place holders of constant expressions in place of expressions of any type. For example, we can write a relation as a continuation to the previous step as follows.

```
grunt> B = FOREACH A GENERATE player_id, null;
```

If you check the schema associated with the relation B, the player_id will be chararray as we have previously assigned but null will default to bytearray.

```
grunt> DESCRIBE B;
```



```
grunt> B = FOREACH A GENERATE player_id, null;
grunt> DESCRIBE B;
B: {player_id: chararray,bytarray}
grunt>
```

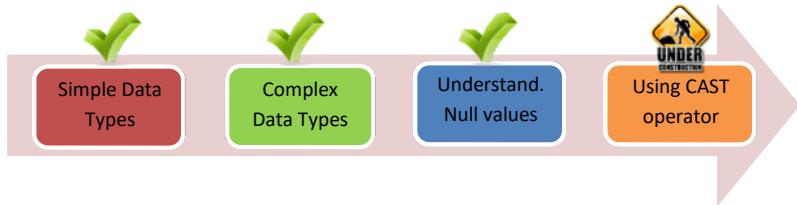
Let us now dump the result to screen.

```
grunt> DUMP B;
```

```
uzair@ubuntu: ~
(priceda01,)
(rodrise01,)
(shielja02,)
(shopkoe01,)
(sonnaan01,)
(soriara01,)
(uptonbj01,)
(wheelda01,)
(zobribe01,)
(andruel01,)
(ariasjo01,)
(blancan01,)
(borboju01,)
(cruzne02,)
(davisch02,)
(feldmsc01,)
(felizne01,)
(francfr01,)
(garkory01,)
(guerrvl01,)
(hamiljo03,)
(harderi01,)
(harrima01,)
(hunteto02,)
```

There are many ways null values are produced either directly or due to a result of an operation. The examples shown here are to help you understand nulls. You will know how nulls can be used or generated with various Pig Latin operators and expressions in the upcoming chapters.

Task 2 is complete!



Task 4: Using Cast Operator

As we know, cast operator is used to convert from one type to another.

Step 1: Let us again load the file `salaries.csv` into Pig and assign schema to all the fields.

```
grunt> A = LOAD 'salaries.csv' USING
PigStorage(',') AS (year: int, team_id: chararray,
league_id: chararray, player_id: chararray, salary:
float);
```

Now that we have the file loaded in Pig, let us convert the salary field from type 'float' to type 'int' and see how the result looks like.

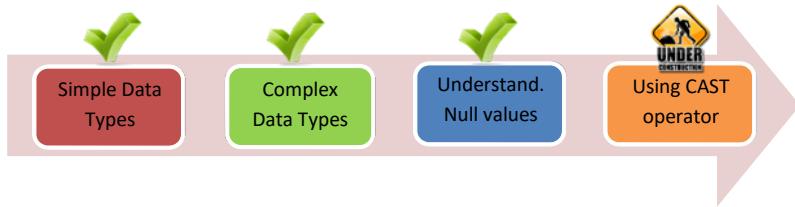
```
grunt> B = FOREACH A GENERATE (int)salary;
```

```
uzair@ubuntu: ~
grunt> A = LOAD 'salaries.csv' USING PigStorage(',') AS (year: int, team_id: cha
rarray, league_id: chararray, player_id: chararray, salary: float);
2015-02-12 05:47:50,899 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> B = FOREACH A GENERATE (int)salary;
grunt>
```

Now dump the result so that we can check how the type conversion has worked.

```
grunt> DUMP B;
```

The result is as shown in the screenshot below. The 'float' type is converted to 'int' and the decimal points have been truncated as 'int' does not support floating point numbers. The same is true for 'double' types too. The decimal points get truncated to form an int.



```
(406090)
(409850)
(401000)
(4200000)
(1750000)
(404730)
(407400)
(427670)
(665000)
(426700)
(3000000)
(975000)
(418580)
(407010)
```

If your process requires to have accurate results, make sure you do not use cast operators as the accuracy is lost in type conversion.

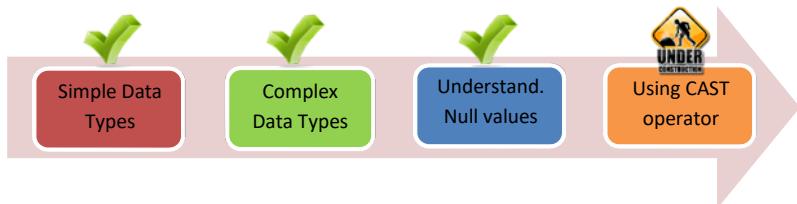
Step 2: Similarly, you can convert this field 'salary' back to 'float' or 'double' but you do not get actual/original decimal points back. You just get the decimal values to '0' for all the values of the field. To understand this, run the following statement.

```
grunt> C = FOREACH B GENERATE (double)salary;
grunt> DUMP C;
```

```
uzair@ubuntu: ~

grunt> C = FOREACH B GENERATE (double)salary;
grunt> DUMP C;
2015-02-12 06:08:20,380 [main] INFO  org.apache.pig.tools.pigstats.ScriptState -
  Pig features used in the script: UNKNOWN
2015-02-12 06:08:20,449 [main] INFO  org.apache.hadoop.conf.Configuration.deprec-
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-12 06:08:20,450 [main] WARN   org.apache.pig.data.SchemaTupleBackend - Sc-
hemaTupleBackend has already been initialized
2015-02-12 06:08:20,452 [main] INFO  org.apache.pig.newplan.logical.optimizer.Lo-
gicalPlanOptimizer - {RULES_ENABLED=[AddForEach, ColumnMapKeyPrune, ConstantCalc-
ulator, GroupByConstParallelSetter, LimitOptimizer, LoadTypeCastInserter, MergeF-
ilter, MergeForEach, PartitionFilterOptimizer, PredicatePushdownOptimizer, PushD
```

The result is as shown below.



```
(950000.0)
(9187500.0)
(1000000.0)
(5450000.0)
(400000.0)
(425000.0)
(2000000.0)
(4312500.0)
(411000.0)
(7500000.0)
```

Step 3: You can also convert integer or floating point numbers to chararray. Let us convert the 'year' field which is type 'int' to 'chararray'.

```
grunt> A = LOAD 'salaries.csv' USING
PigStorage(',') AS (year: int, team_id: chararray,
league_id: chararray, player_id: chararray, salary:
float);
```

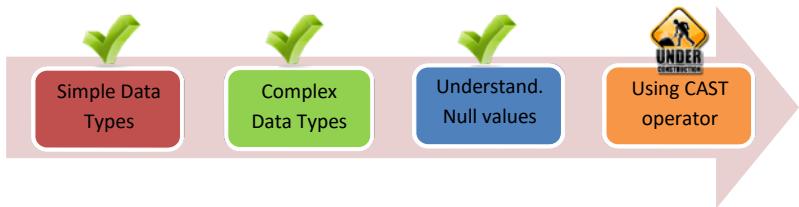
Now, in the next relation, convert the field 'year' from 'int' to 'chararray'.

```
grunt> B = FOREACH A GENERATE (chararray)year;
```

To check if the cast operation was successful, use the DESCRIBE command on relation 'B';

```
grunt> DESCRIBE B;
```

```
uzair@ubuntu: ~
grunt> A = LOAD 'salaries.csv' USING PigStorage(',') AS (year: int, team_id: cha
rarray, league_id: chararray, player_id: chararray, salary: float);
2015-02-12 06:32:34,378 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> B = FOREACH A GENERATE (chararray)year;
grunt> DESCRIBE B;
B: {year: chararray}
grunt>
```



You can also convert back this to any of the numeric types. Remember that you cannot convert a chararray to numeric types when the field contains alphanumeric fields or fields other than numbers. This conversion will result in null. To understand this scenario, let us rewrite the relation 'B' to convert the field 'player_id' to 'int'.

```
grunt> B = FOREACH A GENERATE (int)player id;
```

```
grunt> B = FOREACH A GENERATE (int)player_id;
grunt> DESCRIBE B;
B: {player_id: int}
grunt>
```

You may optionally check the schema of relation 'B'. It shows you that the field 'player_id' is of the type 'int' but when you dump the result of relation 'B', you will have a bunch of null values and nothing else.

```
grunt> DUMP B;
```

The result of casting a field which is alphanumeric from type 'chararray' to 'int' results in nulls as shown in the screenshot below.

Please refer to the table in 'Type Casting in Pig' section to check out more legal ways to convert fields from one type to another.

Task 4 is complete!

LAB CHALLENGE

Challenge

Before you shift your focus to the next chapter, practice the data types a lot more by assigning data types and schema to different data sets. Also refer to the table in 'Type Casting in Pig' section and try to convert types to every possible data type. This makes you perfect with the Pig Data model.

SUMMARY

Similar to any other programming language, Pig Latin also uses data types to describe data to Pig. The data types can be assigned while describing schema to the dataset which is being loaded into Pig. Pig is very good at handling any kind of data ranging from plain text data to nested hierarchical data structures. Pig has two data types: Simple and Complex data types.

The data model in Pig plays a vital role in processing your large datasets with ease and low efforts.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/basic.html>

INDEX

Data Types	92
Pig Latin Nulls.....	97
Pig Latin Schemas	98
Type Casting in Pig Latin	99
AIM	101
Lab Exercise 4: Hands On Data Types	102
Task 1: Working With Simple Data Types	103
Task 2: Working With Complex Data Types	110
Task 3: Understanding Null Values	116
Task 4: Using Cast Operators	119
LAB CHALLENGE.....	123
SUMMARY	124
REFERENCES.....	125

CHAPTER 5: PIG LATIN – PART 1

Theory

Elcomeway otay igpay atinlay! Wondering what this is? Well, this is what Pig Latin is. Pig Latin here is a language game where English words are translated to Pig Latin by taking the first consonant of an English word, moving it to the end of the word and suffixing an *ay*, or if a word begins with a vowel you just add *ay* to the end. For example, *Welcome* becomes *Elcomeway* and *Pig* Latin becomes *Igpay Atinlay*.

Fortunately, the good news is that, this is not what we are dealing with while working with Pig. The concept of Pig Latin programming language is completely different than that of the Pig Latin game and has no relation between them whatsoever. Make sure you do not confuse Pig Latin programming language with Pig Latin game.

What is Pig Latin?

Pig Latin is a high level relational Dataflow scripting language used to explore very large datasets. With Pig Latin, you can create a step-by-step data flow pipeline, where data is processed in every step. Each step in dataflow pipeline is known as a Pig statement or Relation. Pig Latin is a collection of statements or relations. Statements are built using operators, expressions and relations. Every statement in Pig ends with a ‘;’ (semi colon).

Let us look at relations in brief. We have been working with relations in the previous chapters but for the sake of clarity, let us look at an example of a relation below.

```
grunt> employees = LOAD 'employee_data.tsv' ;
```

In the example of a relation above, *employees* is the name of the relation to which the data set *employee_data.tsv* is being loaded. The relation name is known as ‘alias’. Do not confuse alias with variables, as aliases are not variables. Once an alias is assigned, it is permanent. But you can always reuse the same alias and perform other operations to the alias. To understand this, let us look at an example.

```
grunt> employees = LOAD 'employee_data.tsv' AS
          (name, age, salary);
grunt> employees = FILTER employees BY age>50;
grunt> employees = FOREACH employees GENERATE name,
          salary;
```

In the example above, it might look like the same relation name is being assigned again and again, but in reality, a new relation *employees* is being created by overriding the old relation with same name. Pig is smart enough to understand this but is not recommended as it may lead in confusion while reading the program as well as debugging errors.

Similarly, we have field names which describe the fields (*a.k.a columns*) of a relation. In the example above, the relation *employees* has the fields *name*, *age* and *salary*. As we have seen in the previous chapter, we can reference the fields using their names or by their positional notation.

Conventions used in Pig Latin

The following are the conventions used for the syntax in Pig Latin.

() – Parenthesis are used to enclose single or multiple items inside them. Tuples are also indicated using the parenthesis.

[] – Square brackets are used to denote the map data type. Square brackets are also used to enclose single or multiple optional items. When a map data type is used, the optional items can be enclosed using < >.

{ } – Curly braces are used to indicate the bag data type. Curly braces are also used to enclose single or multiple items of which one is required.

Case Sensitivity

There are few rules when it comes to case sensitivity in Pig Latin. The following are such rules:

- The relation names are case sensitive. For example, `employees = LOAD 'employee.tsv'` is not same as `EMPLOYEES = LOAD 'employee.tsv'`.
- The field names are case sensitive. For example, `employees = LOAD 'employee.tsv' AS (name, age, salary)` is not same as `EMPLOYEES = LOAD 'employee.tsv' AS (NAME, AGE, SALARY)`.
- The function names are case sensitive. Examples of function names are `SUM`, `AVG`, `PigStorage()` etc.
- Keywords in Pig Latin such as `LOAD`, `STORE`, `DUMP`, `GENERATE` etc are NOT case sensitive. These keywords can also be written as `load`, `store`, `dump` etc.

Comments in Pig Latin

Comments can be used in the Pig scripts to help others understand your code. Comments in Pig Latin are expressed by -- (double hyphen) for single line comments similar to SQL and multi-line comments begin with /* and end with */ similar to Java. Look at the example below to better understand the concept of comments.

```
A = LOAD 'data'; -- This is a single line comment.  
/* This is a multi-line comment similar to comments  
in Java*/
```

Multi-line comments can be used anywhere in the script as shown below.

```
A = LOAD /* This is a comment */ 'data';
```

Relational Operators in Pig Latin - I

Relational Operators are the heart of Pig Latin. Relational operators are responsible for manipulating data by performing various extraction, transformation and loading operations. Let us look at each operator in detail and understand their importance in Pig Latin.

LOAD

The LOAD operator is used to specify the input path to data on file system. This is the starting point to start working with Pig. As seen in previous chapter in the section 'Pig Latin Schemas', you can optionally specify a schema to the fields in a text file you load. An example of LOAD operator in a Pig statement is as shown below:

```
LOAD 'results/2015/january' USING PigStorage();
```

HDFS is the default file system from which the data is loaded. The default directory is HDFS home directory i.e., /user/[username]. You can also specify the absolute path for the file or directory to be loaded. The example of absolute and relative paths are as shown below.

Relative path:

```
hdfs://nn.cn.com/user/uzair/results/2015/january
```

Absolute path:

```
/results/2015/january
```

You can specify the path to directory or a file. If a directory is specified, all the files inside the directory will be loaded as input. For example, if the directory january has

files 1.txt and 2.txt, both will be loaded. If the specified directory has directories inside it, all the files inside those directories will also be loaded.

By default, the PigStorage() function is used to specify the delimiter and the default delimiter is a tab character. The keyword *USING* is used to specify the load functions. (There are other LOAD functions used to load data from file systems other than HDFS. We shall look at them in the upcoming chapters.) Moreover, if the function PigStorage() is not explicitly defined, the LOAD operator uses built-in PigStorage() function by default and looks for tab-delimited file in the specified path. You can also specify the delimiter in the PigStorage() function as argument if the delimiter in the file is not a tab character. To understand this, consider the example below. The data in the file *clicks.txt* is delimited by a tab character.

```
clicks = LOAD 'clicks.txt';  
clicks = LOAD 'clicks.txt' USING PigStorage();
```

Both the LOAD statements are similar as they load the data from tab-delimited *clicks.txt* file. If the file *clicks.txt* is delimited by a *comma*, you will have to pass a *comma* as argument to the PigStorage() function to specify the delimiter as shown below.

```
clicks = LOAD 'clicks.txt' USING PigStorage(',');
```

Globs similar to Hadoop are also supported in Pig. Globs help you specify multiple files from different directories or specify only specific files from a directory. Pig uses the globs defined in HDFS glob file utilities and not the UNIX shell's glob utilities. Make sure you only use the following Globs as shown in the table below while loading files using the LOAD operator in Pig.

Glob	Description
*	Matches zero or more characters.
?	Matches any single character.
[abc]	Matches a single character from character set (a,b,c).
[a-b]	Matches a single character from the character range (a...b). Note that character a must be lexicographically less than or equal to character b.
[^a]	Matches a single character that is not from character set or range (a). Note that the ^ character must occur immediately to the right of the opening bracket.
/c	Removes (escapes) any special meaning of character c.
{ab,cd}	Matches a string from the string set (ab, cd)

`{ab,c{de,fh}}` | Matches a string from the string set (ab, cde, cfh)

At times you may feel that globs in UNIX shell globs might also work for Pig but this is not true. When you run commands from the UNIX shell using the Hadoop fs command, the UNIX shell may do the substitutions but that does not work while working from Grunt shell. The best example for this is using the expansion glob '...' in UNIX shell. The following command works while running from UNIX shell:

```
$ hadoop fs -ls /records/2014{01,02,03,{04..12}}30
```

But the LOAD statement will not work as shown below since the expansion glob '...' is not supported:

```
LOAD 'records/2014{01,02,03,{04..12}}30';
```

DUMP

DUMP operator is used to display the result of ad-hoc jobs on screen. DUMP comes out very handy when you are debugging or building your Pig script at initial stages to see if everything is working as expected. When you dump a relation, the statements are executed immediately and the results are displayed. The results will not be saved persistently on disk. Make sure you do not use DUMP in the production environment. The syntax is as shown below:

```
DUMP alias;
```

Here, alias is the name of the relation. You can DUMP a relation as soon as you load a file or after performing the transformations.

STORE

While DUMP is used to display the result on screen, STORE is used to save the result persistently on disk. All you need to do is specify the keyword STORE followed by the alias and the absolute or relative path of a directory on file system to save the result. For example,

```
STORE result INTO '/pig/results';
OR
STORE result INTO
'hdfs://nn.cn.com/user/uzair/pig/results';
```

The result will be saved in the 'results' directory with several part files. Make sure the directory is not already present prior to storing the result. If it exists already, the job will fail.

Similar to the LOAD operator, STORE also has the exact same storage functions such as *PigStorage()* or *HBaseStorage()*, while *PigStorage()* being the default storage function unless any other is explicitly specified. By default, the results are stored with fields separated by tab character. You may optionally choose to specify the delimiter of your choice by specifying the USING keyword followed by the store function and pass the delimiter as argument as shown below.

```
STORE result INTO '/pig/results' USING PigStorage(',') ;
```

STORE is used in production environment for obvious reasons.

FILTER

The FILTER operator as the name suggests is used to filter the data based upon a condition. If the condition for that record is true, it will be retained. If false, it will not. This will help remove unwanted data from the relations. The Filter operator in Pig is similar to the WHERE clause in SQL. An example of FILTER statement is,

```
filtered = FILTER clicks BY total_clicks == 250;
```

The conditions are the Boolean expressions which include relational operators such as ==, !=, <, <=, >, >= and logical operators such as NOT, AND, OR for comparing two fields. These can be applied on all the simple types while == and != can be applied for complex types.

For chararrays, you will have to use the keyword MATCHES to test if a string matches a regular expression. For example, if you are looking for fields which contain the string 'apache', you will have to specify it as '.*apache.*' and not just the string 'apache' as shown below.

```
clicks = LOAD 'clicks.txt' AS (user: chararray, ip:  
    chararray, country: chararray,  
    total_clicks:int);
```

```
B = FILTER clicks BY user MATCHES '.*john.*';
```

When filter is applied against a Boolean expression with comparison operators and MATCHES keyword, if a sub expression is null, the result will be null and not true or false even though the other sub expression to which it is compared to is null. For example, in the relation below the result will be null.

```
B = FILTER clicks BY total_clicks == null;
```

You can filter out the null values from your data using '*is not null*' operator or test a field for nulls using '*is null*' operator. Returns true if it is null otherwise returns false. For example,

```
B = FILTER clicks BY total_clicks is not null;
```

FILTER operator works on rows of data. Use FOREACH operator to work on columns of data.

DISTINCT

The DISTINCT operator is used to remove the duplicate records. DISTINCT operator cannot be used on fields. It only works on the entire records of a relation. An example of a DISTINCT statement is as shown below.

```
C = DISTINCT clicks;
```

When DISTINCT operator is used, the order of records is not preserved as Pig first sorts the data to eliminate duplicate records. This results in a reduce phase while processing. Therefore, you can optionally specify number of reducers to use, using the PARALLEL keyword. Moreover, you can also specify a Hadoop partitioner using the keyword PARTITION BY. For example,

```
C = DISTINCT clicks PARTITION BY
org.apache.pig.test.utils.CustomPartitioner
PARALLEL 2;
```

FOREACH...GENERATE

FOREACH...GENERATE adds or removes fields from relation. As the name implies, FOREACH applies a specified transformation to every record of the relation and generates new records. For example, the first statement below load the records as specified but in the second statement, removes all the fields except *user* and *total_clicks*.

```
clicks = LOAD 'clicks.txt' AS (user: chararray, ip:
    chararray, country: chararray, city:
    chararray, total_clicks:int, ctr:float);
D = FOREACH clicks GENERATE user, total_clicks;
```

In the FOREACH statement above, we have referred the fields using field names. You can also refer the fields using their positional notations. If you would like to project the entire fields to a new relation using FOREACH, you can do it by referring all the fields using asterisk (*). Moreover, if there are a lot of fields and you would like to project a range of fields using (..) two periods as shown in the example below.

```

--Projecting all the fields to a new relation
D = FOREACH clicks GENERATE *;

--Projecting range of fields to a new relation
--projects user, ip and country fields.
D = FOREACH clicks GENERATE ..country;

--projects ip, country,city fields.
D = FOREACH clicks GENERATE ip..city;

--projects city, total_clicks, ctr fields.
D = FOREACH clicks GENERATE city..;

```

Schemas can also be specified inside the FOREACH statements. That means, you can assign names to the fields using the AS keyword similar to that of LOAD. For example,

```

clicks = LOAD 'clicks.txt';
D = FOREACH clicks GENERATE $0 AS user, $3 AS
total_clicks;

```

However, note that the AS keyword in FOREACH is specified to each field, while in LOAD, it is specified as whole. We shall look at all the possible ways of extracting data using FOREACH in the lab exercises. There are few advanced use cases of FOREACH operator which we would cover in the next chapter.

GROUP

The GROUP operator is used to group all the records together based on a grouping key. The grouping key is nothing but a field. There can be single as well as multiple grouping keys. It is also possible to group using all the fields just by using *all* keyword. After grouping the similar records together using a group key, you can apply further aggregations as required. Let us look at an example of GROUP statement.

```

grunt> employees = LOAD 'employee_data.tsv' AS
          (name, age, salary);
grunt> grouped = GROUP employees BY age;

```

When you dump the alias *grouped*, you will have all the records grouped according to the *age* as result. You will have two fields i.e., the group key and the bag of records corresponding to the group key. Here is the part where most of the people tend to confuse. The first field which is group key is named as *group* while the second field takes the name of the relation (*employees*, in the example above) itself. Please do not confuse the name of the group key (*group*) with the GROUP operator.

The first field *group* is same type of that of the grouping key while the second field is of type *bag*.

We shall look at them in every detail during the lab exercises.

ORDER BY

Use ORDER BY operator when you require your data to be sorted based on single or multiple fields. You also have the option to sort data in ascending or descending order just by specifying the keywords ASC for ascending and DESC for descending. An example of ORDER BY statement is as follows:

```
grunt> employees = LOAD 'employee_data.tsv' AS
          (name, age, salary);
grunt> order = ORDER employees BY age;
          OR
grunt> order = ORDER employees BY age, salary;
```

Note that you need not specify the parenthesis when you use multiple fields to sort using ORDER BY, while GROUP requires you to specify parenthesis while using multiple fields to group. Also numeric fields are sorted numerically while *chararrays* fields are sorted lexically. *bytearrays* are sorted lexically too but are sorted lexically using byte values instead of character values. Sorting *maps*, *bags* or *tuples* will be greeted by an error. Nulls will appear first or last depending upon the sorting as they are considered as smallest values.

JOIN

JOIN operator is used to join two or more relations based on common field values. A Join will be successful only if the keys for each relation match with each other. If a match is not found in the records, they will be dropped. Joins can also be performed on multiple keys provided that both the relations must have same number of joining keys and must be of same types or the types which can be casted implicitly. Let us look at a simple example of join.

```
grunt> employees = LOAD 'employee_data.tsv' AS
          (emp_id, name, age, salary);
grunt> emp_loc = LOAD 'employee_locations.tsv' AS
          (emp_id, location);
grunt> emp_join = JOIN employees BY emp_id, emp_loc
          BY emp_id;
```

There are two types of joins you can make. They are the *Inner Join* and *Outer Join*. The Inner join is the join as shown in the example above. While in the outer joins, the joins which do not have a match on the other sides are also included, with nulls

filling in the places of missing fields. Outer Joins are further classified into three types: Left, Right and Full joins.

A Left Outer Join is a join where all the records from the left side is included even though there is no match on the right side. Similarly, a Right Outer Join is a join where all the records from the right side is included even though there is no match on the left side. A Full Outer join is a join where all the records from right as well as left side are included even though there is no match on either sides. An example of outer joins is as follows

```
grunt> emp_join = JOIN employees BY emp_id RIGHT
          OUTER, emp_loc BY emp_id;
grunt> emp_join = JOIN employees BY emp_id LEFT
          OUTER, emp_loc BY emp_id
grunt> emp_join = JOIN employees BY emp_id FULL
          OUTER, emp_loc BY emp_id
```

The keyword OUTER is optional and can be safely ignored. We shall look at all the possible ways we can accomplish using Joins in our lab exercise.

UNION

Contrary to joins, UNION operator is used to concatenate two relations. UNION is the best way to concatenate two file as a single input wherever globs were not possible. Let us look at an example of UNION operator.

```
grunt> emp_old = LOAD 'employee_olddata.tsv' AS
          (name, age, salary);
grunt> emp_new = LOAD 'employee_newdata.tsv' AS
          (name, age, salary);
grunt> employees = UNION emp_old, emp_new;
```

The following points have to be noted while working with UNION.

- There is no requirement that both the relations have to have identical schema or have the same number of fields.
- If both the input relations have identical schema, the output will have the same schema as inputs.
- Implicit casts are automatically applied whenever possible and that will be the schema for resulting relation.

- If none of the above schema conditions apply, the resulting relation will not have any schema.
- UNION does not preserves the order of records.
- Union does not eliminate duplicates.

SAMPLE

Sample provides you with a random sample of data. The percentage of sample size which you would require has to be specified using a double value ranging from 0 to 1. The SAMPLE operator provides you a random sample every time you use it and there is no guarantee that the same sample would return every time for the particular size. An example of SAMPLE statement is as shown below.

```
grunt> employees = LOAD 'employee_data.tsv';
grunt> samp = SAMPLE employees 0.15;
```

In the above example the relation *samp* will contain 15% of random sample data. Moreover, instead of specifying a percentage of data to sample, you can also specify a scalar value. For example, the scalar value 500 will return a sample of 500 records from the input approximately.

```
grunt> employees = LOAD 'employee_data.tsv';
grunt> grouped = GROUP employees BY all;
grunt> rows = FOREACH grouped GENERATE
          COUNT(employees) AS num_rows;
grunt> samp = SAMPLE employees 500/rows.num_rows;
```

LIMIT

Use LIMIT operator when you require to limit the total number of output results by a specified value. If the total number of output results are lower than the specified value, all the results will return. For example, the following example will return 25 records as output.

```
grunt> employees = LOAD 'employee_data.tsv' AS
          (emp_id, name, age, salary);
grunt> ordered = ORDER employees BY salary;
grunt> high_25 = LIMIT ordered 25;
```

Please note that, if you do not use the ORDER operator, you will not get the results in sorted order. Also, not all the records in a relation are processed by Pig when

LIMIT operator is used. Using LIMIT is recommended whenever possible as it runs more efficiently than similar query where LIMIT is not used.

That's all in theory for this chapter. We shall work with all the operators described in here in the lab exercises. Not all the Pig Latin operators are covered in this chapter. We shall cover the remaining Pig Latin operators in next chapter.

AIM

The aim of the following lab exercise is to have your hands on relational operators used in Pig Latin.

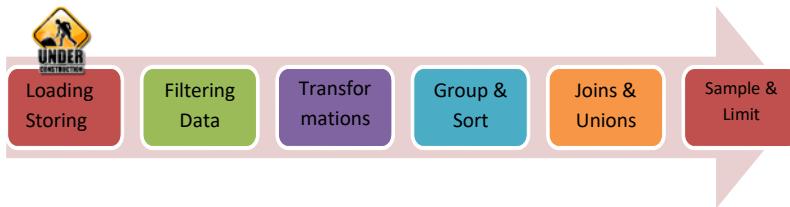
Following steps are required:

- Task 1: Loading and Storing Data
- Task 2: Working with Filters
- Task 3: Applying Transformations
- Task 4: Grouping and Sorting
- Task 5: Working with Joins and Union
- Task 6: Using Sample and Limit operators

Lab Exercise 5: HANDS ON PIG LATIN PART - 1



-
1. **Loading and Storing Data**
 2. **Working with Filters**
 3. **Applying Transformations**
 4. **Grouping and Sorting**
 5. **Working with Joins and Union**
 6. **Using Sample and Limit Operations**



Task 1: Loading and Storing Data

The first step to start working with Pig Latin is to load data into Pig.

Step 1: Download the file *salaries.csv* from the following website and save it to your home directory. (If you have not already done in the previous task).

<https://db.tt/oJ0LzTvN>

Step 2: Start all the Hadoop daemons and Pig in MapReduce mode. Copy the file *salaries.csv* to HDFS using the following command.

grunt> copyFromLocal salaries.csv Pig/salaries.csv

```
grunt> copyFromLocal salaries.csv Pig/salaries.csv
grunt> █
```

Step 3: Now, load the file as shown below.

```
grunt> in_put = LOAD 'Pig/salaries.csv' USING
          PigStorage(',') AS (year, team_id,
          league_id, player_id, salary);

grunt> in_put = LOAD 'Pig/salaries.csv' USING PigStorage(',') AS (year, team_id,
          league_id, player_id, salary);
2015-02-23 13:02:45,391 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-23 13:02:45,614 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> █
```

In the LOAD statement above we have used the relative path to the file *salaries.csv* in HDFS. This is because Pig jobs run from the HDFS home directory by default. You can also use an absolute path to load the file to Pig instead of the relative path as shown below.



Loading
Storing

Filtering
Data

Transforma
tions

Group &
Sort

Joins &
Unions

Sample &
Limit

```
grunt> in_put = LOAD  
'hdfs://localhost:9001/user/<username>Pig/salaries.  
csv' USING PigStorage(',') AS (year, team_id,  
league_id, player_id, salary);
```

Here <username> is your Ubuntu username.

```
grunt> in_put = LOAD 'hdfs://localhost:9001/user/uzair/Pig/salaries.csv' USING  
PigStorage(',') AS (year, team_id, league_id, player_id, salary);  
2015-02-23 13:07:28,303 [main] INFO org.apache.hadoop.conf.Configuration.deprec  
ation - fs.default.name is deprecated. Instead, use fs.defaultFS  
grunt>
```

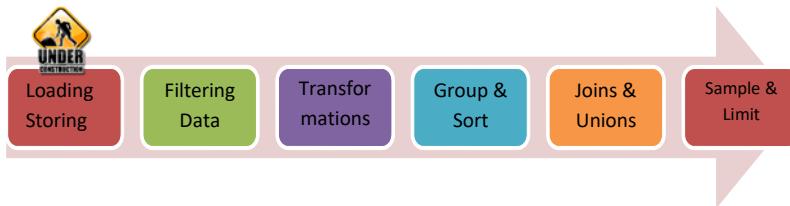
The above statement loads the file *salaries.csv* from HDFS into the relation *in_put*. We use *PigStorage(',')* for specifying the delimiter. It is not necessary to specify a delimiter for this specific file as the fields are delimited by comma (','). The *PigStorage()* default delimiter is a tab character. Moreover, if no loading function is defined, *PigStorage()* will be used by default. There are few other functions similar to *PigStorage()*, which we would cover in upcoming chapters. We also specify the schema for the data using the *AS* clause, so that we can reference the fields by a name rather than the position.

Since we only had a single file, we have specified the path directly to file. If there were multiple files to load, we would have specified a path to the directory instead as shown below.

```
grunt> in_put = LOAD  
'hdfs://localhost:9001/user/<username>Pig' USING  
PigStorage(',') AS (year, team_id, league_id,  
player_id, salary);
```

Step 4: You can also use globs as described in the LOAD section in the theory of this chapter. For example, let us again load the same file again using globs.

```
grunt> input_globs = LOAD 'Pig/s*.csv' USING  
PigStorage(',') AS (year, team_id,  
league_id, player_id, salary);
```



```
grunt> input_globs = LOAD 'Pig/s*.csv' USING PigStorage(',') AS (year, team_id,
  league_id, player_id, salary);
2015-02-23 13:23:30,279 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Here, we have used asterisk (*) to match all the files starting with letter 's' in .csv format in the *Pig* directory. This will load all the files starting with letter 's' in .csv format in the *Pig* directory. If you would like to load all the files of any format starting with letter 's' in the *Pig* directory, you will have to omit .csv. Alternatively, if you prefer to load all the files in the *Pig* directory, simply use the asterisk (*) symbol and nothing else. Since we have only one file in this directory, this feature makes no difference but it does saves you from typing.

Step 5: To check if the load was successful, you can dump the relation to screen and see the contents of the file floating on the screen.

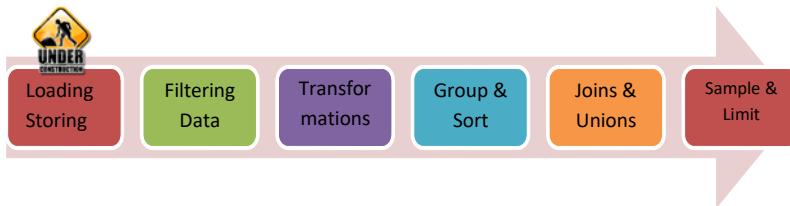
```
grunt> DUMP input_globs;
```

```
uzair@ubuntu: ~
grunt> DUMP input_globs;
2015-02-23 13:34:12,948 [main] INFO org.apache.pig.tools.pigstats.ScriptState -
  Pig features used in the script: UNKNOWN
2015-02-23 13:34:13,145 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-23 13:34:13,208 [main] INFO org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-02-23 13:34:13,366 [main] INFO org.apache.pig.newplan.logical.optimizer.LogicalPlanOptimizer - {RULES ENABLED=[AddForEach, ColumnMapKeyPrune, ConstantCalc]
```

Dumping the result to screen is not recommended in the production environment. Use Dump while in the development phase.

The result is as shown in the screenshot below.

```
(2010,WAS,NL,lannajo01,458000)
(2010,WAS,NL,marqua01,7500000)
(2010,WAS,NL,mockga01,411000)
(2010,WAS,NL,morgany01,426500)
(2010,WAS,NL,morsemi01,410000)
(2010,WAS,NL,nievewi01,700000)
```



Step 6: Alternatively, you can also store the result to a file instead of dumping it to the screen. Enter the following statement as shown below. The result will be stored with fields separated by tab character as the default storage function is also *PigStorage()* which takes tab character as delimiter by default, if not explicitly specified.

```
grunt> STORE input_globs INTO 'final/pig_results';
```

```
grunt> STORE input_globs INTO 'final/pig_results';
2015-02-23 14:41:29,294 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-23 14:41:30,470 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.textoutputformat.separator is deprecated. Instead, use mapreduce.output.textoutputformat.separator
2015-02-23 14:41:30,685 [main] INFO org.apache.pig.tools.pigstats.ScriptState - Pig features used in the script: UNKNOWN
2015-02-23 14:41:31,022 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-23 14:41:31,071 [main] WARN org.apache.pig.data.SchemaTupleBackend - SchemaTupleBackend has already been initialized
```

This action will turn into a MapReduce job under the hood and once it is finished, you will be provided with the *Success* message as shown in the screenshot below.

```
2015-02-23 14:43:19,487 [main] INFO org.apache.hadoop.mapred.ClientServiceDelegate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirecting to job history server
2015-02-23 14:43:19,723 [main] INFO org.apache.hadoop.yarn.client.RMProxy - Connecting to ResourceManager at /0.0.0.0:8032
2015-02-23 14:43:19,781 [main] INFO org.apache.hadoop.mapred.ClientServiceDelegate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirecting to job history server
2015-02-23 14:43:20,022 [main] INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLauncher - Success!
```

Step 7: You can check the contents of the stored file from the grunt shell itself using the following command.

```
grunt> cat final/pig_results/part-m-00000
```



Loading
Storing

Filtering
Data

Transforma-
tions

Group &
Sort

Joins &
Unions

Sample &
Limit

As you can see in the screenshot below, the file stored with a tab character as delimiter, as we have not specified any delimiter explicitly and hence the *PigStorage()* function is picked up by default.

```
2010    WAS    NL    kennead01    1250000
2010    WAS    NL    lannajo01    458000
2010    WAS    NL    marqua01    7500000
2010    WAS    NL    mockga01    411000
2010    WAS    NL    morgany01    426500
2010    WAS    NL    morsemi01    410000
2010    WAS    NL    nieview01    700000
2010    WAS    NL    rodriiv01    3000000
2010    WAS    NL    stammcr01    402000
2010    WAS    NL    taverwi01    400000
2010    WAS    NL    walkety01    650000
2010    WAS    NL    wangch01    2000000
2010    WAS    NL    willijo03    4600000
2010    WAS    NL    zimmej01    401000
2010    WAS    NL    zimmery01    6350000
grunt>
```

You also have the option of specifying a delimiter while storing the result to file along with the schema. The following statement does the trick. Let us change the delimiter to colon (:) by explicitly specifying it in the *PigStorage()* function. Additionally, you can also save the schema so that if you were to load similar data from the same directory, you need not type the schema again as the schema gets attached automatically.

```
grunt> STORE input_globs INTO 'final/results_new'
      USING PigStorage(':', '-schema');
```

Note that we have changed the name of the directory above to store the result. If the same directory is used, the job fails throwing an exception.

It is recommended to always store the schema whenever possible because, you need not type the entire schema again and again for the same dataset. This may not seem worth in our example but consider datasets with tens of number of fields. You would save so much time and great amount of energy as you need not type the schema every time.



Loading
Storing

Filtering
Data

Transforma-
tions

Group &
Sort

Joins &
Unions

Sample &
Limit

```
grunt> STORE input_globs INTO 'final/results_new' USING PigStorage(':', '-schema')
:
2015-02-23 15:06:26,201 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-23 15:06:26,859 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-23 15:06:27,802 [main] INFO  org.apache.pig.tools.pigstats.ScriptState - Pig features used in the script: UNKNOWN
2015-02-23 15:06:28,068 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-23 15:06:28,069 [main] WARN  org.apache.pig.data.SchemaTupleBackend - SchemaTupleBackend has already been initialized
2015-02-23 15:06:28,073 [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]}
```

After the file is stored successfully, you can cat the file and check if the file is stored as desired.

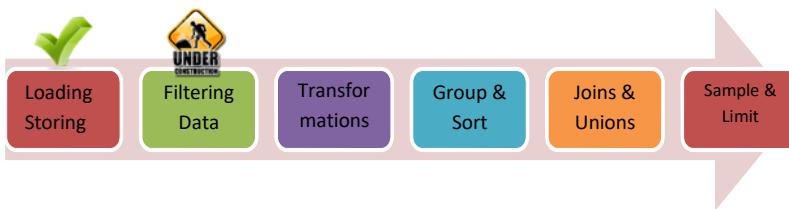
```
grunt> cat final/results_new/part-m-00000
```

As you can see in the screenshot below, the file stored with a colon (:) as delimiter, as we have specified delimiter explicitly and hence the *PigStorage()* function has done its job.

```
2010:WAS:NL:morgany01:426500
2010:WAS:NL:morsemi01:410000
2010:WAS:NL:nievewi01:700000
2010:WAS:NL:rodriiv01:3000000
2010:WAS:NL:stammcr01:402000
2010:WAS:NL:taverwi01:400000
2010:WAS:NL:walkety01:650000
2010:WAS:NL:wangch01:2000000
2010:WAS:NL:willij003:4600000
2010:WAS:NL:zimmejo01:401000
2010:WAS:NL:zimmery01:6350000
grunt> █
```

Finally, you can list the files in this folder to check if the schema has got saved successfully.

Task 1 is complete!



Task 2: Working with Filters

Now that we know how to load and store data in Pig. Let us extract the information from data which we are interested in.

Step 1: Let us load the same file *salaries.csv* again into a new alias to start working with Filter.

```
grunt> salaries = LOAD 'Pig/salaries.csv' USING
      PigStorage(',') AS (year: int, team_id:
      chararray, league_id: chararray,
      player_id: chararray, salary: float);

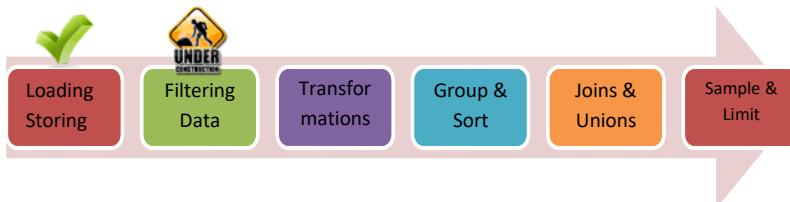
grunt> salaries = LOAD 'Pig/salaries.csv' USING PigStorage(',') AS (year: int,
team_id: chararray, league_id: chararray, player_id: chararray, salary: float);
2015-02-24 11:24:59,995 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 2: We can now use FILTER to filter our data and save it to new relation as shown below.

```
grunt> filtered = FILTER salaries BY salary >
100000;
```

```
grunt> filtered = FILTER salaries BY salary > 100000;
2015-02-24 11:43:53,924 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning IMPLICIT_CAST_TO_FLOAT 2 time(s).
2015-02-24 11:43:53,925 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning IMPLICIT_CAST_TO_DOUBLE 1 time(s).
grunt>
```

The above statement filters data in *salaries* relation from the field *salary* and stores all the data of players which have salary greater than 1, 00,000. You can also add multiple filters in a single statement using AND, OR and other logical operators as per your requirement.



Note the implicit casts to float and double. This is because we have specified 'salary' as float but provided an 'int' as the filter condition. Make sure you specify the correct condition when data accuracy is important.

Step 3: Dump the result and you will have all the filtered data according to your requirement.

```
grunt> DUMP filtered;
```

The result is as shown below.

```
(2010,TBA,AL,daviswa01,404900.0)
(2010,TBA,AL,ekstrmi01,402700.0)
(2010,TBA,AL,garzama01,3350000.0)
(2010,TBA,AL,howeljp01,1800000.0)
(2010,TBA,AL,joycema01,406000.0)
(2010,TBA,AL,kaplega01,1054000.0)
(2010,TBA,AL,longeov01,950000.0)
(2010,TBA,AL,navardl01,2100000.0)
(2010,TBA,AL,niemaje01,1032000.0)
```

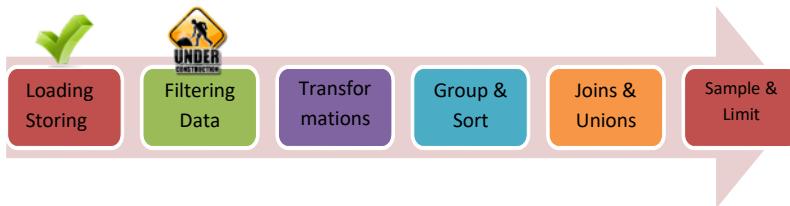
Step 4: Filters can also be applied for chararrays. Let us see how many players are from Team *TEX*.

```
grunt> filtered_teams = FILTER salaries BY team_id
== 'TEX';
```

```
grunt> filtered_teams = FILTER salaries by team_id == 'TEX';
grunt>
```

Dumping the relation *filtered_teams* would have the result as shown below.

```
(2009,TEX,AL,guarded01,1000000.0)
(2009,TEX,AL,hamiljo03,555000.0)
(2009,TEX,AL,harrima01,405500.0)
(2009,TEX,AL,hurleer01,401000.0)
(2009,TEX,AL,jennija01,800000.0)
(2009,TEX,AL,jonesan01,500000.0)
```



Step 5: Let us now use multiple filters using the logical operators. Enter the following statement to find out salaries less than 1,00,000 in team *TEX*:

```
grunt> filtered_sals = FILTER salaries BY team_id
== 'TEX' AND salary < 100000;
```

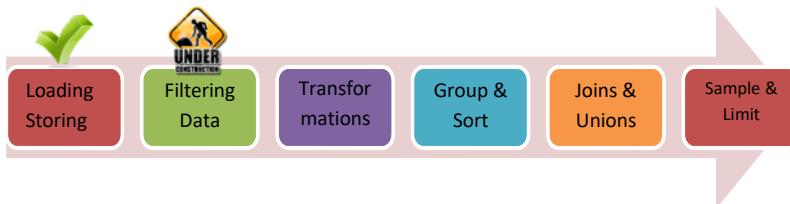
```
grunt> filtered_sals = FILTER salaries BY team_id == 'TEX' AND salary < 100000;
2015-02-24 12:11:04,443 [main] WARN  org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning IMPLICIT_CAST_TO_FLOAT 1 time(s).
grunt>
```

Dumping the relation *filtered_sals* would have the result as shown below.

```
(1986,TEX,AL,loyndmi01,60000.0)
(1986,TEX,AL,mohorda01,60000.0)
(1986,TEX,AL,petrage01,79500.0)
(1986,TEX,AL,russeje01,80000.0)
(1986,TEX,AL,sierrru01,60000.0)
(1986,TEX,AL,willimi02,60000.0)
(1986,TEX,AL,wittbo01,60000.0)
(1986,TEX,AL,wrighri01,70000.0)
(1987,TEX,AL,browebo01,62500.0)
(1987,TEX,AL,brownje01,62500.0)
(1988,TEX,AL,browebo01,98000.0)
(1988,TEX,AL,cecenjo01,62500.0)
(1988,TEX,AL,espyce01,62500.0)
(1988,TEX,AL,kilgupa01,80000.0)
(1988,TEX,AL,stanlmi02,84000.0)
(1989,TEX,AL,brownke01,72500.0)
(1989,TEX,AL,daughja01,68000.0)
(1989,TEX,AL,espyce01,97500.0)
(1989,TEX,AL,halldr01,74000.0)
```

Step 6: Similarly you can check if the chararrays matches a regular expression. Enter the following statement to check how many *player_id* contain the string *brown* in them.

```
grunt> reg_ex = FILTER salaries BY player_id
MATCHES 'brown.*';
```



```
grunt> reg_ex = FILTER salaries BY player_id MATCHES 'brown.*';
2015-02-24 12:27:46,850 [main] WARN  org.apache.pig.newplan.BaseOperatorPlan - Encountered Warning IMPLICIT_CAST_TO_FLOAT 1 time(s).
grunt>
```

Dumping the relation `reg_ex` would have the result as shown below.

```
(1989,CIN,NL,brownto05,1025000.0)
(1989,CLE,AL,brownje01,125000.0)
(1989,TEX,AL,brownke01,72500.0)
(1990,BAL,AL,brownma03,100000.0)
(1990,CIN,NL,brownke02,100000.0)
(1990,CIN,NL,brownto05,2125000.0)
(1990,CLE,AL,brownje01,310000.0)
(1990,MIN,AL,brownja03,100000.0)
(1990,NYN,NL,brownke03,100000.0)
(1990,TEX,AL,brownke01,218000.0)
(1991,CIN,NL,brownto05,2425000.0)
(1991,CLE,AL,brownje01,800000.0)
(1991,TEX,AL,brownke01,355000.0)
(1992,CIN,NL,brownto05,3250000.0)
(1992,MIN,AL,brownja03,112000.0)
```

Similarly there are many case you can use `FILTER` with such as expressions, comparing fields etc.

Task 2 is complete!

Task 3: Applying Transformations

Now that we have seen how to filter the data according to requirement, we can start applying further transformations and extract the information which we require. There is no mandatory sequence that you have to transform your data after filtering. You can straight away start applying transformations and filter the data or vice versa.

Step 1: Let us load the same file `salaries.csv` again into a new alias to start working with `FOREACH` statements.



```
grunt> salaries = LOAD 'Pig/salaries.csv' USING
      PigStorage(',') AS (year: int, team_id:
      chararray, league_id: chararray,
      player_id: chararray, salary: float);

grunt> salaries = LOAD 'Pig/salaries.csv' USING PigStorage(',') AS (year: int,
team_id: chararray, league_id: chararray, player_id: chararray, salary: float);
2015-02-24 11:24:59,995 [Main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 2: Now that we have the file loaded, let us use FOREACH operator and GENERATE just the fields' *year*, *player_id* and *salary*. Enter the following statement to do so.

```
grunt> for_each = FOREACH salaries GENERATE year,
      player_id, salary;
```

```
uzair@ubuntu: ~
grunt> for_each = FOREACH salaries GENERATE year, player_id, salary;
grunt>
```

Dumping the relation *for_each* would have the result as shown below.

```
(2010,gonzaal02,2750000.0)
(2010,gregke01,2000000.0)
(2010,hayhudi01,405000.0)
(2010,hillaa01,4000000.0)
(2010,janssca01,700000.0)
(2010,lindad01,550000.0)
(2010,litscje01,414700.0)
(2010,marcush01,850000.0)
(2010,mccoymi01,400700.0)
(2010,mcdonjo03,1500000.0)
(2010,mcgowdu01,500000.0)
(2010,molinjo01,800000.0)
```



As you can see from the screenshot above, only the fields which we specified have been generated and rest of the fields are skipped.

You can also use project all or only a range of fields as specified in the theory of FOREACH...GENERATE section.

Step 3: We can also perform standard arithmetic operations for integers and floating point numbers such as addition (+), subtraction (-), multiplication (*) and division (/). Let us use FOREACH operator and divide the fields' *salary* with *year* and GENERATE three fields: *year*, *salary*, *salary_year*. As you can see below, you can specify schema in a FOREACH statement using the AS clause, which pretty much describes the third field.

Please note that you would never divide a salary with year in real world applications. This is just shown here for demonstration purposes as we do not have any other integer or floating point numbers in the example.

```
grunt> new_foreach = FOREACH salaries GENERATE
          year, salary, salary/year AS
          salary_year;
```

```
grunt> new_foreach = FOREACH salaries GENERATE year, salary, salary/year AS salary_year;
2015-02-24 13:15:47,999 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Encountered Warning IMPLICIT_CAST_TO_FLOAT 2 time(s).
grunt>
```

We can perform all sorts of arithmetic operations here. We have only performed the division operations for the purpose of demonstration. You are free to perform any arithmetic operation as you desire. Pig also supports the modulo (%) operation for integers.

Dumping the relation *new_foreach* would have the result as shown below.



```
(2010,400700.0,199.35324)
(2010,1500000.0,746.2687)
(2010,500000.0,248.75623)
(2010,800000.0,398.00995)
(2010,409800.0,203.8806)
(2010,7950000.0,3955.2239)
(2010,409900.0,203.93034)
(2010,408300.0,203.13432)
(2010,404300.0,201.14427)
(2010,404600.0,200.99503)
(2010,405800.0,201.89055)
(2010,2000000.0,995.0249)
(2010,409500.0,203.73134)
```

There are many other advanced transformations which can be applied using the FOREACH statement. We shall look at them in the next chapter.

Task 3 is complete!

Task 4: Grouping and Sorting

GROUP:

Step 1: Let us load the same file *salaries.csv* again into a new alias to start working with GROUP statements.

```
grunt> salaries = LOAD 'Pig/salaries.csv' USING
          PigStorage(',') AS (year: int, team_id:
          chararray, league_id: chararray,
          player_id: chararray, salary: float);
```

```
grunt> salaries = LOAD 'Pig/salaries.csv' USING PigStorage(',') AS (year: int,
          team_id: chararray, league_id: chararray, player_id: chararray, salary: float);
2015-02-24 11:24:59,995 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 2: Now that we have the data loaded, let us use the GROUP operator and see how it works. Enter the statement below to group data by *team_id*.

```
grunt> grouped = GROUP salaries BY team_id;
```



```
grunt> grouped = GROUP salaries BY team_id;  
grunt>
```

Dump or Store the relation *grouped*, to check how the data gets grouped when you use the GROUP operator. The result is as shown in the screenshot below.

As you can see from the screenshot above, all the records related to the team (key) **TOR** have been grouped together. The entire result is so big that it cannot accommodate in the screenshot above. Store the relation instead of dumping so that you can have a better view of the result.

Step 3: You can now apply further transformations on the grouped data as desired. For example, enter the following statement to count the total number of records in each team which we just grouped using FOREACH.

```
grunt> agg_group = FOREACH grouped GENERATE group, COUNT(salaries);
```



```
uzair@ubuntu:~
```

```
grunt> agg_group = FOREACH grouped GENERATE group, COUNT(salaries);
grunt>
```

Remember, from the GROUP section, you will have two fields i.e., the group key and the bag of records corresponding to the group key. The first field which is group key is named as *group* while the second field takes the name of the relation (*salaries*, in the example above) itself. Please do not confuse the name of the group key (*group*) with the GROUP operator. You may use DESCRIBE to understand this better.

```
grunt> DESCRIBE grouped;
```

```
grunt> DESCRIBE grouped;
grouped: {group: chararray,salaries: {(year: int,team_id: chararray,league_id: chararray,player_id: chararray,salary: float)}}
grunt> |
```

The result when you dump the relation *agg_grouped* is as shown below.

```
(ANA,247)
(ARI,375)
(ATL,756)
(BAL,775)
(BOS,762)
(CAL,368)
(CHA,732)
(CHN,744)
(CIN,763)
(CLE,781)
(COL,536)
(DET,738)
(FLO,563)
(HOU,728)
(KCA,766)
```

Step 4: You can also group on multiple fields. Enter the following statement to do so. Observe that you will need to enclose the fields in parenthesis.



```
grunt> two_grouped = GROUP salaries BY (team_id,
league_id);
```

```
grunt> two_grouped = GROUP salaries BY (team_id, league_id);
grunt>
```

You may check the schema by using the DESCRIBE keyword.

```
grunt> DESCRIBE two_grouped;
```

```
grunt> two_grouped = GROUP salaries BY (team_id, league_id);
grunt> DESCRIBE two_grouped;
two_grouped: {group: (team_id: chararray, league_id: chararray), salaries: {year: int, team_id: chararray, league_id: chararray, player_id: chararray, salary: float}}
}
grunt>
```

Step 5: We can now apply transformation to this using an aggregate function such as COUNT similar to step 3.

```
grunt> new_agg = FOREACH two_grouped GENERATE
group, COUNT(salaries);
```

```
grunt> new_agg = FOREACH two_grouped GENERATE group, COUNT(salaries);
grunt>
```

You may dump the relation *new_agg* to check the result.

Step 6: You can also group using all the fields of a relation by using the *ALL* clause as shown below. Note that we do not use the *BY* clause when *ALL* is used. Optionally, you may check the schema using DESCRIBE.

```
grunt> group_all = GROUP salaries ALL;
```



```
grunt> group_all = GROUP salaries ALL;
grunt> DESCRIBE group_all;
group_all: {group: chararray,salaries: {(year: int,team_id: chararray,league_id: chararray,player_id: chararray,salary: float)}}
```

You may now apply some transformations using the aggregation functions such as COUNT.

```
grunt> new_agg_all = FOREACH group_all GENERATE COUNT(salaries);
```

```
grunt> group_all = GROUP salaries ALL;
grunt> DESCRIBE group_all;
group_all: {group: chararray,salaries: {(year: int,team_id: chararray,league_id: chararray,player_id: chararray,salary: float)}}
```

```
grunt> new_agg_all = FOREACH group_all GENERATE COUNT(salaries);
grunt>
```

You have the following result when you dump the relation *new_agg_all*.

```
2015-02-24 14:39:51,580 [main] INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLauncher - Success!
2015-02-24 14:39:51,582 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-24 14:39:51,585 [main] INFO org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-02-24 14:39:51,709 [main] INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths to process : 1
2015-02-24 14:39:51,710 [main] INFO org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input paths to process : 1
(21464)
grunt>
```

ORDER BY:

We use the operator ORDER BY to sort data by single or multiple fields.



Step 1: Let us order the *salaries* relation by the salary field using the ORDER BY operator as shown below.

```
grunt> ordered = ORDER salaries BY salary;
```

```
grunt> ordered = ORDER salaries by salary;
grunt>
```

Dump the relation *ordered* to see all the records sorted by the salaries in ascending order. If you would like to have the field *salary* sorted in descending order, just add the keyword DESC at the end as shown below

```
grunt> ordered_des = ORDER salaries BY salary DESC;
```

Step 2: You can also sort on multiple fields using the ORDER BY operator as shown below. Note that there is no requirement of fields to be enclosed in parenthesis, which is in contrary to GROUP operator.

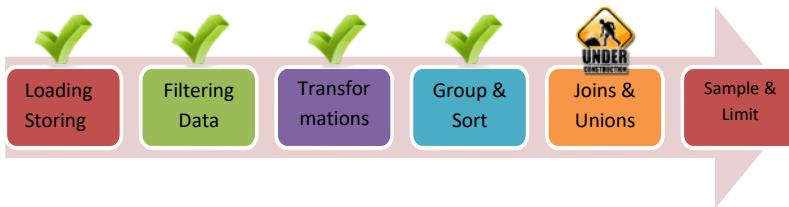
```
grunt> ordered = ORDER salaries BY year, salary;
```

```
grunt> ordered = ORDER salaries BY year, salary;
grunt>
```

You can also use the DESC keyword to sort the fields in descending order but with multiple fields to sort, the DESC keyword is only applied to first field and rest of them are sorted in ascending order.

```
grunt> ordered_des = ORDER salaries BY year DESC,
salary;
```

Task 4 is complete!



Task 5: Working with Joins and Union

Joins:

Step 1: Now that we are dealing with joins, we require a new file to join with our existing file *salaries.csv*. Download the file *AllstarFull.csv* to your home directory from the URL below.

<https://db.tt/KgLilueu>

The file *AllstarFull.csv* contains all the star appearances of players in the matches. The fields include *player_id*, *year*, *gameNum*, *gameID*, *team_id*, *league_id*, *GP* and *startingPos*.

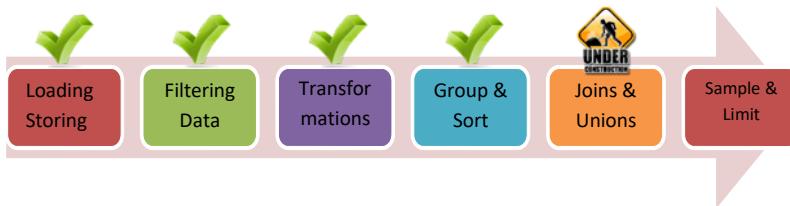
Step 2: Copy the file *AllStarFull.csv* to HDFS using the following command.

```
grunt> copyFromLocal AllstarFull.csv Pig/AllstarFull.csv
```

Step 3: Now load both the files to Pig by entering the statements as shown below.

```
grunt> salaries = LOAD 'Pig/salaries.csv' USING
      PigStorage(',') AS (year: int, team_id:
      chararray, league_id: chararray,
      player_id: chararray, salary: float);

grunt> star_all = LOAD 'Pig/AllstarFull.csv' USING
      PigStorage(',') AS (player_id:
      chararray, year: int, gameNum: int,
      gameID: chararray, team_id: chararray,
      league_id: chararray, GP: chararray,
      startingPos: chararray);
```



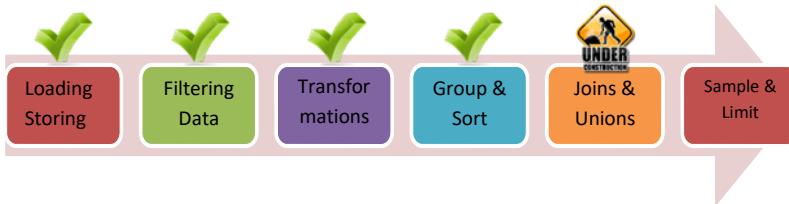
```
grunt> copyFromLocal AllstarFull.csv Pig/AllStarFull.csv
grunt> salaries = LOAD 'Pig/salaries.csv' USING PigStorage(',') AS (year: int, team_id: chararray, league_id: chararray, player_id: chararray, salary: float);
2015-02-24 15:50:07,358 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-24 15:50:07,533 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> star_all = LOAD 'Pig/ AllstarFull.csv' USING PigStorage(',') AS (player_id: chararray, year: int, gameNum: int, gameID: chararray, team_id: chararray, league_id: chararray, GP: chararray, startingPos: chararray);
2015-02-24 15:51:14,759 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 4: Now that we have both the files loaded into two different relations, let us join them using *player_id* as joining key.

```
grunt> joined = JOIN salaries BY player_id,
star_all BY player_id;
```

```
grunt> joined = JOIN salaries BY player_id, star_all BY player_id;
grunt> ■
```

Dump the relation *joined* to screen and you will have the following result with two relations joined based on a joining key which is *player_id* in this case.



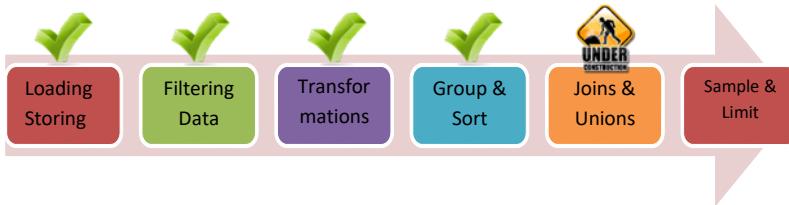
```
uzair@ubuntu:~
(2008,CHN,NL,zambrca01,1.6E7,zambrca01,2004,0,NLS200407130,CHN,NL,1,)
(2008,CHN,NL,zambrca01,1.6E7,zambrca01,2006,0,NLS200607110,CHN,NL,0,)
(2008,CHN,NL,zambrca01,1.6E7,zambrca01,2008,0,ALS200807150,CHN,NL,1,)
(2009,CHN,NL,zambrca01,1.875E7,zambrca01,2004,0,NLS200407130,CHN,NL,1,)
(2009,CHN,NL,zambrca01,1.875E7,zambrca01,2006,0,NLS200607110,CHN,NL,0,)
(2009,CHN,NL,zambrca01,1.875E7,zambrca01,2008,0,ALS200807150,CHN,NL,1,)
(2007,CHN,NL,zambrca01,1.24E7,zambrca01,2004,0,NLS200407130,CHN,NL,1,)
(2007,CHN,NL,zambrca01,1.24E7,zambrca01,2006,0,NLS200607110,CHN,NL,0,)
(2007,CHN,NL,zambrca01,1.24E7,zambrca01,2008,0,ALS200807150,CHN,NL,1,)
(2006,CHN,NL,zambrca01,6500000.0,zambrca01,2004,0,NLS200407130,CHN,NL,1,)
(2006,CHN,NL,zambrca01,6500000.0,zambrca01,2006,0,NLS200607110,CHN,NL,0,)
(2006,CHN,NL,zambrca01,6500000.0,zambrca01,2008,0,ALS200807150,CHN,NL,1,)
(2010,CHN,NL,zambrca01,1.8875E7,zambrca01,2004,0,NLS200407130,CHN,NL,1,)
(2010,CHN,NL,zambrca01,1.8875E7,zambrca01,2006,0,NLS200607110,CHN,NL,0,)
(2010,CHN,NL,zambrca01,1.8875E7,zambrca01,2008,0,ALS200807150,CHN,NL,1,)
(2003,CHN,NL,zambrca01,340000.0,zambrca01,2004,0,NLS200407130,CHN,NL,1,)
(2003,CHN,NL,zambrca01,340000.0,zambrca01,2006,0,NLS200607110,CHN,NL,0,)
(2003,CHN,NL,zambrca01,340000.0,zambrca01,2008,0,ALS200807150,CHN,NL,1,)
(2004,CHN,NL,zambrca01,450000.0,zambrca01,2004,0,NLS200407130,CHN,NL,1,)
(2004,CHN,NL,zambrca01,450000.0,zambrca01,2006,0,NLS200607110,CHN,NL,0,)
(2004,CHN,NL,zambrca01,450000.0,zambrca01,2008,0,ALS200807150,CHN,NL,1,)
(2005,CHN,NL,zambrca01,3760000.0,zambrca01,2004,0,NLS200407130,CHN,NL,1,)
(2005,CHN,NL,zambrca01,3760000.0,zambrca01,2006,0,NLS200607110,CHN,NL,0,)
(2005,CHN,NL,zambrca01,3760000.0,zambrca01,2008,0,ALS200807150,CHN,NL,1,)
```

Step 5: You can also perform joins on multiple keys, provided that both the relations must have same number of joining keys and must be of same types or the types which can be casted implicitly. Let us join these two datasets using two keys (fields): *player_id* and *year*. Enter the following statement to achieve the same.

```
grunt> multi_join = JOIN salaries BY (year,
player_id), star_all BY (year, player_id);
```

Note that the fields have to enclosed in parenthesis when joins are performed using multiple fields.

```
grunt> multi_join = JOIN salaries BY (year, player_id), star_all BY (year, player_id);
grunt>
```



Dump the relation *multi_join* to screen and you will have the following result with two relations joined based on multiple joining keys which are *year* and *player_id* in this case.

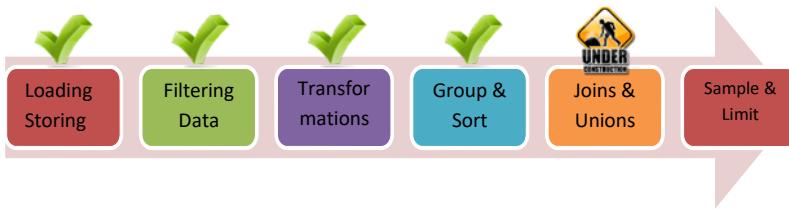
```
uzair@ubuntu: ~
(2007,ATL,NL,mccanbr01,440000.0,mccanbr01,2007,0,NLS200707100,ATL,NL,1,)
(2007,KCA,AL,mechegi01,7400000.0,mechegi01,2007,0,NLS200707100,KCA,AL,0,)
(2007,MIN,AL,morneju01,4500000.0,morneju01,2007,0,NLS200707100,MIN,AL,1,)
(2007,BOS,AL,okajih01,1225000.0,okajih01,2007,0,NLS200707100,BOS,AL,0,)
(2007,DET,AL,ordonna01,1.32E7,ordonna01,2007,0,NLS200707100,DET,AL,1,7)
(2007,BOS,AL,ortizda01,1.325E7,ortizda01,2007,0,NLS200707100,BOS,AL,1,3)
(2007,HOU,NL,oswalro01,1.3E7,oswalro01,2007,0,NLS200707100,HOU,NL,0,)
(2007,BOS,AL,papeljo01,425500.0,papeljo01,2007,0,NLS200707100,BOS,AL,1,)
(2007,SDN,NL,peavyja01,4750000.0,peavyja01,2007,0,NLS200707100,SDN,NL,1,1)
(2007,LAN,NL,pennybr01,7750000.0,pennybr01,2007,0,NLS200707100,LAN,NL,1,)
(2007,DET,AL,polanpl01,4600000.0,polanpl01,2007,0,NLS200707100,DET,AL,1,4)
(2007,NYA,AL,posedjo01,1.2E7,posedjo01,2007,0,NLS200707100,NYA,AL,1,)
(2007,SLN,NL,pujolal01,1.2937813E7,pujolal01,2007,0,NLS200707100,SLN,NL,0,)
(2007,BOS,AL,ramirma02,1.701638E7,ramirma02,2007,0,NLS200707100,BOS,AL,1,)
(2007,NYN,NL,reyesjo01,2875000.0,reyesjo01,2007,0,NLS200707100,NYN,NL,1,6)
(2007,BAL,AL,roberbr01,4200000.0,roberbr01,2007,0,NLS200707100,BAL,AL,1,)
(2007,NYA,AL,rodrial01,2.2708524E7,rodrial01,2007,0,NLS200707100,NYA,AL,1,5)
(2007,LAA,AL,rodrifr03,7000000.0,rodrifr03,2007,0,NLS200707100,LAA,AL,1,)
(2007,DET,AL,rodriv01,1.0567639E7,rodriv01,2007,0,NLS200707100,DET,AL,1,2)
(2007,PHI,NL,rownaaa01,4350000.0,rownaaa01,2007,0,NLS200707100,PHI,NL,1,)
(2007,CLE,AL,sabatcc01,8750000.0,sabatcc01,2007,0,NLS200707100,CLE,AL,1,)
(2007,LAN,NL,saitota01,1000000.0,saitota01,2007,0,NLS200707100,LAN,NL,1,)
(2007,PIT,NL,sanchfr01,2750000.0,sanchfr01,2007,0,NLS200707100,PIT,NL,1,)
(2007,MIN,AL,santajo01,1.3E7,santajo01,2007,0,NLS200707100,MIN,AL,1,)
```

Step 6: We can also perform outer joins in Pig. Refer to JOIN section for more information. Let us now perform a LEFT OUTER JOIN. Enter the following statement to perform a LEFT OUTER JOIN.

```
grunt> left_join = JOIN salaries BY player_id LEFT OUTER, star_all BY player_id;
```

We can safely ignore the OUTER keyword and rewrite the statement as shown below. Not specifying the OUTER keyword does not make any difference.

```
grunt> left_join = JOIN salaries BY player_id LEFT, star_all BY player_id;
```



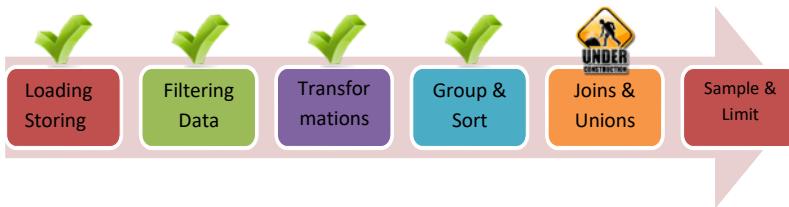
```
grunt> left_join = JOIN salaries BY player_id LEFT, star_all BY player_id;
grunt> ■
```

Dump the relation *left_join* to screen and you will have the following result with two relations left outer joined based on a joining key which is *player_id* in this case.

```
uzair@ubuntu:~
(1989,HOU,NL,youngge02,150000.0,,,...)
(1992,HOU,NL,youngge02,420000.0,,,...)
(1988,HOU,NL,youngge02,85000.0,,,...)
(1991,HOU,NL,youngge02,340000.0,,,...)
(1987,HOU,NL,youngge02,62500.0,,,...)
(1993,COL,NL,youngge02,310000.0,,,...)
(1989,CIN,NL,youngjo02,320000.0,youngjo02,1981,0,ALS198108090,NYN,NL,1,)
(1986,SFN,NL,youngjo02,450000.0,youngjo02,1981,0,ALS198108090,NYN,NL,1,)
(1987,SFN,NL,youngjo02,450000.0,youngjo02,1981,0,ALS198108090,NYN,NL,1,)
(1985,SFN,NL,youngjo02,375000.0,youngjo02,1981,0,ALS198108090,NYN,NL,1,)
(1988,SFN,NL,youngjo02,320000.0,youngjo02,1981,0,ALS198108090,NYN,NL,1,)
(2001,PIT,NL,youngke01,6125000.0,,,...)
(2000,PIT,NL,youngke01,5625000.0,,,...)
(1999,PIT,NL,youngke01,2100000.0,,,...)
(1996,KCA,AL,youngke01,150000.0,,,...)
(2002,PIT,NL,youngke01,5625000.0,,,...)
(1994,PIT,NL,youngke01,160000.0,,,...)
(1995,PIT,NL,youngke01,160000.0,,,...)
(1998,PIT,NL,youngke01,1600000.0,,,...)
(1997,PIT,NL,youngke01,400000.0,,,...)
(2003,PIT,NL,youngke01,6625000.0,,,...)
(1993,PIT,NL,youngke01,109000.0,,,...)
(1986,SEA,AL,youngma01,205000.0,youngma01,1983,0,ALS198307060,SEA,AL,1,)
(1985,SEA,AL,youngma01,159000.0,youngma01,1983,0,ALS198307060,SEA,AL,1,)
```

As you can see in the screenshot above, when a left outer join is performed, all the records from the left side are included even though there are no matching records on the right side. The right side has all the nulls if there is no match.

Similarly, you can perform RIGHT as well as FULL OUTER joins just by specifying the required keyword.



Union:

Step 1: Let us combine both the relations *salaries* and *star_all* using the UNION operator. Please refer to UNION section for more information. Enter the following command to combine two relations into one.

```
grunt> combined = UNION salaries, star_all;
```

```
grunt> combined = UNION salaries, star_all;
grunt> █
```

Step 2: Dump or store the relation *combined* to check if the concatenation of two relation was successful.

Step 3: If your relations are of different schema and you still want to perform UNION, just specify the keyword ONSCHEMA to force the new relation have a common schema. Enter the following to perform a union onschema.

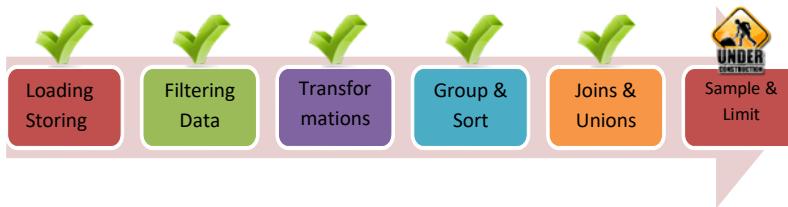
```
grunt> on_schema = UNION ONSCHEMA salaries,
star_all;
```

Use the DESCRIBE operator to check if this has worked.

```
grunt> DESCRIBE on_schema;
```

```
grunt> on_schema = UNION ONSCHEMA salaries, star_all;
grunt> DESCRIBE on_schema;
on_schema: {year: int,team_id: chararray,league_id: chararray,player_id: chararray,salary: float,gameNum: int,gameID: chararray,GP: chararray,startingPos: chararray}
grunt> █
```

Task 5 is complete!



Task 6: Using Sample and Limit Operators

Sample:

Step 1: SAMPLE provides you with a random sample of data. Enter the following statement to sample 15% of the data.

```
grunt> samp = SAMPLE salaries 0.15;
```

```
grunt> samp = SAMPLE salaries 0.15;
grunt>
```

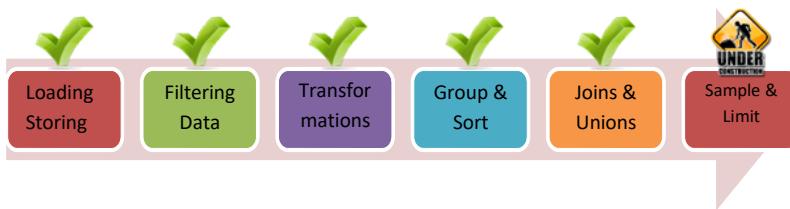
You may dump or store the sample data according to your requirement.

Step 2: Instead of specifying a percentage to sample, you can also specify a scalar expression as shown below.

```
grunt> grouped = GROUP salaries ALL;
grunt> tot_rows = FOREACH grouped GENERATE
          COUNT(salaries) AS total_rows;
grunt> new_sample = SAMPLE salaries
          1000/tot_rows.total_rows;
```

```
grunt> grouped = GROUP salaries ALL;
2015-02-24 17:37:31,338 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Encountered Warning IMPLICIT_CAST_TO_DOUBLE 2 time(s).
grunt> tot_rows = FOREACH grouped GENERATE COUNT(salaries) AS total_rows;
2015-02-24 17:37:35,866 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Encountered Warning IMPLICIT_CAST_TO_DOUBLE 2 time(s).
grunt> new_sample = SAMPLE salaries 25/tot_rows.total_rows;
2015-02-24 17:37:44,462 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Encountered Warning IMPLICIT_CAST_TO_DOUBLE 3 time(s).
2015-02-24 17:37:44,463 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Encountered Warning IMPLICIT_CAST_TO_LONG 1 time(s).
grunt> ■
```

The result will have a sample of approximately 1000 records from the input.



Limit:

Step 1: LIMIT as the name suggests, is used to limit the output records by a specified integer number. Let us see the top 10 salaries paid to the players using the LIMIT operator.

```
grunt> ordered = ORDER salaries BY salary DESC;
grunt> top_10 = LIMIT ordered 10;
```

```
grunt> ordered = ORDER salaries BY salary DESC;
grunt> top_10 = LIMIT ordered 10;
grunt>
```

Step 2: Dump the relation *top_10* to check the top 10 salaries paid to the players. The result is as shown below.

```
2015-02-24 18:05:02,549 [main] INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLauncher - Success!
2015-02-24 18:05:02,615 [main] INFO org.apache.hadoop.conf.Configuration - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-02-24 18:05:02,618 [main] INFO org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-02-24 18:05:02,647 [main] INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths to process : 1
2015-02-24 18:05:02,680 [main] INFO org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input paths to process : 1
(2010,NYA,AL,rodrial01,3.3E7)
(2009,NYA,AL,rodrial01,3.3E7)
(2008,NYA,AL,rodrial01,2.8E7)
(2005,NYA,AL,rodrial01,2.6E7)
(2010,NYA,AL,sabatcc01,2.4285714E7)
(2009,LAN,NL,ramirma02,2.3854494E7)
(2008,NYA,AL,giambja01,2.3428572E7)
(2007,NYA,AL,giambja01,2.3428572E7)
(2007,NYA,AL,rodrial01,2.2708524E7)
(2010,NYA,AL,jeterde01,2.26E7)
grunt> ■
```

Task 6 is complete!

LAB CHALLENGE

Challenge

Please complete the following lab challenges:

- Try loading other datasets and specify schema accordingly. Choose datasets which cover all the data types for fields.
- Apply filters as required using arithmetic expressions as conditions.
- Try to implement all the examples as shown in FOREACH...GENERATE section.
- In the labs we have covered inner joins and left outer join. Try to perform right and full outer joins. Practice joins on different datasets.
- Based upon all the tasks in this exercise, try to use the DISTINCT operator by referring to examples demonstrated in DISTINCT section.
- Practice all the Pig Latin operators specified in different datasets.

SUMMARY

Pig Latin is a high level relational Dataflow scripting language used to explore very large datasets. With Pig Latin, you can create a step-by-step data flow pipeline, where data is processed in every step. Each step in dataflow pipeline is known as a Pig statement or Relation. Pig Latin is a collection of statements or relations. Statements are built using operators, expressions and relations.

Relational Operators are the heart of Pig Latin. Relational operators are responsible for manipulating data by performing various extraction, transformation and loading operations. Let us look at each operator in detail and understand their importance in Pig Latin.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/basic.html>
- Programming Pig by Alan Gates

INDEX

What is Pig Latin?.....	127
Relational Operators in Pig	129
AIM	139
Lab Exercise 5: Hands On Pig Latin - Part -1.....	140
Task 1: Loading and Storing Data.....	141
Task 2: Working With Filters.....	147
Task 3: Applying Transformations.....	150
Task 4: Grouping and Sorting	153
Task 5: Working With Joins and Union	159
Task 6: Using Sample and Limit operators	165
LAB CHALLENGE.....	167
SUMMARY	168
REFERENCES.....	169

CHAPTER 6: PIG LATIN – PART 2

Theory

In the previous chapter, CHAPTER 5: PIG LATIN – PART 1 **Error! Reference source not found.** we have covered some of the relational operators used in Pig Latin. But that is not at all the full list of relational operators used in pig Latin. We shall cover the rest of the Pig Latin relational operators in this chapter. But before we begin, let us have a quick recap on the operators we have already learned.

We have learned how to load and store data using the LOAD and STORE operators. Once the data is loaded to Pig, you may use FOREACH...GENERATE to make aggregations or FILTER operator to filter out unwanted data from input. You may also use GROUP and ORDER BY for grouping and sorting the data as required. If you need to join two or more datasets, you can do it using the JOIN operator. Similarly, if you desire to concatenate two or more datasets, you can achieve the same using the UNION operator. Finally, you can use the SAMPLE operator to sample a percentage of data or number of records from the dataset, use LIMIT operator to limit the result by a specified number of records and use DISTINCT operator to output only the distinct records by removing the duplicates from a relation.

Let us now discuss the Pig Latin operators which have not covered in the previous chapter.

Relational Operators in Pig Latin - II

ASSERT

ASSERT operator is used to make sure that the assertion for the data is true. If the assertion evaluates to false at runtime, the processing fails. For example, let us load data into a relation and apply the following assertion.

Syntax

```
ASSERT <alias> BY <boolean expression>, <Error  
Message if the assertion fails>
```

Let us consider that the data is in following format.

1	Jimmy	45
2	Johnny	25
3	James	36
...		

```
grunt> input = LOAD 'input.txt' AS (x: int, y: int, z: int);
```

You can now use ASSERT operator and make sure that the 'x' field is always greater than 0. If it is not, the processing fails.

```
grunt> ASSERT input BY x > 0, 'x should always be greater than 0';
```

COGROUP

COGROUP operator as the name suggests is similar to that of the GROUP operator, which groups multiple relations at once based on a key. GROUP and COGROUP perform the similar operations. Generally, GROUP is used when working with a single relation, while COGROUP is used when there are multiple relations involved. COGROUP groups the rows based on a grouping key which is nothing but a field (column), and creates bags for each group.

Let us consider an example. Assume we have two datasets `employee_data.tsv` and `employee_locations.tsv`. We can use COGROUP operator to group rows from both the relations based on a grouping key as shown below.

```
grunt> employees = LOAD 'employee_data.tsv' AS  
      (emp_id, name, age, salary);  
grunt> emp_loc = LOAD 'employee_locations.tsv' AS  
      (emp_id, location);  
grunt> cogrouped = COGROUP employees BY emp_id,  
      emp_loc BY emp_id;
```

When you dump the alias `cogrouped`, you will have all the records from both the relations grouped according to the `emp_id` as result. You will have three fields i.e., the group key and the bag of records corresponding to the group key from `employees` relation and bag of records corresponding to the group key from `emp_loc` relation. Since we have had only two relations co-grouped, we have one bag for each relation. If there are n relations, we would have had n bags as output based on group key.

Do not worry, if this sounds like an alien language. We shall have a detail look at this in our lab exercises. Just remember that, GROUP and COGROUP serve the exact purpose but for readability, GROUP is used when working with a single relation, while COGROUP is used when there are multiple relations involved. However, make a note that you can only use COGROUP up to 127 relations only.

CROSS

CROSS operator is used to return the Cartesian product of two or more relations. Each and every record in a relation is combined with all the records in other relation(s). CROSS generates incredibly huge amount of data for the reason above and should be used with caution.

For example, consider two files *dat1* and *dat2*. Let us first load the files and dump to see the contents.

```
grunt> A = LOAD 'dat1' AS (a:chararray,b:chararray);  
grunt> DUMP A;  
  
(x, y)  
(y, z)  
  
grunt> B = LOAD 'dat2' AS (c:int, d:int);  
grunt> DUMP B;  
  
(1, 2)  
(2, 3)
```

Now, lets us perform the cross product if A and B.

```
grunt> C = CROSS A, B;  
grunt> DUMP C;
```

We have the result as follows.

```
(x, y, 1, 2)  
(x, y, 2, 3)  
(y, z, 1, 2)  
(y, z, 2, 3)
```

We shall have a look at CROSS in the lab exercises with a better example.

CUBE

CUBE is a powerful operator which can help you compute aggregations for each and every possible combinations for a dataset on specified group by dimensions. CUBE can be referred to as extended version of GROUP operator. With the GROUP operator we group similar records together based on a grouping key and then apply further aggregations. While CUBE can be used to perform multi-level aggregations

for all the possible combinations on the specified group by dimensions. Let us look at the following example to understand this.

Consider we have a dataset with the following fields: Employee name, gender, country, designation, salary. We would be interested in finding the average salary of the employees for a given country. This is fairly possible by grouping the employees by country and then applying the average function to salary. We can also find the average salary of employees for a given country and also designation by grouping the employees by country and designation and then applying the desired function. However, there might be scenarios where we have to find both the aggregations i.e., finding the average salary of the employees for a given country and finding the average salary of employees for a given country and designation in a single go. This is where CUBE comes into picture. You can easily find the aggregations as mentioned above using the CUBE operator. Not only does CUBE help you compute the aggregations as discussed above but also over multiple dimensions. For example, you can compute all the possible aggregations on country, designation and gender with ease.

All the above explanation above might be hard to digest. Do not worry if it is the case with you. We shall look at CUBE operator with an example and apply it on a dataset in our lab exercises.

ROLLUP

ROLLUP is similar to that of CUBE except that the aggregations are performed in a hierarchical order of specified group by dimensions. By our previous example, the ROLLUP of country, designation and gender will perform all aggregations for a particular country, designation and gender. Then the aggregations for country and designation of any gender is performed. Then the aggregations for a particular country of any designation and any gender is performed. Finally, aggregations for all the employees irrespective of country, designation and gender is performed.

Please check the lab exercises related to ROLLUP to understand this concept better.

RANK

RANK operator is used to rank all the records in a relation. To understand this, consider a file having employee details such as name, country and salary. Let us LOAD them first and then apply the RANK operator.

```
grunt> employees = LOAD 'employee.tsv' AS (name: chararray, country: chararray, salary: chararray);
```

DUMP the relation to see what it contains.

```
grunt> DUMP employees;
(John, USA, 35000)
(Jack, UK, 42000)
(Kim, USA, 25000)
(Mohammed, UK, 25000)
(Zack, USA, 50000)
(Monica, CAN, 10000)
```

Let us now use the RANK operator.

```
grunt> ranked = RANK employees;
```

DUMP the relation to check the result.

```
grunt> DUMP ranked;
(1, John, USA, 35000)
(2, Jack, UK, 42000)
(3, Kim, USA, 25000)
(4, Mohammed, UK, 25000)
(5, Zack, USA, 50000)
(6, Monica, CAN, 10000)
```

As you can see a sequential value has been prepended to every tuple for the *ranked* relation. The RANK operator is pretty much useful when you require to extract the Top n records of a relation.

You may also choose to sort the relation using one or more fields. However, if two or more tuples have the same field values and make a tie, all the tied tuples are assigned a same rank position. For our example, if we chose to rank the *employees* relation and sort them by the *salary* field, the outcome is as follows.

```
grunt> ranked = RANK employees BY salary ASC;
```

DUMP the relation to check the result.

```
grunt> DUMP ranked;
(1, Zack, USA, 50000)
(2, Jack, UK, 42000)
(3, John, USA, 35000)
(4, Kim, USA, 25000)
(4, Mohammed, UK, 25000)
(6, Monica, CAN, 10000)
```

As you can see from the above outcome, *Kim* and *Mohammed* are tied because they have exactly the same salary and have been assigned rank position 4. Also make a note that *Monica* has been assigned rank 6 instead of rank 5.

However, if you do not want to have gaps in the rank positions, you can use the DENSE keyword at the end as shown below.

```
grunt> ranked = RANK employees BY salary ASC DENSE;
```

This will assure that you do not have any gaps in between the rank positions. When we DUMP the relation, this is how the outcome is.

```
grunt> DUMP ranked;
(1, Zack, USA, 50000)
(2, Jack, UK, 42000)
(3, John, USA, 35000)
(4, Kim, USA, 25000)
(4, Mohammed, UK, 25000)
(5, Monica, CAN, 10000)
```

SPLIT

SPLIT operator as the name suggests is used to split a relation into two or more relations basing upon a condition. When you use split, the records in the relation are assigned to multiple relations or are not assigned to any relation at all depending upon the condition you specify. You can use all the possible comparison operators (<, >, == etc) to specify a condition.

Let us consider the example which we have used in the RANK section.

```
grunt> employees = LOAD 'employee.tsv' AS (name:
chararray, country: chararray, salary: chararray);
```

We now use the SPLIT operator to split *employees* depending upon the *country*, *salary* fields.

```
grunt> SPLIT employees INTO employees_usa IF
country=='USA', employees_uk IF country=='UK',
employees_high_sal IF salary > 30000;
```

We now have the records split and stored in their appropriate assigned relations. DUMP these newly assigned relations to check out the result.

```
grunt> DUMP employees_usa;
(John, USA, 35000)
(Kim, USA, 25000)
(Zack, USA, 50000)

grunt> DUMP employees_uk;
(Jack, UK, 42000)
(Mohammed, UK, 25000)

grunt> DUMP employees_high_sal;
(Zack, USA, 50000)
(Jack, UK, 42000)
(John, USA, 35000)
```

SPLIT might look similar to FILTER and it is true to some extent. You can achieve the same result using SPLIT in a single statement while FILTER takes multiple statements depending upon the number of conditions.

FLATTEN

FLATTEN is used to remove a layer of nesting for tuples and bags.

For example, consider a tuple as shown below.

```
(John, (25, 35000))
```

The above tuple is loaded to a relation 'employee'. When the FLATTEN operator is applied as shown below:

```
grunt> sample = FOREACH employee GENERATE $0,
FLATTEN($1);
```

We have the following output:

```
(John, 25, 35000)
```

However, flattening a bag is comparatively complicated. Flattening a bag results in generating tuples. Consider a bag inside a tuple as shown below. The tuple below is a sample result of grouping the *employee* relation by *country*.

```
((Jack, UK, 42000), (Mohammed, UK, 25000))
```

When we apply FLATTEN operator to the above tuple, we have the tuples (Jack, UK, 42000) and (Mohammed, UK, 25000)

Also, when FLATTEN is used with FOREACH operator, we sometimes get a cross product of every record in the bag with all the other expressions in the GENERATE statement. For example, consider a relation with following tuples, which is produced after going through a GROUP statement.

```
(UK, {(Jack, UK, 42000), (Mohammed, UK, 25000)})
```

When FLATTEN is applied to the above statement as shown below, we get the following result.

```
grunt> sample = FOREACH grouped_emp GENERATE $0, FLATTEN($1);  
grunt> DUMP sample;  
(UK, Jack, UK, 42000)  
(UK, Mohammed, UK, 25000)
```

Moreover, if there are multiple bags and they are flattened, we get the cross product of every record in the bag along with all the other expressions in the GENERATE statement.

However, if there is an empty bag in a record and if you try to flatten it, you will end up producing no records at all. For example, consider a file *sample.txt* with a record on which you try to flatten using the FOREACH operator.

```
grunt> cat sample.txt  
{ }, John, 25000  
  
grunt> A = LOAD 'sample.txt' AS (b: bag, {}, name: chararray, int: salary);  
grunt> B = FOREACH A GENERATE FLATTEN(b), salary;  
grunt> DUMP B;  
grunt>
```

As you can see, the empty bag which we tried to flatten has swallowed the entire row. This is because, Pig might not know the schema of the data inside the bag and therefore it does not know what to do with the missing fields. Also, while working with the cross products, crossing a set with an empty set will result in producing an empty set.

Please note that if you do not specify a schema and flatten a bag or a tuple, the records which output due to a FOREACH statement will have a null schema. As there is no schema specified, there is no way Pig can know the outcome of number of fields when FLATTEN is used and so will result in null schema.

Nested FOREACH

In the previous chapter, we have seen the FOREACH...GENERATE operator which can be used to transform the relations and generate fields as desired. The FOREACH statement which we have covered in the previous chapter can be referred to as *outer FOREACH* or simply *FOREACH*. FOREACH can do lot more than this. Not only can we use FOREACH to iterate over the input records but also to apply different relational operators to each and every record in the data processing pipeline. This is known as nested FOREACH or *inner FOREACH*.

Let us look at an example of nested FOREACH statement. Consider a dataset 'sales.tsv' containing sales information of a retail store. The dataset contains the following fields: Transaction ID, Product ID, Transaction Type (Sales or return) and quantity of the product sold as shown below.

```
00001, GRO751, S, 16
00002, VEG859, S, 25
00003, GRO751, R, 6
00004, NUH898, S, 36
```

Now, we need to generate a business report which shows the total sale, total return and net sale of each product. Net sale is the total sale minus the returns. Let us use the nested FOREACH to achieve the same.

Let us LOAD the file.

```
grunt> trans = LOAD 'sales.tsv' AS (transId:
chararray, proId: chararray, tranType: chararray,
quantity: int);
```

We now group the dataset by the product Id.

```
grunt> grouped_trans = GROUP trans BY (proId);
```

Now that we have the sales grouped by product Id, we can apply the nested FOREACH statement and extract the necessary information required from the dataset.

```
grunt> report = FOREACH grouped_trans {
    sale = FILTER trans BY tranType == 'S';
    return = FILTER trans BY tranType == 'R';
    GENERATE group AS proId,
    SUM(sale.quantity) AS totalSale,
    COUNT(return) > 0 ? SUM(return.quantity) : 0
    ) AS totalReturn,
```

```
    SUM(sale.quantity) - (COUNT(return) > 0 ?  
    SUM(return.quantity) : 0) AS netSale;  
}
```

In the above relation, we have used a nested FOREACH statement. Let us walkthrough each line. The first line has a curly brace ({) following the FOREACH operator indicating that it is the beginning of the nested FOREACH statement. The next couple of lines inside the nested FOREACH, filter the transactions based on sales and returns. Next, we use the GENERATE keyword and generate the product id, total sale, total returns and net sale. We use the bincond (?) to check if a condition is true or false and assign the value accordingly. For example, in COUNT(return) > 0 ? SUM(return.quantity) : 0 we check if the return count is greater than 0. If it is, the outcome of SUM(return.quantity) is assigned, else it's value would be zero. The keywords SUM and COUNT are functions which are used to calculate the sum and count respectively. We shall look at the functions in our upcoming chapters.

Finally, you can dump or store the relation *report* according to your requirement.

Please make a note of following while using Nested FOREACH.

- The relational operators can only be applied to relations inside the Nested FOREACH block and cannot be applied to the expressions.
- The Nested FOREACH must always end with the GENERATE keyword.
- CROSS, DISTINCT, FILTER, FOREACH, LIMIT, and ORDER BY are the only relational operators allowed inside the Nested FOREACH block.
- Macros cannot be used inside a nested FOREACH block. We shall cover Macros in our upcoming chapters.
- The FOREACH statements can only be nested to two levels only. Any attempt to nest more than two levels will have a grammar error.
- Parenthesis are required while specifying schema using the AS keyword when FLATTEN is used. However, parenthesis should not be used when FLATTEN operator is not being used.
- While inside the nested block, you can retrieve a field by using a Dereference operator. This would have been illegal if we do this outside the nested block. For example,

```
grunt> report = FOREACH grouped_trans {  
  sale = FILTER trans BY tranType == 'S';  
  fil_sale = sale.tranType;  
  dis_sale = DISTINCT fil_sale;  
  GENERATE group, COUNT(dis_sale); }
```

As you can see above, we have retrieved a field from *sale* relation and assigned it to *fil_sale*. Then we apply DISTINCT operator to extract the distinct values and finally use GENERATE. You might have been tempted to apply the DISTINCT operator directly in the *fil_sale* relation like this: *fil_sale = DISTINCT sale.tranType*. This is not legal as the relational operators cannot be applied to expressions and *sale.tranType* is an expression.

Advanced Joins

In the previous chapter, we have covered Joins which joins two datasets based on the joining key. However, you might come across the scenarios where you would like to optimize your joins according to your datasets. Pig provides the developer to optimize the joins depending upon the requirement. Let us look at few of such scenarios and learn how to optimize joins accordingly when required. Therefore, the developer can choose to optimize the queries and is not dependent on the optimizer, which is the case in SQL.

Fragment - Replicated Joins

Fragment - Replicated Joins are the joins where joining takes place between a smaller and a large dataset. The smaller dataset should be so small that it should be easily able to fit into the main memory. This type of join is very much useful when there is a need to look up in the smaller dataset.

Consider an example of two datasets where there are *employees* and *projectInfo* datasets. The *employees* consists of the employee information as well as the project Id to which the employee is assigned to. While the *projectInfo* has the project Id, city and the country where the project is being developed. The dataset *projectInfo* has very few records and can be considered as a smaller input. If we were to do a default join instead of the Fragment - Replicated Join, the smaller file would be sent to all the nodes in a cluster, loaded into the main memory and then perform the required joining. This would have a reduce phase. But, in the Fragment - Replicated Joins, all the processing is done in the map phase just by loading the smaller dataset into main memory and then perform the join. Following script shows a Fragment - Replicated Join:

```
grunt> employees = LOAD 'employees' AS (name:  
chararray, department: chararray, projectID:  
chararray, salary: int);  
grunt> projects = LOAD 'projectInfo' AS (projectID:  
chararray, city: chararray, country: chararray);  
grunt> rep_join = JOIN employees BY projectID,  
projects BY projectID USING 'replicated';
```

As shown above, we load both the files and tell Pig to perform the Fragment - Replicated Join using the *replicated* keyword at the end.

Please note the following while performing the Fragment - Replicated Joins:

- The smaller dataset should be smaller enough to fit in the main memory. If it does not fit, you will be greeted with an error and the processing fails.
- The larger dataset should be loaded first and then the smaller. There can be more than one smaller datasets and all of them should fit in memory.
- Inner and left-outer joins are only supported by Fragment - Replicated Joins. This is because, while performing a right outer join in the map task, if there is a record in large dataset which does not match any record in smaller dataset, there is no way it would know what to do and hence fails.
- Fragment - Replicated Join is implemented using the Hadoop Distributed cache.

Skewed Joins

While joining two or more datasets, it is always possible to have a good amount skew in the underlying data. This skew would result in having more values for a particular key when compared to other keys. When a default join is performed, all the values corresponding to a key are processed on the same reducer. Therefore, if a particular key has more values and others do not, the reducer which processes the values for the key having significantly more values will take much longer than others. So, we use the optimized version of join known as Skewed join.

When a join is performed using skewed joins, one of the dataset which has to be joined is first sampled. The sampled data is then analyzed for the keys that have great amount of values that skew. Then a normal join is performed for all the keys by allotting same reducer for all the values corresponding to same key except for the

keys which are analyzed before. The keys which contain large amount of values are then split across available reducers and are processed.

Following script shows a skewed join:

```
grunt> employees = LOAD 'employee_data.tsv' AS
        (emp_id, name, age, salary);
grunt> emp_loc = LOAD 'employee_locations.tsv' AS
        (emp_id, location);
grunt> emp_join = JOIN employees BY emp_id, emp_loc
        BY emp_id USING 'skewed';
```

As shown above, we load both the files and tell Pig to perform the Skewed Join using the *skewed* keyword at the end.

Please note the following while performing Skewed Joins:

- Inner and Outer two way skewed joins are only supported. That means you can only perform a skewed join on two input datasets. However, you can break multi-way joins to two way joins and perform skewed joins.
- The dataset which needs to be sampled for number of records per key has to be loaded second.

Merge Joins

In a normal join, all the datasets are first sorted based on the joining key and then the join is performed. This requires to have a map and reduce phase for performing the join. However, there might be cases where both the inputs are already sorted based on the join key and the inputs can be joined directly in the map phase without having to go through the reduce phase. These kind of joins in Pig are known as Merge Joins.

Following script shows a Merge join:

```
grunt> employees = LOAD 'employee_data.tsv' AS
        (emp_id, name, age, salary);
grunt> emp_loc = LOAD 'employee_locations.tsv' AS
        (emp_id, location);
grunt> emp_join = JOIN employees BY emp_id, emp_loc
```

```
BY emp_id USING 'merge';
```

The join is performed by following these steps. Firstly, a MapReduce job is executed on the second relation (*emp_loc*) to sample it. Then an index is built from the sample records that contains the information of the value for the join key (*emp_id*) and the offset of each input split it begins at. Only a record per split is read for this job and this it does not take much time. A second MapReduce job is then executed which takes the first relation (*employees*) and joins both the relations by referring to the index created in the first MapReduce job.

Inner and Outer Merge joins are supported. However, there are few conditions which we have to follow. Inner Merge joins for joining two way inputs has to comply with following conditions:

- The datasets which require an inner merge join should be a result of either a LOAD or ORDER BY statement.
- The data can be filtered or transformed using the FILTER or FOREACH statements before joining. However, the join would not work if the FOREACH has any UDF's (User Defined Functions) involved or if there is a transformation on join keys which would result in changing the sort order.
- While specifying schema, the data types for the Join keys has to be specified.
- The data should be sorted on join keys in ascending order for both the inputs.

Outer merge join for two way inputs and inner merge join for three or more inputs has to comply with following conditions:

- The Join operation should be performed as soon as the data is loaded. There should not be any other operation in between.
- The data should be sorted on join keys in ascending order for both the inputs.
- While specifying schema, the data types for the Join keys has to be specified.

Merge-Sparse Joins

Merge-Sparse join is similar to that of Merge Join but is used only when few records are expected to match during the join. However, if an input is so small that it could

be fit in the main memory, it is recommended to use Fragment-Replicated join. The Merge-Sparse join only works for inner joins. For sparse-merge joins the loader must implement *IndexedLoadFunc* or the join will fail. (We shall look at this in detail in our upcoming chapters while discussing UDF's). Outer joins are not supported as of now. To perform a Merge-Sparse join, use the USING clause followed by the keyword 'merge-sparse' in the Join statement.

Type Construction Operator

Type Construction Operator is used to construct tuple (), bag {} or map [] according to the requirement. This operator can be applied to the expressions while transforming using the relational operators. The following symbols are used for type construction.

- Parenthesis () is used for constructing a tuple.
- Braces { } are used for constructing a bag.
- Square brackets [] are used for constructing a Map.

The examples below show the construction of a tuple, bag and a map.

```
grunt> report = FOREACH employee GENERATE (name,  
gender, salary);
```

The above statement constructs tuples.

```
grunt> report = FOREACH employee GENERATE { (name,  
gender, salary)}, {location, designation};
```

The above statement constructs bags as specified in FOREACH statement.

```
grunt> report = FOREACH employee GENERATE [name,  
location];
```

The above statement constructs a map.

Disambiguate Operator

Disambiguate operator is used to identify field names belonging to a particular relation after JOIN, COGROUP, CROSS, or FLATTEN operators. The symbol (::) is used for disambiguate operator. An example where disambiguate operator can be used is as follows:

```
grunt> A = LOAD 'data1' AS (x, y);  
grunt> B = LOAD 'data2' AS (x, y, z);
```

```
grunt> C = JOIN A BY x, B BY x;  
grunt> D = FOREACH C GENERATE A::y;
```

Disambiguate Operator removes the confusion and lets Pig know which field we are actually referring to.

That's all in theory for this chapter. Let us proceed to the Lab exercises and have our hands on what we have just seen.

AIM

The aim of the following lab exercise is to have your hands on the remaining relational operators used in Pig Latin.

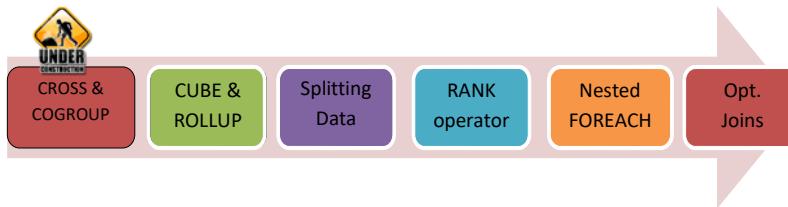
Following steps are required:

- Task 1: Using COGROUP and CROSS.
- Task 2: Using CUBE and ROLLUP.
- Task 3: Using SPLIT operator to Split data
- Task 4: Using the RANK operator
- Task 5: Working with Nested FOREACH
- Task 6: Optimizing Joins

Lab Exercise 6: HANDS ON PIG LATIN PART - 2



1. Using COGROUP and CROSS
2. Using CUBE and ROLLUP
3. Using SPLIT operator to split data
4. Using the RANK operator
5. Working with Nested FOREACH
6. Optimizing joins



Task 1: Using COGROUP and CROSS

COGROUP:

Step 1: Download the files *Sale.csv* and *Sale_Customer.csv* from the following website and save it to your home directory.

Sale.csv - <https://db.tt/8JEZ6M2z>

Sale_Customer.csv - <https://db.tt/JIJqZDMg>

The file *Sale.csv* contains the sales information: Transaction ID, Date of Sale, Purchase ID, Date of Departure and Product ID. The file *Sale_Customer.csv* contains the customer information: Transaction ID and Customer ID.

We shall be using COGROUP to group on the Transaction ID from both the tables. Remember, COGROUP is used to group multiple relations based on a grouping key and create bags for each group.

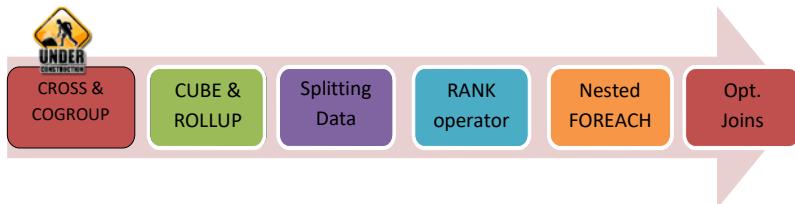
Step 2: Create a new directory in HDFS and name it 'report'. Copy both the files into the directory.

```
$ hadoop fs -copyFromLocal Sale.csv report/Sale.csv
$ hadoop fs -copyFromLocal Sale_Customer.csv
report/Sale_Customer.csv
```

Step 3: Let us now fire up Pig and load these files into Pig by specifying the schema and types. First load the *Sale.csv* file as shown below.

```
grunt> sale = LOAD 'report/Sale.csv' USING
PigStorage(',') AS (transID: int, sale_date:
chararray, purchaseID: int, dep_date: chararray,
prod_id: int);
```

Now load the *Sale_Customer.csv* into Pig.



```
grunt> sale_cust = LOAD 'report/Sale_Customer.csv'
USING PigStorage(',') AS (transID: int, custID:
int);
```

```
uzair@ubuntu: ~
grunt> sale = LOAD 'report/Sale.csv' USING PigStorage(',') AS (transID: int, sale_date: chararray, purchaseID: int, dep_date: chararray, prod_id: int);
2015-03-27 16:07:33,181 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> sale_cust = LOAD 'report/Sale_Customer.csv' USING PigStorage(',') AS (transID: int, custID: int);
2015-03-27 16:08:21,206 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 4: Now use the COGROUP operator and group both the relations based on a grouping key. The grouping key here would be *transID* which is present in both the relations.

```
grunt> grouped = COGROUP sale BY transID, sale_cust
BY transID;
```

```
grunt> sale = LOAD 'report/Sale.csv' USING PigStorage(',') AS (transID: int, sale_date: chararray, purchaseID: int, dep_date: chararray, prod_id: int);
2015-03-27 16:07:33,181 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> sale_cust = LOAD 'report/Sale_Customer.csv' USING PigStorage(',') AS (transID: int, custID: int);
2015-03-27 16:08:21,206 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> grouped = COGROUP sale BY transID, sale_cust BY transID;
grunt>
```



CROSS &
COGROUP

CUBE &
ROLLUP

Splitting
Data

RANK
operator

Nested
FOREACH

Opt.
Joins

Step 5: Let us store this file into a directory '*pig_cogroup*' and then check the result. You can also choose to dump the result.

```
grunt> STORE grouped INTO 'pig_cogroup' USING  
PigStorage( ',' );
```

```
grunt> sale = LOAD 'report/Sale.csv' USING PigStorage(',') AS (transID: int, sal  
e_date: chararray, purchaseID: int, dep_date: chararray, prod_id: int);  
2015-03-27 16:07:33,181 [main] INFO org.apache.hadoop.conf.Configuration.deprec  
ation - fs.default.name is deprecated. Instead, use fs.defaultFS  
grunt> sale_cust = LOAD 'report/Sale_Customer.csv' USING PigStorage(',') AS (tra  
nsID: int, custID: int);  
2015-03-27 16:08:21,206 [main] INFO org.apache.hadoop.conf.Configuration.deprec  
ation - fs.default.name is deprecated. Instead, use fs.defaultFS  
grunt> grouped = COGROUP sale BY transID, sale_cust BY transID;  
grunt> STORE grouped INTO 'pig_cogroup' USING PigStorage( ',' );
```

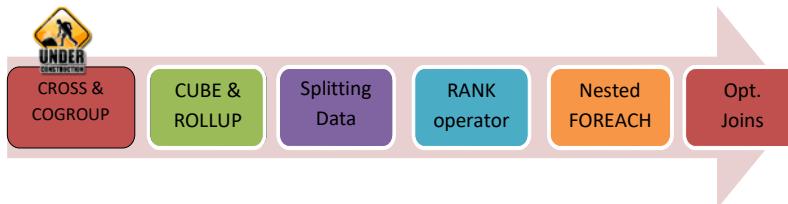
Step 6: After the job is complete, you can check the contents of the output by running the following command.

```
grunt> cat pig_cogroup/part-r-00000
```

You should have the output result similar to the screenshot shown below.

```
99905,{{(99905, 10-24-2011,800, 03-17-2012,2517)},{(99905,803),(99905,801),(99905  
,802)}}  
99906,{{(99906, 11-26-2013,599, 11-06-2014,2116)},{(99906,601),(99906,600)}}  
99907,{{(99907, 06-30-2010,341, 07-01-2010,2213)},{(99907,342),(99907,343),(99907  
,344)}}  
99908,{{(99908, 07-13-2012,655, 08-27-2012,2280)},{(99908,656)}}  
99909,{{(99909, 04-12-2012,170, 04-13-2012,2517)},{(99909,171)}}  
99910,{{(99910, 07-25-2010,395, 09-08-2010,700)},{(99910,397),(99910,396),(99910  
,398)}}  
99911,{{(99911, 09-19-2012,980, 11-03-2012,2948)},{(99911,981),(99911,982)}}  
99912,{{(99912, 03-03-2013,701, 07-26-2013,2299)},{(99912,702),(99912,703)}}  
99913,{{(99913, 04-18-2010,186, 09-10-2010,2072)},{(99913,187),(99913,188),(99913  
,189)}}  
99914,{{(99914, 05-28-2012,166, 07-12-2012,2143)},{(99914,167),(99914,168)}}
```

As you can see, both the relations have been grouped and joined based on the grouping key which is transaction ID in this case.



Once you have co-grouped, you can then use FOREACH operator and perform aggregations as desired.

CROSS:

CROSS is used to return the Cartesian product of two or more relations. This operator results in huge amount of data as output. CROSS is rarely used and you may skip this task if you understand what this does.

Step 1: Download the files *data1.csv* and *data2.csv* from the following website and save it to your home directory.

data1.csv - <https://db.tt/r6MI8WnK>
data2.csv - <https://db.tt/w3WKrXft>

The files *data1.csv* and *data2.csv* contain the following information: name and location.

Step 2: Copy both the files into the directory '*report*' which we have created a while ago.

```
$ hadoop fs -copyFromLocal data1.csv report/data1.csv
$ hadoop fs -copyFromLocal data2.csv report/data2.csv
```

Step 3: Load both the files to Pig as shown below.

```
grunt> data1 = LOAD 'report/data1.csv' USING
PigStorage(',') AS (name: chararray, location:
chararray);

grunt> data2 = LOAD 'report/data2.csv' USING
PigStorage(',') AS (name: chararray, location:
chararray);
```



CROSS &
COGROUP

CUBE &
ROLLUP

Splitting
Data

RANK
operator

Nested
FOREACH

Opt.
Joins

```
grunt> data1 = LOAD 'report/data1.csv' USING PigStorage(',') AS (name: chararray, location: chararray);
2015-03-27 20:47:26,679 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> data2 = LOAD 'report/data2.csv' USING PigStorage(',') AS (name: chararray, location: chararray);
2015-03-27 20:47:42,654 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 4: Use the CROSS operator as shown below to obtain the cross product of both the relations.

```
grunt> crossed = CROSS data1, data2;
```

```
grunt> data1 = LOAD 'report/data1.csv' USING PigStorage(',') AS (name: chararray, location: chararray);
2015-03-27 20:47:26,679 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> data2 = LOAD 'report/data2.csv' USING PigStorage(',') AS (name: chararray, location: chararray);
2015-03-27 20:47:42,654 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> crossed = CROSS data1, data2;
grunt>
```

Step 5: Store the relation *crossed* to check the output. You may also choose to dump the result and check the output directly if you would not like to save the result.

```
grunt> STORE crossed INTO 'pig_cross' USING PigStorage(',') ;
```



CROSS &
COGROUP

CUBE &
ROLLUP

Splitting
Data

RANK
operator

Nested
FOREACH

Opt.
Joins

```
grunt> data1 = LOAD 'report/data1.csv' USING PigStorage(',') AS (name: chararray, location: chararray);
2015-03-27 20:47:26,679 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> data2 = LOAD 'report/data2.csv' USING PigStorage(',') AS (name: chararray, location: chararray);
2015-03-27 20:47:42,654 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> crossed = CROSS data1, data2;
grunt> STORE crossed INTO 'pig_cross' USING PigStorage(',');
```

Step 6: After the job is complete, you can check the contents of the output by running the following command.

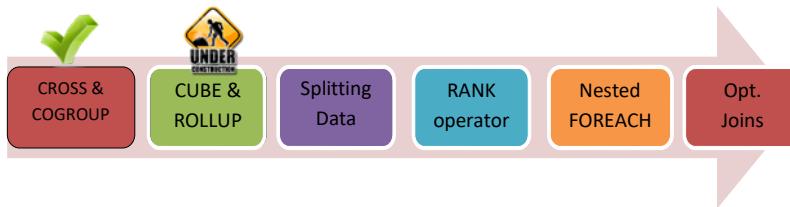
```
grunt> cat pig_cross/part-r-00000
```

You should have the output result similar to the screenshot shown below.

```
Timon Head,Lamorteau,Phyllis Glover,Tavigny
Timon Head,Lamorteau,Winifred Avery,Apeldoorn
Timon Head,Lamorteau,Hilda Peterson,Malartic
Timon Head,Lamorteau,Laurel Malone,Bloomington
Gary Madden,Barnstaple,Kevyn Stevenson,Pievepelago
Gary Madden,Barnstaple,Cynthia Donaldson,Yamuna Nagar
Gary Madden,Barnstaple,Karleigh Mercer,Llandrindod Wells
Gary Madden,Barnstaple,Janna Larsen,Annapolis County
Gary Madden,Barnstaple,Diana Berry,Augusta
Gary Madden,Barnstaple,Nerea Gillespie,Saumur
Gary Madden,Barnstaple,Karleigh Wilkerson,Bordeaux
```

As you can see from the result above, each and every record in a relation is combined with all the records in other relation. You come across such situation where you would require the cross product of two or more relation. Moreover, this is an excessive resource utilizing operation and has to be used only when mostly required.

Task 1 is complete!



Task 2: Using CUBE and ROLLUP

CUBE:

Step 1: Download the file *employees.csv* from the following website and save it to your home directory.

employees.csv - <https://db.tt/0Wc50tYR>

The file *employees.csv* contains the information: Name, gender, country and salary. We shall be using the CUBE operator to find the average salary of an employee for a given country and gender.

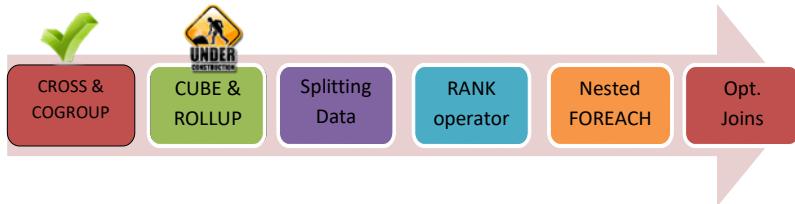
Step 2: Copy the file into the directory '*report*' which we have created in the previous task.

```
$ hadoop fs -copyFromLocal employees.csv
report/employees.csv
```

Step 3: It is now time to load this file to Pig by specifying the schema and data types as shown below.

```
grunt> employees = LOAD 'report/employees.csv'
USING PigStorage(',') AS (name: chararray, gender:
chararray, country: chararray, salary: int);
```

```
grunt> employees = LOAD 'report/employees.csv' USING PigStorage(',') AS (name: chararray, gender: chararray, country: chararray, salary: int);
2015-03-28 16:14:56,258 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-03-28 16:14:56,560 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```



Step 4: Once the data is loaded, we can use the CUBE operator on the gender and country fields as shown below.

```
grunt> cubed = CUBE employees BY CUBE(gender, country);
```

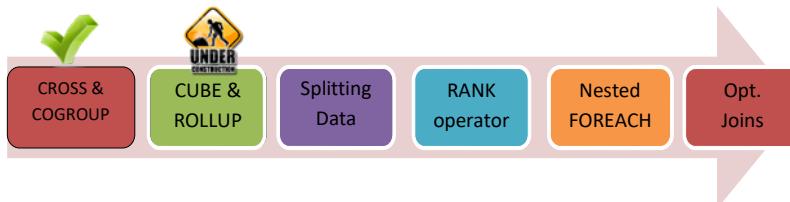
```
grunt> employees = LOAD 'report/employees.csv' USING PigStorage(',') AS (name: chararray, gender: chararray, country: chararray, salary: int);
2015-03-28 16:22:53,603 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> cubed = CUBE employees BY CUBE(gender, country);
grunt>
```

Step 5: You may choose to check the output when CUBE is used by dumping or storing the *cubed* relation. You can also check the schema using the DESCRIBE command instead. However, for clarity, let us dump the relation *cubed* and how the result looks like.

```
grunt> DUMP cubed;
```

The result is as seen in the screenshot below.

```
((F,Falkland Islands),{(F,Falkland Islands,Kaye,21761)}))  
((F,Cocos (Keeling) Islands),{(F,Cocos (Keeling) Islands,Joel,54704)}))  
((F,Côte D'Ivoire (Ivory Coast)),{(F,Côte D'Ivoire (Ivory Coast),Ian,56290)}))  
((F,Holy See (Vatican City State)),{(F,Holy See (Vatican City State),Dana,11711)}))  
((F,),{(F,,Molly,11171),(F,,Cassady,21870),(F,,Alana,62100),(F,,Dillon,92562),(F,,Kyla,86581),(F,,Colby,90938),(F,,Mercedes,52824),(F,,Aspen,81016),(F,,Lionel,80919),(F,,Blaze,12557),(F,,Blossom,23880),(F,,Grace,7441),(F,,Phoebe,28238),(F,,Dalton,91764),(F,,Xerxes,46474),(F,,Tanner,50878),(F,,India,90954),(F,,Elijah,73157),(F,,Phoebe,77966),(F,,Joel,54704),(F,,Shelley,12140),(F,,Herman,13203),(F,,Lionel,80372),(F,,Portia,7597),(F,,Camille,30652),(F,,Dana,11711),(F,,Ian,56290),(F,,Kaye,21761)}))  
((M,Guam),{(M,Guam,August,92188),(M,Guam,Erich,6393)}))  
((M,Laos),{(M,Laos,Quyn,42708)}))
```



This might be hard to look at once but you can see that the first columns are the grouping keys which are gender and country while the second column contains bags of all the rows matching the keys cubed on. This looks like the result of GROUP operator but if you observe carefully, there are some rows which contain the null values corresponding to the aggregations over that column.

The schema for the relation *cubed* is as shown in the screenshot below.

```
grunt> DESCRIBE cubed;
```

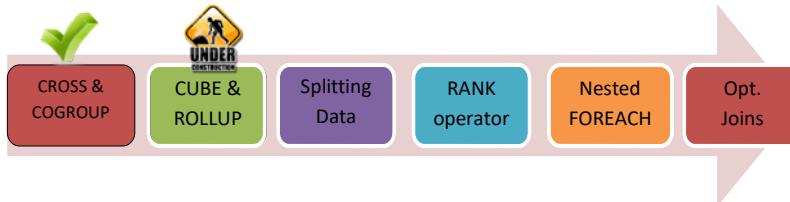
```
grunt> DESCRIBE cubed;
cubed: {group: {gender: chararray,country: chararray},cube: {((gender: chararray,
country: chararray,name: chararray,salary: int))}}
grunt>
```

The grouping keys gender and country are referred to as *group* as seen in GROUP and the bag containing all the matching rows for the grouping keys is referred to as *cube*.

Step 6: Let us now find the average salary of employees for a given country as well as gender.

```
grunt> avg = FOREACH cubed GENERATE FLATTEN(group)
AS (gender, country), COUNT_STAR(cube) AS tot_emp,
AVG(cube.salary) AS avg_sal;
```

```
grunt> DESCRIBE cubed;
cubed: {group: {gender: chararray,country: chararray},cube: {((gender: chararray,
country: chararray,name: chararray,salary: int))}}
grunt> avg = FOREACH cubed GENERATE FLATTEN(group) AS (gender, country), COUNT_S
TAR(cube) AS tot_emp, AVG(cube.salary) AS avg_sal;
grunt>
```



We use the FOREACH...GENERATE operator to project the desired rows. We use the FLATTEN operator to remove a level of nesting i.e., *(gender, country)* becomes *gender, country*. COUNT_STAR is a function which counts all the rows including nulls. (We shall look at this in detail in the next chapter) and AVG function to calculate the salary of employees.

Step 7: Finally, let us dump the relation *avg* to check the result.

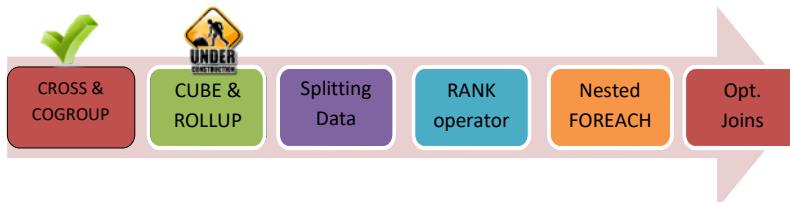
```
grunt> DUMP avg;
```

As seen in the screenshot below, we have the desired aggregation.

```
(M,Russian Federation,1,50038.0)
(M,Trinidad and Tobago,1,93998.0)
(M,Saint Kitts and Nevis,1,71529.0)
(M,Northern Mariana Islands,1,52241.0)
(M,Saint Pierre and Miquelon,1,41720.0)
(M,British Indian Ocean Territory,1,80813.0)
(M,,72,51780.37142857143)
(,Chad,1,23880.0)
(,Guam,3,48351.666666666664)
(,Laos,1,42708.0)
(,Niue,1,84433.0)
(,China,1,8487.0)
(,Ghana,1,95769.0)
(,Japan,1,52019.0)
(,Nauru,2,44756.5)
(,Samoa,1,21123.0)
```

However, you may have observed that there are few null values. These null values can be replaced with *all genders* and *all countries* for the result to make sense. We use the *bincond* operator to achieve the same.

```
grunt> final_avg = FOREACH avg GENERATE (gender is
not NULL ? gender : 'All Genders') as gender,
(country is not NULL ? country : 'All Countries')
as country, tot_emp, avg_sal;
```



Using the *bincond* operator we check whether a gender is not a null value for given row. If it is not null the appropriate gender is generated, else the string *All Genders* is generated. Also the same is followed for countries.

We have the following result when we dump the relation *final_avg*.

```
grunt> DUMP final_avg;
```

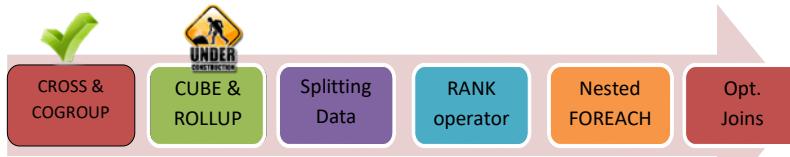
```
(M,Czech Republic,1,73307.0)
(M,Christmas Island,1,88387.0)
(M,Pitcairn Islands,1,26447.0)
(M,Equatorial Guinea,1,93821.0)
(M,Dominican Republic,2,80257.5)
(M,Russian Federation,1,50038.0)
(M,Trinidad and Tobago,1,93998.0)
(M,Saint Kitts and Nevis,1,71529.0)
(M,Northern Mariana Islands,1,52241.0)
(M,Saint Pierre and Miquelon,1,41720.0)
(M,British Indian Ocean Territory,1,80813.0)
(M,All Countries,72,51780.37142857143)
(All Genders,Chad,1,23880.0)
(All Genders,Guam,3,48351.666666666664)
(All Genders,Laos,1,42708.0)
(All Genders,Niue,1,84433.0)
(All Genders,China,1,8487.0)
(All Genders,Ghana,1,95769.0)
(All Genders,Japan,1,52019.0)
(All Genders,Nauru,2,44756.5)
(All Genders,Samoa,1,21123.0)
```

The screenshot above could not fit in the entire result but it should be good enough to get you an idea what CUBE is capable of.

ROLLUP:

ROLLUP is similar to that of CUBE except that the aggregations are performed in a hierachal order of specified group by dimensions.

Step 1: Let us use the dataset used in CUBE task. We will not be loading the file again as it has been already loaded. Let us continue by using the ROLLUP operator as shown below.



```
grunt> rolled = CUBE employees BY ROLLUP(gender, country);
```

```
grunt> rolled = CUBE employees BY ROLLUP(gender, country);
grunt>
```

The rest of the steps is similar to that of CUBE.

Step 2: Let us now find the average salary of employees for a given country as well as gender.

```
grunt> avg = FOREACH rolled GENERATE FLATTEN(group) AS (gender, country), COUNT_STAR(cube) AS tot_emp, AVG(cube.salary) AS avg_sal;
```

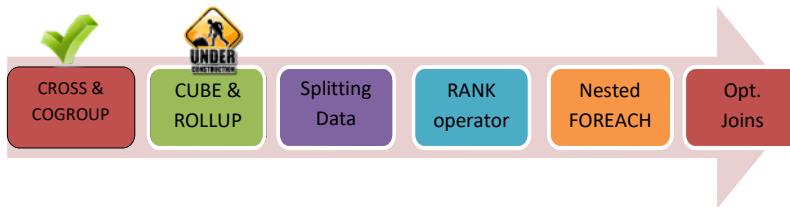
```
grunt> rolled = CUBE employees BY ROLLUP(gender, country);
grunt> avg = FOREACH rolled GENERATE FLATTEN(group) AS (gender, country), COUNT_STAR(cube) AS tot_emp, AVG(cube.salary) AS avg_sal;
grunt>
```

Step 3: Let us make the result to make more sense by adding the string using *bincond* operator as used in CUBE.

```
grunt> final_avg = FOREACH avg GENERATE (gender is not NULL ? gender : 'All Genders') as gender, (country is not NULL ? country : 'All Countries') as country, tot_emp, avg_sal;
```

Finally, dump the relation and you will have the hierachal aggregations as result.

```
grunt> DUMP final_avg;
```



```
(M,Saint Martin,1,76290.0)
(M,Guinea-Bissau,1,14342.0)
(M,Czech Republic,1,73307.0)
(M,Christmas Island,1,88387.0)
(M,Pitcairn Islands,1,26447.0)
(M,Equatorial Guinea,1,93821.0)
(M,Dominican Republic,2,80257.5)
(M,Russian Federation,1,50038.0)
(M,Trinidad and Tobago,1,93998.0)
(M,Saint Kitts and Nevis,1,71529.0)
(M,Northern Mariana Islands,1,52241.0)
(M,Saint Pierre and Miquelon,1,41720.0)
(M,British Indian Ocean Territory,1,80813.0)
(M,All Countries,72,51780.37142857143)
(All Genders,All Countries,100,50983.12244897959)
grunt> █
```

ROLLUP of gender and country will perform all aggregations for a particular gender and country. Then it will perform all aggregations for a particular gender and all countries. And finally it will compute an aggregation for all genders and countries of employees.

Task 2 is complete!

Task 3: Using SPLIT operator to split data

Skip to Step 3 if you have already downloaded the file `employees.csv` in previous task.

Step 1: Download the file `employees.csv` from the following website and save it to your home directory.

`employees.csv - https://db.tt/0Wc50tYR`

The file `employees.csv` contains the information: Name, gender, country and salary.

Step 2: Copy the file into the directory '`report`' which we have created in the previous task.



```
$ hadoop fs -copyFromLocal employees.csv
report/employees.csv
```

Step 3: We shall be using the SPLIT operator to split this data into two different files based on gender of the employees. You can also use split on any of the fields: name, country and salary.

Let us Load the file to Pig so that we can split it in the next step.

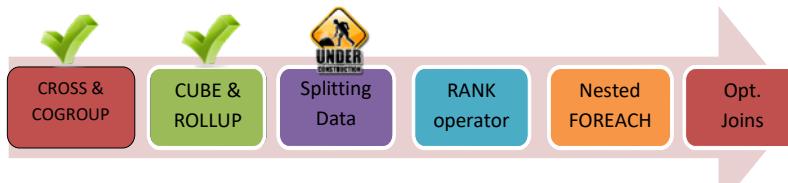
```
grunt> employees = LOAD 'report/employees.csv'
USING PigStorage(',') AS (name: chararray, gender:
chararray, country: chararray, salary: int);
```

```
grunt> employees = LOAD 'report/employees.csv' USING PigStorage(',') AS (name: c
hararray, gender: chararray, country: chararray, salary: int);
2015-03-28 16:14:56,258 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-03-28 16:14:56,560 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 4: Now using the SPLIT operator, we shall split the relation *employees* to two new relations *male_emp* and *female_emp* by applying the split condition on the *gender* column.

```
grunt> SPLIT employees INTO
male_emp IF gender == 'M',
female_emp IF gender == 'F';
```

```
grunt> employees = LOAD 'report/employees.csv' USING PigStorage(',') AS (name: c
hararray, gender: chararray, country: chararray, salary: int);
2015-03-29 11:59:56,604 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> SPLIT employees INTO male_emp IF gender == 'M', female_emp IF gender ==
'F';
grunt>
```



Step 5: In the next step, we can either dump or store these relations which we have just split to check the result. You can also use these relations for further processing if required.

```
grunt> DUMP male_emp;
```

As shown in the screenshot below, we have the employees' data split based on male employees.

```
(Phelan,M,South Sudan,25933)
(Rylee,M,Namibia,16985)
(Karina,M,Zambia,94079)
(Emery,M,Bahamas,5231)
(Cadman,M,Czech Republic,73307)
(Hilda,M,France,58500)
(Rama,M,Morocco,22071)
(Lev,M,Dominican Republic,99135)
(Rashad,M,Ireland,28445)
(Quyn,M,Laos,42708)
(Blossom,M,Georgia,55429)
(Wendy,M,Antarctica,19760)
(Hashim,M,Pitcairn Islands,26447)
(Melodie,M,Nicaragua,19444)
(August,M,Guam,92188)
(Harlan,M,Ireland,92773)
(Marah,M,Saint Pierre and Miquelon,41720)
(Keaton,M,Albania,80244)
(Wang,M,Samoa,21123)
(Deirdre,M,Portugal,51010)
(Raven,M,Luxembourg,95025)
```

You may also dump or store the other relation *female_emp* to check out the result.

```
grunt> DUMP female_emp;
```

As shown in the screenshot below, we have the employees' data split based on male employees.



```
(Dalton,F,Ireland,91764)
(Grace,F,Namibia,7441)
(Blaze,F,Sierra Leone,12557)
(Dana,F,Holy See (Vatican City State),11711)
(Blossom,F,Chad,23880)
(Lionel,F,Cameroon,80919)
(Tanner,F,Ethiopia,50878)
(Portia,F,Jordan,7597)
(Mercedes,F,Namibia,52824)
(Phoebe,F,Algeria,28238)
(Elijah,F,Greece,73157)
(Shelley,F,Bermuda,12140)
(Camille,F,Nauru,30652)
(Cassady,F,Bangladesh,21870)
(Colby,F,Mexico,90938)
(Dillon,F,Poland,92562)
(Alana,F,Belize,62100)
(Ian,F,Côte D'Ivoire (Ivory Coast),56290)
```

Task 3 is complete!

Task 4: Using the RANK operator

Step 1: RANK operator is used to rank all the records in a relation sequentially. Let us rank the *employees* relation on the salary field which we have loaded in the previous task. This will result in ranking all the employees based on their salaries. Highest salary will be ranked 1 and so on. You can also choose to rank in ascending order instead of descending.

```
grunt> ranked = RANK employees BY salary DESC;
```

Please load the file employees.csv before you run this statement, if you have not already loaded. Please check previous task if you have not downloaded the file yet.

```
grunt> ranked = RANK employees BY salary DESC;
grunt>
```



Step 2: Dump or store the relation *ranked* to check the output.

```
grunt> DUMP ranked;
```

The result is as shown below.

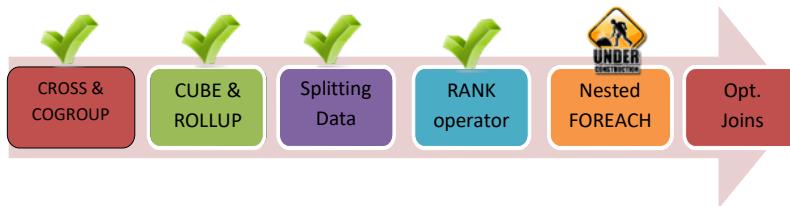
```
(1,Lionel,M,Sudan,99676)
(2,Francis,M,Serbia,99474)
(3,Lev,M,Dominican Republic,99135)
(4,Veronica,M,Ghana,95769)
(5,Raven,M,Luxembourg,95025)
(6,Bruce,M,Bolivia,95005)
(7,Karina,M,Zambia,94079)
(8,Sloane,M,Trinidad and Tobago,93998)
(9,Stacey,M,Equatorial Guinea,93821)
(10,Harlan,M,Ireland,92773)
(11,Dillon,F,Poland,92562)
(12,August,M,Guam,92188)
(13,Dalton,F,Ireland,91764)
(14,India,F,Malaysia,90954)
(15,Colby,F,Mexico,90938)
(16,Tatyana,M,Pakistan,90482)
(17,Basil,M,Christmas Island,88387)
(18,Kyla,F,Costa Rica,86581)
(19,Keefe,M,Niue,84433)
(20,Allegra,M,Bhutan,82160)
(21,Aspen,F,Suriname,81016)
```

Step 3: However, if there is a tie in salary for two or more employees, you will have the same rank position. The next rank position will have gaps. If you do not want to have gaps in the rank positions, you can use the **DENSE** keyword at the end as shown below.

```
grunt> ranked = RANK employees BY salary DESC
DENSE;
```

The relation *employees* may not have any tied positions and you might not be able to check this practically. However, you are free to apply the same on any other dataset.

Task 4 is complete!



Task 5: Working with Nested FOREACH

This task covers most of the important relational operators we have learned in this and previous chapter. This serves as a revision for all the important relational operators you have come across. If you do not understand the field position reference at any point, please use the DESCRIBE operator to check the same.

Step 1: Download the files *Sale.csv* and *Product.csv* from the following website and save it to your home directory. You might have already downloaded and copied the files *Sale.csv* in Task 1. You may skip it if you already have it. You only need to download the *Product.csv* file and save it to your home directory.

Sale.csv - <https://db.tt/8JEZ6M2z>
Product.csv - <https://db.tt/JIJqZDMg>

The file *Sale.csv* contains the sales information: Transaction ID, Date of Sale, Purchase ID, Date of Departure and Product ID. The file *Product.csv* contains the products information: Product ID, price, margin, Product Type, length, feature ID, Start Location and Serve Region.

We shall be using the Nested FOREACH and other relational operators and find the most purchased product in a service region and count of that product's quantity.

Step 2: Create a new directory in HDFS and name it 'report'. Copy all the files into the directory.

You need not create the directory and copy the files Sale.csv and Sale_Customer.csv in Task 1.

```
$ hadoop fs -copyFromLocal Sale.csv report/Sale.csv
$ hadoop fs -copyFromLocal Sale_Customer.csv
report/Sale_Customer.csv
$ hadoop fs -copyFromLocal Product.csv
report/Product.csv
```



Step 3: Load both the files into Pig by specifying the schema and types. First load the *Sale.csv* file as shown below.

```
grunt> sale = LOAD 'report/Sale.csv' USING
PigStorage(',') AS (transID: int, sale_date:
chararray, purchaseID: int, dep_date: chararray,
prod_id: int);
```

Then, load the *Product.csv* into Pig.

```
grunt> product = load 'report/Product.csv' USING
PigStorage(',') AS (prod_id: int, price: float,
margin: float, productType: chararray, length:
float, featureId: int, startLocation: chararray,
serveRegion: chararray);
```

```
grunt> sale = LOAD 'report/Sale.csv' USING PigStorage(',') AS (transID: int, sale_date: c
hararray, purchaseID: int, dep_date: chararray, prod_id: int);
2015-03-29 14:36:17,540 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - f
s.default.name is deprecated. Instead, use fs.defaultFS
grunt> product = load 'report/Product.csv' USING PigStorage(',') AS (prod_id: int, price:
float, margin: float, productType: chararray, length: float, featureId: int, startLocati
on: chararray, serveRegion: chararray);
2015-03-29 14:36:44,720 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - f
s.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 4: As a next step, let us join the relations *sale* and *product* by *prod_id*.

```
grunt> productTrans = JOIN sale BY prod_id, product
BY prod_id;
```

This way we have both the datasets joined. We can then extract the required columns and perform aggregations as required.



Step 5: Now, let us extract the columns which we are interested in. As we need to find out the most number of products purchased in a region, we will have to have the columns: *transID*, *prod_id* and *serveRegion*. We use the normal FOREACH operator and extract the columns we are interested in.

Check the schema using the DESCRIBE operator so that you get an idea how the schema is after performing the JOIN operation.

```
grunt> DESCRIBE productTrans;
```

```
grunt> DESCRIBE productTrans;
productTrans: {sale:::transID: int,sale:::sale_date: chararray,sale:::purchaseID: int,sale:::dep_date: chararray,sale:::prod_id: int,product:::prod_id: int,product:::price: float,product:::margin: float,product:::productType: chararray,product:::length: float,product:::featureId: int,product:::startLocation: chararray,product:::serveRegion: chararray}
grunt> ■
```

Now using FOREACH generate the required columns as highlighted above. Observe the use of Disambiguate operator (::) which helps you identify the fields of a relation when there are same alias names to fields in both the relations.

```
grunt> itemsets = FOREACH productTrans GENERATE
transID, sale:::prod_id, serveRegion;
```

```
grunt> DESCRIBE productTrans;
productTrans: {sale:::transID: int,sale:::sale_date: chararray,sale:::purchaseID: int,sale:::dep_date: chararray,sale:::prod_id: int,product:::prod_id: int,product:::price: float,product:::margin: float,product:::productType: chararray,product:::length: float,product:::featureId: int,product:::startLocation: chararray,product:::serveRegion: chararray}
grunt> itemsets = FOREACH productTrans GENERATE transID, sale:::prod_id, serveRegion;
grunt>
```

Step 6: Now that we have extracted the required fields, let us make sure we have the distinct records by removing same products purchased many times in the same transaction. Use the DISTINCT operator on *itemsets* relation to get all the distinct records.



```
grunt> distinctItemsets = DISTINCT itemsets;
```

Once we have the distinct records, let us group the relation *distinctItemsets* by *prod_id* and *serveRegion*. For simplicity, instead of using the field alias, you can use the position to refer the field. We have the *prod_id* and *serveRegion* on position 1 and 2.

```
grunt> inp = GROUP distinctItemsets BY ($1,$2);
```

```
grunt> distinctItemsets = DISTINCT itemsets;
grunt> inp = GROUP distinctItemsets BY ($1,$2);
grunt>
```

The schema for relation *inp* is now as shown below.

```
inp: {group: (sale::prod_id: int,
product::serveRegion: chararray), distinctItemsets:
{ (sale::transID: int, sale::prod_id: int,
product::serveRegion: chararray) }}
```

Step 7: Now use FOREACH again and generate group by flattening and count of number of products purchased. We flatten the group to remove a level of nesting and use COUNT function to count the number of products purchased.

```
grunt> interResult = FOREACH inp GENERATE
FLATTEN(group), COUNT(distinctItemsets.$1) as
count;
```

```
inp = GROUP distinctItemsets BY ($1,$2);
grunt> describe inp;
inp: {group: (sale::prod_id: int,product::serveRegion: chararray),distinctItemsets: { (sale::transID: int,sale::prod_id: int,product::serveRegion: chararray) }}
grunt> interResult = FOREACH inp GENERATE FLATTEN(group), COUNT(distinctItemsets.$1) as count;
grunt> 
```



The schema for relation *interResult* is now as shown below.

Step 8: We now have the count of each product purchased for a given service region. We now have to find which product purchased most in each service region with count. To do that, first we have to group the relation *interResult* by *regionServe*.

```
grunt> interResultGrp = GROUP interResult BY $1;
```

Now, we use the Nested FOREACH to sort the count and limit it by one record. Then we generate the group which is *regionServe*, product id and count which we have sorted and limited in the foreach block.

```
grunt> finalResult = FOREACH interResultGrp {  
  ord = ORDER interResult BY count DESC;  
  top = LIMIT ord 1;  
  GENERATE  
  group,FLATTEN(top.prod_id),FLATTEN(top.count);  
}
```

```
grunt> interResultGrp = GROUP interResult BY $1;  
grunt> finalResult = FOREACH interResultGrp {  
  >> ord = ORDER interResult BY count DESC;  
  >> top = LIMIT ord 1;  
  >> GENERATE group,FLATTEN(top.prod_id),FLATTEN(top.count);  
  >> }  
grunt>
```

Step 9: Dump or store the relation *finalResult* to check the product purchased most in each service region. This job might take a while to complete depending upon your machine.

```
grunt> DUMP finalResult;
```



The result is as shown in the screenshot below.

```

2015-03-29 17:02:11,557 [main] INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLauncher - Success!
2015-03-29 17:02:11,560 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-03-29 17:02:11,566 [main] INFO org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-03-29 17:02:11,609 [main] INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths to process : 1
2015-03-29 17:02:11,623 [main] INFO org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input paths to process : 1
( Asia,1034,53)
( Canada,2100,49)
( East Coast,1167,55)
( East Europe,2561,52)
( Midwest,2030,49)
( Mountain States,1057,49)
( Northeast,972,46)
( Pacific,2877,50)
( South,1534,51)
( South America,947,46)
( West Coast,1880,47)
( West Europe,1951,49)
grunt>

```

We finally have the region, product ID that was most purchased in that region and the count of that product's quantity.

Task 5 is complete!

Task 6: Optimizing Joins

In this task we optimize joins for better performance of our Hadoop cluster by smartly utilizing the available resources.

We shall be using the same files which we used in Task 5 of Lab Exercise 5. Please download the files and upload them to HDFS as instructed in steps 1 and 2 of Lab Exercise 5: Task 5.

Step 1: load both the files to Pig by entering the statements as shown below.



```
grunt> salaries = LOAD 'Pig/salaries.csv' USING
PigStorage(',') AS (year: int, team_id: chararray,
league_id: chararray, player_id: chararray, salary:
float);

grunt> star_all = LOAD 'Pig/AllstarFull.csv' USING
PigStorage(',') AS (player_id: chararray, year:
int, gameNum: int, gameID: chararray, team_id:
chararray, league_id: chararray, GP: chararray,
startingPos: chararray);
```

```
grunt> salaries = LOAD 'Pig/salaries.csv' USING PigStorage(',') AS (year: int, team_id: c
hararray, league_id: chararray, player_id: chararray, salary: float);
2015-03-29 17:15:58,814 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - f
s.default.name is deprecated. Instead, use fs.defaultFS
grunt> star_all = LOAD 'Pig/AllstarFull.csv' USING PigStorage(',') AS (player_id: chararr
ay, year: int, gameNum: int, gameID: chararray, team_id: chararray, league_id: chararray,
GP: chararray, startingPos: chararray);
2015-03-29 17:16:39,293 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - f
s.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

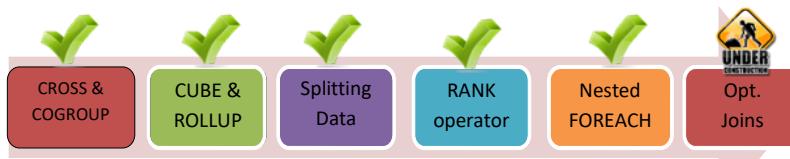
Step 2: Let us now perform a Fragment - Replicated Join. Run the following statement to perform a Fragment - Replicated Join. Please check the theory for Fragment - Replicated Joins to know when it should be used.

```
grunt> joined = JOIN salaries BY player_id,
star_all BY player_id USING 'replicated';
```

```
grunt> joined = JOIN salaries BY player_id, star_all BY player_id USING 'replicated';
grunt>
```

Step 3: Store or Dump the relation *joined* to check the result.

```
grunt> DUMP joined;
```



Similarly, check the theory for Skewed Joins, Merge Joins and Merge-Sparse Joins and perform the joins. Consider this as a Lab Challenge. This should help you understand the syntax and you would know when to use these joins to optimize your Pig queries.

Task 6 is complete!

LAB CHALLENGE

Challenge

Please complete the following lab challenges:

- Use COGROUP and CROSS operators on different datasets.
- Understand how Nested FOREACH works and apply it on other datasets by following the explanation provided in Theory.
- Use the Type Construction operator and construct all the three different types.
- In the labs we have covered Fragment – Replication Joins. Try to perform all the other advanced joins. Practice joins on different datasets.
- Practice all the Pig Latin operators specified in different datasets.

SUMMARY

Pig Latin is a high level relational Dataflow scripting language used to explore very large datasets. With Pig Latin, you can create a step-by-step data flow pipeline, where data is processed in every step. Each step in dataflow pipeline is known as a Pig statement or Relation. Pig Latin is a collection of statements or relations. Statements are built using operators, expressions and relations.

Relational Operators are the heart of Pig Latin. Relational operators are responsible for manipulating data by performing various extraction, transformation and loading operations. Let us look at each operator in detail and understand their importance in Pig Latin.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/basic.html>

INDEX

Relational Operators in Pig Latin - II	171
Nested FOREACH	179
Advanced Joins	181
Type Construction Operator	181
Disambiguate Operator	185
AIM	187
Lab Exercise 6: Hands On Pig Latin - Part -2.....	188
Task 1: Using COGROUP and CROSS	189
Task 2: Using CUBE and ROLLUP	195
Task 3: Using SPLIT operator to split data.....	201
Task 4: Using the RANK operator	205
Task 5: Working With Nested FOREACH	206
Task 6: Optimizing Joins.....	211
LAB CHALLENGE.....	214
SUMMARY	215
REFERENCES.....	216

CHAPTER 7: FUNCTIONS

Theory

We have covered the basics of Pig Latin in couple of the previous chapters. It is now time that we dig a little deeper and cover an important topic in Pig called *Functions*. We have used few of these functions while working in labs such as *PigStorage()*, *COUNT()* etc.

Functions are basically divided into two types:

- Built-In Functions
- User Defined Functions (UDFs)

Built-In Functions are the set of functions which get shipped directly with Pig. They are just Built-In. All you need to do is specify the name of the function and use it in your Pig Latin script. However, User Defined Functions are the functions which are not built-in with Pig and are developed by users for custom processing. If you do not find a Built-In function to process your data as per your requirement, you can develop a function using any of these six programming languages: Java, Jython, Python, JavaScript, Ruby and Groovy. You will then have to register the function in your Pig script and use the fully qualified name to refer the same.

We shall look at the various Built-In functions in this chapter. We shall then cover User Defined Functions in the upcoming chapters.

Built-In Functions can be categorized into six types as follows:

- Load/Store Functions
- Math Functions
- Eval Functions
- String Functions
- Datetime Functions
- Tuple, Bag, Map Functions

Let us look at these functions in detail. We have only listed the most important functions here as there are so many Built-In functions available. Please refer to the URL in REFERENCES section for the complete list. If you do not find a Built-in function here which can help you for processing the data as per your requirement, you can always write a UDF in any of the programming languages as mentioned above.

Load/Store Functions

Load/Store functions help you to load and store data to and from Pig. With the help of these functions, you can load and store almost every possible data format. If you have a format which is not available in these functions, you can write your own function.

PigStorage

We have been working with this function for quite a while now to load and store data into Pig. PigStorage is the default function used by Pig, meaning, it will be used even if you do not specify any function while loading or storing data to Pig.

PigStorage is used to load or store structured text files to and from Pig. The default delimiter to separate the fields is a tab character and can be changed according to the requirement. The input data can be a file, a directory or a glob. PigStorage also supports compression.

Points to remember when using PigStorage function:

- When data is loaded make sure the correct delimiter is specified, as Pig looks for a tab character by default or the explicitly specified delimiter.
- You can also specify a custom delimiter while storing or use the default delimiter which is again the tab character.
- Field delimiters can also be other characters. However, they should be represented in Unicode using UTF-16 encoding.
- Simple as well as Complex data types can be used with PigStorage.
- You can also specify to store the schema while loading data. The schema will be stored as a hidden file in the output directory. This is very much helpful when you have to load the data with same schema again and again.
- You may also choose not to load the stored schema, if a schema is already stored before. But if do not specify that you do not want to load schema, the schema gets attached automatically.
- Using the PigStorage function, you can tag the input path or file name of the file at the beginning of each record. A new column with input path or file name as specified will be added when the file gets loaded.

BinStorage

While PigStorage is used to load and store data which is in human readable files or structured text data, BinStorage is used to load and store machine readable data a.k.a. binary data. BinStorage is used by Pig to load as well as store the temporary data generated by multiple MapReduce jobs. Unlike PigStorage, BinStorage function does not take any parameters. The input data can be a file, a directory or a glob but BinStorage does not support compression.

Points to remember when using PigStorage function:

- BinStorage function does not take any parameters.
- Simple as well as Complex data types can be used with BinStorage.

HBaseStorage

HBaseStorage is used to load or store data from HBase tables to and from Pig. HBase is a Hadoop ecosystem project which is an open source, distributed, column-oriented, non-relational database build on top of Hadoop and HDFS. The HBaseStorage function takes two parameters. First is the list of columns separated by space and the second argument is optional used to specify options as desired.

For example, consider an *employees* table in HBase. We have two column families' *emp_details* and *billing_details*. The column family *emp_details* has the columns: *emp_id*, *name* and *age* while the column family *billing_details* has columns *rate_per_hour*, *hours_billed* and *total*. The following load statement is used to load the HBase table to Pig.

```
grunt> LOAD 'hbase://employees' USING
org.apache.pig.backend.hadoop.hbase.HBaseStorage(
  'emp_details:emp_id emp_details:name
  emp_details:age billing_details:rate_per_hour
  billing_details:hours_billed
  billing_details:total', '-loadKey=true -limit=10')
AS (id, emp_id: int, name: chararray, age: int,
  rate_per_hour: double, hours_billed: double, total:
  double);
```

We begin the load statement by specifying the table in HBase using the HBaseStorage function. The columns belonging to each specific column family is specified followed by the options. Here the option *loadKey* loads the row key as the first value in every tuple returned from HBase. It is by default set to *false*. Next we limit the maximum number of rows to retrieve per region to 10. There are many

other options which you can use with HBaseStorage. Check the link in REFERENCES for more information. Finally, we specify the schema.

You can also use the HBaseStorage function to store a relation in Pig to HBase. For example, first load the file *employees.csv* from HDFS to Pig.

```
grunt> emp = LOAD 'employees.csv' AS (id:  
bytearray, name: chararray, age: int, salary:  
double);
```

Now, store the relation in HBase using the HBaseStorage function as shown below.

```
grunt> STORE emp INTO 'hbase://employees' USING  
org.apache.pig.backend.hadoop.hbase.HBaseStorage  
'emp_details:name emp_details:age  
emp_details:salary';
```

As you can see, we load four fields when loading data to Pig but only three columns are specified when storing to HBase. This is because, the first column is always the HBase rowkey.

Other Load/Store Functions

Apart from the above Load/Store functions, we also have few other functions. Let us have a brief look at them.

JsonLoader: JsonLoader is used to load JSON data to Pig.

JsonStorage: JsonStorage is used to store JSON data to Pig. You can also assign schema to the data which is encoded in the standard JSON format for both JsonLoader and JsonStorage.

AvroStorage: AvroStorage is used to load and store data from and to Avro files. Avro is an advanced concept which is used for serialization of data.

TrevniStorage: TrevniStorage is similar to that of AvroStorage. TrevniStorage is used to load and store data from Trevni files. Trevni is a column oriented storage format.

TextLoader: TextLoader is used to load data in UTF-8 format.

PigDump: PigDump is used to store data in UTF-8 format.

AccumuloStorage: AccumuloStorage is used to load and store data from an Accumulo Table. Accumulo is a sorted, distributed key/value store which is a robust, scalable, high performance data storage and retrieval system.

OrcStorage: OrcStorage is used to load or store data from ORC files. ORC files are the Optimized Row Columnar file formats which improve the efficiency of storing Hive data.

Please check the URLs in REFERENCES for more information about all the above functions.

Eval Functions

Eval functions are the functions which evaluates one or more expression and return a result. All the functions below require you to pass expressions as arguments to the functions. Let us look at few of the most used Eval functions.

TOKENIZE

TOKENIZE function is used to split a string and output a bag of words as tuples. To understand this function, consider an example as shown below.

Let us load a file *pig_phil.txt* to Pig and dump it to see the contents.

```
grunt> phil = LOAD 'pig_phil.txt' AS (lines: chararray);  
grunt> DUMP phil;  
(Pigs Eat Anything)  
(Pigs Live Anywhere)  
(Pigs Are Domestic Animals)  
(Pigs Fly)
```

Now, let us use the TOKENIZE function with the FOREACH operator to split the split the lines (strings) to a bag of words as tuples.

```
grunt> token = FOREACH phil GENERATE TOKENIZE(lines);
```

Dump the relation *token* to check how TOKENIZE works.

```
grunt> DUMP token;  
({(Pigs), (Eat), (Anything)} )  
({(Pigs), (Live), (Anywhere)} )  
({(Pigs), (Are), (Domestic), (Animals)} )  
({(Pigs), (Fly)} )
```

Now that we have the lines tokenized as a bag of tuples, we can perform further aggregations such as count etc.

COUNT

COUNT is used to compute the number of elements in a bag. However, null elements are not counted with COUNT. Also, before you can use COUNT, you will have to use the GROUP ALL or GROUP BY operator accordingly, to group the elements you want to count.

COUNT_STAR

COUNT_STAR is similar to COUNT, except that it also counts the null elements. Use COUNT_STAR when you also need the count of null elements present in your data.

IsEmpty

IsEmpty is used to check if a bag or a map is empty. This function is pretty much useful if you want to filter out the empty bags or maps from your data.

SUM

SUM function helps you compute the sum of the numeric values in a single-column bag. NULL values are ignored by the SUM function. Similar to that of COUNT and COUNT_STAR, SUM function can only be used after the GROUP or GROUP ALL operator is applied on the data.

SUBTRACT

SUBTRACT function requires you to pass two bags as arguments and returns a new bag with tuples which are only present in the first bag but not in the second bag. Tuples which match in both the bags will not be returned. The passed arguments must be bags. If the arguments passed are not bags, you will be greeted by an error.

MAX

MAX is used to calculate the maximum of the numeric values or chararrays in a single-column bag. Similar to that of COUNT, COUNT_STAR and SUM, MAX function can only be used after the GROUP or GROUP ALL operator is applied on the data.

MIN

MIN does exactly the opposite of what MAX does. MIN is used to calculate the minimum of the numeric values or chararrays in a single-column bag. MIN also requires the data to be grouped before it is applied.

AVG

AVG is used to compute the average of numeric values in a single column bag. You have to group the data before you can apply the AVG function.

CONCAT

CONCAT function is used to concatenate to or more expression of the same data type.

These are few of the important eval functions used in Pig Latin. There are a couple functions more which can be accessed from the URL in REFERENCES section.

Math Functions

Math functions are used to perform all sorts of mathematical operations on your data. There is quite a big list of Math functions available. We shall cover few of the important Math functions here.

RANDOM

RANDOM function is used to return a random number of type double. The random number will be greater than or equal to 0.0 and less than 1.0.

CIEL

CIEL function is used if you would like to round up the value of an expression to nearest integer. For example, if we have a value of 7.3 and we use the CIEL function, we have the value of 7.3 rounded up to 8.

FLOOR

FLOOR function is exactly opposite of CIEL. FLOOR function will round down the value of an expression to nearest region. For example, if we have a value of 2.6 and we use the CIEL function, we have the value of 2.6 rounded up to 2.

ROUND

ROUND function is used to round off an expression to integer or long if the expression is float or double respectively. The values are rounded up to the nearest integer or long value. For example, if we have a value of 10.62 and we use the CIEL function, we have the value of 10.62 rounded up to 11.

ROUND_TO

ROUND_TO function gives you more power of rounding a value. With ROUND_TO you can round the decimal places of a value to a fixed number of decimal digits.

ABS

ABS function provides you with the absolute value of an expression. If the result is not negative ($x \geq 0$), the result is returned. If the result is negative ($x < 0$), the negation of the result is returned.

Above are few of the Math functions. Apart from these, we have many more math functions available. Please check the URL in REFERENCES section for more information.

Datetime Functions

Datetime functions are the functions which deal in manipulating date and time. You can use the date/time functions to format the date and time in several ways. There are Datetime functions to get the date or time in milliseconds, seconds, minutes hours, weeks, months etc.

ToDate

ToDate function is used to generate a DateTime object depending upon the parameters supplied.

You can generate a datetime object by converting it from a string and specifying the format you require. You can also generate the DateTime object in milliseconds.

There are many other similar DateTime functions which generate DateTime objects to milliseconds, to string, to unix time etc.

GetDay

GetDay function is used to extract the day of the month from a DateTime object. There are many other similar DateTime functions which extract data from DateTime objects such as GetHour, GetMilliSecond, GetMinute, getHour etc.

DaysBetween

DaysBetween function is used to return the number of days between two date objects. Similarly, there are functions which return the number of minutes, hours, seconds, milliseconds etc between two date objects.

There are many DateTime functions available and is not possible to list all of them here.

String Functions

String functions are used to manipulate or test against the strings according to the requirement. Let us look at few examples of String functions to get a basic idea about their usage.

LOWER

LOWER function converts the characters of an entire string to lower case.

UPPER

On contrary to LOWER function, UPPER function converts the characters of an entire string to upper case.

LTRIM

LTRIM function removes the leading white space from a string.

RTRIM

RTRIM does the opposite of LTRIM function. RTRIM function removes the trailing white space from a string.

TRIM

TRIM function on the other hand is combination of LTRIM and RTRIM functions. TRIM function removes the leading and trailing white spaces from a string.

The above are few String functions which help you manipulate the strings. This is not at all close to full list of String functions. Please refer to URL in REFERENCES section for complete list of String functions.

We shall also cover most of the String functions in our lab exercises.

Tuple, Bag, Map Functions

Tuple, Bag and Map functions are used to convert an expression or key-value pairs to tuples, bags and maps.

TOTUPLE

TOTUPLE function is used to convert one or more expressions to type tuple.

TOBAG

Similar to TOTUPLE function, TOBAG function converts one or more expressions to tuples and then places them in a bag.

TOMAP

TOMAP function converts a pair of key-value expressions to type Map. There are couple of points to remember before you use the TOMAP function. Make sure you supply even number of expressions as parameters. Also the key expressions should be of type chararray as only chararrays can be keys into the map and the value expression can be of any type.

TOP

TOP function is used to return top 'n' tuples from a bag of tuples. The TOP function takes three parameters. The first parameter is to specify the number of top tuples to

return. The second parameter is the column (field) for which the tuples need to be compared to and the third parameter is the relation. The top tuples are displayed in descending order by default. However, this can be changed to ascending order too.

We shall cover all these functions in our lab exercises.

That's all the theory we have for this chapter. Let us proceed to the Lab Exercises and get our hands on what we have just learned.

AIM

The aim of the following lab exercise is to have your hands on the built-in functions used in Pig.

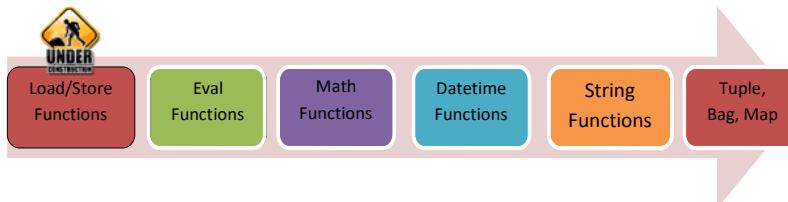
Following steps are required:

- Task 1: Using Load/Store Functions.
- Task 2: Using Eval Functions
- Task 3: Using Math Functions
- Task 4: Using Datetime Functions
- Task 5: Using String Functions
- Task 6: Using Tuple, Bag, Map Functions

Lab Exercise 7: HANDS ON FUNCTIONS



1. **Using Load/Store Functions**
2. **Using Eval Functions**
3. **Using Math Functions**
4. **Using Datetime Functions**
5. **Using String Functions**
6. **Using Tuple, Bag, Map Functions**



Please note that there are a lot of built-in functions available and is not possible to demonstrate each and every function here. We have only used few of the important built-in functions in the Lab Exercises to help you get an idea about their usage. Please check the URL in References section for complete list of built-in functions.

Task 1: Using Load/Store Functions

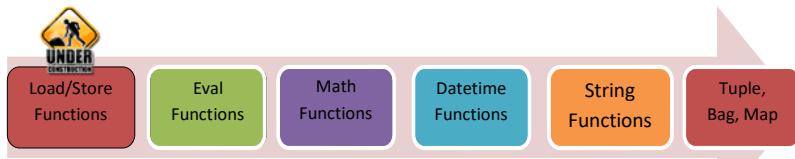
Step 1: Start all the Hadoop daemons as well as Pig in grunt mode. We shall be using the file *Sale.csv* to load into Pig using the *PigStorage* function. Please check Lab Exercise 6 Task 1 and Task 2 to get the file and save it in HDFS.

Step 2: Load the file to Pig using the *PigStorage* function by specifying the delimiter, schema.

```
grunt> sale = LOAD 'report/Sale.csv' USING
PigStorage(',') AS (transID: int, sale_date:
chararray, purchaseID: int, dep_date: chararray,
prod_id: int);
```

```
grunt> sale = LOAD 'report/Sale.csv' USING PigStorage(',') AS (transID: int, sal
e_date: chararray, purchaseID: int, dep_date: chararray, prod_id: int);
2015-04-15 23:32:58,249 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

In the statement above, we have loaded the file *Sale.csv* into the relation *sale* using the *PigStorage* function by specifying *comma* as the delimiter. We specify a *comma* as a delimiter because, the columns in the file *Sale.csv* are separated with a comma. You can choose to specify any other delimiter according to the delimiter you have in your input. However, if your input data columns are separated by a tab character, you need not specify the delimiter as tab character, i.e., you need not mention *PigStorage('/t')* in the statement above as *PigStorage* function takes a tab character as a delimiter by default. Next, we specify the schema for the data so that we can reference the fields by a name rather than the position.



Step 3: Optionally, you can add a tag as first column to the data you load using the `tagPath` or `tagFile` options. This tagging helps you in identifying the path of the file or the name of the file to which the data belongs to so that you can further process the data accordingly.

The same load statement from the previous task looks as shown below with the tagging options included. Please note that you will also have to specify the name of the column in the schema.

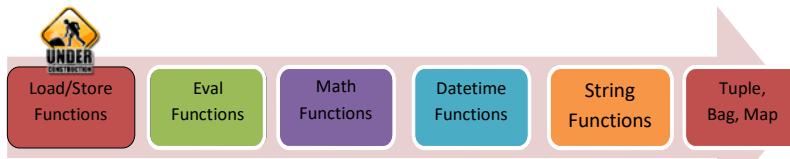
```
grunt> sale = LOAD 'report/Sale.csv' USING
PigStorage(',', '-tagPath') AS (filepath, transID:
int, sale_date: chararray, purchaseID: int,
dep_date: chararray, prod_id: int);
```

```
grunt> sale = LOAD 'report/Sale.csv' USING PigStorage(',', '-tagPath') AS (filepath, transID: int, sale_date: chararray, purchaseID: int, dep_date: chararray, prod_id: int);
2015-04-16 06:21:14,310 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

You can also use the option `tagFile` instead to add a first column containing the name of the file(s) you load to Pig as shown below.

```
grunt> sale = LOAD 'report/Sale.csv' USING
PigStorage(',', '-tagFile') AS (filename, transID:
int, sale_date: chararray, purchaseID: int,
dep_date: chararray, prod_id: int);
```

```
grunt> sale = LOAD 'report/Sale.csv' USING PigStorage(',', '-tagFile') AS (filename, transID: int, sale_date: chararray, purchaseID: int, dep_date: chararray, prod_id: int);
2015-04-16 06:23:30,024 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```



Step 4: Now that we have seen how to load a file to Pig using the `PigStorage` function, let us see how to store it.

We can store the above relation `sale` to a file using the following statement along with the schema. We can also change the delimiter from `comma` to any other character. We shall use the `'|'` (pipe) character as delimiter instead of `,` (comma).

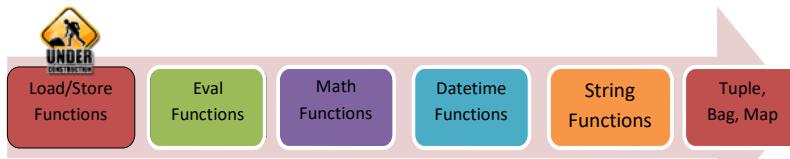
```
grunt> STORE sale INTO 'store' USING
PigStorage('|', '-schema');
```

```
grunt> sale = LOAD 'report/Sale.csv' USING PigStorage(',', '-tagFile') AS (trans
ID: int, sale_date: chararray, purhaseID: int, dep_date: chararray, prod_id: int
);
2015-04-16 01:13:41,939 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> STORE sale INTO 'store' USING PigStorage('|', '-schema');
```

The above statement stores the relation `'sale'` into a file called `'store'` along with the schema. Make sure the output directory is not available already. If it is available, the job fails. The fields are now separated by a `'|'` (pipe) character instead of a `,` (comma). You may also choose to use other characters as field delimiters. For such characters, you will have to represent them in Unicode using the UTF-16 encoding. For example, to use a π sign as a delimiter, you will have to specify the Unicode of π which is `\u03A0` as shown below.

```
grunt> STORE sale INTO 'store1' USING
PigStorage('\u03A0', '-schema');
```

Once it is stored, you can check out the output.



We save the schema using `-schema` because, if we were to load similar data from the same directory, we need not type the schema again as the schema gets attached automatically. You can also check the `.pig_schema` file in output directory which contains the schema we just saved. However, if you do not want to load the schema automatically when you load this while to Pig, you will have to specify `noSchema` option in the load statement.

For example, if you were to load the file to Pig for further processing, you need not mention all the schema again. Your load statement will look something like below.

```
grunt> sale = LOAD 'report/Sale.csv' USING
PigStorage('\u03A0');
```

But, if you do not want to have that schema loaded automatically, all you need to mention is the `noSchema` option as shown below.

```
grunt> sale = LOAD 'report/Sale.csv' USING
PigStorage('\u03A0', '-noSchema');
```

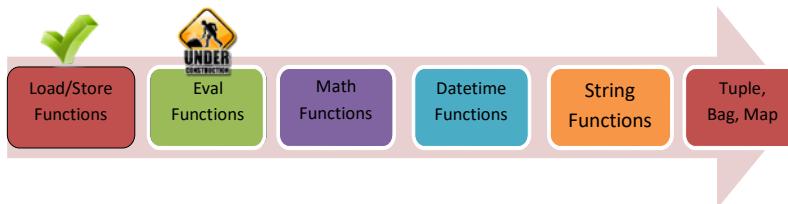
This completes the demo on PigStorage function. Now that you know how to deal with Load/Store functions, try to load the data using other Load/Store functions as a Lab challenge. Please use the link in REFERENCES section if you get stuck in-between.

Task 1 is complete!

Task 2: Using Eval Functions

To work with Eval functions, let us write a Pig Latin script to find the count of each word in the input file.

Step 1: Let us use the book *Pride and Prejudice* by Jane Austen and count how many times each word has been repeated throughout the book. Please download this book from the URL below and save it to your home directory.



<https://db.tt/UGPx4xJn>

Step 2: Create a new directory in HDFS and name it *books*. Ignore if you have already created in the previous tasks. Copy the book you have just downloaded to that folder in HDFS by running the following command. Make sure you have started all the Hadoop daemons.

```
$ hadoop fs -mkdir books
$ hadoop fs -copyFromLocal pg1342.txt books/pg1342.txt
```

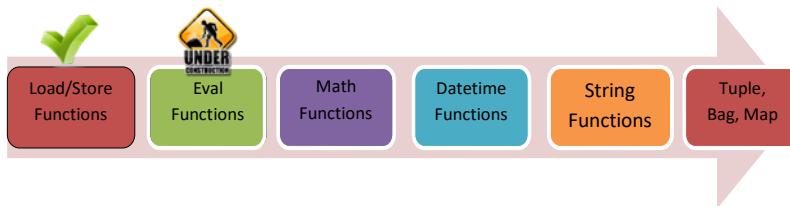
Step 3: Load the file to Pig using PigStorage function as shown below.

```
grunt> book = LOAD 'books/pg1342.txt' USING
PigStorage() AS (lines:chararray);
```

```
grunt> book = LOAD 'books/pg1342.txt' USING PigStorage() AS (lines:chararray);
2015-04-19 00:26:45,822 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 4: Dump the relation to screen to check how the loaded file looks like using the DUMP operator as shown below. The output is as shown in the screenshot below.

```
grunt> DUMP book;
(As it happened that Elizabeth had _much_ rather not, she endeavoured in)
(her answer to put an end to every entreaty and expectation of the kind.)
(Such relief, however, as it was in her power to afford, by the practice)
(of what might be called economy in her own private expences, she)
(frequently sent them. It had always been evident to her that such an)
(income as theirs, under the direction of two persons so extravagant in)
(their wants, and heedless of the future, must be very insufficient to)
(their support; and whenever they changed their quarters, either Jane or)
(herself were sure of being applied to for some little assistance)
(towards discharging their bills. Their manner of living, even when the)
(restoration of peace dismissed them to a home, was unsettled in the)
(extreme. They were always moving from place to place in quest of a cheap)
(situation, and always spending more than they ought. His affection for)
(her soon sunk into indifference; hers lasted a little longer; and)
(in spite of her youth and her manners, she retained all the claims to)
(reputation which her marriage had given her.)
```



Step 5: Each line is a tuple in the above output. In order to count word repetition count, we first have to tokenize each word as a tuple. Remember, TOKENIZE is used to split a string and output a bag of words as tuples. Then we use flatten to remove the extra layer of tuple so that we end up with each word as a tuple. Run the following Pig statement to achieve the same.

```
grunt> words = FOREACH book GENERATE
FLATTEN(TOKENIZE(lines)) AS word;
```

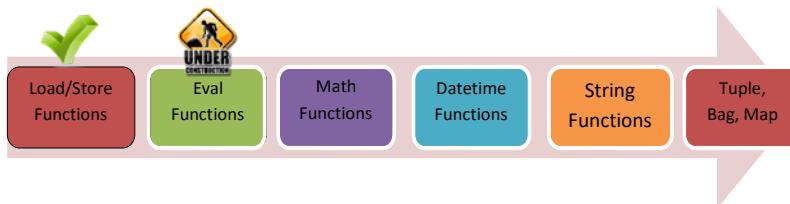
```
grunt> words = FOREACH book GENERATE FLATTEN(TOKENIZE(lines)) AS word;
grunt>
```

Again, dump the relation *words* and check out how the output is at this time.

```
grunt> DUMP words;
```

```
(principal)
(office)
(is)
(located)
(at)
(4557)
(Melan)
(Dr.)
(S.)
(Fairbanks)
(AK)
(99712.)
(but)
(its)
(volunteers)
(...)
```

As you can see from the screenshot above, each line is split and into each word as tuples. You may also experiment the above statement without using the FLATTEN operator and check out the output to understand why we have had to use the FLATTEN operator.



Step 6: Now to perform the count of each word, we have to first somehow group similar words as a group. So, we use the GROUP operator as shown below.

```
grunt> grouped = GROUP words BY word;
```

```
grunt> grouped = GROUP words BY word;
grunt>
```

You may choose to dump the relation *grouped* to screen. However, we continue to the next step.

Step 7: Now that we have all the similar words grouped, we can just count number of times the words has been repeated and output the count using the COUNT function.

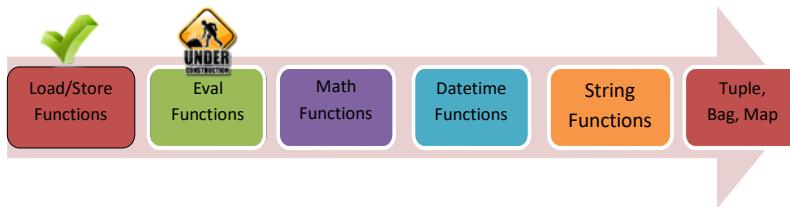
```
grunt> count = FOREACH grouped GENERATE group,
COUNT(words);
```

```
grunt> grouped = GROUP words BY word;
grunt> count = FOREACH grouped GENERATE group, COUNT(words);
grunt>
```

If you like to include the nulls, you should use the function COUNT_STAR instead of COUNT.

```
grunt> count = FOREACH grouped GENERATE group,
COUNT_STAR(words);
```

Dump the relation *count* and you should have the count of each word used in the book. We have the output as shown in the screenshot below. You can see that the output is not accurate. Similar words having different punctuations are treated as different words. We can definitely clean the punctuations and get the accurate output. We shall perform all the cleaning when we deal with the String functions.



```
(Hertfordshire,29)
(International,1)
(Unfortunately,1)
(Wickham--when,1)
(_Boulanger_--,1)
(accomplished!,1)
(accomplished.,3)
(acknowledged.,1)
(acknowledged;,1)
(acknowledging,1)
(acquaintance.,10)
(acquaintances,9)
(acquiescence.,2)
(advantageous.,1)
(altered--what,1)
(amazing!--but,1)
```

Similarly, you can use all the other eval functions based upon your requirement by carefully checking the conditions for each function.

Task 2 is complete!

Task 3: Using Math Functions

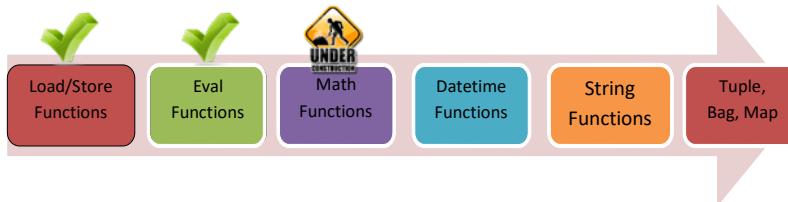
Step 1: Download this file *mathfunc.csv* from the URL below to work on this task and save it in the Home Directory. The file contains the fields of type int, int and float.

<https://db.tt/Ni44xcWn>

Step 2: Copy the file to the directory *Pig* in HDFS and load the file to Pig.

```
grunt> A = LOAD 'Pig/mathfunc.csv' USING
PigStorage(',') AS (a: int, b: int, c: float);
```

```
grunt> A = LOAD 'Pig/mathfunc.csv' USING PigStorage(',') AS (a: int, b: int, c:
double);
2015-04-20 02:07:09,745 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```



Step 3: Now you can apply any possible Math function to the fields above. For instance, let us apply the Square root function to field *a*, absolute function to field *b* and Round function on field *c*.

```
grunt> B = FOREACH A GENERATE SQRT(a), ABS(b),  
ROUND(c);
```

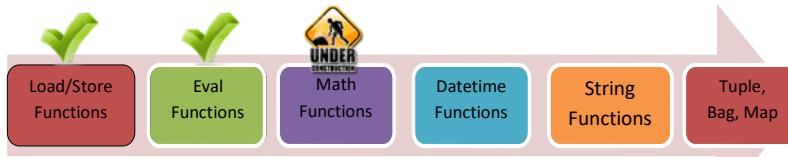
The result is as shown below when you dump the relation *B*.

```
ne.util.MapRedUtil - Total input paths to process : 1  
(1.0,0,0)  
(1.4142135623730951,1,1)  
(1.7320508075688772,2,1)  
(2.0,3,2)  
(2.23606797749979,4,2)  
(2.449489742783178,5,3)  
(2.6457513110645907,6,3)  
(2.8284271247461903,7,4)  
(3.0,8,4)  
(3.1622776601683795,9,5)  
(3.3166247903554,10,5)  
(3.4641016151377544,11,6)  
(3.605551275463989,12,6)  
(3.7416573867739413,13,7)  
(3.872983346207417,14,7)  
(4.0,15,8)
```

Step 4: You can also use the trigonometric math functions on this data too. Let us apply the Sin, Cos and Tan functions for instance.

```
grunt> C = FOREACH A GENERATE SIN(a), COS(b),  
TAN(c);
```

The result is as shown below when you dump the relation *C*.



```
(0.8414709848078965,1.0,0.0)
(0.9092974268256817,0.5403023058681398,0.5463024898437905)
(0.1411200080598672,-0.4161468365471424,1.5574077246549023)
(-0.7568024953079282,-0.9899924966004454,14.101419947171719)
(-0.9589242746631385,-0.6536436208636119,-2.185039863261519)
(-0.27941549819892586,0.28366218546322625,-0.7470222972386603)
(0.6569865987187891,0.9601702866503661,-0.1425465430742778)
(0.9893582466233818,0.7539022543433046,0.3745856401585947)
(0.4121184852417566,-0.14550003380861354,1.1578212823495777)
(-0.5440211108893698,-0.9111302618846769,4.637332054551185)
(-0.9999902065507035,-0.8390715290764524,-3.380515006246586)
(-0.5365729180004349,0.004425697988050785,-0.995584052213885)
(0.4201670368266409,0.8438539587324921,-0.29100619138474915)
(0.9906073556948704,0.9074467814501962,0.22027720034589682)
(0.6502878401571168,0.1367372182078336,0.8714479827243187)
(-0.2879033166650653,-0.7596879128588213,2.706013866772691)
(-0.9613974918795568,-0.9576594803233847,-6.799711455220379)
(-0.7509872467716762,-0.27516333805159693,-1.326364327785607)
```

Similarly, apply and play with all the other possible math functions. This, of course, is just a sample data to demonstrate how the math functions work. Try these functions on other data too.

Task 3 is complete!

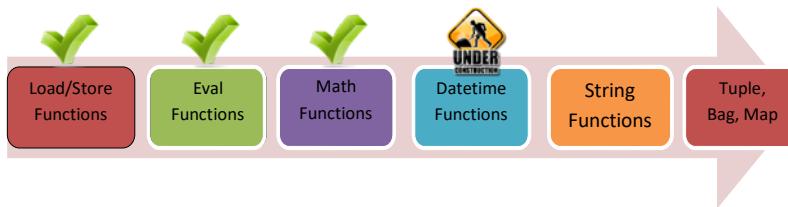
Task 4: Using Datetime Functions

Step 1: Download this file *datetimefunc.csv* from the URL below to work on this task and save it in the Home Directory. The file contains one date per line.

<https://db.tt/EdypvOVA>

Step 2: Copy the file to the directory *Pig* in HDFS and load the file to Pig.

```
grunt> A = LOAD 'Pig/datetimefunc.csv' AS (inp: chararray);
```



```
grunt> A = LOAD 'Pig/datetimefunc.csv' AS (inp: chararray);
2015-04-20 04:48:56,930 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 3: Now, let's use the `ToDate` function and convert the date which is of type `chararray` to type `datetime`.

```
grunt> B = FOREACH A GENERATEToDate(inp,
'yyyy/MM/dd') AS (dt:datetime);
```

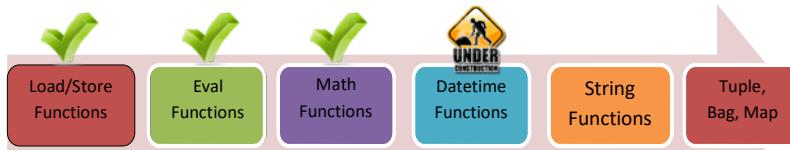
Please note the uppercase and lowercase letters in the date format. **D** means Day in year but **d** refers to Day in month. Similarly **M** means Month in year while **m** refers to Minute in hour.

```
grunt> A = LOAD 'Pig/datetimefunc.csv' AS (inp: chararray);
2015-04-20 12:04:07,913 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> B = FOREACH A GENERATEToDate(inp, 'yyyy/MM/dd') AS (dt:datetime);
grunt>
```

Step 4: Now let us use the `DaysBetween` function to output the dates greater than 2015/08/09 and filter the rest.

```
grunt> C = FILTER B by DaysBetween(dt,
(datetime)ToDate('2015/08/09', 'yyyy/MM/dd')) >=
(long) 0;
```

Please see the explicit casting for the `datetime` and `long` types. We use it to inform Pig to cast the date to type `datetime` and 0 to `long` type.



```
grunt> A = LOAD 'Pig/datetimefunc.csv' AS (inp: chararray);
2015-04-20 12:04:07,913 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> B = FOREACH A GENERATE ToDate(inp, 'yyyy/MM/dd') AS (dt:datetime);
grunt> C = FILTER B by DaysBetween(dt, (datetime)ToDate('2015/08/09', 'yyyy/MM/dd')) >= (long)0;
grunt>
```

You can now dump the relation *C* to check the result.

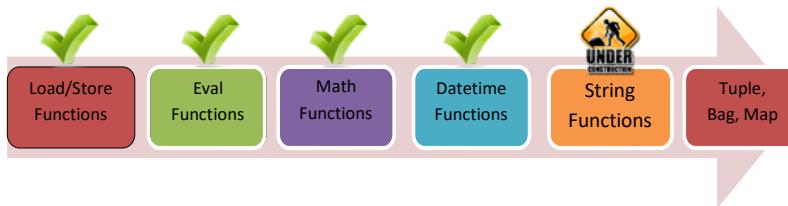
```
grunt> DUMP C;
```

```
(2015-09-08T00:00:00.000-07:00)
(2016-01-09T00:00:00.000-08:00)
(2016-04-09T00:00:00.000-07:00)
(2015-09-19T00:00:00.000-07:00)
(2015-08-22T00:00:00.000-07:00)
(2015-09-19T00:00:00.000-07:00)
(2015-11-15T00:00:00.000-08:00)
(2016-04-08T00:00:00.000-07:00)
(2015-11-21T00:00:00.000-08:00)
(2016-02-07T00:00:00.000-08:00)
(2015-11-12T00:00:00.000-08:00)
(2015-09-05T00:00:00.000-07:00)
(2015-12-17T00:00:00.000-08:00)
(2015-11-03T00:00:00.000-08:00)
(2015-08-29T00:00:00.000-07:00)
(2015-10-23T00:00:00.000-07:00)
```

As you can see in the result above, we have the output dates greater than 2015/08/09.

There are tons of datetime functions available. You may use them according to your requirement.

Task 4 is complete!



Task 5: Using String Functions

For this task we shall again run the word count script which we have used during Task 2: Using Eval Functions. We shall use the String functions to achieve more accurate results.

Please follow the steps 1-4 in Task 2: Using Eval Functions before proceeding with this task.

Step 1: Once the file is loaded, let us use the String function TRIM function to remove the leading and trailing white spaces from a string.

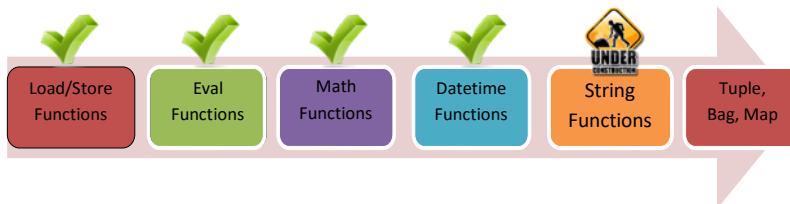
```
grunt> trimmed = FOREACH book GENERATE TRIM(lines)
AS trimmed_lines;
```

```
grunt> book = LOAD 'books/pg1342.txt' USING PigStorage() AS (lines:chararray);
2015-04-20 12:21:49,165 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> trimmed = FOREACH book GENERATE TRIM(lines) AS trimmed_lines;
grunt>
```

Step 2: Now convert all the strings to lower case using the LOWER function, so that same words with different case are not counted again and again.

```
grunt> lowered = FOREACH trimmed GENERATE
LOWER(trimmed_lines) AS lowered_lines;
```

```
grunt> book = LOAD 'books/pg1342.txt' USING PigStorage() AS (lines:chararray);
2015-04-20 12:21:49,165 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> trimmed = FOREACH book GENERATE TRIM(lines) AS trimmed_lines;
grunt> lowered = FOREACH trimmed GENERATE LOWER(trimmed_lines) AS lowered_lines;
grunt>
```



Step 3: Now that we have converted all the strings to lower case, we are now left with the punctuations. Let us remove the punctuations using the REPLACE function so that same words with different punctuations are not counted again and again.

REPLACE function is used to replace existing characters in a string with new characters. The REPLACE function takes three arguments. The first argument is the string which has to be updated. Second is the regular expression to which the string is to be matched, in quotes and the third is the new characters replacing the existing characters, in quotes.

For our example, we specify the string which is *lowered_lines* in this case as the first argument. Punctuations which have to be replaced as second argument with an empty string as third argument.

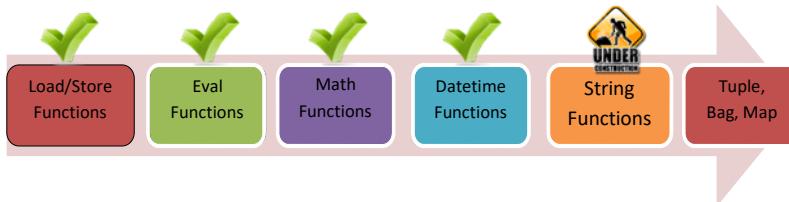
```
grunt> cleaned = FOREACH lowered GENERATE
REPLACE(lowered_lines, '[\p{Punct},\p{Cntrl}]',
'') AS cleaned_lines;
```

```
grunt> cleaned = FOREACH lowered GENERATE REPLACE(lowered_lines, '[\p{Punct},\p{Cntrl}]', '') AS cleaned_lines;
grunt>
```

Step 4: The rest of the steps are similar to that of those in the task 2. All we have to do is flatten and tokenize the tuples and then count the number of times each word has repeated.

```
grunt> words = FOREACH cleaned GENERATE
FLATTEN(TOKENIZE(cleaned_lines)) AS word;
```

Please make a note that we could have also performed all the data cleansing in a single line of code instead of four lines as shown below. We have broken this cleansing process in parts for easy understanding.



```
grunt> words = FOREACH book GENERATE
FLATTEN(TOKENIZE(REPLACE(LOWER(TRIM(lines)),
`[\p{Punct},\p{Cntrl}]', ''))) AS word;
```

Next, group the *words* by *word* and generate the count for each word.

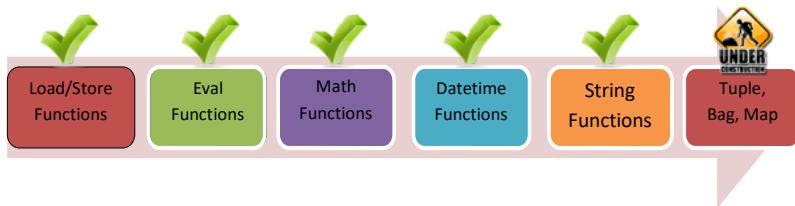
```
grunt> grouped = GROUP words BY word;
grunt> count = FOREACH grouped GENERATE group,
COUNT(words);
```

Dump the relation *count* to check out the accurate word count of the words in the book.

```
(understanding,20)
(unembarrassed,2)
(unexpectedfor,1)
(unfortunately,3)
(unjustifiable,2)
(unnecessarily,1)
(unwillingness,3)
(variancethere,1)
(wateringplace,1)
(whatsomething,1)
(worthlessness,1)
(accomplishment,3)
(acknowledgment,3)
(acquaintancean,1)
(advantageously,2)
(affectionately,4)
(characteristic,2)
(characterthere,1)
```

Similarly, use all the other string functions accordingly when required.

Task 5 is complete!



Task 6: Using Tuple, Bag, Map Functions

ToTuple

Step 1: As a continuation to the previous task, let us convert the result which is the word and its count to tuple. Run the following statement to do so using the TOTUPLE function.

```
grunt> tupled = FOREACH count GENERATE TOTUPLE ($0, $1);
```

Dump the relation *tupled* to check the output.

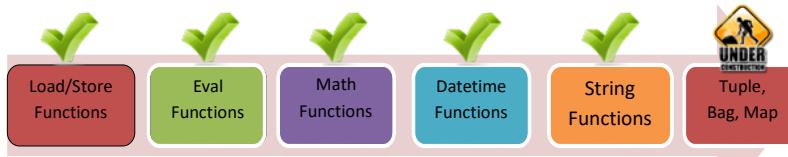
```
grunt> DUMP tupled;
```

The result is as shown in the screenshot below.

```
((condescendingly,1))
((congratulations,12))
((conscientiously,1))
((disappointments,1))
((fellowcreatures,2))
((gbnewbypglaforg,1))
((generoushearted,1))
((guardianshiphow,1))
((halfexpectation,1))
((hardheartedness,1))
((httpwwwpglaforg,1))
```

ToBag

Step 1: Similar to the TOTUPLE function, we can convert an expression to individual tuples and then place them in a bag. Run the following statement to do so using the TOBAG function.



```
grunt> bagged = FOREACH count GENERATE TOBAG($0, $1);
```

Dump the relation *bagged* to check the output.

```
grunt> DUMP bagged;
```

The result is as shown in the screenshot below.

```
((premeditation),(1))
((prepossession),(2))
((probabilities),(1))
((proportionate),(1))
((purchaseslook),(1))
((qualification),(1))
((recollections),(8))
((reprehensible),(3))
((retrospective),(1))
((satisfactions),(1))
((selfattracted),(1))
```

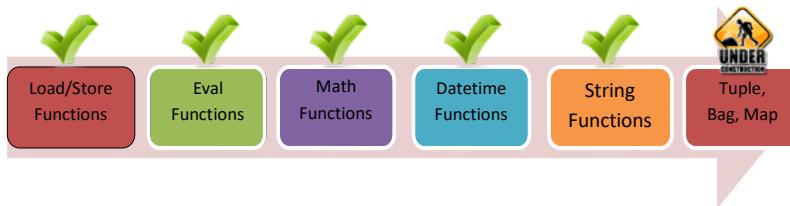
ToMap

Step 1: The TOMAP function is no different than the TOTUPLE and TOMAP functions. TOMAP function converts a pairs of expressions into a map. Run the following statement to do so using the TOBAG function.

```
grunt> mapped = FOREACH count GENERATE TOMAP($0, $1);
```

Dump the relation *bagged* to check the output.

```
grunt> DUMP mapped;
```



The result is as shown in the screenshot below.

```
([congratulation#1])  
([congratulatory#1])  
([considerations#1])  
([contemptuously#1])  
([contradictions#1])  
([correspondence#5])  
([correspondents#1])  
([counterbalance#1])  
([desiresplendid#1])  
([disappointment#17])  
([disapprobation#7])  
([discontinuance#1])  
([discretion#1])
```

Task 6 is complete!

LAB CHALLENGE

Challenge

Please complete the following lab challenges:

- Try to load other Load/Store functions apart from PigStorage.
- Try to use all the built-in functions found in the URL in REFERENCES section.

SUMMARY

Functions are the essential part of any programming language and Pig is no exception. There are two types of functions in Pig. They are the Built-in functions and User defined functions (UDFs). Built-In functions are further divided into Load/Store functions, Eval Functions, Math functions, Datetime functions, String functions and Tuple, Bag, Map functions.

We have covered the Built-In functions for this chapter. User defined functions shall be covered in the upcoming chapters.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/func.html>

INDEX

Load/Store Functions	218
Eval Functions	222
Math Functions	224
Datetime Functions	225
String Functions	225
Tuple, Bag, Map Functions	226
AIM	228
Lab Exercise 7: Hands On Functions	229
Task 1: Using Load/Store Functions	230
Task 2: Using Eval Functions	233
Task 3: Using Math Functions	237
Task 4: Using DateTime functions	239
Task 5: Using String Functions	242
Task 6: Using Tuple, bag, Map Functions	245
LAB CHALLENGE	248
SUMMARY	249
REFERENCES	250

CHAPTER 8: USER DEFINED FUNCTIONS – PART 1

Theory

In the previous chapters, we have been working with the operators and functions which ship directly with Pig. In this chapter we extend the functionality of Pig by writing our own functions according to the requirements. These functions are known as User Defined Functions or simply UDFs. User Defined Functions helps you write your own custom function to process your data. There might be cases where you cannot find a built-in function suitable for your needs and this is where a user defined function helps you. Pig UDFs can be written and implemented in Java, Jython, Python, JavaScript, Ruby and Groovy. However, since Pig is written in Java, UDFs written in Java are most efficient with greater support while the others have limited support. With UDFs every part of processing can be customized including data load/store, column transformation, and aggregation.

We can write UDFs for Eval, Filter and Load/Store functions. In this chapter, we shall learn how to write UDFs for Eval and Filter functions. The next chapter will cover the Load/Store UDFs. We use Java to write the user defined functions throughout this and the upcoming chapter as Pig has the most extensive support for Java.

Eval Functions

Eval functions are one of the most used User Defined Functions in Pig. As we know, eval functions evaluate one or more expression and return a result. All the eval functions extend the Java class `org.apache.pig.EvalFunc` which is the base class for all the eval functions.

We shall look at writing the eval functions in the lab exercise.

Algebraic Interface

Using the advantage of Hadoop's combiner is always recommended as it lowers the amount of data transmitted over the network between the map and reduce tasks as well as reduces the load on reducers. The advantage of using a combiner is much more when we are working on the sets of data on which we intend to aggregate and then compute. There are two types of functions where a combiner can be used. They are the Distributive function and the Algebraic function.

The best example of a distributive function is the SUM function. The SUM function is directly applied to the input data and obtain the result or applied to the subsets of input and again applied on the result of subsets to obtain the final result. However, this is not the case with algebraic function. A function is an algebraic function only

when it can be divided into initial, intermediate and final functions where the final function is possibly different from the initial function. The best example for this is the COUNT function, in which count is used for the initial function while the sum is used as the intermediate and final functions. Because, when the count function is used, internally the similar input data is first counted in the map phase (initial), summed in the combiner phase (intermediate) and then again summed during the reducer phase (final).

An Eval function is algebraic when it implements the Algebraic Interface. The Algebraic interface consists of three functions that allow our UDF to declare initial, intermediate, and final classes. These classes should extend *evalFunc*. The three functions are the *getInitial()*, *getIntermed()* and *getFinal()* functions.

The Algebraic interface look like this:

```
public interface Algebraic{  
    public String getInitial();  
    public String getIntermed();  
    public String getFinal();  
}
```

These functions return the name of the class where the actual work gets done. Let us look at implementation of COUNT function to understand this concept better. The first step is to extend the *evalFunc* class and implement the algebraic interface.

```
public class COUNT extends EvalFunc<Long> implements  
Algebraic {
```

Next, we specify the algebraic functions to return the names of the classes as shown below.

```
public String getInitial() {  
    return Initial.class.getName();  
}  
public String getIntermed() {  
    return Intermediate.class.getName();  
}  
public String getFinal() {  
    return Final.class.getName();  
}
```

We now need build the Initial, Intermediate and Final classes. Let us look at the implementation of the Initial class.

```

static public class Initial extends EvalFunc<Tuple> {

    public Tuple exec(Tuple input) throws IOException {
        DataBag bag = (DataBag) input.get(0);
        Iterator it = bag.iterator();
        if (it.hasNext()) {
            Tuple t = (Tuple) it.next();
            if (t != null && t.size() > 0 && t.get(0) != null)
                return mTupleFactory.newTuple(Long.valueOf(1));
        }
        return mTupleFactory.newTuple(Long.valueOf(0));
    }
}

```

The `exec` function of the `Initial` class is called with an input of a bag with single tuple. It then verifies for null records in the bag. If it finds a null, it returns zero and if it does not, it returns one. The return type of the initial is a tuple of type `Long`.

Next is the Intermediate class. The Intermediate class calculates the sum of the counts produced in the Initial phase. The implementation of intermediate class is as follows:

```

static public class Intermediate extends EvalFunc<Tuple> {
    public Tuple exec(Tuple input) throws IOException {
        try {
            return mTupleFactory.newTuple(sum(input));
        } catch (ExecException ee) {
            throw ee;
        } catch (Exception e) {
            int errCode = 2106;
            String msg = "Error while computing
count in " + this.getClass().getSimpleName();
            throw new ExecException(msg, errCode,
PigException.BUG, e);
        }
    }
    static protected Long sum(Tuple input) throws
ExecException, NumberFormatException {
        DataBag values = (DataBag) input.get(0);
        long sum = 0;
        for (Iterator<Tuple> it = values.iterator();
it.hasNext();) {
            Tuple t = it.next();

```

```

        sum += (Long)t.get(0);
    }
    return sum;
}

```

The Intermediate phase has the output of initial phase as input. The sum of the counts obtained from initial phase is performed in the intermediate phase. As you can see from the code above, the sum is returned as output. The intermediate `exec` function may be called zero, one or more times.

Lastly, we have to implement the Final class. The implementation of Final class is as shown below.

```

static public class Final extends EvalFunc<Long> {

    public Long exec(Tuple input) throws IOException {
        try {
            return sum(input);
        } catch (Exception ee) {
            int errCode = 2106;
            String msg = "Error while computing count
in " + this.getClass().getSimpleName();
            throw new ExecException(msg, errCode,
PigException.BUG, ee);
        }
    }
}

```

The `exec` function of *Final* class is called in the reduce phase and is guaranteed to be called only one time. The sum of counts obtained from the intermediate phase is again performed in the final phase and produces the end result.

Accumulator Interface

Accumulator Interface in Pig is used to decrease memory usage by passing subsets of input instead of passing all the input at once to a function. This is very much useful when there is huge amount of data to be passed through a function but the function does not require to read all the data at once. When the accumulator interface is implemented, instead of calling the UDF once with the entire input set in one bag, Pig calls the UDF multiple times with subset of the records. Once all the records are passed, the result is asked and finally, cleanup function is called to reset its state before the records for the next group are passed.

The Accumulator interface look like this:

```
public interface Accumulator <T> {  
  
    /* Tuples are passed to the UDF. The tuples inside  
    the bag may be zero or more for the current key*/  
  
    public void accumulate(Tuple b) throws IOException;  
  
    /* Called when all tuples from current key have been  
    passed to the accumulator.  
     * @return the value for the UDF for this key. */  
  
    public T getValue();  
  
    /* Called after getValue() to prepare processing for  
    next key. */  
  
    public void cleanup();  
}
```

To implement Accumulator Interface, every UDF must extend the *EvalFunc* class and implement all the required functions in that UDF class. Moreover, the interface must be parameterized by the return type of the function. If a function is algebraic but can be used in a FOREACH statement with accumulator functions, it needs to implement the Accumulator interface in addition to the Algebraic interface. The accumulate function is guaranteed to be called one or more times, passing one or more tuples in a bag, to the UDF. The *getValue()* function is called after all the tuples for a particular key are passed to get the final result. Finally, *cleanup()* is called to reset itself before the next value is passed.

To understand better, let us look at the implementation of accumulator interface in COUNT function.

```
public class COUNT extends EvalFunc<Long> implements  
Algebraic, Accumulator<Long>{  
    ...  
    /* Accumulator interface implementation */  
  
    private long intermediateCount = 0L;  
  
    @Override  
    public void accumulate(Tuple b) throws  
IOException {  
        try {
```

```

        DataBag bag = (DataBag)b.get(0);
        Iterator it = bag.iterator();
        while (it.hasNext()){
            Tuple t = (Tuple)it.next();
            if (t != null && t.size() > 0 && t.get(0) != null) {
                intermediateCount += 1;
            }
        }
    } catch (ExecException ee) {
        throw ee;
    } catch (Exception e) {
        int errCode = 2106;
        String msg = "Error while computing
min in " + this.getClass().getSimpleName();
        throw new ExecException(msg, errCode,
PigException.BUG, e);
    }
}

@Override
public void cleanup() {
    intermediateCount = 0L;
}

@Override
public Long getValue() {
    return intermediateCount;
}
}
}

```

Accumulator interface definitely helps you decrease memory usage but there is a lot of data traffic between map and reduce nodes. Therefore, Pig prefers to use Algebraic Interface over the Accumulator Interface while implemented in a UDF.

Understanding Pig Types

Almost all the data types in Pig use the native Java types. For example, *chararray* type in Pig is represented *String* class in Java or the Pig type *int* is represented by the *Integer* class in Java. Please refer to the Pig data types in the Simple Types in CHAPTER 4: PIG LATIN DATA TYPES for more information. We can construct the appropriate Java objects normally for most of the Pig types, except for tuples and bags. Tuples and Bags are interfaces and are not classes. The UDFs cannot instantiate bags or tuples but instead use the factory classes for bags and tuples. The factory classes are *TupleFactory* and *BagFactory*. This enables the users to build their own implementations of tuple and bag so that Pig can use them.

TupleFactory can be used to construct tuples by calling the static method *TupleFactory.getInstance()* to get an instance of *TupleFactory* and create new tuples using the either *newTuple()* or *newTuple(int size)* methods. The *newTuple()* method is used when you do not know the number of fields in the tuple when it is constructed. You can then add fields using *Tuple*'s *append(Object val)* method, which will append the field to the end of the tuple. On the other hand, the method *newTuple(int size)* is used to allocate the number of fields the tuple will have. This method will create a tuple with the allocated number of fields with all nulls. Once it is created, you can set the fields using the *Tuple*'s *set(int fieldNum, Object val)* method.

Similarly, *BagFactory* is used to construct bags by calling *BagFactory.getInstance()* to get an instance of *BagFactory* and create new empty bags using the *newDefaultBag()* method. Once a bag is created, you can add tuples to the bag using *DataBag*'s *add(Tuple t)* method.

The code below is the builtin TOKENIZE showing how tuples and bags are created. As we know the TOKENIZE function is used to split a string and output a bag of words as tuples.

```
package org.apache.pig.builtin;

import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.BagFactory;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.Tuple;
import org.apache.pig.data.TupleFactory;

public class TOKENIZE extends EvalFunc<DataBag> {

    TupleFactory mTupleFactory = TupleFactory.getInstance();
    BagFactory mBagFactory = BagFactory.getInstance();

    public DataBag exec(Tuple input) throws IOException
    try {
        DataBag output = mBagFactory.newDefaultBag();
        Object o = input.get(0);
        if (!(o instanceof String)) {
            throw new IOException("Expected input to
be chararray, but got " + o.getClass().getName());
        }
        StringTokenizer tok = new StringTokenizer((String)o, "
\", () *", false);
    }
```

```
    while (tok.hasMoreTokens())
output.add(mTupleFactory.newTuple(tok.nextToken()));
    return output;
} catch (ExecException ee) {
    // error handling goes here
}
}
```

Schemas

Type information in Pig is validated before running a script. It is required that the UDFs too specify the type information and so *EvalFunc* class has a method which allows you to specify the type validation for input and output. However, the UDFs do not often communicate their output schema with Pig as Pig uses Java reflection to determine the same. This works fine until the UDF returns a scalar or a map but if the UDF returns a tuple or a bag, Pig has no idea to deal with it as Pig assumes that the tuple contains a single field of type bytearray. Therefore, it is required to specify the schema to avoid failures.

We shall look at specifying the schemas in UDFs in the lab exercises.

Error Handling

A UDF might encounter errors while processing and there are few choices on how to handle them. The most common error handling technique is to issue a warning and returning null. This error might be a result of a corrupt input data or a case of dividing a value with zero. For such cases we can just issue a warning and return a null value. Doing this does not end the job but allows it to continue till the end. This is very much useful when you are processing tons of records and you would definitely not want the entire job to fail just because of a malformed record. Also it does not make a great difference in the end result when few records are corrupt as it is always possible to have few corrupt records out of billion records. The warnings are aggregated by Pig and provided to the user after the job is complete.

Another type of error can occur when few resources are temporarily unavailable. This kind of error is not tolerable. The UDF throws an exception and all the tasks fail. However, in this case Hadoop automatically restarts the job and the resources might be available when retried and there might not be any error. The default number of retries is three. If the error still persists, the job will not be retried and will be marked as failed as Hadoop does not know how to handle this error.

Reporting Progress

At times a UDF needs to report its progress to Pig so that it knows that the job is running as it should and is not dead. All the tasks must send heartbeats to Pig and if no heartbeat is received for five minutes, the task is killed.

Less complex UDFs which takes very less time to execute are not required to send heartbeats. However, if a UDF is complex enough and runs more than five minutes, it is important that it reports its progress. To make your UDF to report its progress, all you need to do is call the *progress* function in the *exec* method provided in the *EvalFunc* class.

The following code shows the *progress()* for the UPPER UDF.

```
public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try{
            progress();
            String str = (String)input.get(0);
            return str.toUpperCase();
        }
        catch(Exception e){
            throw new IOException("Caught exception processing
input row ", e);
        }
    }
}
```

Overloading Functions

Writing type specific versions of UDFs helps increase the overall performance and accuracy. For example, MAX(int) should return an int or MAX(long) should return an long. This can be achieved using the *getArgToFuncMapping()* method provided by *EvalFunc* class. The *getArgToFuncMapping()* method returns a list of *FuncSpecs* which has a mapping from the input schema to the appropriate class it should handle. If *getArgToFuncMapping()* method returns null, the current UDF will be used. Pig provides a *FuncSpecs* class that describe a UDF. Using *FuncSpecs* you can describe a set of expected input arguments. Pig will then check which type fits the best for the given input and chooses that corresponding type class stored in same package for execution.

To understand this better, check the *getArgToFuncMapping()* of Pig's built-in MAX function below.

```
public List<FuncSpec> getArgToFuncMapping() throws
FrontendException {
    List<FuncSpec> funcList = new ArrayList<FuncSpec>();
```

```

        funcList.add(new FuncSpec(this.getClass().getName(),
Schema.generateNestedSchema(DataType.BAG, DataType.BYTEARRAY)));

        funcList.add(new FuncSpec(DoubleMax.class.getName(),
Schema.generateNestedSchema(DataType.BAG, DataType.DOUBLE)));

        funcList.add(new FuncSpec(FloatMax.class.getName(),
Schema.generateNestedSchema(DataType.BAG, DataType.FLOAT)));

        funcList.add(new FuncSpec(IntMax.class.getName(),
Schema.generateNestedSchema(DataType.BAG, DataType.INTEGER)));

        funcList.add(new FuncSpec(LongMax.class.getName(),
Schema.generateNestedSchema(DataType.BAG, DataType.LONG)));

        funcList.add(new FuncSpec(StringMax.class.getName(),
Schema.generateNestedSchema(DataType.BAG, DataType.CHARARRAY)));

        funcList.add(new FuncSpec(DateTimeMax.class.getName(),
Schema.generateNestedSchema(DataType.BAG, DataType.DATETIME)));

        funcList.add(new FuncSpec(BigDecimalMax.class.getName(),
Schema.generateNestedSchema(DataType.BAG,
DataType.BIGDECIMAL)));

        funcList.add(new FuncSpec(BigIntegerMax.class.getName(),
Schema.generateNestedSchema(DataType.BAG,
DataType.BIGINTEGER)));

        return funcList;
    }
}

```

In the example above, the main class handles the *bytearray* input types and if the input is of other type, the appropriate class is loaded to the execution pipeline. Pig chooses the best match for the input type. If no match is found, Pig returns an error.

Filter Functions

Filter functions are another commonly used UDF which returns True/False (Boolean) value. The filter functions can be used anywhere in the pig script including in the statement where the FILTER operator is used.

We shall look at more information on writing the filter functions in the lab exercise.

Registering and Defining UDFs

Before we jump to the lab exercises, we have a couple of Pig Latin UDF commands to cover. These commands help pig in finding the location of the UDF and make our lives a lot easier.

Registering UDFs

Since the UDFs are custom built functions and not the functions which are already built-in, we have to specify the location of the UDF. To do this, we use the REGISTER command at the beginning of Pig script. The UDF which is a jar file can be stored in the local, remote or distributed file system such as HDFS. The syntax for REGISTER command is simply by specifying the REGISTER command followed by the path to the UDF ending with a semi-colon as shown in the example below.

```
grunt> REGISTER /pig/udf/quality.jar;
grunt> A = LOAD 'user_data';
grunt> B = FOREACH A GENERATE quality.Checker($0);
```

In the example above, we have first provided the location of the UDF *quality.jar* to Pig using the REGISTER command. Later in the script, we have called the *Checker* function in the *Quality* class we have just registered on the first field of data we loaded.

When registering a UDF in the local file system, you can specify glob pattern using “*” to match the jars available in the local file system. The path which you specify can be of the relative path or the absolute path.

Additionally, you can also register the UDF from Command Line Interface by exporting an environment variable. To do this we can specify the *PIG_OPTS* environment variable using the *-Dpig.additional.jars.uris* option as shown in the example below.

```
$ export PIG_OPTS= "-Dpig.additional.jars.uris=/pig/udf/quality.jar"
```

Once you export this environment variable and register your UDF, you need not again register the UDF in Pig script using the REGISTER command. It is always recommended that you register the UDF using the REGISTER command within the Pig script instead of exporting the environment variable in the command line interface.

Defining UDFs

At times, the package names for your Java UDFs might be so long and you would definitely not want to type that long package in your Pig script while using the UDF. To overcome this, we have a DEFINE command which can be used to specify an alias so that there is no need of typing the entire package name of the UDF.

For example, consider the UDF with the following package name: *com.consultantsnetwork.pig.quality.Checker*. It is always pain to

type the entire package name while using the UDF. So we use the `DEFINE` command followed by the alias and the package name of the UDF. Once defined, we can use the alias for UDF instead of the long package name as shown in the example below.

```
grunt> REGISTER /pig/udf/quality.jar;
grunt> DEFINE checker
com.consultantsnetwork.pig.quality.Checker();
grunt> A = LOAD 'user_data';
grunt> B = FOREACH A GENERATE checker($0);
```

That's all the theory we have for this chapter. Let us proceed to the Lab Exercises and get our hands on what we have just learned. We need an Integrated Development Environment (IDE) to write our Java code in to develop Pig UDFs. We will be using Eclipse as our IDE throughout the lab exercises. You may choose to use any other IDE which you feel you are comfortable with. Let's kick start our lab exercises by installing Eclipse on to our machines.

AIM

The aim of the following lab exercise is to install Eclipse IDE and develop our own Eval and Filter functions.

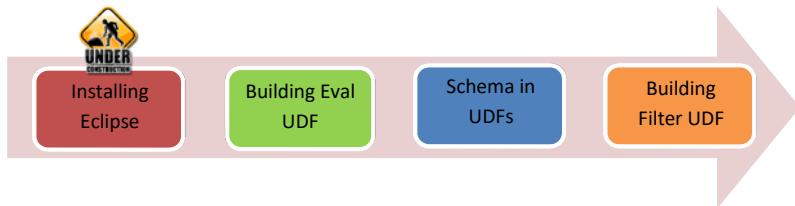
Following steps are required:

- Task 1: Installing Eclipse IDE.
- Task 2: Building an Evaluation Function
- Task 3: Implementing Schema in UDFs
- Task 4: Building a Filter Function

Lab Exercise 8: BUILDING EVAL AND FILTER UDFs



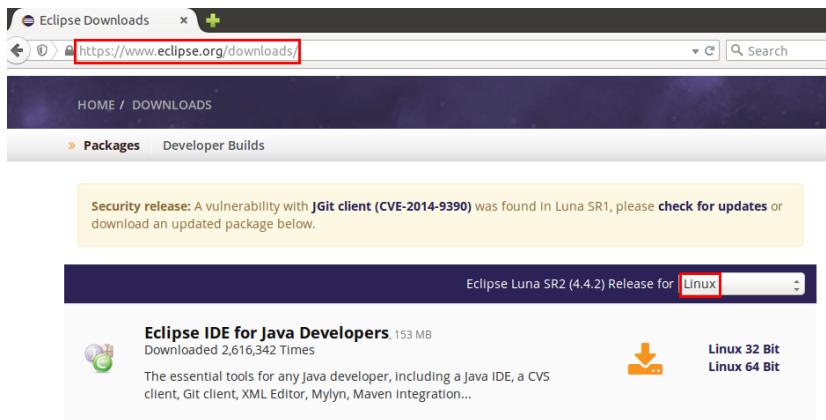
1. **Installing Eclipse IDE**
2. **Building an Eval Function**
3. **Implementing Schema in UDFs**
4. **Building a Filter UDF**



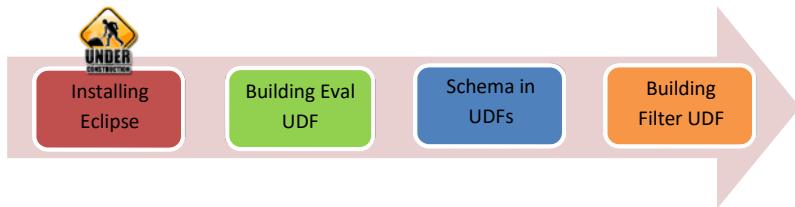
In order to build our own piece of awesomeness, we need to have an integrated development environment to write Java code in and develop Pig UDFs. Let us begin with installing Eclipse IDE. You are free to choose any other IDE and use it for developing. Eclipse is an integrated development environment (IDE) used for building integrated web and application development tooling. Eclipse has the base workspace and an extensible plug-in system for customizing the environment. Eclipse is most popular with Java developers and is also used by C++ and PHP developers too.

Task 1: Installing Eclipse IDE

Step 1: We first have to download Eclipse to our machines. To do so, open your internet browser and navigate to <https://www.eclipse.org/downloads/>. Make sure Linux is selected in the drop down menu as shown in the screenshot below.



Step 2: Download the Linux 64 bit version of Eclipse IDE for Java Developers 4.4.2 edition as shown in the screenshot. We are about to download the 64 bit version of Eclipse because; remember we have installed Hadoop and Pig in a 64 bit machine running Ubuntu LTS 14.10 Operating system and hence our machine is 64 bit compatible. You can also install the 32 bit version for developing Pig UDFs.



Eclipse Luna SR2 (4.4.2) Release for Linux

Eclipse IDE for Java Developers, 153 MB
 Downloaded 2,616,342 Times
 The essential tools for any Java developer, including a Java IDE, a CVS client, Git client, XML Editor, Mylyn, Maven Integration...
 Linux 32 Bit Linux 64 Bit

Package Solutions Filter Packages ▾

Eclipse IDE for Java EE Developers, 253 MB
 Downloaded 1,355,243 Times
 Tools for Java developers creating Java EE and Web applications,  Linux 32 Bit Linux 64 Bit

Please note that there might be a different version of Eclipse available when you work on this. Make sure you download the latest available version. The installation procedure remains same.

Step 3: Click on the mirror link as highlighted in the screenshot. Please see that your mirror link might vary depending upon your geographical location.

[Downloads Home](#)

» [Source code](#)

» [More Packages](#)

Eclipse downloads - mirror selection

All downloads are provided under the terms and conditions of the [Eclipse Foundation Software User Agreement](#) unless otherwise specified.

Download `eclipse-java-luna-SR2-linux-gtk-x86_64.tar.gz` from:

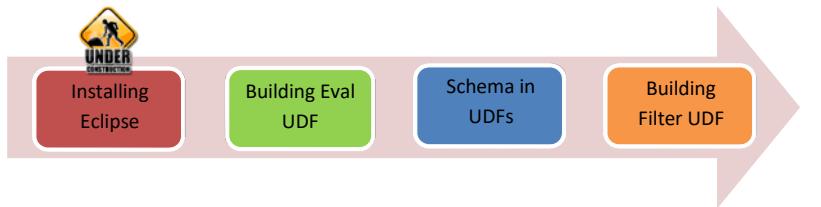


[Taiwan] Computer Center, Shu-Te University (http)

Checksums: [\[MD5\]](#) [\[SHA1\]](#) [\[SHA-512\]](#)

...or pick a mirror site below.

Step 4: Save the file to your local disk when prompted by your browser. This might take a while to download depending upon your internet connection. The file may be downloaded to your Downloads directory. Go to the Downloads directory and copy/paste it to the Home directory.



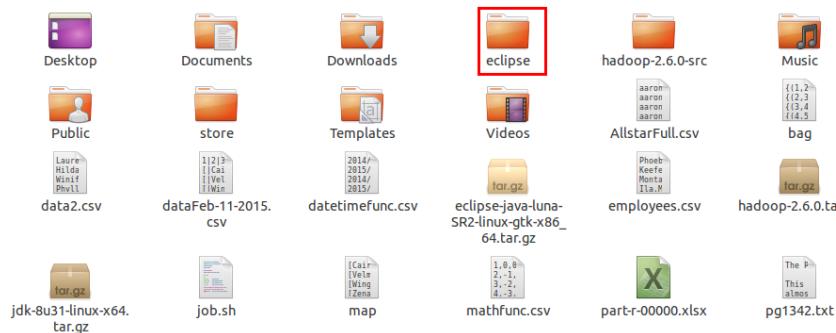
Step 5: Extract the contents of the Eclipse package by running the following command from the command line interface:

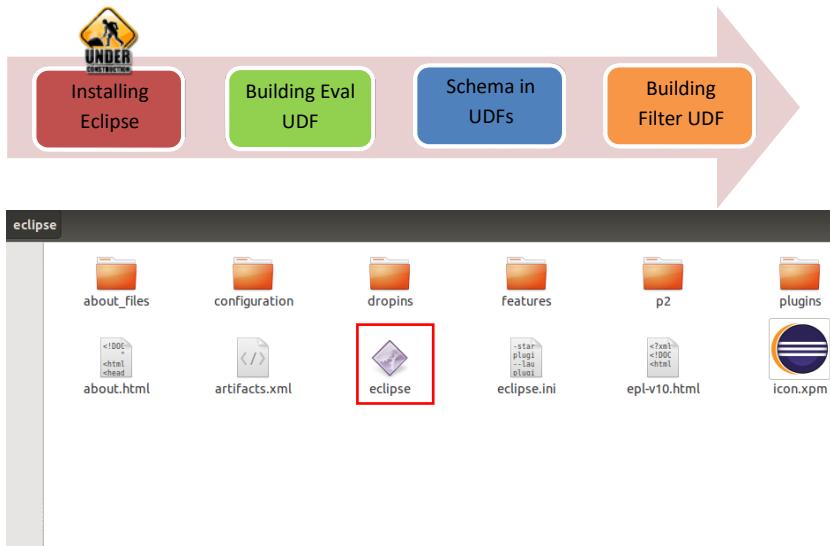
```
$ tar -xvzf eclipse-java-luna-SR2-linux-gtk-x86_64.tar.gz
```

You can also extract the package by right clicking on the package and clicking Extract here.

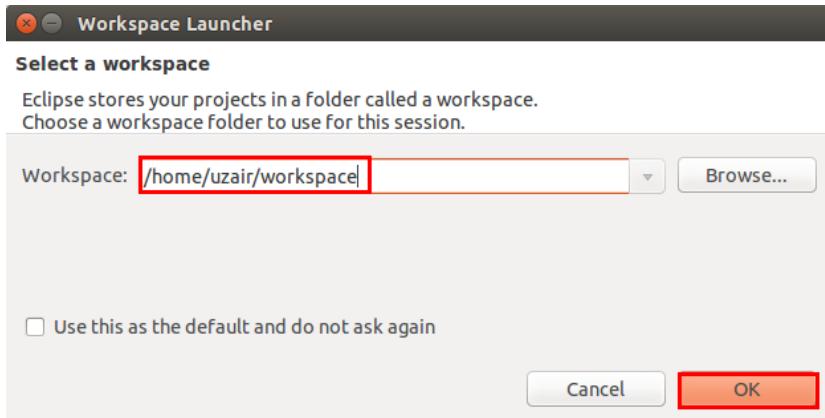
```
uzair@ubuntu:~$ tar -xvzf eclipse-java-luna-SR2-linux-gtk-x86_64.tar.gz
```

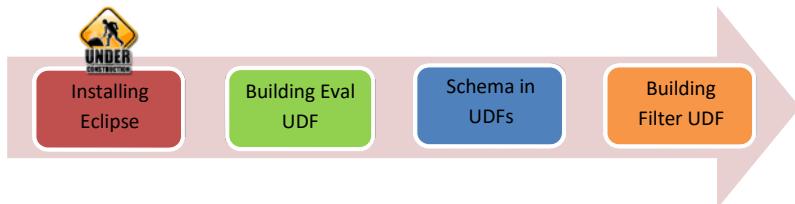
Step 6: Once the extraction is complete you will have *eclipse* directory. Go into the directory and open the *Eclipse Application* as shown in the screenshot below.



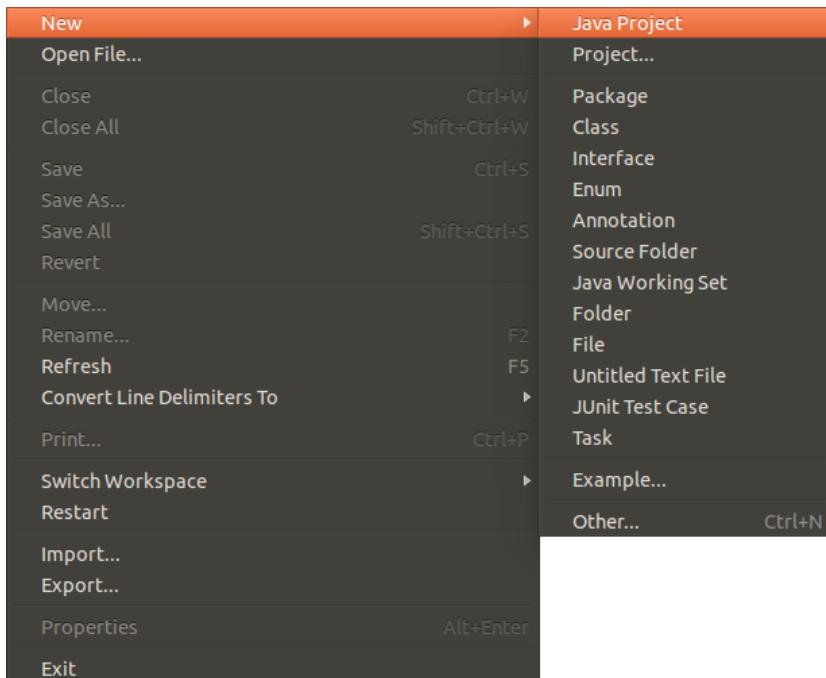


Step 7: Eclipse will start loading all the required stuff to function. In few moments, it will then pop-up asking where to save the workspace. You can choose to save your workspace in a custom directory or just go with the default workspace which Eclipse has suggested.





Step 8: Once Eclipse has been loaded, close the welcome screen (if any), click on *File*, hover on *New* and then click *Java Project*.



Step 9: In the dialog box enter any project name as you like. I have named my project as Pig. Leave all the settings to default and click the Finish button at the bottom of the dialog as shown in the screenshot. Alternatively, you can also change the workspace to save the files relate to the project by unchecking the Use default location box and specifying your custom path but it is not necessary. You can continue to work in your default workspace.

New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: **Pig**

Use default location

Location: `/home/uzair/workspace/Pig` [Browse...](#)

JRE

Use an execution environment JRE: `JavaSE-1.7`

Use a project specific JRE: `java-7-openjdk-amd64` [Configure JREs...](#)

Use default JRE (currently 'java-7-openjdk-amd64')

Project layout

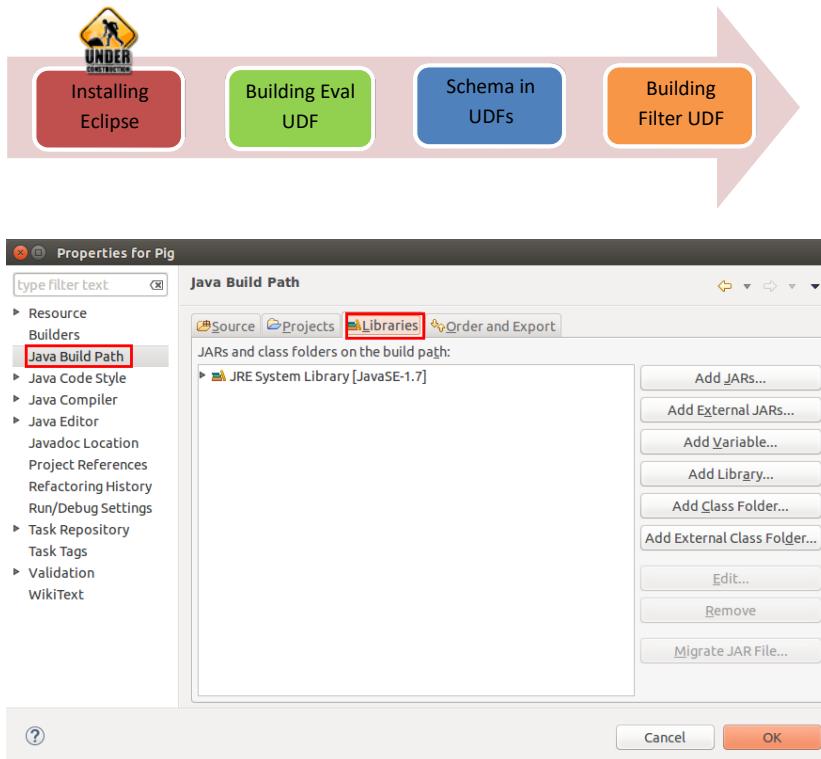
Use project folder as root for sources and class files

Create separate folders for sources and class files [Configure default...](#)

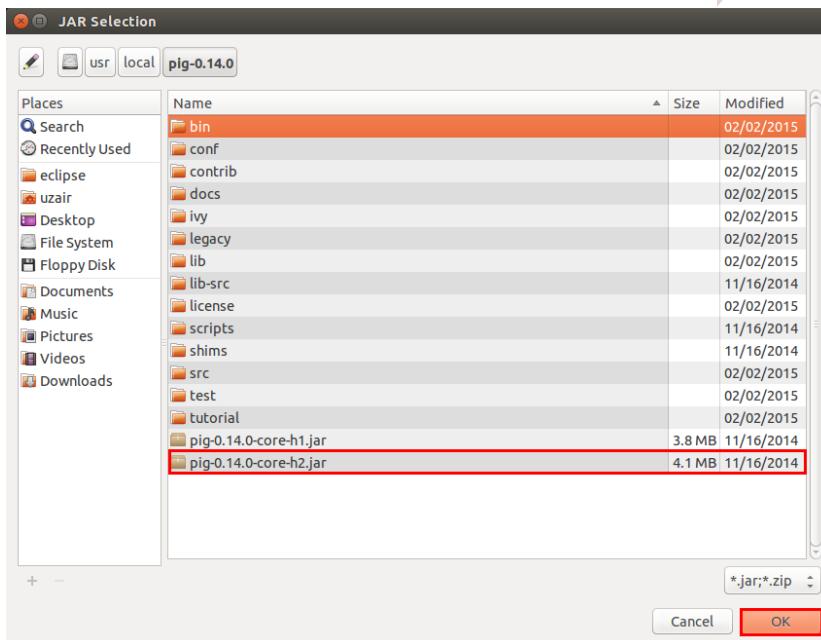
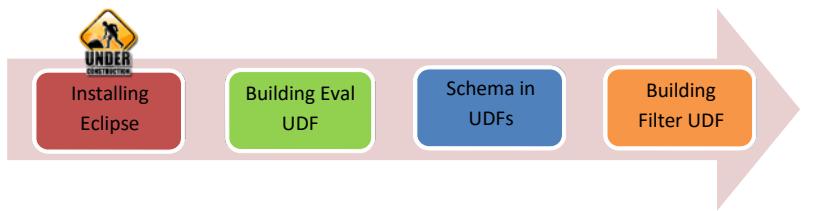
Working sets

< Back [Next >](#) Cancel **Finish**

Step 10: Finally include Pig libraries in the build path of the project so as Eclipse can understand that you are developing a Pig UDF. To do this *Right Click* on the project name we have just created and click on properties. Now in the properties dialog click on *Java Build Path* and then click on *Libraries* as shown in the screenshot below.



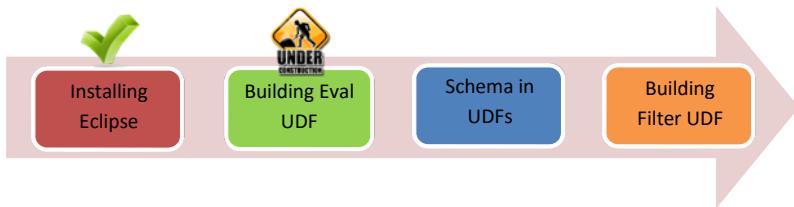
Under *Libraries*, click on *Add external Jars* as seen in the screenshot above. Browse through the Pig installation directory and select Pig core Jar file as shown below. The Pig Core Jar file consists of all the Pig libraries and helps Eclipse to understand that we are developing Pig UDF. If we do not add this Pig Core Jar File, Eclipse shows errors and does not compile the MapReduce applications. Make sure you only add the *pig-0.14.0-core-h2.jar* file only.



You should also add the Hadoop libraries. Click on *Add External Jars* again and navigate to your Hadoop directory. Once in the Hadoop directory navigate to *share/hadoop/common* directory and add the *hadoop-common-2.6.0.jar*. Then again click on *Add External Jars* and navigate to the *lib* directory inside the *common* directory. Add all the jars available in the *lib* directory.

You should now be able to find the Pig core jar file in the build path. Click *OK* and you have successfully added the Hadoop build path for your first Pig UDF.

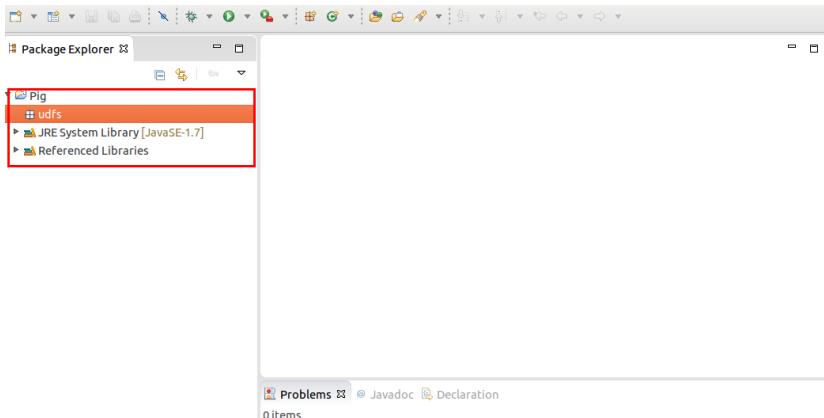
Task 1 is complete!



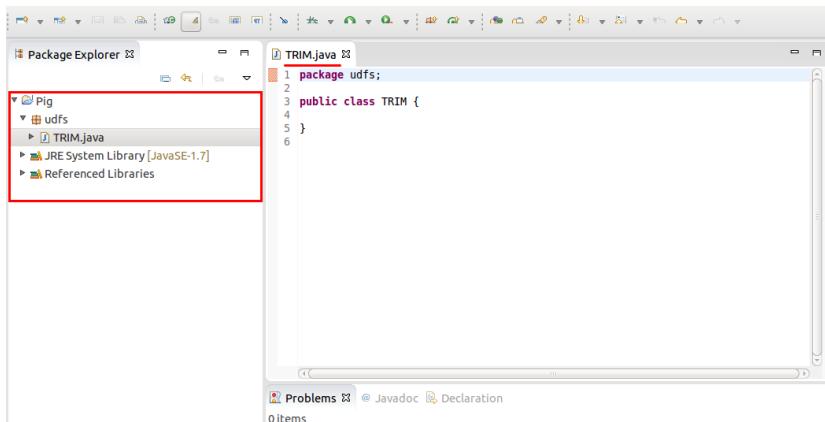
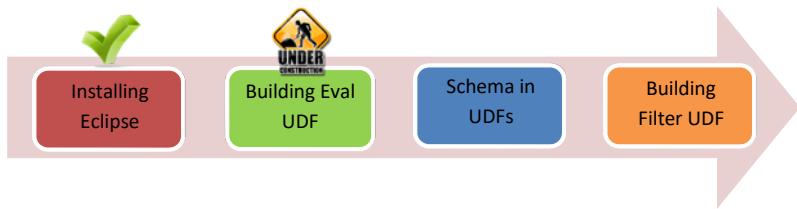
Task 2: Building an Eval Function

For simplicity and better understanding, let us build an eval UDF for a built-in function, *TRIM*. The procedure to build a Eval UDF remains the same. The *TRIM* UDF is used to trim the leading and trailing whitespaces from a string.

Step 1: We have created a new project *Pig* in the previous task. Let us now begin with where we have left and create a new package and name it *udfs*. To do this, right click on the project name, hover over *New* and then click on *Package*. Enter the package name as *udfs* in the popup dialog. Finally click on *Finish*. You should see the project name and package name similar to the screenshot below.



Step 2: Now that we have the package created, let us begin building our UDF by creating a new class. To create a new class, right click on the package, hover over *New* and then click on *Class*. You should have a popup asking for the name of the class. Enter *TRIM* for the class name and click on *Finish*. Your class is now created and left with *TRIM.java* class under the package as shown in the screenshot below.



Step 3: The next step is to add the imports required so that we can access all the classes and interfaces required. We need to add these 4 imports for our UDF.

```

import java.io.IOException;

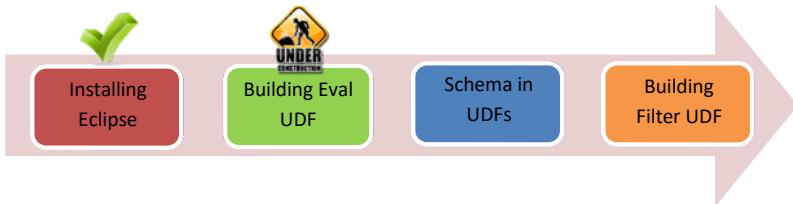
import org.apache.pig.EvalFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.Tuple;

```

First is the java specific import for IO exceptions and the rest are Pig's imports.

*If you have a problem adding imports and come across an error due to non-adding of imports, you can simply ask Eclipse to automatically add imports by using the shortcut **Ctrl + Shift + O**.*

Step 4: The UDF class *TRIM* extends the *EvalFunc* class which is the base class for all the eval functions. The eval function has to be parameterized by the type of return value, which is a *String* for this UDF. Next, we have to implement the function *exec* which takes a tuple as an input and makes sure the input is not null and it exists.

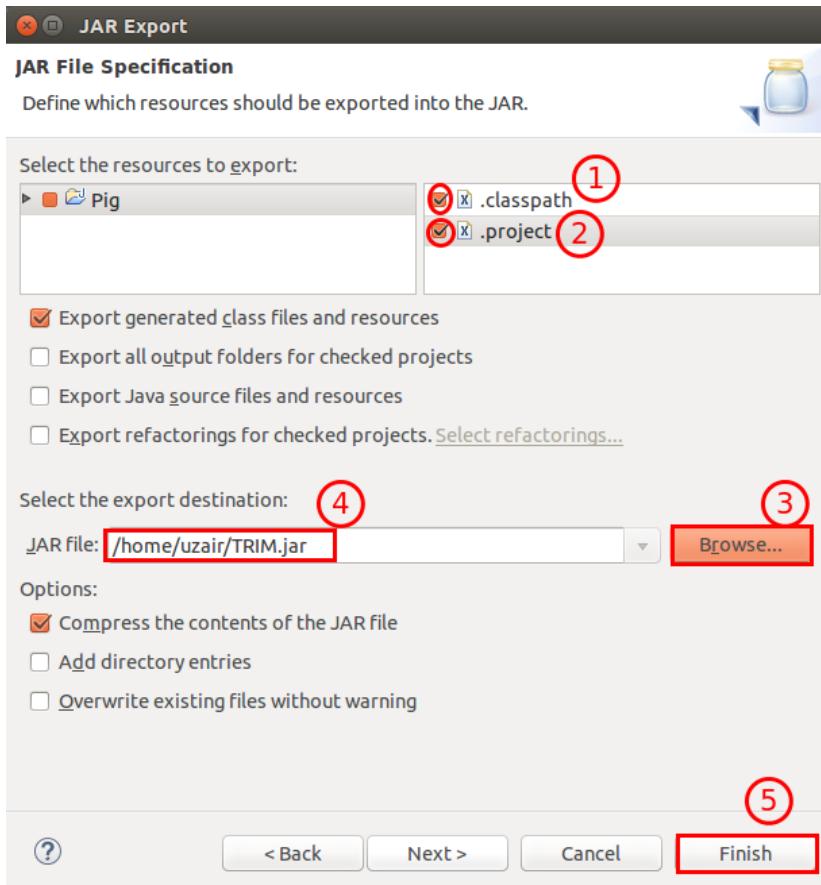
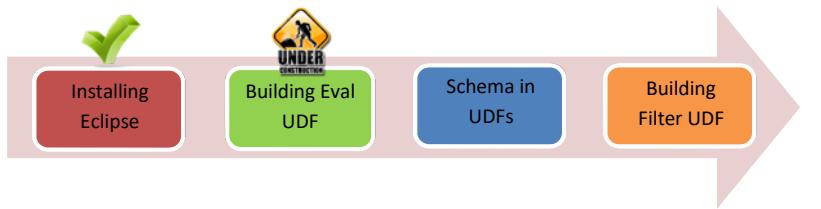


The main part is inside the try/catch blocks. We get the first item from the input and cast it to *String* so that we can make sure we are working with a String. We then use the Java *trim()* function to remove the trailing and leading white spaces and return the trimmed string. Finally, we do the error handling by issuing a warning and returning null in the *catch* block.

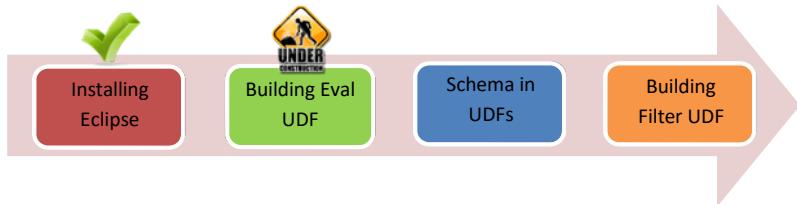
```
public class TRIM extends EvalFunc<String> {
    @Override
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0) {
            return null;
        }
        try {
            String str = (String) input.get(0);
            if (str == null)
                return null;
            if (str.length() == 0)
                return str;
            return str.trim();
        } catch (ExecException e) {
            log.warn("Error reading input: " + e.getMessage());
            return null;
        }
    }
}
```

Step 5: Now that we have the UDF implemented, let us export the jar file. To do so, right click on the *udfs* package name and click on *Export...*

- You will have a popup asking you to choose the export destination. In the *Export* pop up, click the arrow button for Java so that it pop down its contents. Then click on *JAR file* and click on *Next >* button.
- In the *Jar Export* pop up, check the boxes for *.classpath* and *.project*. Click on *browse* and save the UDF to your home directory. You will be prompted to name your jar file, name it *TRIM.jar*
- Click *Finish* to complete the process.



Step 6: We can now use the UDF in our Pig script. Start all the Hadoop daemons and Pig in Grunt mode so that we can start writing Pig Latin in the Grunt shell.



Step 7: To use the TRIM UDF, download the file *spaced_data.csv* from the following URL and save it to your HDFS home directory.

<https://db.tt/f1R3rrMu>

The data consists of leading and trailing whitespaces as shown below.

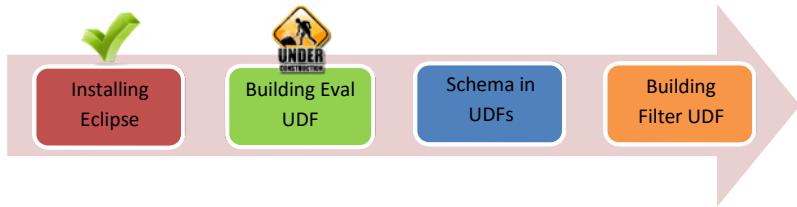
```
Barbara
Nicholas
Judy
Douglas
  Tammy
  Jack
Walter
  Shawn
  Joyce
  Andrew
Ruby
  Sandra
  Helen
  Victor
  Jennifer
John
Joseph
Carlos
  Stephanie
```

Step 8: Register the *TRIM.jar* UDF by specifying the path to the jar file in the Grunt shell.

```
grunt> REGISTER '/home/uzair/TRIM.jar';
```

Make sure you specify the correct path to the jar file and not just the path shown above. The path on your file system will be different.

```
grunt> REGISTER '/home/uzair/TRIM.jar';
grunt>
```



Step 9: Now that we have registered the jar file, let us load the input data, then apply the UDF and check the output.

```
grunt> A = LOAD 'spaced_data' AS (name: chararray);
grunt> B = FOREACH A GENERATE udfs.TRIM(name);
```

```
grunt> REGISTER '/home/uzair/TRIM.jar';
grunt> A = LOAD 'spaced_data.csv' AS (name: chararray);
2015-05-09 14:21:26,057 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> B = FOREACH A GENERATE udfs.TRIM(name);
grunt>
```

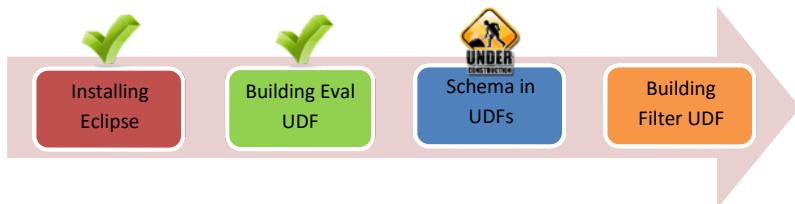
Step 10: Now dump the relation *B* and check the output.

```
grunt> DUMP B;
```

```
(Janet)
(Barbara)
(Nicholas)
(Judy)
(Douglas)
(Tammy)
(Jack)
(Walter)
(Shawn)
(Joyce)
(Andrew)
(Ruby)
(Sandra)
(Helen)
(Victor)
```

As you can see, the UDF has worked by removing the leading and trailing whitespaces.

Task 2 is complete!



You can download the *TRIM.jar* and *TRIM.java* files from the URL below.

TRIM.jar - <https://db.tt/sgiR3Uor>

TRIM.java - <https://db.tt/HmFSt3qI>

Task 3: Implementing Schemas in UDFs

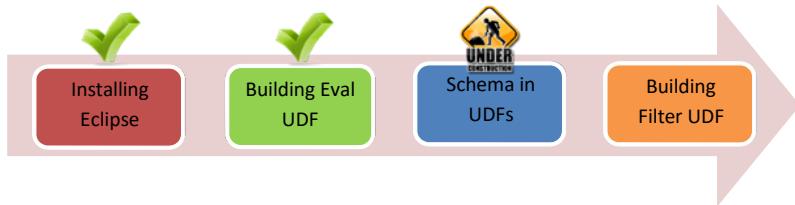
We will be implementing schema for the TRIM UDF seen in the previous task.

Step 1: We first need to import the *schema* class and the *DataType* so that we can access it while implementing schema. The import we need to add is as shown below.

```
import org.apache.pig.impl.logicalLayer.schema.Schema;
import org.apache.pig.data.DataType;
```

Make sure you add this import along with all the other imports added in previous task.

```
1 package udfs;
2
3 import java.io.IOException;
4
5 import org.apache.pig.EvalFunc;
6 import org.apache.pig.backend.executionengine.ExecException;
7 import org.apache.pig.data.DataType;
8 import org.apache.pig.data.Tuple;
9 import org.apache.pig.impl.logicalLayer.schema.Schema;
10
11 public class TRIM extends EvalFunc<String> {
12     @Override
13     public String exec(Tuple input) throws IOException {
14         if (input == null || input.size() == 0) {
15             return null;
16         }
17         try {
18             String str = (String) input.get(0);
19             if (str == null)
20                 return null;
21             if (str.length() == 0)
22                 return str;
23         }
24     }
25 }
```



Step 2: To implement schema in the UDF, we have to override the `outputSchema(Schema)` method as shown below. Next, we return the desired schema (`chararray`, in this case) by using the `FieldSchema()` which takes arguments as the name of the field and the type of that field.

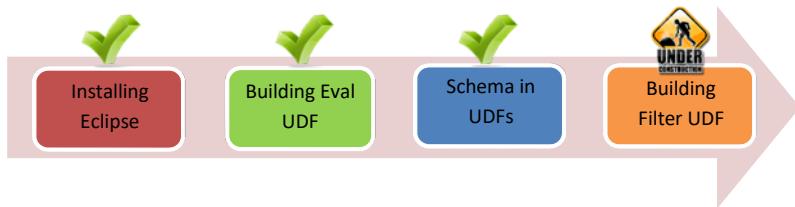
```
@Override
public Schema outputSchema(Schema input) {
    return new Schema(new Schema.FieldSchema(null,
    DataType.CHARARRAY));
```

We only have a single field for this UDF, and hence we have specified the schema only for this field. If you have multiple fields, you can specify the schema by declaring a Schema object and then appending all the fields' schemas to it. For example, consider three fields `name`, `age` and `city`, we implement schema as shown below.

```
public Schema outputSchema(Schema input) {
    Schema s = new Schema();
    s.add(new Schema.FieldSchema("name",
    DataType.CHARARRAY));
    s.add(new Schema.FieldSchema("age",
    DataType.INTEGER));
    s.add(new Schema.FieldSchema("city",
    DataType.CHARARRAY));
    return s;
}
```

Implementing schema also allows you to verify, if the correct type is passed. If it receives a type which is not expected, an error can be thrown.

Task 3 is complete!



Task 4: Building a Filter UDF

Let us build a filter UDF which filters out players with high salaries (greater than 100,000) from the input data containing details of players and their salaries.

Step 1: Create a new Java class in Eclipse under *udfs* and name the class as *IsHighSalary*.

Step 2: The next step is to add the imports required so that we can access all the classes and interfaces required. We need to add these 4 imports for our UDF.

```
import java.io.IOException;
import org.apache.pig.FilterFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.Tuple;
```

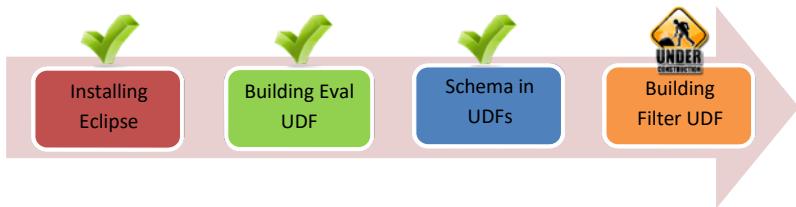
First is the java specific import for IO exceptions and the rest are Pig's imports.

If you have a problem adding imports and come across an error due to non-adding of imports, you can simply ask Eclipse to automatically add imports by using the shortcut Ctrl + Shift + O.

Step 3: The UDF class *IsHighSalary* extends the *FilterFunc* class which is the base class for all the Filter functions. Unlike eval function, the filter function need not be parameterized by the type of return value. Next, we have to implement the function *exec* which takes a tuple as an input and makes sure the input is not null and it exists.

The main part is inside the try/catch blocks. We get the first item from input, cast it to *Integer* and verify if the input record is greater than 100,000. If it is, we return the record. Finally, we do the error handling by issuing a warning and returning null in the *catch* block.

Please refer to the *IsHighSalary* UDF below.



```

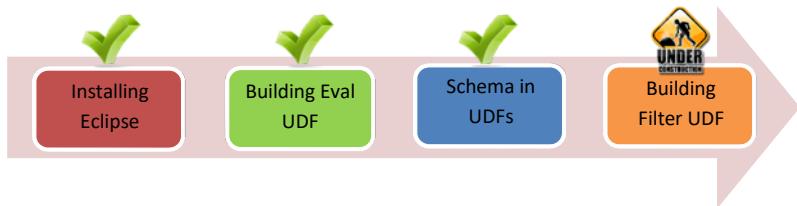
public class IsHighSalary extends FilterFunc {

    @Override
    public Boolean exec(Tuple tuple) throws IOException {
        if (tuple == null || tuple.size() == 0) {
            return false;
        }
        try {
            Object object = tuple.get(0);
            if (object == null) {
                return false;
            }
            float f = (Float) object;
            return f > 100000;
        } catch (ExecException e) {
            log.warn("Error reading input: " +
e.getMessage());
            return null;
        }
    }
}

```

Step 4: Now that we have the UDF implemented, let us export the jar file. To do so, right click on the *udfs* package name and click on *Export...*

- You will have a popup asking you to choose the export destination. In the *Export* pop up, click the arrow button for Java so that it pop down its contents. Then click on *JAR file* and click on *Next >* button.
- In the *Jar Export* pop up, check the boxes for *.classpath* and *.project*. Click on *browse* and save the UDF to your home directory. You will be prompted to name your jar file, name it *IsHighSalary.jar*
- Click *Finish* to complete the process.



Step 5: To use the *IsHighSalary* UDF, download the file *salaries.csv* from the following URL and save it to your HDFS home directory.

<https://db.tt/3s2oghDi>

Please ignore this step if you already have the file saved in your HDFS.

Step 6: Register the *IsHighSalary.jar* UDF by specifying the path to the jar file in the Grunt shell.

```
grunt> REGISTER '/home/uzair/IsHighSalary.jar';
```

Make sure you specify the correct path to the jar file and not just the path shown above. The path on your file system will be different.

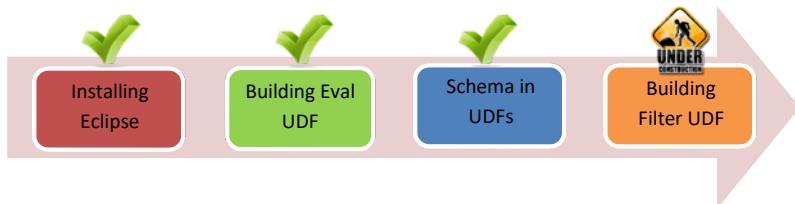
```
grunt> REGISTER /home/uzair/IsHighSalary.jar
2015-05-11 15:00:26,757 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 7: Now that we have registered the jar file, let us load the input data and then apply the UDF and check the output.

```
grunt> salaries = LOAD 'Pig/salaries.csv' USING
      PigStorage(',') AS (year: int, team_id:
      chararray, league_id: chararray,
      player_id: chararray, salary: float);

grunt> high_salaries = FILTER salaries BY
      udfs.IsHighSalary(salary);

grunt> DUMP high_salaries;
```



The output has all the filtered records with salaries greater than 100,000 as shown in the screenshot below.

```
(2010,TOR,AL,litscje01,414700.0)
(2010,TOR,AL,marcush01,850000.0)
(2010,TOR,AL,mccoymi01,400700.0)
(2010,TOR,AL,mcdonjo03,1500000.0)
(2010,TOR,AL,mcgowdu01,500000.0)
(2010,TOR,AL,molinjo01,800000.0)
(2010,TOR,AL,morrobr01,409800.0)
(2010,TOR,AL,overbly01,7950000.0)
(2010,TOR,AL,richmsc01,409900.0)
(2010,TOR,AL,romerri01,408300.0)
(2010,TOR,AL,ruizra01,404300.0)
(2010,TOR,AL,rzepcma01,404000.0)
(2010,TOR,AL,snidetr01,405800.0)
(2010,TOR,AL,tallebr01,2000000.0)
```

You can download the *IsHighSalary.jar* and *IsHighSalary.java* files from the URL below.

IsHighSalary.jar - <https://db.tt/Kp7ffDsn>
IsHighSalary.java - <https://db.tt/YR1atDoK>

Task 4 is complete!

SUMMARY

User Defined Functions helps you write your own custom function to process your data. There might be cases where you cannot find a built-in function suitable for your needs and this is where a user defined function helps you. Pig UDFs can be written and implemented in Java, Jython, Python, JavaScript, Ruby and Groovy. However, since Pig is written in Java, UDFs written in Java are most efficient with greater support while the others have limited support. With UDFs every part of processing can be customized including data load/store, column transformation, and aggregation.

We have covered the Eval and Filter UDFs for this chapter. We shall cover Load/Store UDFs in the next chapter.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/udf.html>
- Programming Pig by Alan Gates

INDEX

Eval Functions	252
Filter Functions	261
Registering and Defining UDFs.....	261
AIM	264
Lab Exercise 8: Bulding Eval and Filter Functions.....	265
Task 1: Installing Eclipse IDE	266
Task 2: Building an Eval Function	274
Task 3: Implementing Schemas in UDFs	280
Task 4: Building a Filter Function	282
SUMMARY	286
REFERENCES.....	287

CHAPTER 9: USER DEFINED FUNCTIONS – PART 2

Theory

We have learned and built Eval and Filter User Defined Functions in the previous chapter. In this chapter, we shall cover the Load/Store User Defined Functions which we left out in the previous chapter. Load/Store UDFs are used to instructing Pig to get data in and out according to our requirement of handling and storing data. This chapter serves as an extension to the previous chapter. As in the previous chapter, we write and implement UDFs in Java for this chapter too. However, you can also build your Load/Store Pig UDF in Jython, Python, JavaScript, Ruby and Groovy languages too.

As we know Pig uses Hadoop's MapReduce framework under the hood, Pig's Load/Store Function API sits on top of the Hadoop's *InputFormat* and *OutputFormat* classes. Therefore, you will be able to create custom Load/Store function implementations using the existing Hadoop *InputFormat* and *OutputFormat* classes with very less effort. The *InputFormat* class helps you with reading the input data while the *OutputFormat* class to write data and you need not bother about this stuff while building UDFs.

Let us first look at the process to implement your own load functions.

Load Functions

Pig's Load function sits on top of the hadoop's *InputFormat*. This interface helps Hadoop to read data. The Hadoop *InputFormat* does the following: It first checks if the data is available as described in the location. Then it determines how the input data is split into *InputSplit* to serve the map tasks and finally provides the *RecordReader* implementation used to create key-value pairs from *InputSplit* as input to those map tasks. The key-value pairs are then used by Load function to generate a Pig Tuple. All this happens under the hood and we need not worry about any of these while building a load UDF.

All the Load UDFs have to extend the *LoadFunc* abstract class which is the base class for all the load functions. There are four mandatory steps to be taken care of while we create our own Load function as shown below.

1. We need to first let Pig know the load function the input format, which should be used to read the input data.
2. Next, the Load function has to know the location where the input data is available in order to prepare itself to read.

3. Next, the load function has to get ready to read the input data from record reader.
4. The final stage is to start reading the input records from record reader and returning tuples to Pig.

There are few optional steps which we shall discuss later in this chapter.

Let us now look at the methods associated with the above steps which we need to override to build a Load UDF provided by the *LoadFunc* class.

getInputFormat()

This method is used to let the load function know which input format should be used to read input data. You may choose to specify the *InputFormat* shipped directly with Hadoop or a custom *InputFormat*. Moreover, if you want your text based *InputFormat* or file based *InputFormat* to read all the data inside the sub-directories of a directory recursively, you have to use *PigTextInputFormat* and *PigFileInputFormat* classes provided in *org.apache.pig.backend.hadoop.executionengine.mapReduceLayer*. These Pig *InputFormat* classes help you overcome the limitation in Hadoop *TextInputFormat* and *FileInputFormat* classes as Hadoop *InputFormat* cannot read the data recursively to multiple levels of directories inside a directory while Pig *InputFormat* can read all the data inside the sub-directories of a directory recursively.

setLocation()

This method is used to let the load function know the location where the input data is available for loading. This method can end up getting called multiple times and hence utmost care should be taken to ensure that this does not create a problem when used repeatedly.

prepareToRead()

This method is called prior to the input split being read. This method is called in each map task and a copy of the *RecordReader* is passed as an argument. *RecordReader* is the class used by *InputFormat* to read the input splits. The *RecordReader* will then be used by *getNext()* to return a tuple representing a record of data back to pig. The record reader and input split are passed as arguments for this method.

getNext()

This method is the final and crucial method for a load function as all the reading of data takes place here. When this method is called the tuples are read by the load function. This method is called repeatedly until the last record is read from its split. Once all the records are read, null is returned and this indicates that there are no more records to be read.

We shall be looking at all these methods in our lab exercises while we build our own Load Function.

Apart from the above methods which need to be overridden while building a load UDF, there are few interfaces which can be optionally implemented to achieve more complex features for your Load Function.

LoadMetadata

This interface is used when dealing with metadata systems. There are many data storage mechanisms which also allow you to store the schema associated with the data. This interface provides the method `getSchema()` which provides the schema associated with the data to Pig. Therefore, if the schema is obtained automatically by the load function, the programmer need not specify the schema while loading the data using the LOAD operator. This decreases a lot of effort and is there is also no chance of errors while specifying the schema.

This interface also has the method `getStatistics()` which returns the statistics about the data being loaded, if available. If not, it returns null. Similarly, the interface also has a `getPartitionKeys()` method. This method is useful when the data storages allow partitioning the data. With this we can only read the data from the partition which is required for our job. The `getPartitionKeys()` method returns an array of strings which are nothing but the field names on which the data is partitioned. These keys can be used to partition the data. If this returns null, there are no partition keys available. Once we have these keys, we can use the `setPartitionFilter()` method to filter out the keys and only return the values provided in the filter.

Please check out the code for LoadMetadata interface from the URL below for better understanding.

<http://svn.apache.org/viewvc/pig/trunk/src/org/apache/pig/LoadMetadata.java?view=markup>

LoadPushDown

This interface has the methods which help you extract only the fields you require from the data which is stored in columnar format. Therefore, you can only load the fields you require and ignore the rest. This optimization leads in great performance gain. The method `pushProjection()` provided by the LoadPushDown interface helps you get the fields required in the Pig script. The `pushProjection()` method takes `requiredFieldList` as the argument which contains the list of `requiredField`. The `requiredField` contains index, alias, type (which is reserved for future use), and subFields. Pig will use the column index `requiredField.index` to communicate with the load function about the fields required by the Pig script.

Please check out the code for LoadPushDown interface from the URL below for better understanding.

<http://svn.apache.org/viewvc/pig/trunk/src/org/apache/pig/LoadPushDown.java?view=markup>

LoadCaster

LoadCaster interface provides the methods which allow you to control how the binary data loaded by the loader is casted to specific data types. This interface contains so many methods which are enough for converting a binary type to any possible Pig data types such as *bytesToLong()*, *bytesToBoolean()*, *bytesToDateTIme()* etc.

Please check out the code for LoadCaster interface from the URL below for better understanding.

<http://svn.apache.org/viewvc/pig/trunk/src/org/apache/pig/LoadCaster.java?view=markup>

Store Functions

Similar to Load functions, Pig's Store function sits on top of the hadoop's *OutputFormat* class. This interface helps Hadoop to store output results of a job. All the Store UDFs have to extend the *StoreFunc* abstract class which is the base class for all the store functions.

There are four mandatory steps to be taken care of while we create our own Store function as shown below.

1. We need to first let Pig know the store function the output format, which should be used to store the output data.
2. Next, the Store function has to know the location where the output data has to be stored in order to prepare itself to store.
3. Next, the store function has to get ready to store the output data from record writer.
4. The final stage is to start writing the output records from record writer.

There are few optional steps which we shall discuss later in this chapter.

Let us now look at the methods associated with the above steps which we need to override to build a Store UDF provided by the *StoreFunc* class.

getOutputFormat()

This method is used to let the store Function know which output format should be used to store output data. You may choose to specify the *OutputFormat* shipped directly with Hadoop or a custom *OutputFormat*.

getStoreLocation()

This method is used to let the store function know the location where the output data has to be stored. This method can end up getting called multiple times and hence utmost care should be taken to ensure that this does not create a problem when used repeatedly.

prepareToWrite()

This method is called in each map or reduce task prior to the output being written. When this method is called, a copy of *RecordWriter* is passed as an argument. *RecordWriter* is a class used by the *OutputFormat* to write output records. The *RecordWriter* will then be used by *putNext()* to write a tuple representing a record of data.

putNext()

This method is the final and crucial method for a store function as all the writing of output data takes place here. When this method is called the tuples are written by the store function. This method is called repeatedly until the last record is written out in the format expected by the output format.

We shall be looking at all these methods in our lab exercises while we build our own Store Function.

Apart from the above methods which need to be overridden while building a store UDF, there are few interfaces which can be optionally implemented to achieve more complex features for your Store Function.

StoreMetadata

This is an optional interface used when the storage system is capable of storing schemas and statistics. This can be achieved by the methods *storeSchema()* and *storeStatistics()* provided by this interface.

Please check out the code for StoreMetadata interface from the URL below for better understanding.

<http://svn.apache.org/viewvc/pig/trunk/src/org/apache/pig/StoreMetadata.java?view=markup>

StoreResources

This interface provides you with methods which can help you to put hdfs files or local files to distributed cache. The methods `getCacheFiles()` and `getShipFiles()` are used to put a list of files from distributed file system and local file system respectively.

Please check out the code for `StoreResources` interface from the URL below for better understanding.

<http://svn.apache.org/viewvc/pig/trunk/src/org/apache/pig/StoreResources.java?view=markup>

cleanupOnFailure()

This method is called by Pig when a job fails during a store function. The output location which contains incomplete results will be deleted from the file system. This method takes the output location and the job object as the parameters so that it knows where and what to clean.

You need not worry about implementing this method in your function until your output location is on HDFS as the default implementation automatically takes care of this. However, if your output location is other than HDFS, you will have to implement this method.

That's all the theory we have for this chapter. Let us proceed to the Lab Exercises and get our hands on what we have just learned. We need an Integrated Development Environment (IDE) to write our Java code in to develop Pig UDFs. We will be using Eclipse as our IDE throughout the lab exercises. You may choose to use any other IDE which you feel you are comfortable with. Please check the previous lab exercises to install eclipse if you haven't installed it yet.

AIM

The aim of the following lab exercise is to develop our own Load and Store functions.

Following steps are required:

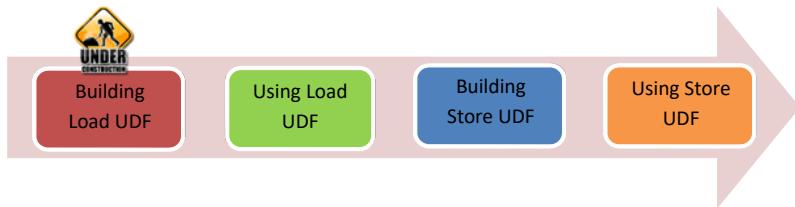
- Task 1: Building a Load Function
- Task 2: Using the Load UDF
- Task 3: Building a Store Function
- Task 4: Using the Store UDF

This lab exercise assumes you to have knowledge about Hadoop's Input and Output format along with knowledge on how to implement Custom input and output formats. Kindly proceed with this lab exercise only after you understand Hadoop's Input and Output formats.



Lab Exercise 9: BUILDING LOAD AND STORE UDFs

- 1. Building a Load Function**
- 2. Using the Load UDF**
- 3. Building a Store Function**
- 4. Using the Store UDF**

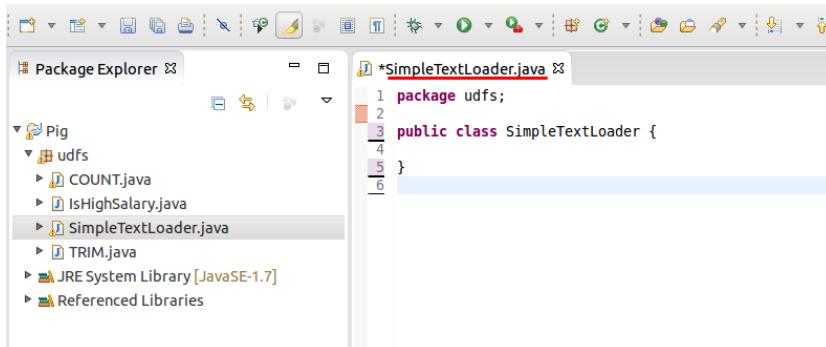


Task 1: Building a Load Function

For simplicity and better understanding, let us build a load UDF which loads text data using new line (\n) as line delimiter and tab character (\t) as default field delimiter. The default field delimiter can be changed according to the requirement by passing the required field delimiter to the constructor as seen in the *PigStorage()* built-in function.

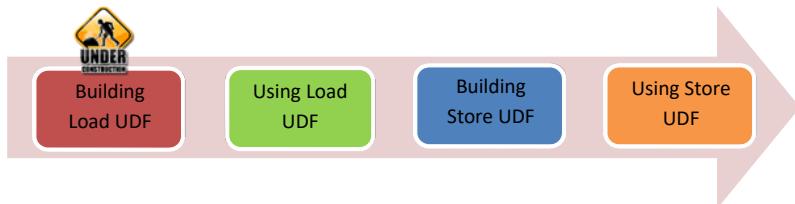
Step 1: Open Eclipse IDE installed in the previous lab exercise. If you have not installed Eclipse yet, please refer to Task 1 in the previous lab exercise to install the same.

Step 2: We have created a new project *Pig* in the previous task with *udfs* as a package. In the *udfs* package, create a new class and name it *SimpleTextLoader*. You should have *SimpleTextLoader.java* workspace as shown in the screenshot below.



Step 3: The next step is to add the imports required so that we can access all the classes and interfaces required. We need to add these imports below for our UDF.

```
import java.io.IOException;
import java.util.ArrayList;
```



```

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.pig.LoadFunc;
import org.apache.pig.PigException;
import org.apache.pig.backend.executionengine.ExecException;
import
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.Pig
Split;
import org.apache.pig.data.DataByteArray;
import org.apache.pig.data.Tuple;
import org.apache.pig.data.TupleFactory;

```

First are the java specific import and the rest are Hadoop and Pig specific imports.

If you have a problem adding imports and come across an error due to non-adding of imports, you can simply ask Eclipse to automatically add imports by using the shortcut Ctrl + Shift + O.

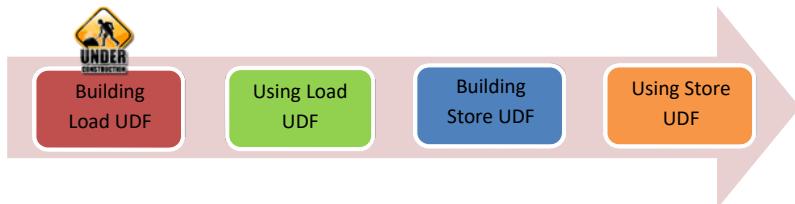
Step 4: The UDF class *SimpleTextLoader* extends the *LoadFunc* abstract class which is the base class for all the load functions. Next, we declare all the required variables and implement a constructor. This constructor helps you set the desired field delimiter while the default field delimiter is a tab character. The code for this implementation is as shown below.

```

public class SimpleTextLoader extends LoadFunc {
    protected RecordReader in = null;
    private byte fieldDel = '\t';
    private ArrayList<Object> mProtoTuple = null;
    private TupleFactory mTupleFactory =
        TupleFactory.getInstance();
    private static final int BUFFER_SIZE = 1024;

    public SimpleTextLoader() {
    }

```



```

public SimpleTextLoader(String delimiter) {
    this();
    if (delimiter.length() == 1) {
        this.fieldDel = (byte) delimiter.charAt(0);
    } else if
    (delimiter.length() > 1 && delimiter.charAt(0) == '\\\\') {
        switch (delimiter.charAt(1)) {
    case 't':
        this.fieldDel = (byte) '\\t';
        break;

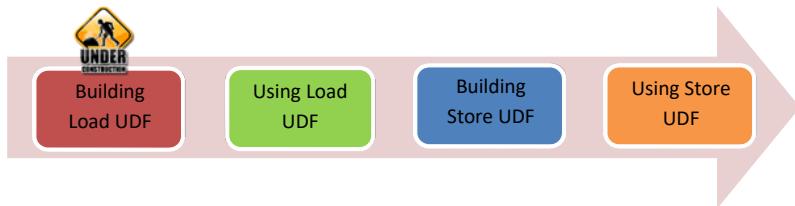
    case 'x':
        fieldDel =
    Integer.valueOf(delimiter.substring(2), 16).byteValue();
        break;

    case 'u':
        this.fieldDel =
    Integer.valueOf(delimiter.substring(2)).byteValue();
        break;

    default:
        throw new RuntimeException("Unknown
delimiter " + delimiter);
    }
} else {
    throw new RuntimeException("PigStorage
delimiter must be a single character");
}
}

```

Step 5: Next, we determine the *InputFormat*, so that Pig knows which input format has to be used while reading the input data. The *getInputFormat()* is used to get an instance of the input format. For this Load UDF, this method returns a *TextInputFormat()* which is the input format used to read the text data. Make sure you override this method so that you can instruct Pig to use the input format required for your load function.



```
@Override
    public InputFormat getInputFormat() {
        return new TextInputFormat();
    }
```

Step 6: Now that we have instructed Pig to use the input format required, we have to set the location of input data using the *setLocation()*. This method sets the path to input data as shown in the code below.

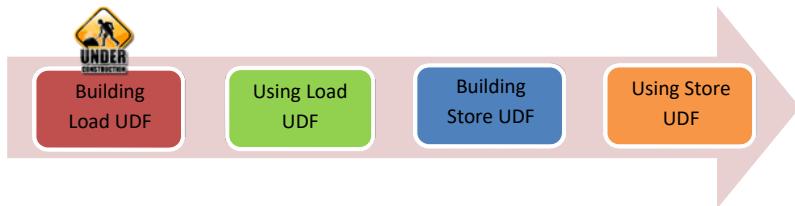
```
@Override
    public void setLocation(String location, Job job)
throws IOException {
    FileInputFormat.setInputPaths(job,
location);
}
```

Step 7: The next step is to prepare the load function to read data using the *prepareToRead()*. It takes record reader and input split as arguments.

```
@Override
    public void prepareToRead(RecordReader reader,
PigSplit split) {
    in = reader;
}
```

Step 8: The final step is to start reading the records from *RecordReader* using the *getNext()*. This method returns next tuple available in the process line to be read until *null* is returned. Once *null* is returned, it understands that the entire input is read and stops reading. This implementation is built using the *readFile()* as shown in the code below.

```
@Override
    public Tuple getNext() throws IOException {
try {
    boolean notDone = in.nextKeyValue();
    if (!notDone) {
        return null;
    }
}
```



```

Text value = (Text) in.getCurrentValue();
byte[] buf = value.getBytes();
int len = value.getLength();
int start = 0;

for (int i = 0; i < len; i++) {
    if (buf[i] == fieldDel) {
        readField(buf, start, i);
        start = i + 1;
    }
}

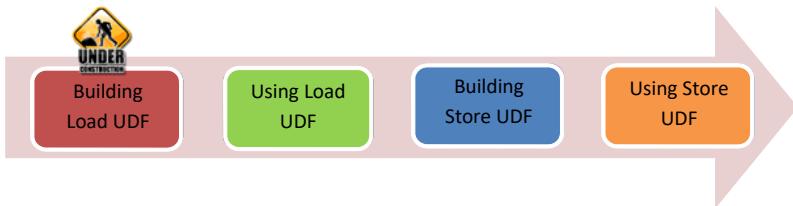
readField(buf, start, len);

Tuple t = mTupleFactory.newTupleNoCopy(mProtoTuple);
mProtoTuple = null;
return t;
} catch (InterruptedException e) {
    int errCode = 6018;
    String errMsg = "Error while reading input";
    throw new ExecException(errMsg, errCode,
                           PigException.REMOTE_ENVIRONMENT, e);
}
}

private void readField(byte[] buf, int start, int end) {
    if (mProtoTuple == null) {
        mProtoTuple = new ArrayList<Object>();
    }

    if (start == end) {
        mProtoTuple.add(null);
    } else {
        mProtoTuple.add(new DataByteArray(buf, start, end));
    }
}
}

```



Step 9: Now that we have the UDF implemented, let us export the jar file. To do so, right click on the *udfs* package name and click on *Export...*

- You will have a popup asking you to choose the export destination. In the *Export* pop up, click the arrow button for Java so that it pop down its contents. Then click on *JAR file* and click on *Next >* button.
- In the *Jar Export* pop up, check the boxes for *.classpath* and *.project*. Click on *browse* and save the UDF to your home directory. You will be prompted to name your jar file, name it *SimpleTextLoader.jar*
- Click *Finish* to complete the process.

We shall use this load UDF in the next task by loading a text file.

You can download the *SimpleTextLoader.jar* and *SimpleTextLoader.java* files from the URL below.

SimpleTextLoader.jar - <https://db.tt/NN3Bfx8>
SimpleTextLoader.java - <https://db.tt/hCe1DEns>

Task 1 is complete!

Task 2: Using the Load UDF

Step 1: We can now use the UDF exported to jar file in the previous task in our Pig script. Start all the Hadoop daemons and Pig in Grunt mode so that we can start writing Pig Latin in the Grunt shell.

Step 2: To use the *SimpleTextLoader* UDF, download the file *Items.tsv* from the following URL and save it to your HDFS home directory.

Items.tsv - <https://db.tt/ZtPvkfbT>

You may skip this step if you already have this file saved in your HDFS home directory from previous lab exercises.



Step 3: Register the *SimpleTextLoader.jar* UDF by specifying the path to the jar file in the Grunt shell.

```
grunt> REGISTER '/home/uzair/SimpleTextLoader.jar';
```

Make sure you specify the correct path to the jar file and not just the path shown above. The path on your file system will be different.

```
grunt> REGISTER '/home/uzair/SimpleTextLoader.jar';
2015-06-06 17:37:55,918 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 4: Now that we have registered the jar file, let us load the input data using the UDF we have just created.

```
grunt> A = LOAD 'Items.tsv' USING
udfs.SimpleTextLoader() AS (Id: int, LabelName:
chararray, TagLabelCategory: chararray,
TagSubcategory: chararray, ServiceLevel: chararray,
LeadTime: float, StockOnHand: int, StockOnOrder:
int, LotMultiplier: int);
```

```
grunt> A = LOAD 'Items.tsv' USING udfs.SimpleTextLoader() AS (Id: int, LabelName
: chararray, TagLabelCategory: chararray, TagSubcategory: chararray, ServiceLeve
l: chararray, LeadTime: float, StockOnHand: int, StockOnOrder: int, LotMultiplie
r: int);
2015-06-07 04:56:04,924 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 5: Verify if the input data has been successfully loaded using the load UDF by dumping the relation 'A' as shown below.



grunt> DUMP A;

If the data gets dumped to the screen as shown in the screenshot below, you have successfully built a load UDF to load text data into Pig.

```
(914,Cherry Warheads 100g,ILS40,ILS46,0.9,7.0,700,600,50)
(915,Orange Rocky Road 500g,ILS3014,ILS0000,0.95,3.0,55,400,10)
(916,Lemon Saf-T-Pops 100g,ILS40,ILS46,0.95,3.0,594,250,50)
(917,Raspberry Warheads 500g,ILS2002,ILS0000,0.9,7.0,673,80,10)
(918,Banana Caramel Creams 500g,ILS40,ILS46,0.9,7.0,1258,340,20)
(919,Strawberry Stick Candy 200g,ILS40,ILS46,0.97,1.0,0,50,50)
(920,Strawberry Tootsie Roll 200g,ILS3002,ILS0006,0.97,1.0,0,100,50)
(921,Watermelon Stick Candy 200g,ILS40,ILS46,0.95,3.0,213,220,10)
(922,Blackberry Gummi Fish 100g,ILS2002,ILS0000,0.9,7.0,959,300,20)
(923,Blackberry Caramels 500g,,,0.95,3.0,278,420,20)
(924,Strawberry Smarties 500g,ILS40,ILS42,0.97,1.0,0,50,10)
(925,Blueberry Smarties 200g,ILS40,ILS42,0.97,1.0,0,20,20)
(926,Banana Bit-o-Honey 200g,ILS60,ILS66,0.95,3.0,53,600,50)
(927,Raspberry Gummi Bears 200g,ILS60,ILS66,0.95,3.0,463,130,10)
(928,Grape Warheads 200g,ILS60,ILS66,0.97,1.0,0,50,50)
(929,Apple Candy Sticks 200g,ILS3002,,0.9,7.0,1264,120,10)
(930,Strawberry Black Jacks 100g,,,0.97,1.0,0,30,10)
(931,Lemon Black Jacks 200g,ILS40,ILS42,0.95,3.0,42,350,10)
(932,Grape Jawbreaker 200g,ILS60,ILS66,0.95,3.0,179,440,20)
(933,Grape Dum Dum Pops 200g,ILS3006,ILS0000,0.95,3.0,175,150,50)
(934,Orange Bit-o-Honey 100g,ILS40,ILS46,0.9,7.0,1423,220,20)
(935,Strawberry Sour Patch Kids 200g,ILS60,ILS66,0.9,7.0,331,50,50)
(936,Grape Caramel Squares 500g,ILS40,ILS46,0.97,1.0,0,40,20)
(937,Apple Abba-Zaba 200g,ILS40,ILS46,0.9,7.0,0,410,10)
```

Task 2 is complete!

Task 3: Building a Store function

Now that we have seen how to build a load function, let us complete the process by building a store function which can help us store the way we like.

For the store UDF, let us implement a store function which splits the output data into directories and files based on the field value. For example, in the file *Items.tsv* which we used in the previous task, has a *TagLabelCategory* field. We can use this field and instruct Pig to split output into directories based on the Tag label category.



For this storer, we also have to implement a custom *OutputFormat* as there are no output formats shipped by Hadoop suit our requirement. We do not go deeper into the discussion on building a custom *OutputFormat*, however, you will be able to get the code of custom *OutputFormat* from the steps below.

Step 1: Create a new Java class in Eclipse under the *udfs* package and name the class as *MultiStorage*.

Step 2: The next step is to add the imports required so that we can access all the classes and interfaces required. We need to add these imports below for our UDF.

```

import java.io.ByteArrayOutputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.text.NumberFormat;
import java.util.HashMap;
import java.util.Map;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.OutputFormat;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.TaskID;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

```



```

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.ReflectionUtils;
import org.apache.pig.StoreFunc;
import org.apache.pig.backend.executionengine.ExecException;
import
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MRC
onfiguration;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.StorageUtil;

```

First are the java specific imports and the rest are Pig's imports.

If you have a problem adding imports and come across an error due to non-adding of imports, you can simply ask Eclipse to automatically add imports by using the shortcut Ctrl + Shift + O.

Step 3: The UDF class *MultiStorage* extends the *StoreFunc* abstract class which is the base class for all the store functions. Next, we declare all the required variables and implement a constructor. The variables include:

outputPath: User specified output Path where the output storage takes place.
splitFieldIndex: The field index based on which the output has to be split.
fieldDel: The output field delimiter.
comp: The compression type for the output data, which is optional. The compression types are specified as enumeration types.

```

public class MultiStorage extends StoreFunc {

    private Path outputPath;
    private int splitFieldIndex = -1;
    private String fieldDel;
    private Compression comp;

    enum Compression {
        none, bz2, bz, gz;
    };
}

```



```

public MultiStorage(String parentPathStr, String
splitFieldIndex) {
    this(parentPathStr, splitFieldIndex, "none");
}

public MultiStorage(String parentPathStr, String
splitFieldIndex, String compression) {
    this(parentPathStr, splitFieldIndex, compression,
"\t");
}

public MultiStorage(String parentPathStr, String
splitFieldIndex,
    String compression, String fieldDel) {
    this.outputPath = new Path(parentPathStr);
    this.splitFieldIndex =
Integer.parseInt(splitFieldIndex);
    this.fieldDel = fieldDel;
    try {
        this.comp = (compression == null) ?
Compression.none : Compression
        .valueOf(compression.toLowerCase());
    } catch (IllegalArgumentException e) {
        System.err.println("Exception when converting
compression string: " + compression + " to enum. No
compression will be used");
        this.comp = Compression.none;
    }
}

```

Step 4: We need to implement a custom *OutputFormat* for this function. The implementation of the custom *OutputFormat* for this UDF is *MultiStorageOutputFormat* as shown below.

```

public static class MultiStorageOutputFormat extends
TextOutputFormat<String, Tuple> {

    private String keyValueSeparator = "\t";
    private byte fieldDel = '\t';

```



```

@Override
public RecordWriter<String, Tuple>
getRecordWriter(TaskAttemptContext context) throws
IOException, InterruptedException {

    final TaskAttemptContext ctx = context;

    return new RecordWriter<String, Tuple>() {

        private Map<String, MyLineRecordWriter> storeMap =
            new HashMap<String, MyLineRecordWriter>();

        private static final int BUFFER_SIZE = 1024;

        private ByteArrayOutputStream mOut =
            new ByteArrayOutputStream(BUFFER_SIZE);

        @Override
        public void write(String key, Tuple val) throws
IOException {
            int sz = val.size();
            for (int i = 0; i < sz; i++) {
                Object field;
                try {
                    field = val.get(i);
                } catch (ExecException ee) {
                    throw ee;
                }
                StorageUtil.putField(mOut, field);

                if (i != sz - 1) {
                    mOut.write(fieldDel);
                }
            }
            getStore(key).write(null, new
Text(mOut.toByteArray()));
        }
    };
}

```



```

        mOut.reset();
    }

    @Override
    public void close(TaskAttemptContext context)
throws IOException {
    for (MyLineRecordWriter out : storeMap.values()) {
        out.close(context);
    }
}

private MyLineRecordWriter getStore(String fieldValue)
throws IOException {
    MyLineRecordWriter store = storeMap.get(fieldValue);
    if (store == null) {
        DataOutputStream os = createOutputStream(fieldValue);
        store = new MyLineRecordWriter(os, keyValueSeparator);
        storeMap.put(fieldValue, store);
    }
    return store;
}

private DataOutputStream createOutputStream(String
fieldValue) throws IOException {
    Configuration conf = ctx.getConfiguration();
    TaskID taskId = ctx.getTaskAttemptID().getTaskID();

        // Check whether compression is enabled, if so
        // get the extension and add them to the path
    boolean isCompressed = getCompressOutput(ctx);
    CompressionCodec codec = null;
    String extension = "";
    if (isCompressed) {
        Class<? extends CompressionCodec> codecClass =
            getOutputCompressorClass(ctx,
GzipCodec.class);
        codec = (CompressionCodec)
ReflectionUtils.newInstance(codecClass,
ctx.getConfiguration());
    }
}

```

```

        extension = codec.getDefaultExtension();
    }

    NumberFormat nf = NumberFormat.getInstance();
    nf.setMinimumIntegerDigits(4);

    Path path = new Path(fieldValue+extension, fieldValue
+ '-' + nf.format(taskId.getId())+extension);
    Path workOutputPath =
((FileOutputCommitter) getOutputCommitter(ctx)).getWorkPath();
    Path file = new Path(workOutputPath, path);
    FileSystem fs = file.getFileSystem(conf);
    FSDataOutputStream fileOut = fs.create(file, false);

    if (isCompressed)
        return new
DataOutputStream(codec.createOutputStream(fileOut));
    else
        return fileOut;
}

};

}

public void setKeyValueSeparator(String sep) {
    keyValueSeparator = sep;
    fieldDel = StorageUtil.parseFieldDel(keyValueSeparator);
}

protected static class MyLineRecordWriter extends
TextOutputFormat.LineRecordWriter<WritableComparable,
Text> {

    public MyLineRecordWriter(DataOutputStream out, String
keyValueSeparator) {
        super(out, keyValueSeparator);
    }
}
}

```

Step 4: Next, we determine the *OutputFormat*, so that Pig knows which output format has to be used while storing the output data. The *getOutputFormat()* is used to get an instance of the output format. For this Store UDF, this method returns a *MultiStorageOutputFormat()* which is the custom output format. We shall look at this custom *OutputFormat* in the next steps for this tasks.



```

@Override
    public OutputFormat getOutputFormat() throws
IOException {
    MultiStorageOutputFormat format = new
MultiStorageOutputFormat();
    format.setKeyValueSeparator(fieldDel);
    return format;
}

```

Step 5: Now that we have instructed Pig to use the output format required, we have to set the output location using the `setStoreLocation()`. This method sets the path to store output data as shown in the code below.

```

@Override
    public void setStoreLocation(String location, Job job)
throws IOException {

job.getConfiguration().set(MRConfiguration.TEXTOUTPUTFORMAT
AT_SEPARATOR, "");
    FileOutputFormat.setOutputPath(job, new
Path(location));
    if (comp == Compression.bz2 || comp == Compression.bz)
{
    FileOutputFormat.setCompressOutput(job, true);
    FileOutputFormat.setOutputCompressorClass(job,
BZip2Codec.class);
} else if (comp == Compression.gz) {
    FileOutputFormat.setCompressOutput(job, true);
    FileOutputFormat.setOutputCompressorClass(job,
GzipCodec.class);
}
}

```

Step 7: The next step is to prepare the store function to write data using the `prepareToWrite()`. It takes record write as an argument.



```

@Override
    public void prepareToWrite(RecordWriter writer) throws
IOException {
    this.writer = writer;
}

```

Step 8: The final step is to start writing the key-value pairs from *RecordWriter* using the *putNext()*. The store function takes the tuples and produces key-value pairs which are expected by the output format. The following shows the implementation:

```

private RecordWriter<String, Tuple> writer;

@Override
public void putNext(Tuple tuple) throws IOException {
    if (tuple.size() <= splitFieldIndex) {
        throw new IOException("split field index:" +
this.splitFieldIndex + " >= tuple size:" + tuple.size());
    }
    Object field = null;
    try {
        field = tuple.get(splitFieldIndex);
    } catch (ExecException exec) {
        throw new IOException(exec);
    }
    try {
        writer.write(String.valueOf(field), tuple);
    } catch (InterruptedException e) {
        throw new IOException(e);
    }
}

```

Step 8: Now that we have the UDF implemented, let us export the jar file. To do so, right click on the *udfs* package name and click on *Export...*

- You will have a popup asking you to choose the export destination. In the *Export* pop up, click the arrow button for Java so that it pop down its contents. Then click on *JAR file* and click on *Next >* button.



- In the *Jar Export* pop up, check the boxes for *.classpath* and *.project*. Click on browse and save the UDF to your home directory. You will be prompted to name your jar file, name it *SimpleTextLoader.jar*
- Click **Finish** to complete the process.

We shall use this store UDF in the next task by storing the output in multiple directories and files.

You can download the *MultiStorage.jar* and *MultiStorage.java* files from the URL below.

MultiStorage.jar - <https://db.tt/Rh94N0T0>
MultiStorage.java - <https://db.tt/Qrc2qUBY>

Task 3 is complete!

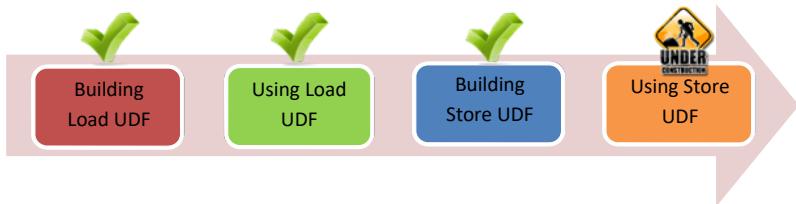
Task 4: Using the Store UDF

Step 1: Register the *MultiStorage.jar* UDF by specifying the path to the jar file in the Grunt shell.

```
grunt> REGISTER '/home/uzair/MultiStorage.jar';
```

```
grunt> REGISTER '/home/uzair/MultiStorage.jar';
2015-06-07 07:55:39,201 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

Step 2: Follow the steps 2 through 4 from Task 2: Using the Load UDF and return to the task for storing the data in multiple directories based upon the field.



Step 3: Now that we have the data loaded, let us store the data in multiple directories based on the field TagLabelCategory.

To achieve this, enter the following statement into the Grunt shell using the MultiStorage function. This function takes four arguments. The user specified path to store the data, the field index based on which the output has to be split, compression type (optional) and field delimiter.

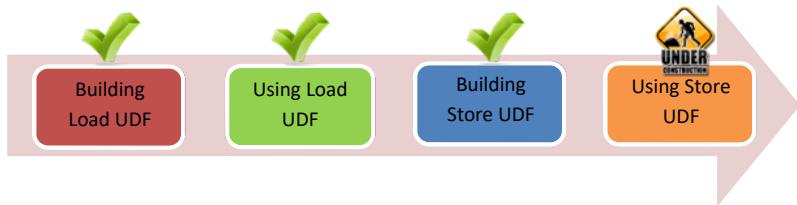
```
grunt> STORE A INTO '/pig/multiout' USING
udfs.MultiStorage('/pig/multiout', '2', 'none',
'\\t');
```

```
grunt> REGISTER '/home/uzair/MultiStorage.jar';
2015-06-07 07:55:39,201 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> STORE A INTO '/pig/multiout' USING udfs.MultiStorage('/pig/multiout', '2',
,'none', '\\t');
```

What does this do? Well, it splits the output and stores in the path /pig/multiout based upon the value of 3rd field which is TagLabelCategory. The fields are zero based count and hence we pass '2' to refer the 3rd field. We do not use any compression as it is very small dataset we are working in and hence we specify none. Finally, the delimiter is specified which is tab spaced.

Step 4: Once you have seen the success message, run the following *fs* command from the grunt shell to list the contents from /pig/multiout/ directory.

```
grunt> fs -ls /pig/multiout
```



```
grunt> fs -ls /pig/multiout/
Found 9 items
drwxr-xr-x  - uzair supergroup          0 2015-06-07 08:05 /pig/multiout/ILS200
2
drwxr-xr-x  - uzair supergroup          0 2015-06-07 08:05 /pig/multiout/ILS26
drwxr-xr-x  - uzair supergroup          0 2015-06-07 08:05 /pig/multiout/ILS300
2
drwxr-xr-x  - uzair supergroup          0 2015-06-07 08:05 /pig/multiout/ILS300
6
drwxr-xr-x  - uzair supergroup          0 2015-06-07 08:05 /pig/multiout/ILS301
4
drwxr-xr-x  - uzair supergroup          0 2015-06-07 08:05 /pig/multiout/ILS40
drwxr-xr-x  - uzair supergroup          0 2015-06-07 08:05 /pig/multiout/ILS60
-rw-r--r--  1 uzair supergroup          0 2015-06-07 08:05 /pig/multiout/_SUCCE
SS
drwxr-xr-x  - uzair supergroup          0 2015-06-07 08:05 /pig/multiout/null
grunt> clear
```

You should have the directories split based on the TagLabelCategory field. There are a total of 7 Tag label categories including a null as shown in the screenshot above.

Task 4 is complete!

SUMMARY

User Defined Functions helps you write your own custom function to process your data. There might be cases where you cannot find a built-in function suitable for your needs and this is where a user defined function helps you. Pig UDFs can be written and implemented in Java, Jython, Python, JavaScript, Ruby and Groovy. However, since Pig is written in Java, UDFs written in Java are most efficient with greater support while the others have limited support. With UDFs every part of processing can be customized including data load/store, column transformation, and aggregation.

We have covered Load/Store functions in this chapter. We have already covered the Eval and Filter UDFs in the previous chapter. This completes the user defined functions section.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/udf.html>

INDEX

Load Functions.....	289
Store Functions.....	292
AIM	295
Lab Exercise 9: Bulding Load and Store UDFs	296
Task 1: Building a Load Function.....	297
Task 2: Using the Load UDF.....	302
Task 3: Building a Store Function.....	304
Task 4: Using the Store UDF.....	313
SUMMARY	316
REFERENCES.....	317

CHAPTER 10: EMBEDDING PIG LATIN

Theory

We have been using Pig, a data flow language, till this point to perform ETL operations by tapping in into large datasets, performing operations, using built-in and user defined functions and finally aggregating the results. This works good for most of the data processing applications as we perform all the above ETL operations and obtain the results either to std-out or to a file. However, few applications require the data flow operations to iterate themselves till the desired result is achieved and Pig lacks this controlled environment of processing data. There is no facility in Pig Latin where you can use the conditional statements such as *if*, *while*, *for* etc to iterate a piece code over and over again. Therefore, to overcome this inability, Pig is embedded with low-level programming languages such as Java, Python, Groovy etc. This is similar to embedding database API with the scripting languages such as PHP.

In this chapter, we shall look at embedding Pig in Python and Java. However, Python is most preferred language for embedding Pig in. There are many reasons behind it but the main reason is that Python is one of the most popular languages used by data scientists and so most of them would be very happy to embed Pig with python.

Embedding Pig in Python

Embedding Pig in Python is possible in batch mode only and not possible in Interactive mode. To run the embedded Pig Latin in Python script, we have to run it using the Pig binaries. For example, if we have a Python script with name *helloworld.py* with Pig Latin embedded in it, we have to run it using the following command from the command shell.

```
$ pig helloworld.py
```

Pig then looks for the *!#* line in the script and calls the interpreter for Python. As the Pig native API is a Java class, Jython interpreter is used to run the Python scripts which have Pig Latin embedded in them. We need to make sure that we have the Jython jar file available in the Pig class path. For example, the following line of code is used to call the interpreter.

```
#!/usr/bin/python
```

Next, we have to explicitly import the Pig class and its objects in the python script as shown below.

```
from org.apache.pig.scripting import Pig
```

Finally, the goal of embedding Pig with Python can be achieved by fulfilling the following steps.

Compile

Compile is a static Pig method in which the Pig Latin code is first compiled. The Pig Latin code consists of all the required loading, transforming and storing the output operations. We can also specify variables as parameters within the Pig Latin statements and can be later assigned the values in the **Bind** phase. The parameters will be indicated by a dollar sign followed by a sequence of alpha-numeric or underscore characters. An example of compile method in Python containing Pig Latin code is as shown below.

```
P = Pig.compile("""data = LOAD '$input'; STORE data  
INTO '$output';""")
```

In the example above, *Pig.compile* static method does the initial compilation of Pig Latin code by verifying the syntax only. No verification is done for type checking during this phase. Once the compilation is done, the method returns a *Pig* object which can then be bounded to variables in the next phase. These variables are provided by the control flow script.

In addition to the above compile method where we compile a block of Pig Latin statements, we can also assign a name to this block so that we can import this block to another block of Pig Latin statements so that we need not type the entire code again the other block. An example of this scenario is as shown below:

```
A = Pig.compile("load", """data = LOAD  
'$input';""")  
  
B = Pig.compile("""import load; STORE data INTO  
'$output';""")
```

In the example above, we have first compiled the code in first line and named the block of code as *load*. Next, we use the same block of code in second statement simplifying importing it along with the other Pig statements.

There might also be few scenarios where you would like to embed Pig script stored in file. You can achieve this too, using the *compileFromFile* method. An example for this method is as shown below:

```
P = Pig.compileFromFile("pig_script.pig")
```

You can also assign a name as seen in the *compile* method when you pass a Pig script as shown below.

```
P = Pig.compileFromFile("first", "pig_script.pig")
```

Bind

After the Pig Latin code is compiled in the first phase, the next phase is to bind the variables in our control flow script with the variables of the Pig Latin script. This is done by the *bind* method which binds the variables and returns a *BoundScript* object. An example of bind is as shown below.

```
in = "in_data"
out = "out_data"

Q = P.bind({'input':in, 'output':out})
```

Moreover, if all the control script variables and Pig Latin variables are declared with same names, you may simply call the *bind* method without any arguments. An implicit bind is performed in this case. Pig internally looks for the parameters automatically from the variables specified by the user in the script. The bind method without arguments is as shown below.

```
Q = P.bind()
```

Also, there might be a scenario where you would want to run your compiled Pig Latin script in parallel with a multiple set of variables in a single go instead of running compiled Pig Latin separately for different variables multiple times and iterate over the data. This is possible by assigning a list of maps of parameters instead of assigning map of parameters to the *bind*. For example consider calculating salaries of employees from different departments in a company as shown below.

```
#!/usr/bin/python
from org.apache.pig.scripting import Pig
P = Pig.compile("""
input = LOAD '$dept' USING PigStorage();
...
STORE output INTO '$dept_out';
""")
params = [{ 'dept': 'IT', 'dept_out': 'it.out' },
...
{ 'dept': 'Procurement', 'dept_out': 'proc.out' }]
bound = P.bind(params)
stats = bound.run()
```

As you can see from the example above, we use different sets of variables and run the script in parallel against the inputs from all the departments at the same time.

Run

In the previous phase we have a *BoundScript* object generated by the *bind* method. Now, we can run it using the *runSingle* method. This method is legal only when a single set of variables are bounded with the script. Calling *runSingle* method when list of maps of parameters are bounded with the script will throw an exception. The *runSingle* method returns a *PigStats* object. This object helps you get all the statistics, error messages, status messages etc. An example showing *runSingle* method is as follows.

```
result = Q.runSingle()
```

We can then check if the result was successful or get error messages using the methods provided by the *PigStats* object as shown below.

```
if result.isSuccessful():
    print "Pig job succeeded"
else:
    raise "Pig job failed"
```

Or we can get the error messages

```
result.getErrorMessage()
```

Please refer to the URL in REFERENCES section for all the PigStats methods.

However, if you have your Pig script bounded with list of maps of parameters, you will have to call the *run* method instead of calling the *runSingle* method. Pig will then start processing the data in parallel.

Embedding Pig in Java

Pig Latin can be embedded in Java programming language and extend its capability by enabling controlled environment for processing data.

PigServer

In order to embed Pig Latin in Java, we have to use the *PigServer* class. *PigServer* provides you with methods required for embedding Pig Latin in Java. There are a lot of methods provided by the *PigServer* and can be found from the URL in REFERENCES section.

The embedded Pig Latin script in Java can be run in local as well as MapReduce mode. When we choose to run in local mode, the data present in the local file system will be processed while in MapReduce mode, the data present in the HDFS will be processed. All we have to do is specify *local* or *mapreduce* as argument while creating an instance of *PigServer* class. For example, the following line of code shows creating an instance of *PigServer* class and running Pig in MapReduce mode.

```
PigServer pigServer = new PigServer("mapreduce");
```

Pig statements are then embed into Java using the *registerQuery()* method as shown below.

```
pigServer.registerQuery("A = LOAD '" + inputFile +  
" USING PigStorage(',')");
```

Next, the results can then be stored using the *or store()* method as shown below.

```
pigServer.store("A", "output");
```

Moreover, you may also use the *openIterator()* method to retrieve the results. This method is used to execute Pig Latin code up to and the specified alias. For example, if we have the following Pig Latin code:

```
pigServer pigServer = new PigServer();  
pigServer.registerQuery("A = load 'input';");  
pigServer.registerQuery("B = filter A by $0 > 0;");  
pigServer.registerQuery("C = order B by $1;");
```

And then when we use the *openIterator()* method as shown below:

```
pigServer.openIterator("B");
```

Only the Pig Latin till the alias *B* is executed and the result would be unsorted filtered results. However, if you want the filtered as well as sorted results, you would specify something as shown below.

```
pigServer.openIterator("C");
```

Optionally, you may use the *shutdown()* method to free the resources used by *PigStorage* instance so as to prevent a possibility of memory leak.

Do not worry if you are confused at this point. We shall be looking at an example for embedding Pig in Java in our Lab Exercises and you should be able to easily understand then.

That's all the theory we have for this chapter. Let us proceed to the Lab Exercises and get our hands on what we have just learned.

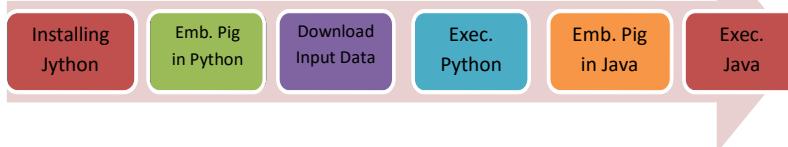
AIM

The aim of the following lab exercise is to have your hands on embedding Pig Latin code in Python and Java programming languages.

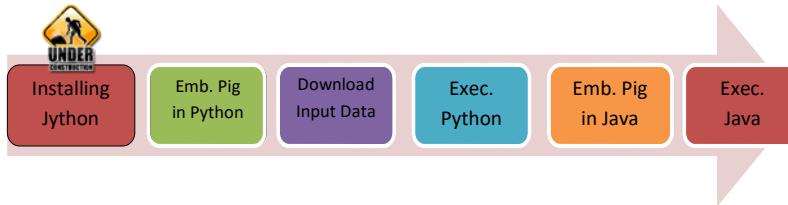
Following steps are required:

- Task 1: Installing Jython
- Task 2: Embedding Pig in Python
- Task 3: Downloading Input Data
- Task 4: Executing Pig Embedded Python Script
- Task 5: Embedding Pig in Java
- Task 6: Executing Pig Embedded Java Program

Lab Exercise 10: EMBEDDING PIG LATIN



1. **Installing Jython**
2. **Embedding Pig in Python**
3. **Downloading Input Data**
4. **Executing Pig Embedded python Script**
5. **Embedding Java in Python**
6. **Executing Pig Embedded Java Program**



Task 1: Installing Jython

We start our lab exercises by installing and configuring Jython by making it available in the Pig class path.

Step 1: Open your web browser and navigate to the URL as shown below.

<http://www.jython.org/downloads.html>

Step 2: Now, click on *Download Jython 2.7.0 - Standalone Jar* and save the jar file to your home directory. We just need the standalone jar and not the full installer as we are only interested in Jython for interpretation purposes. We shall be using the most stable version which is Jython 2.7.0.

ABOUT
[Welcome](#)
[Download](#)
[Installation](#)
[Mailing Lists](#)
[License](#)
[Foundation](#)
[Archived Sites](#)
[Acknowledgments](#)
DOCUMENTATION
[Current Docs](#)
[Core Development](#)
[Jython Book](#)
[WIKI LINKS](#)

Downloads

The most current stable release of Jython is 2.7.0. For production purposes, please use this version.

Please use the link below to download the Java installer. Once downloaded, please double-click on the JAR file to start the installation process. You may also want to read the [Installation instructions](#) or the [Release Notes](#).

Jython 2.7.0

[Download Jython 2.7.0 - Installer : Executable jar for installing Jython](#)
[Download Jython 2.7.0 - Standalone Jar](#) For embedding Jython in Java applications

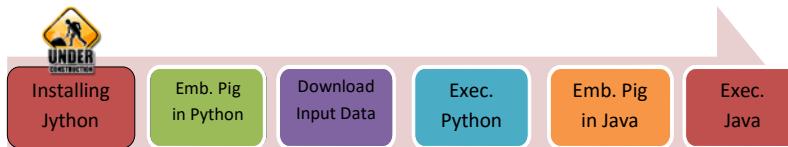
Jython 2.5.4rc1

[Download Jython 2.5.4rc1 - Installer : Executable jar for installing Jython](#)
[Download Jython 2.5.4rc1 - Standalone Jar](#) For embedding Jython in Java applications

Step 3: Make a new directory in */usr/local* directory and name it *jython-install*. Assuming you have downloaded the Jython jar file to your home directory, copy the Jython jar file to the directory you have just created. Please use the following commands from the command line interface.

```
$ sudo mkdir /usr/local/jython-install
```

```
$ sudo mv jython-standalone-2.7.0.jar
/usr/local/jython-install
```



```
uzair@ubuntu:~$ sudo mkdir /usr/local/jython-install
[sudo] password for uzair:
uzair@ubuntu:~$ sudo mv jython-standalone-2.7.0.jar /usr/local/jython-install
uzair@ubuntu:~$
```

Step 4: Finally, export the *jython-standalone-2.7.0.jar* to the *PIG_CLASSPATH* variable in your *.bashrc* file. To do this, open the *.bashrc* file from command line interface using the following command.

```
$ sudo gedit .bashrc
```

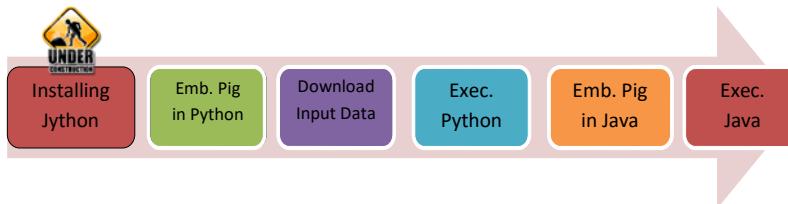
Once the *.bashrc* file has opened, copy and paste the line below as shown in the screenshot.

```
export
PIG_CLASSPATH=$PATH:/usr/local/jython-
install/jython-standalone-2.7.0.jar

if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
fi

export JAVA_HOME=/usr/local/jdk1.8.0_31
export PATH=$PATH:$JAVA_HOME/bin
export HADOOP_PREFIX=/usr/local/hadoop-2.6.0
export PATH=$PATH:$HADOOP_PREFIX/bin
export PATH=$PATH:$HADOOP_PREFIX/sbin
export HADOOP_MAPRED_HOME=$HADOOP_PREFIX
export HADOOP_COMMON_HOME=$HADOOP_PREFIX
export HADOOP_HDFS_HOME=$HADOOP_PREFIX
export YARN_HOME=$HADOOP_PREFIX
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_PREFIX/lib/native
export HADOOP_OPTS="-Djava.net.preferIPv4Stack=true"
export HADOOP_OPTS="-Djava.library.path=$HADOOP_PREFIX/lib"
export PIG_PREFIX=/usr/local/pig-0.14.0
export PIG_CLASSPATH=$PATH:/usr/local/jython-install/jython-standalone-2.7.0.jar
export PATH=$PATH:$PIG_PREFIX/bin
export PATH
```

Save and then refresh the *.bashrc* file using the following command as shown below.



```
$ . .bashrc
```

This completes the installation and configuration of Jython. We can now proceed with embedding Pig Latin in Python script.

Task 1 is complete!

Task 2: Embedding Pig in Python

We shall embed Pig Latin in Python to calculate the PageRank of websites using the data obtained from crawling the World Wide Web. According to Wikipedia, *PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.*

You may learn more about PageRank from the URL in REFERENCES section.

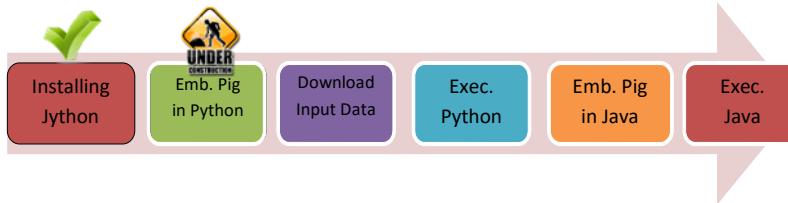
The input data obtained from crawling the internet is in the following format. Every record contains a URL, initial rank of 1 and a bag containing tuples with outbound links of that URL as shown below.

http://www.google.com/intl/en/privacy/tools.html	1	{ (http://www.google.com/intl/en/privacy/faq.html), (http://www.google.com/intl/en/privacy/blogs.html) }
http://www.google.com/intl/en/privacy/faq.htm	1	{ (http://www.google.com/intl/en/privacy/tools.html) }
http://www.google.com/intl/en/privacy/blogs.html	1	{ }

An empty bag represents no outbound links for that particular URL.

The algorithm to calculate PageRank is as shown below:

$$PR(A) = (1-d) + d (PR(T1)/C(T1) + \dots + PR(Tn)/C(Tn))$$



Where

$PR(A)$ is the PageRank of page A

$PR(T1)$ is the PageRank of pages $T1$ which link to page A,

$C(Ti)$ is the number of outbound links on page Ti and d is a damping factor which can be set between 0 and 1.

Step 1: Open *gedit* from terminal using the following command. We shall be using *gedit* as editor for the script we are about to create.

```
$ sudo gedit
```

Step 2: Once the editor has opened, enter the following line of code which tells Pig to call the interpreter for Python.

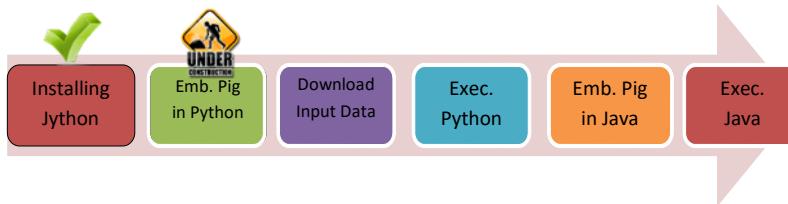
```
#!/usr/bin/python
```

Step 3: Next, we need to explicitly import the Pig class to be able to use it in our Python script.

```
from org.apache.pig.scripting import *
```

Step 4: The next step is where all the embedding takes place which is the *Compile* phase. We embed all the Pig Latin code in this phase as shown below.

```
P = Pig.compile("""
previous_pageRank =
    LOAD '$docs_in'
    USING PigStorage('\t')
    AS (url: chararray, pageRank: float, links:{link: (url: chararray) } );
outbound_pageRank =
    FOREACH previous_pageRank
    GENERATE pageRank / COUNT (links) AS pageRank,
    FLATTEN (links) AS to_url;
```



```

new_pageRank =
    FOREACH (COGROUPOUTBOUND_PAGERank BY to_url,
previous_pageRank BY url INNER)
    GENERATE group AS url, (1 - $d) + $d * SUM
    (outbound_pageRank.pageRank) AS pageRank, FLATTEN
    (previous_pageRank.links) AS links;

STORE new_pageRank
    INTO '$docs_out' USING PigStorage('\t');"""

```

The above Pig Latin code takes the input data and calculates the page rank for each webpage using the algorithm which we have seen earlier in this task. The variables `$docs_in`, `$d` and `$docs_out` are the parameters which are provided during the bind phase. Your script at this point of time should look similar to the one shown in screenshot below.

```

#!/usr/bin/python
from org.apache.pig.scripting import *

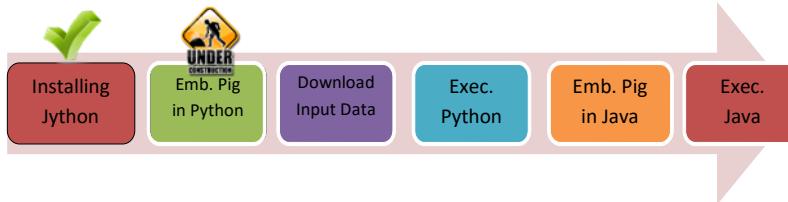
P = Pig.compile("""| 

previous_pageRank =
    LOAD '$docs_in'
    USING PigStorage('\t')
    AS ( url: chararray, pageRank: float, links:{ link: ( url: chararray ) } );

outbound_pageRank =
    FOREACH previous_pageRank
    GENERATE
        pageRank / COUNT ( links ) AS pageRank,
        FLATTEN ( links ) AS to_url;

new_pageRank =
    FOREACH
        ( COGROUPOUTBOUND_PAGERank BY to_url, previous_pageRank BY url INNER )
    GENERATE
        group AS url,
        ( 1 - $d ) + $d * SUM ( outbound_pageRank.pageRank ) AS pageRank,
        FLATTEN ( previous_pageRank.links ) AS links;

STORE new_pageRank
    INTO '$docs_out',
    USING PigStorage('\t');
""")
```



Step 5: The next step is to bind the variables in our control flow script with the variables of the Pig Latin script. This is done by the *bind* method which binds the variables and returns a *BoundScript* object. Once binded, we finally run it using the *runSingle* method. We control the flow by iterating until the condition is satisfied and get the stats of the job using the *PigStats* object.

```
params = {'d': '0.5', 'docs_in': 'data/pagerank_data_simple'}

for i in range(10):
    out = "out/pagerank_data_" + str(i + 1)
    params["docs_out"] = out
    Pig.fs("rm -r " + out)
    stats = P.bind(params).runSingle()
    if not stats.isSuccessful():
        raise 'failed'
    params["docs_in"] = out
```

All the above code does is binds the parameters using the *bind()* method and triggers the *runSingle()* method for each iteration until the condition is met i.e., 10 times.

Step 6: Finally, save this script and name it as *pagerank.py*. Change the permissions of the script, if required.

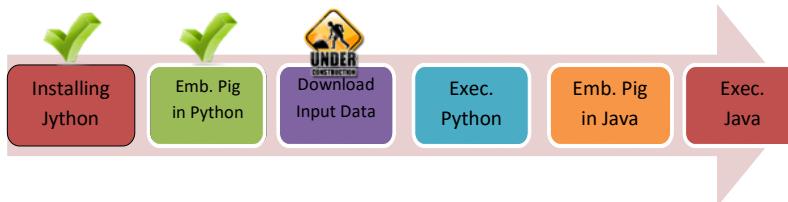
This completes embedding Pig Latin in Python script. You may download this script from the URL below and save it to your home directory.

pagerank.py - <https://db.tt/0pDknuw6>

Task 2 is complete!

Task 3: Downloading Input Data

Step 1: Download the file *pagerank_data_simple* from the URL below and save it to your home directory.



pagerank_data_simple – <https://db.tt/Y7EeeWZ9>

Step 2: Start all the Hadoop daemons. Make a new directory in HDFS and name it *data*. Finally copy the file *pagerank_data_simple* to the *data* directory

```
$ hadoop fs -mkdir data
$ hadoop fs -copyFromLocal pagerank_data_simple
data
```

Now that we have the input data ready, let us execute the Pig embedded Python script.

Task 3 is complete!

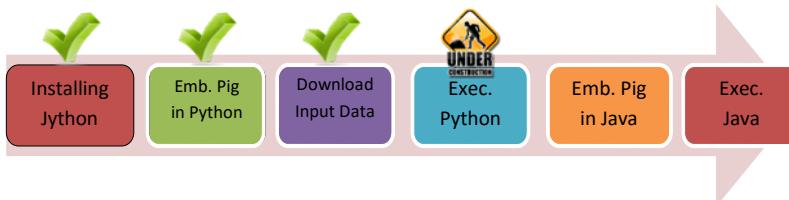
Task 4: Executing Pig Embedded Python Script

Step 1: From the terminal execute the following. Make sure you have all the Hadoop daemons up and running.

```
$ pig pagerank.py
```

```
uzair@ubuntu:~$ pig pagerank.py
15/06/29 18:10:37 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/06/29 18:10:37 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/06/29 18:10:37 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
2015-06-29 18:10:37,764 [main] INFO org.apache.pig.Main - Apache Pig version 0.
14.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-06-29 18:10:37,770 [main] INFO org.apache.pig.Main - Logging error messages to: /home/uzair/pig_1435626637761.log
2015-06-29 18:10:40,124 [main] WARN org.apache.hadoop.util.NativeCodeLoader - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2015-06-29 18:10:40,757 [main] INFO org.apache.pig.Main - Run embedded script: python
2015-06-29 18:10:41,185 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
```

This job will run for a fair amount of time depending upon the hardware of your machine and will end up showing you the *Success* message as shown in the screenshot below indicating that the job has been successfully completed.



```

2015-06-29 18:46:11,918 [main] INFO org.apache.hadoop.yarn.client.RMProxy - Connecting to ResourceManager at /0.0.0.0:8032
2015-06-29 18:46:11,928 [main] INFO org.apache.hadoop.mapred.ClientServiceDelegate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirecting to job history server
2015-06-29 18:46:11,995 [main] INFO org.apache.hadoop.yarn.client.RMProxy - Connecting to ResourceManager at /0.0.0.0:8032
2015-06-29 18:46:12,000 [main] INFO org.apache.hadoop.mapred.ClientServiceDelegate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirecting to job history server
2015-06-29 18:46:12,101 [main] WARN org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLauncher - Encountered Warning DIVIDE_BY_ZERO 166 times.
2015-06-29 18:46:12,101 [main] INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLauncher - Success!
2015-06-29 18:46:12,218 [main] INFO org.apache.pig.Main - Pig script completed in 16 minutes, 56 seconds and 137 milliseconds (1016137 ms)

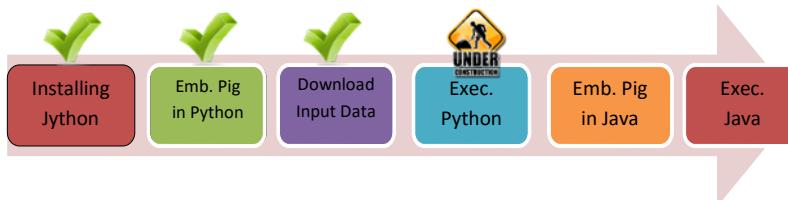
```

Step 2: The job at this point of calculating the PageRank has been completed. However, the results are not in a sorted order. We need to run the following pig statements in order to check the maximum and minimum PageRank values for the URLs. Make sure you have started Pig in Grunt shell and execute the following Pig statements.

```

grunt> A = LOAD 'pagerank_data_10' AS (url: chararray, pagerank: float, links:{ link: ( url: chararray ) });
grunt> B= GROUP A ALL;
grunt> C = FOREACH B generate FLATTEN(TOBAG(MAX(A.pagerank), MIN(A.pagerank))) as pagerank;
grunt> D = JOIN C BY pagerank, A BY pagerank;
grunt> E = FOREACH D GENERATE C::pagerank, A::url;
grunt> DUMP E;

```



```

grunt> A = LOAD 'out/pagerank_data_10' AS ( url: chararray, pagerank: float, links:{ link: ( url: chararray ) } );
2015-06-29 19:04:07,901 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> B= GROUP A ALL;
grunt> C = FOREACH B generate FLATTEN(TOBAG(MAX(A.pagerank), MIN(A.pagerank))) as pagerank;
grunt> D = JOIN C BY pagerank, A BY pagerank;
grunt> E = FOREACH D GENERATE C::pagerank, A::url;
grunt> DUMP E;
2015-06-29 19:04:46,701 [main] INFO org.apache.pig.tools.pigstats.ScriptState - Pig features used in the script: HASH_JOIN,GROUP_BY
2015-06-29 19:04:46,994 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-06-29 19:04:46,995 [main] INFO org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-06-29 19:04:47,002 [main] INFO org.apache.pig.newplan.logical.optimizer.LogicalPlanOptimizer - {RULES_ENABLED=[AddForEach, ColumnMapKeyPrune, ConstantCalculator, GroupByConstParallelSetter, LimitOptimizer, LoadTypeCastInserter, Merge]}

```

The result is as shown below with the maximum and minimum values.

```

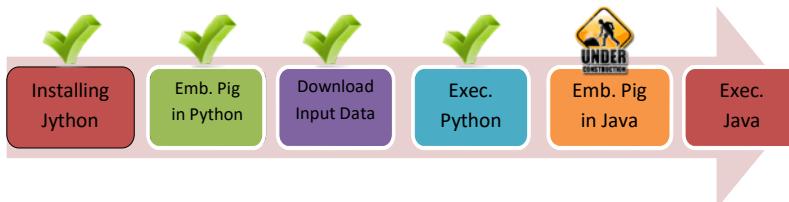
ne.mapReduceLayer.MapReduceLauncher - Success!
2015-06-29 19:07:17,394 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-06-29 19:07:17,398 [main] INFO org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2015-06-29 19:07:17,424 [main] INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths to process : 1
2015-06-29 19:07:17,424 [main] INFO org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input paths to process : 1
(0.50249785,http://google-web-toolkit.googlecode.com/svn/javadoc/latest/index.html?overview-summary.html)
(10.335426,http://www.google.com/intl/en/privacy_faq.html)
(10.335426,http://www.google.com/intl/en/privacy_faq.html)
grunt>

```

Task 4 is complete!

Task 5: Embedding Pig in Java

Now that we have completed embedding Pig in python, let us look at the procedure of embedding Pig in Java. For the sake of simplicity, let us embed Pig Latin in Java to extract all the items from the file *Items.tsv*.



Step 1: Download the file *Items.tsv* from the URL below and save it to your HDFS home directory, if you have not already downloaded during the previous lab exercises.

Items.tsv - <https://db.tt/ZtPvkfbT>

Step 2: Before we proceed with the code, we first have to export few libraries in the *.bashrc* file. We need to setup the environment variable *PIG_JAVA_CLASSPATH*. To do so, open the *.bashrc* file and add the line as shown below.

```
export
PIG_JAVA_CLASSPATH=$PIG_JAVA_CLASSPATH:/usr/local/pig-0.14.0/pig-0.14.0-core-
h2.jar:$HADOOP_PREFIX/share/hadoop/tools/lib/*:$HADOOP_PREFIX/share/hadoop/common/*
:$HADOOP_PREFIX/share/hadoop/hdfs/*:$HADOOP_PREFIX/share/hadoop/mapreduce/*
:$HADOOP_PREFIX/share/hadoop/common/lib/*:$PIG_PREFIX/lib/*:$HADOOP_PREFIX/share/hadoop/yarn/*
```

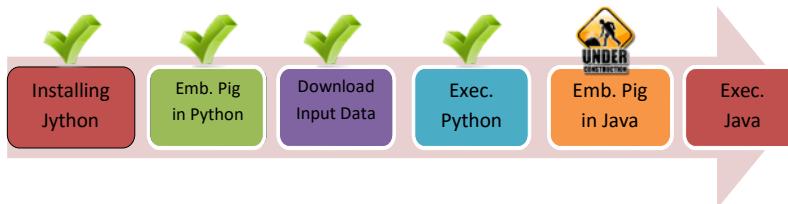
```
##!/usr/share/bash-completion/bash_completion
elf [-f /etc/bash_completion ]; then
. /etc/bash_completion
fi

export JAVA_HOME=/usr/local/jdk1.8.0_31
export PATH=$PATH:$JAVA_HOME/bin
export HADOOP_PREFIX=/usr/local/hadoop-2.6.0
export PATH=$PATH:$HADOOP_PREFIX/bin
export HADOOP_CLASSPATH=$HADOOP_PREFIX/lib
export HADOOP_HOME=$HADOOP_PREFIX
export HADOOP_HDFS_HOME=$HADOOP_PREFIX
export HADOOP_COMMON_HOME=$HADOOP_PREFIX
export HADOOP_HDFS_HOME=$HADOOP_PREFIX
export YARN_HOME=$HADOOP_PREFIX
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_PREFIX/lib/native
export HADOOP_OPTS="-Djava.net.preferIPv4Stack=true"
export HADOOP_CONF_DIR=$HADOOP_PREFIX/lib
export PIG_PREFIX=/usr/local/pig-0.14.0
export PIG_CLASSPATH=$PIG_PREFIX:/usr/local/python-Install/python-standalone-2.7.0.jar
export PIG_JAVA_CLASSPATH=$PIG_JAVA_CLASSPATH:/usr/local/pig-0.14.0/pig-0.14.0-core-h2.jar:$HADOOP_PREFIX/share/hadoop/tools/lib/*:$HADOOP_PREFIX/share/hadoop/common/*:$HADOOP_PREFIX/share/hadoop/hdfs/*:$HADOOP_PREFIX/share/hadoop/mapreduce/*:$HADOOP_PREFIX/share/hadoop/common/*:$HADOOP_PREFIX/lib/*:$HADOOP_PREFIX/share/hadoop/yarn/*
```

```
export PATH=$PATH:$PIG_PREFIX/bin
export PATH
```

Make sure you specify correct path to these libraries in case you have installed Hadoop and Pig somewhere else other than the above paths.

Save and refresh the *.bashrc* file.



Step 3: Once done with `.bashrc`, create a new empty document in your home directory and name it `Extract.java`.

Step 4: The next step is to add the imports required so that we can access all the classes and interfaces required. We need to add these imports below.

```
import java.io.IOException;
import org.apache.pig.PigServer;
```

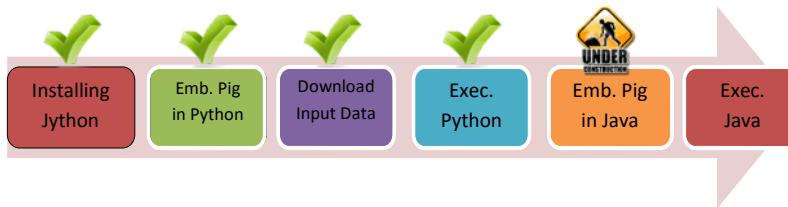
Step 5: Next we have to create a new instance of `PigServer` and pass in the argument `mapreduce` as we would run Pig in MapReduce mode. You may also choose to run local mode. For that you will have to specify `local` in the argument.

Extract.java

```
import java.io.IOException;
import org.apache.pig.PigServer;

public class Extract {
    public static void main(String[] args) {
        try {
            PigServer pigServer = new
PigServer("mapreduce");
            runItemQuery(pigServer, "Items.tsv");
        }
        catch(Exception e) {
        }
    }
    public static void runItemQuery(PigServer pigServer,
String inputFile) throws IOException {

        pigServer.registerQuery("A = load '" + inputFile +
"' using PigStorage();");
        pigServer.registerQuery("B = foreach A generate $1
as item;");
        pigServer.store("B", "items");
    }
}
```



Once a new instance of *PigServer* is created, we implement a new method called *runItemQuery()* which takes in the arguments of the *pigServer* object and the name of the input file stored in the HDFS home directory. Within the *runItemQuery()* method, we use the *registerQuery()* and *store()* methods to embed the Pig Latin code. First we load the input file and then extract the items from the input data. Finally store the extracted data in the *items* file.

Step 6: Finally save the file. With this we have successfully completed embedding Pig Latin in Java. All we need to do is test our script by executing it.

You may download the file *Extract.java* and *Extract.class* from the URLs below.

Extract.java - <https://db.tt/oUW1RBc0>

Extract.class - <https://db.tt/LfJ2Hos1>

Task 5 is complete!

Task 6: Executing Pig Embedded Java Program

Step 1: From the terminal run the following command to compile the java program using the Pig Jar.

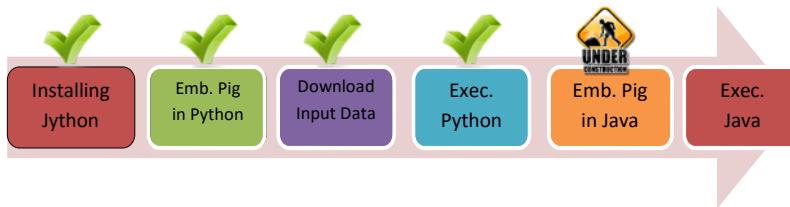
```
$ javac -cp /usr/local/pig-0.14.0/pig-0.14.0-core-h2.jar Extract.java
```

This will create a new *Extract.class* file in your current working directory which is the home directory.

Step 2: Finally execute the program using the following command from your terminal.

```
$ java -cp $PIG_JAVA_CLASSPATH:. Extract
```

Observe the '.' in the class path. It indicates the path to the compiled class file which is the current directory. If you have saved the class file somewhere else, make sure you include the correct path.



```
uzair@ubuntu:~$ javac -cp /usr/local/pig-0.14.0/pig-0.14.0-core-h2.jar Extract.java
uzair@ubuntu:~$ java -cp $PIG_JAVA_CLASSPATH:. Extract
15/06/30 19:21:38 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/06/30 19:21:38 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/06/30 19:21:38 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
15/06/30 19:21:40 INFO Configuration.deprecation: mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
15/06/30 19:21:40 INFO Configuration.deprecation: fs.default.name is deprecated. Instead, use fs.defaultFS
15/06/30 19:21:40 INFO executionengine.HExecutionEngine: Connecting to hadoop file system at: hdfs://localhost:9000
15/06/30 19:21:43 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/06/30 19:21:45 INFO Configuration.deprecation: fs.default.name is deprecated. Instead, use fs.defaultFS
15/06/30 19:21:47 INFO Configuration.deprecation: fs.default.name is deprecated. Instead, use fs.defaultFS
15/06/30 19:21:48 INFO Configuration.deprecation: mapred.textoutputformat.separator is deprecated. Instead, use mapreduce.output.textoutputformat.separator
```

This will start the job and provides you with a success message as shown in the screenshot below.

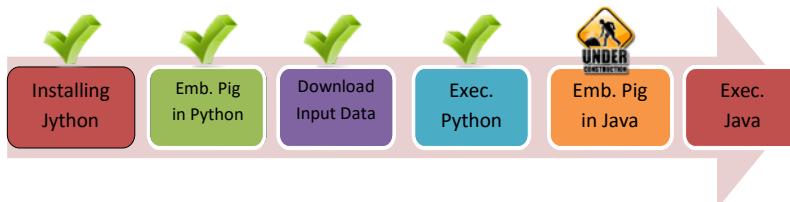
```
Job Stats (time in seconds):
JobId  Maps  Reduces  MaxMapTime  MinMapTime  AvgMapTime  MedianMapTime  MaxRed
uceTime  MinReduceTime  AvgReduceTime  MedianReduceTime  Alias  Feature  Outputs
job_1435700448803_0003  1  0  28  28  28  28  0  0  0  0  0
A,B  MAP_ONLY  items,
```

Input(s):
Successfully read 1035 records (62063 bytes) from: "hdfs://localhost:9000/user/uzair/Items.tsv"

Output(s):
Successfully stored 1035 records (26273 bytes) in: "items"

Counters:
Total records written : 1035
Total bytes written : 26273
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

If you have encountered a strange error while running the program, make sure you have included all the libraries in the class path. Also make sure that you have the same version of Java for both Java compiler and JRE. If you have compiled the java program with a higher version of compiler and try to run the program with a lower version of JRE, you will be greeted by an error.



Step 3: Once the job is complete, check the output of the job using the following command.

```
$ hadoop fs -cat items/p*
```

As you can see from the screenshot below, we have successfully extracted the item names by embedding Pig Latin in Java program.

```
Banana Sweethearts 200g
Banana Sugar Daddy 500g
Cherry Peach Blossoms 500g
Blackberry Jawbreaker 200g
Cherry Sugar Daddy 100g
Cherry Gummi Cherries 500g
Blackberry Bit-o-Honey 500g
Banana Rocky Road 200g
Cherry Warheads 500g
Apple Sugar Daddy 500g
Raspberry Rocky Road 200g
Blueberry Sour Patch Kids 100g
Banana Sour Patch Kids 200g
Banana Sweethearts 100g
Banana Gummi Bears 100g
Blueberry Boston Fruit Slices 100g
Cherry Abba-Zaba 500g
Blackberry Look! 500g
```

Task 6 is complete!

SUMMARY

We have been using Pig, a data flow language, till this point to perform ETL operations by tapping in into large datasets, performing operations, using built-in and user defined functions and finally aggregating the results. This works good for most of the data processing applications as we perform all the above ETL operations and obtain the results either to std-out or to a file. However, few applications require the data flow operations to iterate themselves till the desired result is achieved and Pig lacks this controlled environment of processing data. There is no facility in Pig Latin where you can use the conditional statements such as *if*, *while*, *for* etc to iterate a piece code over and over again. Therefore, to overcome this inability, Pig is embedded with low-level programming languages such as Java, Python, Groovy etc. This is similar to embedding database API with the scripting languages such as PHP.

We have covered embedding Pig Latin in Python as well as Java programming languages during the lab exercises.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/cont.html>
- <https://techblog.wordpress.com/2011/07/29/pagerank-implementation-in-pig/>
- <https://en.wikipedia.org/wiki/PageRank>
- <http://pig.apache.org/docs/r0.13.0/api/org/apache/pig/PigServer.html>

INDEX

Embedding Pig in Python	319
Embedding Pig in Java.....	322
AIM	325
Lab Exercise 10: Embedding Pig Latin	326
Task 1: Installing Jython	327
Task 2: Embedding Pig in Python	329
Task 3: Downloading Input Data	332
Task 4: Executing Pig Embedded in Python Script	333
Task 5: Embedding Pig in Java.....	335
Task 6: Executing Pig Embedded in Java Program	338
SUMMARY	341
REFERENCES.....	342

CHAPTER 11: ADMINISTERING & TESTING PIG

Theory

We have been using Pig all this time to get familiar with developing Pig Latin scripts to process big data. In this chapter, we shall take a break from developing stuff and focus on administering Pig along with tools for developing and testing. Let us kick start this chapter by looking at some administration stuff.

Administering Pig

Let us start administering Pig by customizing Pig behaviour using the Java properties supported by Pig and Hadoop. The Java properties are the configuration values in the form of key/value pairs. These properties can be used to override or specify specific configuration values while running a job to determine its behaviour.

Pig Properties

We can specify Pig properties by any of the following ways.

- We can specify a list of properties to `pig.properties` file and adding the directory which contains the `pig.properties` file to classpath.
- We can specify a Pig property from command line interface simply by specifying `-D` switch followed by the name of the property in the `PIG_OPTS` environment variable. For example,

```
$ export PIG_OPTS=-Dpig.exec.reducers.max=300
```

- We can also specify a `pig.properties` file from command line interface by specifying `-P` switch followed by the name of the property. For example,

```
$ pig -P conf/pig.properties
```

- Finally, we can specify a Pig property from the Grunt command line interface or from Pig script using the `set` command. For example,

```
grunt> set pig.disable.counter true
```

The precedence order followed when a similar property is specified multiple times using any of the mechanisms above is as follows:

1. The `set` command has the top most precedence. The properties specified using any of the other mechanisms will be overridden, if the same

property is specified using the `set` command from the command line interface.

2. The property file specified using `-P` switch from the command line interface has next precedence.
3. The third precedence is for the properties specified using `-D` switch from the command line interface.
4. Finally, the directory in which the `pig.properties` file set in classpath has precedence.

Hadoop Properties

We can also specify Hadoop properties in the exact similar ways as seen above in the Pig properties.

- We can specify the Hadoop configuration files containing Hadoop properties as shown below.

```
$ include hadoop-site.xml
```

- We can specify a Pig property from command line interface simply by specifying `-D` switch followed by the name of the property in the `PIG_OPTS` environment variable. For example,

```
$ export PIG_OPTS=-Dio.sort.mb=320
```

- We can also specify a `property_file` from command line interface by specifying `-P` switch followed by the name of the property. For example,

```
$ pig -P conf/property_file
```

- Finally, we can specify a Hadoop property from the command line interface using the `set` command. For example,

```
grunt> set hadoop.security.authorization true
```

The precedence is similar to that of the Pig properties. The `set` command has the higher precedence and so on. All these Pig and Hadoop properties are available to UDF's too via the `UDFContext` object. You can get access to these properties by calling the `getJobConf` method.

Finally, you may list all the properties you have set to use by Pig using the following command:

```
$ -h properties
```

Strict Output Location Check

When we use Pig to process data, we are processing huge amount of data. It requires good amount of resources as well as processing time. Therefore, we cannot risk to lose any of the output data which we obtain after so much efforts.

It is possible to have multiple STORE statements within a Pig script. Hence, if you have mistakenly set the same output location for multiple STORE statements, you will be risking all the precious output data. To overcome this situation, we can use the property *pig.location.check.strict* and set it to *true*. Once the property is set to *true*, Pig will enforce strict check for output locations and lets you know when there are similar locations set for multiple STORE statements.

Blacklist/Whitelist Pig Operators and Commands

Pig provides a feature for administrators to blacklist/whitelist Pig operators and commands. This feature is very much helpful in the production environment to make Pig safe. For example, we have commands such as *sh* and *fs* which can lead to harmful results when triggered by inexperienced users. Therefore, blacklisting or whitelisting them will facilitate administrators to have more control over user scripts.

We can blacklist or whitelist Pig operators and commands using the following Pig properties.

- ***pig.blacklist***

This property is used to blacklist a set of Pig operators or commands. All the blacklisted commands or operators have to be listed by using commas as delimiters. For example, *pig.blacklist=kill, rm, filter, limit*. Setting this property will disallow users from using these commands and operators.

- ***pig.whitelist***

This property is used to whitelist a set of Pig operators or commands. All the whitelisted commands or operators have to be listed by using commas as delimiters. For example, *pig.whitelist=group, load, store, cross*. Setting this property will disallow users from using all the Pig commands and operators except specified in the whitelist.

Please make sure that there are no conflicts between the blacklist and whitelist properties, i.e., both these lists should be distinct and no command or operator

should appear in both the lists. If the lists are not distinct, Pig will throw out an error.

Statistics

By now, you might have ran a lot of Pig Latin scripts if you have been following the lab exercises till this point. You might have noticed a bunch of stats are displayed after the job has been completed. These are the stats which show you the insights and provides you with the details which help you understand what all has happened while the job was being executed. These stats help you tweak your configuration settings further to provide you a better throughput. Therefore, understanding what these stats mean is very crucial to know if your settings are correctly configured and are using all the resources as it should.

To demonstrate this better, we shall look at these stats by running a Pig Latin script during the Lab Exercises. Once we understand these stats, we shall further check out tuning and optimizing stuff in the next chapter.

Testing Pig Scripts

Testing is an essential phase apart from development for any programming language and Pig Latin is no exception. Testing your Pig scripts helps you make sure that there are no bugs in your code and is compatible with different versions of Pig and Hadoop or when there is a change made in the scripts. Testing has always been employed by the development teams for a couple of decades now to ensure the quality of products.

PigUnit

PigUnit provides you with a JUnit testing framework which helps you write unit tests to test your Pig Latin scripts. Unit tests are nothing but chunks of code that execute parts of software under controlled environments and verify the outcomes with expected results. To start testing our scripts, all we need is a Pig Latin script, `pigunit.jar`, input data and a JUnit Java class.

PigUnit can be run on local mode and is recommended for testing. However, you can also run the tests in Pig's mapreduce, Tez or Tez local modes. To enable these modes, we have to set the property `pigunit.executype` to `mr`, `tez` or `tez_local`. For example to run Pig unit in MapReduce mode, we need to set the following property:

```
-Dpigunit.executype=mr
```

We shall be looking at an example in our lab exercises which uses PigUnit to test Pig Latin scripts.

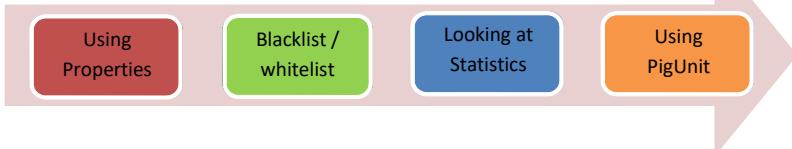
AIM

The aim of the following lab exercise is to have your hands on administering Pig and testing Pig Latin code.

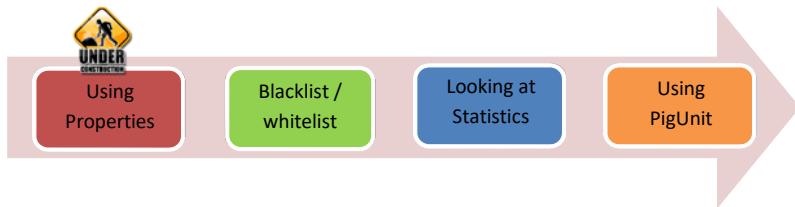
Following steps are required:

- Task 1: Using Pig Properties
- Task 2: Blacklist/Whitelist Pig commands/operators
- Task 3: Looking at statistics
- Task 4: Using PigUnit to test Pig Latin scripts

Lab Exercise 11: HANDS ON ADMINISTERING AND TESTING PIG



-
1. **Using Pig Properties**
 2. **Blacklist/Whitelist Pig commands/operators**
 3. **Looking at Statistics**
 4. **Using PigUnit to test Pig Latin scripts**



Task 1: Using Pig Properties

Using pig.properties file

Step 1: As learned in the theory part, we can use all the Pig properties by placing them in a pig.properties file and add it to the classpath. The location of the pig.properties file is as shown below.

PIG_PREFIX/conf/pig.properties

Step 2: Navigate to the path above and open the pig.properties file to check all the Pig properties as shown in the screenshot below.

```
pig.properties x
#####
#
# == Logging properties
#
# Location of pig log file. If blank, a file with a timestamped slug
# ('pig_1399336559369.log') will be generated in the current working directory.
#
# pig.logfile=
# pig.logfile=/tmp/pig-err.log

# Log4j configuration file. Set at runtime with the -4 parameter. The source
# distribution has a ./conf/log4j.properties.template file you can rename and
# customize.
#
# log4jconf=./conf/log4j.properties

# Verbose Output.
# * false (default): print only INFO and above to screen
# * true: Print all log messages to screen
#
# verbose=false

# Omit timestamps on log messages. (default: false)
#
# brief=false

# Logging level. debug=OFF|ERROR|WARN|INFO|DEBUG (default: INFO)
#
# debug=INFO
```



Using Properties

Blacklist / whitelist

Looking at Statistics

Using PigUnit

Step 3: To be able to use these properties we have to uncomment the property by removing '#' and specify the value accordingly for that property. For example, let us set the location for Pig to store its log file using the property *pig.logfile*. To do so, remove the '#' and set the location as */usr/local/pig-0.14.0/pig_logs* as shown below.

```
pig.logfile = /usr/local/pig-0.14.0/pig_logs
```

You may provide any other directory to store logs as you seem fit.

```
pi properties x

# Pig configuration file. All values can be overwritten by command line
# arguments; for a description of the properties, run
#
#      pig -h properties
#
#####
#
# == Logging properties
#
#
# Location of pig log file. If blank, a file with a timestamped slug
# ('pig_1399336559369.log') will be generated in the current working directory.
#
# pig.logfile=/usr/local/pig-0.14.0/pig_logs
# pig.logfile=/tmp/pig-err.log

# Log4j configuration file. Set at runtime with the -4 parameter. The source
# distribution has a ./conf/log4j.properties.template file you can rename and
# customize.
#
# log4jconf=./conf/log4j.properties
```

Make sure you save the file after making changes. You might have to create *pig_logs* directory and set appropriate permissions to the directory.

Step 4: Finally, we need to add this *conf* directory to classpath so that Pig can use the properties we have set. To do this, open *.bashrc* file and add the following line.

```
export PIG_CONF_DIR=$PIG_PREFIX/conf
```



Using Properties

Blacklist / whitelist

Looking at Statistics

Using PigUnit

```
export HADOOP_OPTS="-Djava.net.preferIPv4Stack=true"
export HADOOP_OPTS="-Djava.library.path=$HADOOP_PREFIX/lib"
export PIG_CLASSPATH=$PIG_CLASSPATH:/usr/local/jython-install/jython-standalone-2.7.0.jar
export PIG_JAVA_CLASSPATH=$PIG_JAVA_CLASSPATH:/usr/local/pig-0.14.0/pig-0.14.0-core-h2.jar:$HADOOP_PREFIX/share/hadoop/common/*:$HADOOP_PREFIX/share/hadoop/hdfs/*:$HADOOP_PREFIX/share/hadoop/mapred/common/lib/*:$PIG_PREFIX/lib/*:$HADOOP_PREFIX/share/hadoop/yarn/*
export PIG_CONF_DIR=$PIG_PREFIX/conf
```

You can now have all the logs stored in the directory specified in the property. You can now change other properties in the similar way as per requirement. However, you need not set the class path every time you change a property.

Step 5: You can also specify a pig.properties file from command line interface by specifying -P switch followed by the name of the property. For example,

```
$ pig -P $PIG_PREFIX/conf/pig.properties
```

```
uzair@ubuntu:~$ pig -P $PIG_PREFIX/conf/pig.properties
15/07/21 00:49:59 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/07/21 00:50:00 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/07/21 00:50:00 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/07/21 00:50:00 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
2015-07-21 00:50:00,473 [main] INFO org.apache.pig.Main - Apache Pig version 0.
14.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-07-21 00:50:00,506 [main] INFO org.apache.pig.Main - Logging error messages to: /usr/local/pig-0.14.0/pig_logs/pig_1437465000462.log
2015-07-21 00:50:00,580 [main] INFO org.apache.pig.impl.util.Utils - Default bootstrap file /home/uzair/.pigbootup not found
2015-07-21 00:50:01,549 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2015-07-21 00:50:01,551 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
```

Using Pig properties from CLI

Step 1: Instead of specifying many properties at once using pig.properties file, you can also specify single property from command line interface. This can be achieved by specifying -D switch followed by the name of the property in the PIG_OPTS environment variable. For example,

```
$ export PIG_OPTS=-Dpig.logfile =
/home/uzair/piglogs
```



Using Properties

Blacklist / whitelist

Looking at Statistics

Using PigUnit

```
uzair@ubuntu:~$ export PIG_OPTS=-Dpig.logfile=/home/uzair/piglogs
uzair@ubuntu:~$ pig
15/07/21 01:51:37 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/07/21 01:51:37 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/07/21 01:51:37 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
2015-07-21 01:51:38,033 [main] INFO org.apache.pig.Main - Apache Pig version 0.
14.0 (r1640057) compiled Nov 16 2014, 18:02:05
2015-07-21 01:51:38,038 [main] INFO org.apache.pig.Main - Logging error messages to: /home/uzair/piglogs/pig_1437468698027.log
2015-07-21 01:51:38,156 [main] INFO org.apache.pig.impl.util.Utils - Default bootstrap file /home/uzair/.pigbootup not found
2015-07-21 01:51:40,056 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2015-07-21 01:51:40,057 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2015-07-21 01:51:40,063 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:9000
2015-07-21 01:51:42,306 [main] WARN org.apache.hadoop.util.NativeCodeLoader - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2015-07-21 01:51:44,472 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt>
```

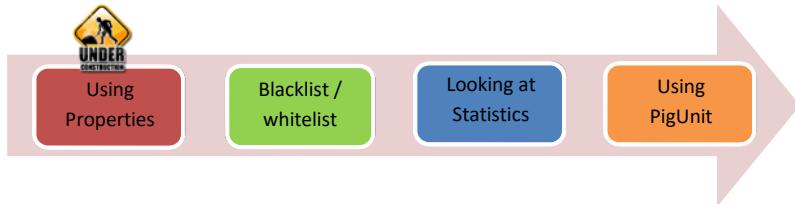
As you can see from the screenshot above, the `-D` pig property has taken precedence over the property set in `pig.properties` file.

Using `set` command

Step 1: We can also specify Pig properties using the `set` command directly from the command line interface. We shall use the `set` command to write all the logs in a file `newpiglogs.log`

```
grunt> set pig.logfile /usr/local/pig-0.14.0/pig_logs/newpiglogs.log
```

```
grunt> set pig.logfile /usr/local/pig-0.14.0/pig_logs/newpiglogs.log
grunt>
```



This will start logging all the logs to the log file mentioned above. Similarly, you can use many other properties by specifying from the `set` command. The `set` command takes the highest precedence. Similar properties specified by other mechanisms will be void when the property is specified using the `set` command. We have already covered the `set` command in chapter 3 along with labs.

Task 1 is complete!

Task 2: Blacklist/Whitelist Pig commands/operators

We have learned from theory that we can blacklist/whitelist Pig commands/operators using the properties `pig.blacklist` and `pig.whitelist` respectively. Let us check the procedure to blacklist/whitelist Pig operators/commands.

Step 1: Navigate to `$PIG_PREFIX/conf` directory and open `pig.properties` file. In the `pig.properties` file scroll down to the part with the title *Security Features* as shown below.

```
pig.properties x
#####
#
# Security Features
#
# Comma-delimited list of commands/operators that are disallowed. This security
# feature can be used by administrators to block use of certain commands by
# users.
#
# * <blank> (default): all commands and operators are allowed.
# * fs, set (for example): block all filesystem commands and config changes from pig scripts.
#
# pig.blacklist=
# pig.blacklist=fs, set

# Comma-delimited list of the only commands/operators that are allowed. This
# security feature can be used by administrators to block use of certain
# commands by users.
#
# * <blank> (default): all commands and operators not on the pig.blacklist are allowed.
# * load, store, filter, group: only LOAD, STORE, FILTER, GROUP
#   from pig scripts. All other commands and operators will fail.
#
# pig.whitelist=
# pig.whitelist=load, store, filter, group
#####
#
```



Step 2: You may now uncomment the properties `pig.blacklist` and `pig.whitelist` and specify the commands or operators you wish to blacklist or whitelist according to your requirement.

For the sake of demonstration, let us simply blacklist the `set` and `fs` commands. Simply remove the `#` before the line `pig.blacklist=fs,set` as shown in the screenshot below.

```
pig.properties x
#####
#
# Security Features
#
# Comma-delimited list of commands/operators that are disallowed. This security
# feature can be used by administrators to block use of certain commands by
# users.
#
# * <blank> (default): all commands and operators are allowed.
# * fs,set (for example): block all filesystem commands and config changes from pig scripts.
#
# pig.blacklist=


pig.blacklist=fs,set


# Comma-delimited list of the only commands/operators that are allowed. This
# security feature can be used by administrators to block use of certain
# commands by users.
#
# * <blank> (default): all commands and operators not on the pig.blacklist are allowed.
# * load,store,filter,group: only LOAD, STORE, FILTER, GROUP
#   from pig scripts. All other commands and operators will fail.
#
# pig.whitelist=
# pig.whitelist=load,store,filter,group
#####
#
```

Step 3: Now start Pig in Grunt mode and try to use the `fs` and `set` commands. For example,

```
grunt> fs -ls
```

```
grunt> fs -ls
2015-07-21 04:46:41,356 [main] ERROR org.apache.pig.tools.grunt.Grunt - ERROR 18
56: FS command is not permitted.
Details at logfile: /home/uzair/piglogs/pig_1437479165536.log
grunt>
```



As you can see from the screenshot above, you will be thrown an error saying that you are not permitted to use that command.

Step 4: Similarly, you can whitelist Pig commands or operators simply by specifying the commands or operators in the `pig.whitelist` property which you desire to use and the rest of the commands or operators (Which are not whitelisted) will be not be available for usage.

Please note that you will have to restart Pig for the changes in properties to take effect.

Once you are done with this lab exercise, make sure you remove the commands or operators from the blacklist and whitelist properties as the commands or operators may not be available for other users. Also, make sure you do not specify similar commands or operators in both blacklist and whitelist properties as this may result an error.

Task 2 is complete!

Task 3: Looking at Statistics

To understand the bunch of stats that are displayed at the end of the job, we have to first run a Pig script and wait for it to complete and show the stats.

Step 1: Kindly run the Pig script from Task 4: Running a Pig Script from Lab Exercise 3. Please come back here as soon as you run the Pig script.

Make sure you delete the output folder, if exists, from Hadoop Distributed file system before you run the script or the job fails.

Step 2: Once the job is complete, you should see a success message similar to the one as shown in the screenshot below. We then have all the stats followed by the success message.



```

HadoopVersion  PigVersion      UserId  StartedAt      FinishedAt      Features
2.6.0  0.14.0  uzair  2015-07-21 05:12:28  2015-07-21 05:22:09  GROUP_BY,ORDER_BY

Success!

Job Stats (time in seconds):
Jobid  Maps  Reduces  MaxMapTime  MinMapTime  AvgMapTime  MedianMapTime  MaxRe
ducetime  Alias  Feature  Outputs
job_1437456705991_0001  1  1  64  64  64  55  55
GROUP_BY,COMBINER
job_1437456705991_0002  1  1  43  43  43  37  37
job_1437456705991_0003  1  1  41  41  41  30  30
er/uzair/wc_out,

Input(s):
Successfully read 2909 records (150467 bytes) from: "hdfs:/user/uzair/books/pg10773.txt"

Output(s):
Successfully stored 4083 records (41293 bytes) in: "hdfs:/user/uzair/wc_out"

Counters:
Total records written : 4083
Total bytes written : 41293
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

```

We shall look at the stats part by part as it is harder accommodate all the stats in a single screenshot.

As in the screenshot below, the first couple of lines provide you with the basic details such as Hadoop version, Pig version, The ID of the user from which the job was initiated, the time at which the job has started and completed and the features used during the job.

```

HadoopVersion  PigVersion      UserId  StartedAt      FinishedAt      Features
2.6.0  0.14.0  uzair  2015-07-21 05:12:28  2015-07-21 05:22:09  GROUP_BY,ORDER_BY

Success!

```

Next few lines of stats are the *Job Stats* which provide the statistics of each and every MapReduce job. The stats include the number of map and reduce tasks ran for each job, time taken during each phase of the job, the features used in each job and the outputs written by the jobs, if any.



```
Job Stats (time in seconds):
Jobid    Maps    Reduces  MaxMapTime    MinMapTime    AvgMapTime    MedianMapTime
ductime   Alias   Feature Outputs
job_1437456705991_0001  1      1      64      64      64      64      55      55
GROUP_BY, COMBINER
job_1437456705991_0002  1      1      43      43      43      43      37      37
job_1437456705991_0003  1      1      41      41      41      41      30      30
er/uzair/wc_out,
```

MaxReduceTime	MinReduceTime	AvgReduceTime	MedianRe
55	55	file,words,wordsAggregated,wordsGroupedG	
37	37	wordsSorted	SAMPLER
30	30	wordsSorted	ORDER_BY
			hdfs:/us

Please see that the screenshot above has been adjusted by half to fit the page.

Next, we have the sections *Input(s)* and *Output(s)* which provide the information about the number of records read from input and stored to output. Since we have had only one input and output for this job, we are provided with the information about the locations where the data is read and stored respectively.

```
Input(s):
Successfully read 2909 records (150467 bytes) from: "hdfs:/user/uzair/books/pg10773.txt"

Output(s):
Successfully stored 4083 records (41293 bytes) in: "hdfs:/user/uzair/wc_out"
```

We then have the *Counters* section which provides us with the stats of total number of records and bytes written along with the spill records which have been spilled to local disk to avoid running out of memory. Please note that if you have run this Pig script on local mode, the *Counters* section will not be available. This is because counters are not reported by Hadoop in local mode.



```
Counters:
Total records written : 4083
Total bytes written : 41293
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0
```

Finally, we have the *Job DAG* section which lets you know about the flow of data between MapReduce jobs. For this job, the data flow was linear.

```
Job DAG:
job_1437456705991_0001 -> job_1437456705991_0002,
job_1437456705991_0002 -> job_1437456705991_0003,
job_1437456705991_0003
```

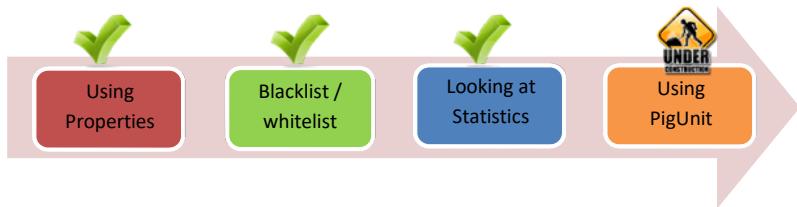
Task 3 is complete!

Task 4: Using PigUnit to test Pig Latin Scripts

Step 1: We require the following jar files to proceed with PigUnit Testing.

- junit.jar
- hamcrest-core.jar
- pig.jar
- pigunit.jar

All the above files can be downloaded from the internet. However, for this demonstration we build pig.jar and pigunit.jar using Apache Ant. Meanwhile, please download junit-4.12.jar and hamcrest-core-1.3.jar from the URLs below.



junit.jar - <http://bit.ly/My9IXz>

Navigate to the above link from your web browser and click on *jar* as shown in the screenshot below.

The Central Repository

SEARCH | ADVANCED SEARCH | BROWSE | QUICK STAT

New: About Central Advanced Search | API Guide | Help

Search Results

Groupid	Artifactid	Version	Updated	Download
junit	junit	4.12	04-Dec-2014	pom jar javadoc sources jar
junit	junit	4.12-beta-3	09-Nov-2014	pom jar javadoc sources jar
junit	junit	4.12-beta-2	25-Sep-2014	pom jar javadoc sources jar
junit	junit	4.12-beta-1	28-Jul-2014	pom jar javadoc sources jar
junit	junit	4.11	15-Nov-2012	pom jar javadoc sources jar
junit	junit	4.11-beta-1	16-Oct-2012	pom jar javadoc sources jar

< 1 2 > displaying 1 to 20 of 2

hamcrest-core.jar - <http://bit.ly/1gb125b>

Navigate to the above link from your web browser and click on *jar* as shown in the screenshot below.

The Central Repository

SEARCH | ADVANCED SEARCH | BROWSE | QUICK STATS

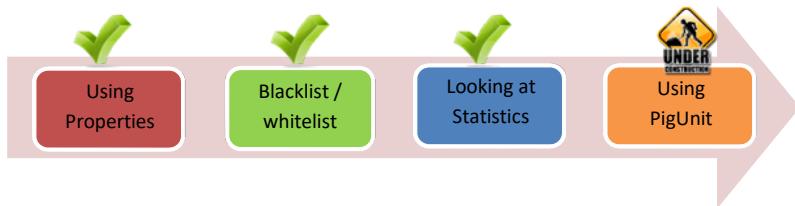
New: About Central Advanced Search | API Guide | Help

Search Results

Groupid	Artifactid	Version	Updated	Download
org.hamcrest	hamcrest-core	1.3	10-Jul-2012	pom jar javadoc sources jar
org.hamcrest	hamcrest-core	1.3.BC2	22-Dec-2010	pom jar javadoc sources jar
org.hamcrest	hamcrest-core	1.2.1	22-Dec-2010	pom jar javadoc sources jar
org.hamcrest	hamcrest-core	1.2	22-Sep-2010	pom jar javadoc sources jar
org.hamcrest	hamcrest-core	1.1	19-Jul-2007	pom jar sources jar

< 1 > displaying 1 to 5 of 5

Step 2: In order to generate *pigunit.jar* and *pig.jar*, we have to download the following packages.



- Apache Pig Source
- Apache Ant

Please navigate to following URL from your web browser to proceed with downloading Apache Pig Source package.

<http://www.apache.org/dyn/closer.cgi/pig>

Now, click on any of the mirror links as shown in the screenshot below.



Other mirror sites are suggested below. Please use the backup mirrors only to download PGP and MD5 signatures to working.

HTTP

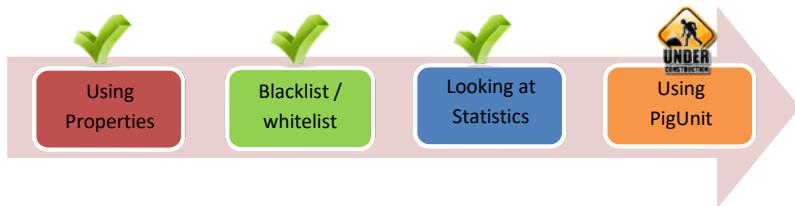
<http://ftp.wayne.edu/apache/pig>

<http://apache.cs.utah.edu/pig>

<http://supergsego.com/apache/pig>

As we are working with pig-0.14.0, click on that version as shown below.

Name	Last modified	Size	Description
Parent Directory		-	
latest/	05-Jun-2015 13:31	-	
pig-0.13.0/	17-Feb-2015 17:49	-	
pig-0.14.0/	17-Feb-2015 17:49	-	
pig-0.15.0/	05-Jun-2015 13:31	-	



Finally click on the source tar file and download the file as shown in the screenshot below.

Index of /apache/pig/pig-0.14.0

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
Parent Directory		-	
pig-0.14.0-src.tar.gz	20-Nov-2014 03:31	14M	
pig-0.14.0.tar.gz	20-Nov-2014 03:31	114M	

Apache/2.2.15 (Red Hat) Server at ftp.wayne.edu Port 80

Once the package has been downloaded, extract the contents of the package.

Step 3: Let us proceed by downloading Apache Ant to build pigunit.jar from source. To download Apache Ant, navigate to the following URL from your web browser.

<http://ant.apache.org/bindownload.cgi>

Once the page is opened, scroll down a bit and click on the tar.gz link as shown in the screenshot to download Apache Ant.

Current Release of Ant

Currently, Apache Ant 1.9.6 is the best available version, see the [release notes](#).

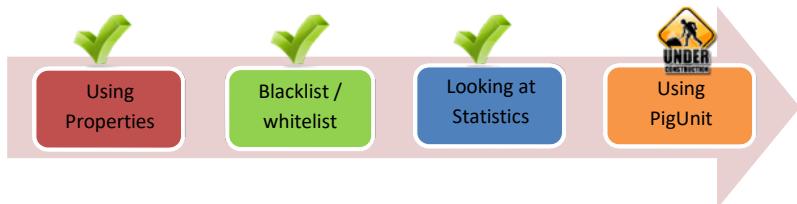
Note

Ant 1.9.6 was released on 02-Jul-2015 and may not be available on all mirrors for a few days.

Tar files may require gnu tar to extract

Tar files in the distribution contain long file names, and may require gnu tar to do the extraction.

- .zip archive: [apache-ant-1.9.6-bin.zip](#) [PGP] [SHA1] [SHA512] [MD5]
- .tar.gz archive: [apache-ant-1.9.6-bin.tar.gz](#) [PGP] [SHA1] [SHA512] [MD5]
- .tar.bz2 archive: [apache-ant-1.9.6-bin.tar.bz2](#) [PGP] [SHA1] [SHA512] [MD5]



Step 4: After Ant has downloaded, extract the contents and add the location to Ant in .bashrc.

```

export JAVA_HOME=/usr/local/jdk1.8.0_31
export PATH=$PATH:$JAVA_HOME/bin
export ANT_HOME=/usr/local/apache-ant-1.9.6
export PATH=$PATH:$ANT_HOME/bin
export HADOOP_PREFIX=/usr/local/hadoop-2.6.0
export PATH=$PATH:$HADOOP_PREFIX/bin
export PATH=$PATH:$HADOOP_PREFIX/sbin
export HADOOP_MAPRED_HOME=$HADOOP_PREFIX
export HADOOP_COMMON_HOME=$HADOOP_PREFIX
export HADOOP_HDFS_HOME=$HADOOP_PREFIX
export YARN_HOME=$HADOOP_PREFIX
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_PREFIX/lib/native
export HADOOP_OPTS="-Djava.net.preferIPv4Stack=true"
export HADOOP_OPTS="-Djava.library.path=$HADOOP_PREFIX/lib"

```

You may verify if the installation was successful by checking the Apache Ant version using the following command.

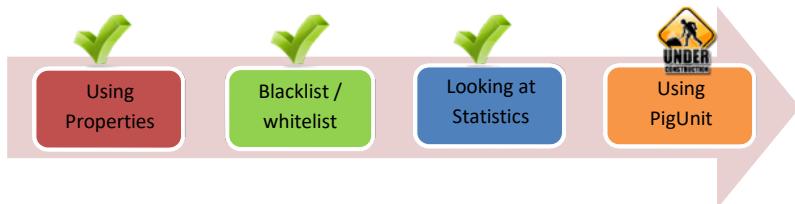
```
$ ant -version
```

Step 5: From the terminal, navigate to the Apache Pig Source directory and run the following command to build pigunit.jar and pig.jar files. Assuming, you have extracted the Apache Pig source package in *Downloads* directory, run the following commands.

```

$ cd Downloads/pig-0.14.0-src
$ ant pigunit-jar

```



```
uzair@ubuntu:~$ cd Downloads/pig-0.14.0-src
uzair@ubuntu:~/Downloads/pig-0.14.0-src$ ant pigunit-jar
Buildfile: /home/uzair/Downloads/pig-0.14.0-src/build.xml

ivy-download:
    [get] Getting: http://repo2.maven.org/maven2/org/apache/ivy/ivy/2.2.0/ivy-
2.2.0.jar
    [get] To: /home/uzair/Downloads/pig-0.14.0-src/ivy/ivy-2.2.0.jar
    [get] Not modified - so not downloaded

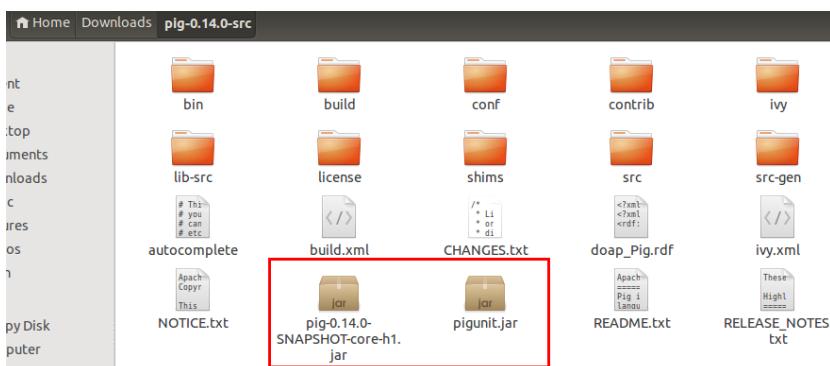
ivy-init-dirs:

ivy-probe-antlib:

ivy-init-antlib:
```

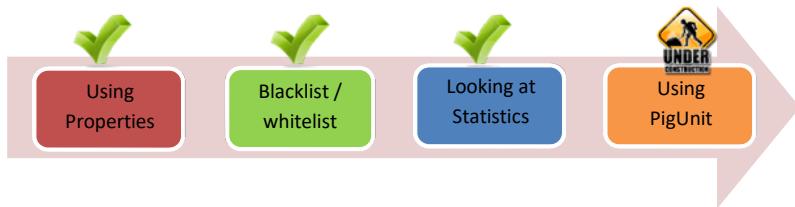
Step 6: After the build is complete, you should be having the following jars as shown in the screenshot below.

pigunit.jar
pig-0.14.0-SNAPSHOT-core-h1.jar



Step 7: With this we have all the necessary jars required for testing. Open *Eclipse IDE* and create a new package and name it *test*. Now, create a new class and name it *WordCountPigTest*.

We shall be testing out WordCount script using PigUnit.



Step 8: Right click the package *test*, hover over *Build Path* and click on *Configure Build path..* In the pop up click on *Libraries* and then click on *Add External JARs..* Select all the four jars i.e., *junit-4.12.jar*, *hamcrest-core-1.3.jar*, *pigunit.jar* and *pig-0.14.0-SNAPSHOT-core-h1.jar*.

Step 9: Let us now check out the code. The WordCount Pig Script is as shown below.

```
A = LOAD 'input' USING PigStorage() AS
(a:chararray);

B = FOREACH A GENERATE FLATTEN(TOKENIZE((a))) as
word:chararray;

C = GROUP B BY word;

D = FOREACH C GENERATE COUNT(B), group;

STORE D INTO 'output' using PigStorage();
```

The corresponding PigUnit test is as shown below.

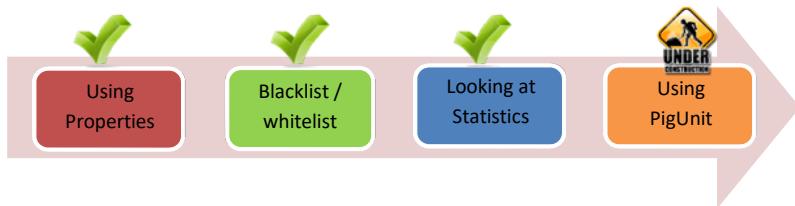
```
package test;

import org.apache.pig.pigunit.PigTest;
import org.apache.pig.tools.parameters.ParseException;
import org.junit.Test;

import java.io.IOException;

public class WordCountPigTest {

    @Test
    public void testWordCount() throws IOException,
ParseException {
        PigTest test = new PigTest("wordcount.pig", new
String[]{});
```



```

String[] input = {
    "hello hello hello",
    "hello hello hello"
};

String[] output = {
    "(6,hello)"
};

test.assertOutput("A", input, "D", output);
}
}

```

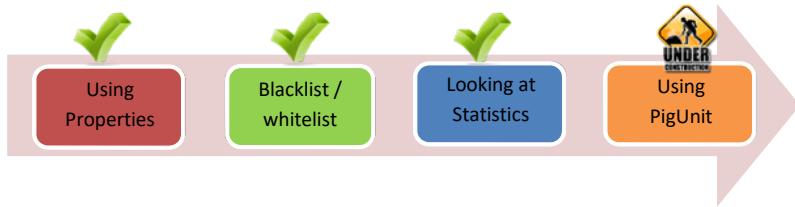
The first few lines are the imports required. In the next couple of lines, we are providing the Pig script *wordcount.pig* to initiate the test. Next couple of lines is the input with *hello* repeated three times in two lines and the expected output. Finally we assert that the alias *A* in the Pig script should be replaced by the *input* file and the alias *D* in the Pig script should be the *output* file. This is where PigUnit executes the script and performs the replacements. The LOAD and STORE operators are ignored and considers the above replacements.

You may download the Pig script and PigUnit code from the URLs below.

wordcount.pig - <https://db.tt/5DMJ3uZT>
WordCountPigTest.java - <https://db.tt/WPOrTVy3>

Step 10: Finally, run the script as JUnit test. To do this, right click on the class, hover over *Run As* and click on *JUnit Test*. Make sure you do not have any errors in your code. You will have errors, if you have not added all the jars from the previous steps to build path.

Once you run the program, you will be able to see the progress in the console. PigUnit chooses local mode to run PigUnit tests by default. However, nothing is stopping you to run PigUnit tests in MapReduce, Tez or Tez Local mode. All you need to do is specify the property as seen in theory section.



Once the testing is done, you should see the *Success* message as shown in the screenshot below. Also you will have *green* indication as shown in the screenshot below.

Package Explorer JUNIT

Finished after 10.274 seconds

Runs: 1/1 Errors: 0 Failures: 0

test.WordCountPigTest [Runner: JUnit 4] (10.1)

```

1 package test;
2
3 import org.apache.pig.pigunit.PigTest;
4
5 public class WordCountPigTest {
6     @Test
7     public void testWordCount() throws IOException, ParseException {
8         PigTest test = new PigTest("/home/uzair/wordcount.pig", new String{
9             "hello hello hello",
10            "hello hello hello"
11        });
12        String[] output = {
13            "(6,hello)"
14        };
15        test.assertOutput("A", input, "D", output);
16    }
17 }
18
19
20
21
22

```

Failure Trace

Problems Javadoc Declaration Console

<terminated> src [JUnit] /usr/local/java/jre1.7.0_45/bin/java (24-Jul-2015 6:39:11 pm)

Output(s): Successfully stored records in: "file:/tmp/temp252595658/tmp-360506842"

Job DAG: job_local_0001

15/07/24 18:39:22 INFO mapReduceLayer.MapReduceLauncher: Success!

15/07/24 18:39:22 WARN data.SchemaTupleBackend: SchemaTupleBackend has already been initialized

15/07/24 18:39:22 INFO input.FileInputFormat: Total input paths to process : 1

15/07/24 18:39:22 INFO util.MapRedUtil: Total input paths to process : 1

Although, every minute step of this process is demonstrated here, but still if you come across a strange error, try to solve it by searching the solution to the problem over internet.

Task 4 is complete!

SUMMARY

Administration and Testing play very important role in Apache Pig. With Administration you have much more control over Pig installation while testing helps you make sure that all your scripts are error free and can be run with different versions of Pig and Hadoop.

In this chapter, we have seen how to use Pig properties, administer Pig by blacklisting or whitelisting Pig commands or operators to avoid any problems during production stage, understood the statistics displayed at the end of the job and finally tested a Pig script using PigUnit.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/test.html>
- <http://pig.apache.org/docs/r0.14.0/admin.html>

INDEX

Administering Pig	344
Testing Pig Scripts	347
AIM	348
Lab Exercise 11: Hands on Administration and Testing Pig.....	349
Task 1: Using Pig properties.....	350
Task 2: Blacklist/Whitelist Pig commands/operators .	354
Task 3: Looking at statistics.....	356
Task 4: Using PigUnit to test Pig script.....	359
SUMMARY	368
REFERENCES.....	369

CHAPTER 12: OPTIMIZING PIG

Theory

Apache Pig ships with default settings and not all the settings are appropriate for a particular environment. We have to set custom settings and make few tweaks based upon our requirement for optimal performance. All these optimizations made in Pig will have greater impact on the performance. In this chapter, we shall look at tweaks and tips to optimize Apache Pig in order to increase its performance.

The need for Optimization

First things first. Why do we have to think so much about optimizing Pig? The answer is simple. To increase overall performance and efficiency by utilizing all the available resources to core. Therefore to optimize, we need to first identify the features in Pig and MapReduce which degrade the overall performance in the system. The following features are considered as the most common features which degrade Pig's performance.

Number of MapReduce Jobs

Once the Pig Latin scripts are executed, the scripts are turned into a series of MapReduce jobs under the hood. There is some overhead attached with each MapReduce job to the query. As MapReduce is built for large batch jobs, the developers have not put much extra efforts to minimize the overheads such as job start time. Every MapReduce job takes anywhere between 5 to 10 seconds to start, assuming there are idle resources in the Hadoop cluster for the job to run. Therefore, in a busy cluster, with less to no idle resources to run the job, Pig has to wait for some time for the resources to be available again. Moreover, the intermediate data generated in between the MapReduce jobs is stored in HDFS. The data is written with replications over the network and read after each completed MapReduce job. Also, the MapReduce jobs are executed in series, i.e., a MapReduce job does not start before all the tasks from previous MapReduce job is finished. Therefore, slower running reduce tasks can increase the overall time taken for completion of a job.

Hence we can conclude from the above observations that the performance of Pig scripts can be significantly increased when multiple MapReduce jobs can be merged as a single MapReduce job.

Key Skew at Reducer

A MapReduce job is divided into mapper and reducer phases on many nodes of a Hadoop cluster. During the map phase, a map task is spawned for each input split of data where the size of input split is equal to HDFS block size. The block size ranges from 64 MB to 512 MB depending upon the cluster configuration. Therefore, the

input data is evenly distributed to the map tasks and the work load is even during the map phase. However, during the reduce phase, the work is distributed based upon the key chosen to transform data during operations such as GROUP, ORDER BY, JOIN etc. For this kind of transformations, if the data is uniformly distributed for the specified key, the work is uniformly distributed too and everything is just fine. But most of the times, this is not the scenario. The data is power law distributed. Therefore, the reducer which gets the key with highest skew in data will take some time to complete when compared to other reducers and ultimately will lead in longer durations to complete the job.

Data at Shuffle Phase

After the map phase is completed, the intermediate data generated by mappers is sent to reducers by performing a hash using hash partitioner (Hadoop's default partitioner). This process is known as shuffle. Most of the time is spent at this phase during the life cycle of a MapReduce job. Therefore, it is recommended to compress the data which has to be shuffled using the compression algorithms which do not use too much of resources such as LZO, Google Snappy etc. This reduces the amount of data which has to shuffle between the map and reduce phases.

List of Optimization Rules

Optimization rules are applied by Pig while generating the logical plan for a Pig script. There are a bunch of optimization rules supported by Pig and all are enabled by default. You may disable the optimization rules using the `set` command for the Pig property `pig.optimizer.rules.disabled` followed by the rules to be disabled separated by a comma. For example

```
set pig.optimizer.rules.disabled <comma-separated rules>
```

You may also use the option `all` to disable all the optimization rules at once. However, there are few optimization rules which have to be mandatorily used and cannot be disabled.

Alternatively, the rules can be disabled using the command line options `-t` and `-optimizer_off`. The option `all` to disable all the rules at once can be applied here too. For example

```
$ pig -optimizer_off [optimization rule | all]
```

Let us now look at the various optimization rules supported by Pig in detail. Most of these optimizations should look familiar if you have been working on optimizations for database queries.

ConstantCalculator

This optimization rule simplifies:

1. The expressions which evaluate to a constant. For example,

```
filtered = FILTER employees BY age < 40-5; is  
simplified as filtered = FILTER employees BY age <  
35;
```

2. The eval functions used in the Pig statements. For example,

```
data = FOREACH input GENERATE UPPER('log'); is  
simplified as data = FOREACH input GENERATE 'LOG';
```

SplitFilter

This optimization rule splits filter conditions. Two or more filter conditions in a single statement are split into two separate filter statements and are pushed up individually. For example,

```
emp_data = LOAD 'emp_data' USING PigStorage(',') AS  
(emp_id:int, emp_name: chararray, emp_age: int,  
emp_desgn: chararray);  
  
emp_billing = LOAD 'emp_billing' USING  
PigStorage(',') AS (emp_id: int, emp_rate: float);  
  
joined = JOIN emp_data BY emp_id, emp_billing BY  
emp_id;  
  
filtered = FILTER joined BY emp_rate > 30 AND  
emp_age < 25;
```

The filtered statement above is split into two filter statements below by the SplitFilter optimization rule so that they can be pushed up individually.

```
filtered_1 = FILTER joined BY emp_rate > 30;  
  
filtered_2 = FILTER joined BY emp_age < 25;
```

PushUpFilter

This optimization rule pushes up the FILTER statements upstream in the data flow so that the number of records flowing in the pipeline is reduced. To understand this

better, refer to the example seen in the *SplitFilter* rule. In the example, after the *SplitFilter* does its job of splitting the filter conditions in the *filtered* statement into *filtered_1* and *filtered_2*, both these relations are pushed up before the join statement and after the load statements by *PushUpFilter* so that the irrelevant data gets filtered out before the two datasets are joined. This helps increase efficiency.

MergeFilter

MergeFilter and *SplitFilter* go well together. It is just that *SplitFilter* rule is applied before *PushUpFilter* and *MergeFilter* is applied after *PushUpFilter*. *MergeFilter* simply merges multiple filter statements on same dataset into single filter statement. For example,

```
emp_data = LOAD 'emp_data' USING PigStorage(',') AS  
(emp_id:int, emp_name: chararray, emp_age: int,  
emp_desgn: chararray);  
  
filtered_1 = FILTER emp_data BY emp_id < 225;  
  
filtered_2 = FILTER emp_data BY emp_age > 30;
```

The two filter statements above will be merged to a single statement as shown below:

```
filtered = FILTER emp_data BY (emp_id < 225 AND  
emp_age > 30);
```

PushDownForEachFlatten

This optimization rule pushes down the FLATTEN operation in FOREACH statements downstream in the data flow pipeline. This is because the FLATTEN operation with FOREACH generates significantly more data when compared to the input data. In order to reduce the amount of data flow through the pipeline, this operation is applied. For example, in the Pig script below, the FOREACH statement will be pushed down after the JOIN statement.

```
A = LOAD 'input1' AS (a, b, c);  
B = LOAD 'input2' AS (x, y, z);  
C = FOREACH A GENERATE FLATTEN($0), B, C;  
D = JOIN C BY $1, B BY $1;
```

LimitOptimizer

This optimization rule is similar to that of *PushUpFilter* optimization rule. The idea behind this rule is to push the LIMIT operator upstream in the data flow so that the

number of records flowing in the pipeline is reduced. This optimization significantly decreases the number of records flow through the data pipeline.

Moreover, if there is ORDER BY query within the script followed by the LIMIT operator to achieve top-k results, the LIMIT is pushed into the ORDER BY.

ColumnMapKeyPrune

This optimization rule simply gets the loader to loads the columns which are required for processing in the pipeline and ignores the columns which are specified in the loader but have not used anywhere else in the script. However, if the loader is unable to perform this optimization, a FOREACH statement is inserted right after the loader. For example,

```
emp_data = LOAD 'emp_data' USING PigStorage(',') AS  
(emp_id:int, emp_name: chararray, emp_age: int,  
emp_desgn: chararray);  
  
ordered = ORDER emp_data BY emp_name;  
  
result = FOREACH ordered GENERATE emp_id, emp_name,  
emp_desgn;
```

In the Pig Latin script above, the column *emp_age* is not used anywhere else in the script except in the loader. Therefore, having the contents of that column flowing throughout the pipeline is just not required. So this column is pruned while loading to significantly increase performance and efficiency.

This optimization rule is also applied for map keys too. The unused map keys are pruned using this rule.

AddForEach

AddForEach optimization rule is similar to the *ColumnMapKeyPrune* rule. The objective of this rule is to prune the column as soon as it is not required by the script. For example,

```
emp_data = LOAD 'emp_data' AS (emp_id, emp_name,  
emp_age);  
  
ordered = ORDER emp_data BY emp_id;  
  
filtered = FILTER ordered BY emp_age > 30;
```

In the Pig Latin script above, the column `emp_name` is pruned at the loader by `ColumnMapKeyPrune`. However, `emp_id` is not used again after the ORDER BY statement and hence it can be pruned. This is done by adding a FOREACH statement before the FILTER operation as shown below.

```
emp_data = LOAD 'emp_data' AS (emp_id, emp_name,  
emp_age);  
  
ordered = ORDER emp_data BY emp_id;  
  
pruned = FOREACH ordered GENERATE emp_age;  
  
filtered = FILTER pruned BY emp_age > 30;
```

MergeForEach

This optimization rule merges two or more FOREACH statements into one FOREACH statement. By doing this, the same dataset is not iterated multiple times. However, this optimization rule can only be applied if the following preconditions are true.

- The FOREACH statements are consecutive.
- The first FOREACH statement does not contain FLATTEN operator within it.
- The second FOREACH is not nested. However, the first FOREACH can be nested.

GroupByConstParallelSetter

This optimization rule sets number of reducers to use to 1 for a GROUP ALL statement. This is set to 1 because, even though if the users set reducers more than 1 using PARALLEL, only one reducer will be used and the rest will produce empty results. In order to avoid this, `GroupByConstParallelSetter` automatically sets the number of reducers to 1 for GROUP ALL statement.

PartitionFilterOptimizer

This optimization rule pushes a filter condition upstream to the loader. With this optimization the loader which supports this optimization only loads the data which satisfies the filter condition and ignores the rest. For example,

```
emp_data = LOAD 'emp_data' AS (emp_id, emp_name,  
emp_age) USING HCatLoader();  
  
filtered = FILTER emp_data BY emp_age > 30 AND  
emp_name == 'John';
```

In the Pig Latin script above, the loader will only load the input data satisfying the filter condition. This helps avoid loading unnecessary data.

PredicatePushDownOptimizer

This optimization rule is similar to that of *PartitionFilterOptimizer*. The filter condition will be pushed upstream to the loader. But if the loader does not support, the filter condition will be evaluated by Pig separately. This means that the filter condition pushed to loader is just a hint for the loader. If the loader does not support the optimization, it will load all the records irrespective of the filter condition.

Best Practices to Enhance Performance

We have seen Optimization rules in the previous section which help optimize Pig Queries and enhance performance. However, the optimization rules are just not enough to increase the performance and efficiency of Pig scripts. Not all the best practices can be made into optimization rules as most of the practices might be specific to data and application. So following the best practices at the right time is important for significant increase in performance. Let us now look at few of the best practices to enhance performance.

Specifying Types Explicitly

It is always recommended to explicitly specify the data types for fields while loading the input data. This explicitly specification of types will speed up queries twice more efficiently. For instance, the fields containing numerical computations are considered as type double by Pig. However, the input data most of the times will be an integer or long. As the computation for type integer is faster than the type double, there will be a significant speed up in computation. So, always remember to specify types explicitly while loading input data to Pig.

Filtering Data Early and Frequently

We load lots of data to Pig. However, we do not require to process most of the data we load. So, we should always try to get rid of the data as soon as possible and make the amount of data in the pipeline as low as possible to downstream. Doing this will significantly decrease the amount of data being shuffled and sorted during the MapReduce jobs in turn increasing efficiency.

However, filtering out which reduces only little amounts of data and higher costs of application might just not help you increasing the performance. So, consider this case before applying this practice and ask yourself if this is really worth it.

Projecting Data Early and Frequently

In the previous section, we have seen optimization rules *ColumnMapKeyPrune* and *AddForEach* which project only the necessary fields and prune the rest. These

projections help you get rid of unnecessary fields and transfer absolutely necessary fields to downstream. It is always a best practice to project only the fields required and eliminate the rest using a FOREACH statement. Doing so has resulted in 50% drop in total time of the script being processed.

Reducing Operators in Pipeline

In the previous section, we have seen optimization rules *MergeFilter* and *MergeForEach* which merge multiple FILTER and FOREACH statements into single FILTER or FOREACH statements. The optimization rules do their job but there might be some cases where they may not. So, keep an eye for such optimizations which could help you reduce the number of operators in pipeline. Reducing the number of operators in the data pipeline increases the performance of a Pig script.

Using the LIMIT operator

The LIMIT operator can help you achieve the top N results of your query. Pushing this operator upstream can reduce the amount of data transfer to downstream as well as overall time consumption to process data.

Using the DISTINCT operator

The DISTINCT operator helps you find the distinct elements in a field. But, it is also possible to find the distinct elements using FOREACH and then GROUP BY operator to generate group key. However, using FOREACH and GROUP BY causes a lot of data flow in the pipeline and increases overall time taken. So, it is always advised to use DISTINCT operator to find distinct elements in a field.

An example of using DISTINCT and GROUP BY is as shown below. First let us use GROUP BY.

```
ratings = LOAD 'ratings' AS (user_id, movie_id, rating);  
  
movies = FOREACH ratings GENERATE movie_id;  
  
grouping = GROUP movies BY movie_id;  
  
key = FOREACH grouping GENERATE group AS key;  
  
DUMP key;
```

Now let us look how this can be achieved using DISTINCT operator.

```
ratings = LOAD 'ratings' AS (user_id, movie_id, rating);
```

```
movies = FOREACH ratings GENERATE movie_id;  
  
distinct_rec = DISTINCT movies;  
  
DUMP distinct_rec;
```

Implementing Algebraic Interface for UDFs

It is always recommended to write Pig UDFs that implement Algebraic Interface. The reason behind this is that the combiners are used when an UDF which implements Algebraic Interface is working on grouped data. The MapReduce jobs which use combiners run much faster than those which don't. This significantly increases overall performance of the job.

Using the Accumulator Interface

Implementing Algebraic Interface for Pig UDFs definitely increases performance. However, it is just not possible to implement Algebraic Interface for UDFs all the time. So, in such cases, it is recommended to implement Accumulator Interface for Pig UDFs, if the UDF can take input in chunks so as to reduce the amount of memory utilized by the Pig scripts.

If a Pig UDF is Algebraic and can also use Accumulator Interface, it has to implement both the Algebraic as well as Accumulator Interface. Pig chooses the best interface it feels provides the best performance. First priority is given to Algebraic Interface, second to Accumulator Interface and finally comes the default interface.

Remove Nulls in Data

Removing nulls in data before performing a JOIN or COGROUP operation can improve the performance of the Pig script significantly. The reason for this is, when a JOIN or COGROUP operation is performed on the input data containing nulls, all the null keys are sent to a single reducer task. Then an extra step is required to get rid of nulls by flattening the grouping of empty bag which results in an empty row. This flattening operation happens after the reducer task and we have used much more resources to get rid of nulls. So, in order to increase performance and efficiency, care should be taken to get rid of nulls before performing a JOIN or COGROUP operation.

An example of this case is as shown below. Consider a JOIN operation as shown below.

```
ratings = LOAD 'ratings' AS (user_id, movie_id,  
rating);  
  
users = LOAD 'users' AS (user_id, age, location);
```

```
joined = JOIN ratings BY user_id, users By user_id;
```

The above script is rewritten by Pig as follows:

```
ratings = LOAD 'ratings' AS (user_id, movie_id, rating);
```

```
users = LOAD 'users' AS (user_id, age, location);
```

```
cogrouped = COGROUP ratings BY user_id INNER, users BY user_id INNER;
```

```
flat = FOREACH cogrouped GENERATE FLATTEN(RATINGS), FLATTEN(users);
```

The null keys will be removed when we perform FLATTEN operation after cogrouping. But this removal happens after the reducer tasks have been executed. Therefore, it is advised to filter nulls before performing a join as shown below.

```
ratings = LOAD 'ratings' AS (user_id, movie_id, rating);
```

```
users = LOAD 'users' AS (user_id, age, location);
```

```
filter1 = FILTER ratings BY user_id IS NOT NULL;
```

```
filter1 = FILTER users BY user_id IS NOT NULL;
```

```
joined = JOIN ratings BY user_id, users By user_id;
```

This will make sure that the nulls are removed before joining. This practice can help your queries perform ten times faster.

Using Join Optimizations

There are two kinds of join optimizations to make use of. They are the Regular join optimizations and Specialized join optimizations. We have already looked at Regular Join Optimizations in Chapter 6 in the section [Advanced Joins](#).

Regular join optimization come into a picture when a dataset with large records is used to join as last input in a join operation. This is because the last input in a join is streamed instead of bringing it to memory thereby avoiding spilling the data. For

example, considering the input *users* having large records when compared to the input *ratings*, we join *users* last so as to make use of the regular join optimization as shown below.

```
joined = JOIN ratings BY user_id, users By user_id;
```

Combining Small Files

Small files are not ideal for processing with Hadoop. In order to achieve efficiency, it is recommended to combine multiple small files as one. This is because a separate map task is spawned for each input file (input split). So as to overcome this situation Pig can now process several small files in a single map using the following properties.

The *pig.splitCombination* property has to be set to *true* to combine small files while the size of split can be set using *pig.maxCombinedSplitSize*. When a value (size in bytes) is set using this property, small files are combined until this limit is reached as the input for each Map task.

Using Compression for Intermediate Results

A lot of intermediate data is generated and written between MapReduce jobs for Pig scripts. Compressing intermediate data can save so much HDFS space and in turn increase execution speed. This intermediate data can be compressed using LZO compression. The property *pig.tmpfilecompression* has to be set to *true* to enable compression for intermediate files generated during MapReduce jobs from Pig scripts. This is set to *false* by default. Once this is set to true, use the property *pig.tmpfilecompression.codec* to specify the compression codec to use. Pig currently supports *gz* and *lzo* compression codecs. *Gzip* codec is also supported and provides better compression but slowdowns execution. Hence, it is not recommended.

Direct Fetch from HDFS

Pig can directly read data from HDFS when a DUMP operator is used instead of launching MapReduce jobs in order to reduce latency. However, there is a catch. The result is only fetched when the operators FILTER, FOREACH, LIMIT, STREAM and UNION are used.

The fetch is disabled if there are other operators present other than the operators mentioned above, there is no LIMIT operator or in the presence of implicit splits. Also, direct fetch does not support the UDFs which interact with distributed cache. To know if your query is direct fetch supported, run the EXPLAIN operator and check for "No MR jobs. Fetch only." in the MapReduce part of the plan.

The Direct fetch option is enabled by default and can be turned off by setting the property *opt.fetch* to false or start Pig with the "*-N*" or "*-no_fetch*" option.

Using Automatic Local Mode

Pig can be configured to use local mode automatically to execute small jobs locally instead of using MapReduce mode. Local mode is preferred over MapReduce mode for small MapReduce jobs because the jobs on Hadoop cluster could be slow as it has overhead of job start up and job scheduling. Smaller MapReduce jobs run in local mode if the property *pig.auto.local.enabled* is set to true. This is set to false by default. You can also configure the maximum size of input up to which the jobs should run in local mode using the property *pig.auto.local.input.maxbytes*. The default value is set to 100MB. This only works when the maximum reducers required by the job is less than or equal to 1.

User Jar Cache

The UDF jars are copied to distributed cache by Pig to other task nodes. These jars are copied to a temporary location in HDFS. However, for scheduled jobs, these jars do not change often and it is discouraged to copy lots of jars in HDFS. So as to avoid copying this jars to HDFS, Pig has a solution to configure user level jar cache which is only read by users for security reasons. To enable this feature, set the property *pig.user.cache.enabled* to true (false by default) and the jars get copied to the jar cache location under a directory named with the hash (SHA) of the jar. This hash is used to identify the jar for subsequent uses of jar. If the jar with same name and hash is already available in cache, the jar is not copied. The property *pig.user.cache.location* is used to specify the path on HDFS to use as staging directory for user jar cache.

There are no issues with using the User Jar Cache. In case of no access to jar cache, Pig will employ the default behaviour.

Using Parallel Features

Pig allows users to set the number of reducer tasks for MapReduce jobs using two of the following parallel features. They are

- Number of reducers set by user
- Number of reducers set by Pig

User can set number of reducers at script level using the command *default_parallel* and specifying the number of tasks. For example, to use 15 reducers:

```
SET default_parallel 15;
```

Pig also provides the option of setting number of reducers within the Pig script at operator level using the PARALLEL clause. This will override the value set using the *default_parallel* at script level. The PARALLEL clause can be applied to any of the

operators which start a reduce phase: COGROUP, CROSS, DISTINCT, GROUP, JOIN (inner), JOIN (outer), and ORDER BY. For example,

```
joined = JOIN ratings BY user_id, users BY user_id  
PARALLEL 15;
```

If the user has not set reducers using the options above, Pig will set number of reducers based on the size of input data. The property `pig.exec.reducers.bytes.per.reducer` is used to set the number of input bytes per reducer. It is set to 1GB by default. We also have to set the property `pig.exec.reducers.max` for maximum number of reducers where 999 being the default.

The formula to determine the number of reducers is

```
Number of reducers = MIN (pig.exec.reducers.max,  
total input size (in bytes) / bytes per reducer)
```

Combiners in Pig

Combiners can be made use of to decrease the amount of data shuffled between map and reduce tasks and also reduces disk IO. The combiner in Pig is invoked depending upon the statements used in Pig script.

The following are the cases which invoke a combiner:

- A combiner is used when a Non-nested FOREACH statement is used in the script. However, if there is a DISTINCT operator in a nested FOREACH statement, the combiner is invoked.
- A combiner is used when all the projections in a FOREACH statement are either expressions on the grouped columns or expressions on Algebraic UDFs.

For example,

```
users = LOAD 'users' AS (user_id, age, location,  
no_ratings);  
  
grouped = GROUP users BY age;
```

```
result = FOREACH grouped GENERATE
SUM(users.no_ratings), COUNT(DISTINCT(user_id)),
group.age;
```

The following are the cases which do not invoke a combiner:

- If the script does contains any of the cases above.
- If there is a statement between GROUP and FOREACH statement with an exception of LIMIT operator. In some cases, even if there is no statement between GROUP and FOREACH statements, Pig might just rearrange the script using the *PushUpFilter* optimizer preventing use of a combiner.

Multi-Query Execution

Multi-Query Execution is used to process an entire Pig script or batch of statements at once. Multi-Query Execution is enabled by default. This can be turned off using “-M” or “-no_multiquery” options. Once turned off, execution will be triggered when it sees DUMP or STORE statement. For example, running a pig script without multi-query execution is as shown below:

```
$ pig -M pig_script.pig
Or
$ pig -no_multiquery pig_script.pig
```

Advanced Optimization using Apache Tez

We have been using MapReduce engine to execute our Pig scripts. However, Pig supports an alternative execution engine called Apache Tez which can boost performance with the features: optimized job flow, edge semantics and container reuse.

We have already seen how to start Pig in Task 5: Starting Pig in MapReduce Mode and Tez Mode.

DAG Generation in Tez

Each Pig script is translated into one or more Tez Directed Acyclic Graph (DAG). Each Tez DAG consists of a number of vertices and edges connecting vertices. For example, a simple join involves 1 DAG which consists of 3 vertices: load left input, load right input and join.

Container Reuse in Tez

Apache Tez overcomes the setbacks in MapReduce. The overhead to start a MapReduce job is very high. The performance and efficiency is reduced due to this

overhead and impacts overall time taken by the job as learned at the starting of this chapter. With Tez, the job start up time is very much reduced as it can reuse the same container and there is no need to start a new container by the application master every time and start a JVM for every task. This is turned on by default and there is no reason why you should turn it off. However, JVM reuse may hurt when there is a static variable as it can live across different jobs. So, in case you have a static variable in EvalFunc/LoadFunc/StoreFunc, make sure you implement a cleanup function and register with JVMReuseManager.

Parallel Features

Parallel features are similar to that of MapReduce in Tez when specified by the user. However, if the user has not specified the number of reducers, unlike MapReduce Tez handles parallelism differently. Tez submits a DAG as a unit and automatic parallelism is managed in two parts. They are

- Before submitting a DAG, Pig estimates parallelism of each vertex statically based on the input file size of the DAG and the complexity of the pipeline of each vertex.
- At runtime, Tez adjusts vertex parallelism dynamically based on the input data volume of the vertex. However, currently Tez can only decrease the parallelism dynamically and does not increase. So in the previous step, Pig overestimates the parallelism.

The properties which are set for parallelism in MapReduce are similar for controlling parallelism in Tez.

LAB EXERCISE 12

"There are no activities required for this lab"

SUMMARY

Apache Pig ships with default settings and not all the settings are appropriate for a particular environment. We have to set custom settings and make few tweaks based upon our requirement for optimal performance. All these optimizations made in Pig will have greater impact on the performance. In this chapter, we shall look at tweaks and tips to optimize Apache Pig in order to increase its performance.

In this chapter, we have seen the Optimization rules applied by Pig to increase performance and efficiency as well as best practices to juice more performance from our Pig cluster. We have also seen the cases when combiners are used and when they are not.

REFERENCES

- <http://pig.apache.org/docs/r0.14.0/perf.html>
- <https://www.cs.duke.edu/courses/fall11/cps216/Lectures/gates.pdf>
- <http://sites.computer.org/debull/A13mar/p34.pdf>

INDEX

The Need For Optimization.....	371
List of Optimization Rules	372
Best Practices to Enhance Performance	377
Combiners in Pig	383
Multi-Query Execution.....	384
Advanced Optimizations using Apache Tez	384
Lab Exercise 12	386
SUMMARY	387
REFERENCES.....	388

CHAPTER 13: THE HADOOP ECOSYSTEM & PIG

Theory

Hadoop, an open source top level Apache project has gained wide popularity due to its cutting edge technology and features. It is being adopted worldwide to process huge amounts of data stored in disk drives and extract information worth billions in a quick and cost effective manner. Hadoop is being adopted in almost every industry you can think of. Therefore, a lot of applications (e.g. Pig) are developed and are being developed to help working with Hadoop and make our lives easy. These applications which are developed for working with Hadoop are termed as Hadoop Ecosystem projects. This chapter deals with the important Hadoop Ecosystem projects and know how they differ from or can be used with Pig in brief.

Apache Pig is one of the most important Hadoop ecosystem application. As we have seen throughout the chapter, Apache Pig provides its users a high level scripting language called PigLatin to run queries and perform ETL operations on BigData. These queries are then converted to a series of MapReduce jobs under the hood. Now that we are well versed with Apache Pig, it is time to shift our focus on other Hadoop Ecosystem projects and learn about them in brief.

Apache Hive

Apache Hive is the data warehouse system built on top of Hadoop which helps its users to easily query, summarize and analyse large datasets stored in distributed systems. Apache Hive provides a SQL like language called HiveQL to query the data. The HiveQL queries are then translated into a series of MapReduce jobs under the hood similar to that of Apache Pig. HiveQL is similar to that of SQL containing the data definition language (DDL) statements to create tables and data manipulation language (DML) statements to extract and transform data. Furthermore, HiveQL has support to load unstructured data into the tables with the help of delimiters. HiveQL is also extensible as it supports the user defined functions (UDF) and user defined aggregate functions (UDAF) implemented in Java. HiveQL does not support updating and deleting of rows in the existing tables. Once the data is loaded in the table we can perform multiple queries on the data using a single HiveQL statement. If you are a SQL professional, Hive makes you feel right at home allowing you to run queries in SQL style. The purpose of Hive is similar to that of Pig but the process to analyse data is different.

Even though Hive provides SQL like query language, there are few differences. Let us now look at a comparison between Hive and RDBMS.

Comparing Hive and RDBMS

RDBMS	Hive
<ul style="list-style-type: none">• RDBMS support online transaction processing (OLTP) with real time reads and writes for data stored in databases and online analytical processing (OLAP) for real time reporting• RDBMS strictly requires schema on write. Schema should be present before data is loaded.• Loading is slower and queries are fast.• There are hundreds of built-in functions available.• Indexes are supported in RDBMS.	<ul style="list-style-type: none">• Hive only supports online analytical processing (OLAP). Updates and inserts are not supported.• Hive does not have requirement of schema on write. Schema can be associated at the time of read.• Loading is fast but queries are slow.• Functions in Hive are few when compared to RDBMS.• Indexes are supported in Hive as well.

Let us also look how Hive is different from Pig.

Comparing Hive and Pig

Pig	Hive
<ul style="list-style-type: none">• Pig is a procedural data flow language.• Easy to Optimize by using various optimization techniques.• Syntax for PigLatin is new and users have to learn it by sparing some time.• Pig is recommended for programmers and developers. Complex problems such as joins are easily handled by Pig.	<ul style="list-style-type: none">• Hive is SQL like declarative language.• Difficult to optimize as Hive depends on its own optimizer.• Syntax is similar to that of SQL and users familiar with SQL language feel right at home while working with Hive.• Hive is recommended for analytics.

-
- Data is represented in the form of variables in Pig. You can easily store intermediate data at any point of the pipeline.
 - Structured and unstructured data is efficiently processed by pig.
 - Writing user defined functions is easier in Pig
 - Data is represented in the form of tables in Hive. Storing intermediate data is not so easy. It requires to create new table and insert intermediate data.
 - Hive efficiently processes structured data.
 - Writing UDFs and UDAFs is complex in Hive.
-

Apache HBase

Apache HBase is an open source, column oriented and distributed database for Hadoop which supports random, real-time reads and writes of data. The data in HBase is represented by tables. HBase is not a relational database and does not support structured query language (SQL) but is capable of hosting very large tables with billions of rows and millions of columns. It is a NoSQL database and uses HDFS as underlying storage system. NoSQL databases differ from traditional relational databases and do not require data to be normalized.

HBase stores data into tables in columns rather than storing it in rows unlike relational database. The advantage of column oriented databases such as HBase is that, it makes it easy to compute aggregates on the columns which consists similar items by just reading that specific column and producing the result. Whereas in row oriented databases such as RDBMS, the entire row is read even though the query requires only one column and produces the result.

HBase is a Master-slave architecture similar to that of Hadoop. The master is *HBase Master* and the slaves are *Region Servers*. The HBase Master server is responsible for monitoring and load balancing all Region Servers in the HBase cluster, and is the interface for all metadata changes. The region servers are the slave nodes which are deployed in each machine. They store the data and process IO requests.

HBase Data Model

Data model in HBase is designed in such a way as to accommodate data that could vary in field size, data type and columns. The data model in HBase is made up of logical components such as Tables, Rows, Column Families, Columns, Cells and Versions. Let us now look at each of the component in detail.

Tables	Tables in HBase are logical collection of rows which are stored in separate partitions known as region servers. The table in a HBase looks similar to the representation below.
Rows	Rows in HBase are sorted lexicographically by a row key. The row keys are always unique in a table.
Column families	The data present in a row is grouped together as column families. The column families have one or more columns associated with it and these columns in a family are stored together in a low level storage file known as HFile. The table below has two column families: employee details and billing. The employees' column family consists of name of the employee and their designation while the billing contains the details of rate per hour and total billed for a particular month.
Columns	Columns are the part of column families which are identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon.
Cell	The cell stores data value which is the unique combination of row key, column family and the column. The data type of the value stored in a cell is always a byte array.
Version	The version is denoted by a timestamp for the data stored in a cell. We can have more than one version of the same cell and we have full control on which cell to keep.

Comparing Column Oriented and Row Oriented Databases

Column-Oriented	Row-oriented
<ul style="list-style-type: none"> Aggregations are more efficient in column-oriented databases as only the column is read. 	<ul style="list-style-type: none"> Aggregations are not efficient as the entire row is read.

<ul style="list-style-type: none"> • Best suited for Online analytical processing • Column-oriented databases can achieve high rate of compressions due to very few distinct values. • Data is stored and retrieved one row at a time and hence could read unnecessary data if only some of the data in a row is required. 	<ul style="list-style-type: none"> • Best suited for Online Transaction processing. • Higher compression rates cannot be achieved.
---	--

HBase features

- HBase is scalable by adding servers as our data grows.
- HBase provides random read or write access to large datasets.
- Tables inside HBase can be queried using Apache Hive in batch mode.
- Rows and columns are stored sequentially and hence there are no real indexes.
- Automatic failover support mechanism when nodes fail and hence it is fault tolerant.
- HBase automatically partitions and distributes the tables to other nodes when the data grows beyond the capacity of a single node.
- HBase does not require expensive hardware to work and can be deployed on commodity hardware.

Loading/Storing to HBase using Pig

Pig can be used to load or store data from or to HBase using *HBaseStorage* function. While loading data, we need to specify the location of table, name of the table and the columns of column family which we need to load. For example, consider a table in HBase called *employees*. The table contains two column families, *emp_info* and *emp_bill*. The key for the table is *emp_id*. The column family *emp_info* consists the columns *name*, *age*, *location* while the column family *emp_bill* contains *rate_per_hour*, *no_hours*, *total_hours*. This table in HBase can be load to Pig as shown below.

```
employees = LOAD 'hbase://employees' USING
org.apache.pig.backend.hadoop.hbase.HBaseStorage(
'emp_info:name, emp_bill:*, '-loadkey true') AS
(emp_id, name: chararray, rate_per_hour: float,
no_hours: float, total_hours: float);
```

The *-loadkey* option when set to true loads the row key as first column. The default value is false.

Similarly, we can also perform store data from Pig to an HBase table. We just need to specify STORE instead of LOAD to store data from Pig to HBase. The example is as shown below.

```
STORE employees INTO 'hbase://employees' USING
org.apache.pig.backend.hadoop.hbase.HBaseStorage(
'emp_info:name, emp_bill:*'');
```

If you observe carefully, we have not specified the *emp_id* field. This is because the row key is always assumed to be the first field in the Pig tuple and need not be referenced while storing and can be safely omitted.

Apache HCatalog

Apache HCatalog is a table and metadata management service for Hadoop. HCatalog helps MapReduce, Hive, Pig or any other tool users to read and write data easily from HDFS. The data in HCatalog is presented in a relational view in the form of tables. The tables are partitioned with schemas defined consistently. As a result of this abstraction in HCatalog's table, users need not worry about the location and format of the data stored in HDFS. HCatalog supports data from text files, sequence files and RCFile format in tabular view. For the matter of fact, HCatalog supports reading and writing files in any format for which a Hive SerDe can be written. However, a custom format can also be implemented by specifying *InputFormat*, *OutputFormat*, and *SerDe*. HCatalog is built on the top of Hive metastore to store metadata. Therefore, it takes in all the components from Hive Data Definition Language (DDL).

HCatalog and Pig

HCatalog provides load and store functions to load and store data from HCatalog to Pig. Let us now look at them in brief.

HCatLoader

The load function to load data to Pig from HCatalog is *HCatLoader*. All you need to do is specify the name of the table to load data to Pig from HCatalog using *HCatLoader*. As the schema is already defined in HCatalog, you need not again

specify the schema or data types while loading. This will get automatically loaded by the function. Moreover, as the tables in HCatalog are partitioned, when a FILTER statement is specified just after the LOAD statement, the filter condition is pushed up to the load statement which as a result returns only required partitions. An example of working with *HCatLoader* is as shown below.

Consider a table *movies* available in HCatalog. Let us load it to Pig using the *HCatLoader* function.

```
movies      =      LOAD      'movies'      USING
org.apache.hcatalog.pig.HCatLoader();

--Using Filter on ratings will cause it to push up to the
loader.

top_rated = FILTER movies BY ratings == 5;
```

HCatStorer

The store function to store data from Pig to HCatalog is *HCatStorer*. Similar to loader function, we have to specify the name of the table where the records have to be stored. However, as a constructor argument in the store function, we have to specify the partition key value pair which represents the partition. This is a strict requirement when writing to a partition table. An example of working with *HCatStorer* is as shown below.

Let us simply store the filtered records from the above statement to HCatalog table, *top_movies*.

```
STORE      top_rated      INTO      top_movies      USING
org.apache.hcatalog.pig.HCatStorer('ratings == 5');
```

Other Hadoop Ecosystem Projects

Let us now briefly look at other Hadoop ecosystem projects which make our lives a lot easier.

Apache Zookeeper

ZooKeeper is an open source, robust distributed coordination service for distributed applications. ZooKeeper is an open source Apache Software Foundation project available for free and is ready to use. ZooKeeper helps in overcoming many of the common challenges faced by distributed applications. ZooKeeper can be used for synchronization, sequential consistency and coordination between distributed applications. It helps in maintaining the configuration information which can be shared to all the nodes in a distributed system. ZooKeeper also helps in group

services such as leader election and many more. ZooKeeper is reliable and fast yet very simple to work with. With ZooKeeper you can build reliable, distributed data structures for group membership, leader election, coordinated workflow, and configuration services. You can also build generalized distributed data structures like locks, queues, barriers, and latches.

ZooKeeper provides an eventually consistent view of its *znodes* which are nothing but files or directories in a file system. It provides basic CRUD operations such as creating, updating, and deleting *znodes*. It provides an event-driven model in which clients can watch for changes to specific *znodes*, for example if a new child is added to an existing *znode*. ZooKeeper is a high availability service as it consists of a set of ZooKeeper servers known as *Ensemble* (cluster), with each of the server holding an in-memory image of the distributed file system to serve client read requests. Each server also holds a persistent copy on disk.

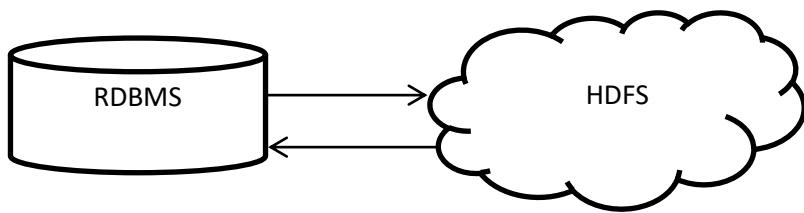
One of the servers in the *Ensemble* dynamically chosen by consensus as the leader, and all other servers are followers. The leader is responsible for all writes and for updating the changes to its followers. When the majority of followers update a change successfully, the write succeeds and the data is still available even if the leader fails. When a leader fails, a new leader is again dynamically chosen by consensus within the *Ensemble*. This eliminates the single point of failure scenario and the *Ensemble* keeps on doing its work as it should.

When a client connects to ZooKeeper, it is provided with the list of servers in the *Ensemble*. The client connects to one of the servers in the ensemble at random until a connection is established. Once connected, ZooKeeper creates a session with a pre specified timeout period by client. The ZooKeeper client automatically sends heartbeats periodically to keep the session alive if no operations are performed for a while, and automatically handles failover. If the connection between ZooKeeper and client fails, the client automatically detects this and retries to connect to a different server in the *Ensemble*. After it is reconnected, the same client session is retained while the failure has occurred

Apache Sqoop

Apache Sqoop is a tool designed for importing and exporting data from HDFS to relational database management systems such as MySQL and Oracle. Sqoop can be used to import data from RDBMS and HDFS, perform MapReduce jobs on the imported data or use high level tool such as Hive and export the processed or transformed data back to RDBMS. Sqoop is an open source top level Apache Software Foundation project. The import/export process is automated and does not require any user input once started. The schema is attached from the database for the data which is being imported. Sqoop uses MapReduce to import and export the

data, which provides parallel operation as well as fault tolerance. Sqoop can manage both of these processes by using the Sqoop Import and Sqoop Export functions.



Similar to that of Hadoop, Sqoop is also written in Java which provides an API known as Java Database Connectivity a.k.a JDBC. If the database platform from which the data is being imported or exported is native to Java, Sqoop can work directly. If the database is not native to Java, we can hook into the database using Sqoop connectors.

Apache Flume

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of streaming data into HDFS. The streaming data can be from multiple sources such as application logs, server logs, GPS tracking, social media updates, digital sensors, event logs, emails and almost any possible data source. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failover and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application. Flume is an open source top level Apache Software Foundation project. There are other technologies such as Facebook Scribe and Apache Kafka similar to that of Flume, but Flume is being the choice for most of the users who deal with such operations.

Apache Oozie

Apache Oozie is a workflow scheduler system to manage data processing jobs for Apache Hadoop and its ecosystem. Oozie workflow is collection of jobs triggered by different tools such as MapReduce or Pig jobs arranged in a Directed Acyclic Graph (DAG), specifying a sequence of actions to be executed. Oozie is extensible, scalable and reliable service. Oozie can be easily integrated with rest of the Hadoop ecosystem projects and supports almost all the Hadoop jobs such as Java map-reduce, Streaming map-reduce, Pig, Hive, Sqoop and Distcp. It also supports system specific jobs such as Java programs and shell scripts.

Cascading

Cascading is open source software abstraction layer for data processing built on the top of Hadoop. Cascading is used to build complex data processing workflows in Java, which can be very challenging if built with MapReduce. Cascading is similar to that of Pig but all the coding is done in Java. Cascading allows more control to the user but users still have to code low level.

MongoDB

MongoDB is open source document-oriented NoSQL database designed to deliver high performance, high availability, and automatic scaling. Each record in MongoDB is a document similar to JSON objects. The data structure is made up of field and value pairs. The values of fields may include other documents, arrays, and arrays of documents.

Apart from the above projects, there is a huge list of Hadoop ecosystem projects available. Most of them are available under Apache. You can check out the complete list from the URL available in the REFERENCES section.

LAB EXERCISE 13

"There are no activities required for this lab"

SUMMARY

Hadoop, an open source top level Apache project has gained wide popularity due to its cutting edge technology and features. It is being adopted worldwide to process huge amounts of data stored in disk drives and extract information worth billions in a quick and cost effective manner. Hadoop is being adopted in almost every industry you can think of. Therefore, a lot of applications (e.g. Pig) are developed and are being developed to help working with Hadoop and make our lives easy. These applications which are developed for working with Hadoop are termed as Hadoop Ecosystem projects

REFERENCES

- <https://hadoopecosystemtable.github.io/>

INDEX

Apache Hive	390
Apache HBase	392
Apache HCatalog.....	395
Other Hadoop Ecosystem Projects	396
Lab Exercise 13	400
SUMMARY	401
REFERENCES.....	402