## IS3S602 ADCANCED DATABASES

Apache Pig Tutorial – Part 1

*Apache Pig is a tool used to analyze large amounts of data by represeting them as data flows. Using the PigLatin scripting language operations like ETL (Extract, Transform and Load), adhoc data anlaysis and iterative processing can be easily achieved.*

Pig is an abstraction over MapReduce. In other words, all Pig scripts internally are converted into Map and Reduce tasks to get the task done. Pig was built to make programming MapReduce applications easier. Before Pig, Java was the only way to process the data stored on HDFS.

Dataset

The dataset is a simple text (movies\_data.csv) file that lists movie names and details like release year, rating and runtime.

A sample of the dataset is as follows:

1,The Nightmare Before Christmas,1993,3.9,4568

2,The Mummy,1932,3.5,4388

3,Orphans of the Storm,1921,3.2,9062

4,The Object of Beauty,1991,2.8,6150

5,Night Tide,1963,2.8,5126

6,One Magic Christmas,1985,3.8,5333

7,Muriel's Wedding,1994,3.5,6323

8,Mother's Boys,1994,3.4,5733

9,Nosferatu: Original Version,1929,3.5,5651

10,Nick of Time,1995,3.4,5333

# Load the data into HDFS

* Start the Cloudera VM by going to:
* C:\Virtual Machines\cloudera-quickstart-vm-5.8.0-0-vmware\ and double-clicking the vmx file.
* Open Hue form the quickstart page. Login in is cloudera/cloudera.
* Download the movies\_data.csv file from Blackboard.
* Go to the file browser in Hue and upload the data.

The file has a total of **49590** records.

To start pig, open a terminal shell and type:

$ pig

grunt>

This command presents you with a grunt shell. The grunt shell allows you to execute PigLatin statements to quickly test out data flows on your data step by step without having to execute complete scripts. Pig is now installed and we can go ahead and start using Pig to play with data.

Pig Latin

To learn Pig Latin, let’s question the data. Before we start asking questions, we need the data to be accessible in Pig.

Use the following command to load the data:

grunt> movies = LOAD '/user/cloudera/movies\_data.csv' USING PigStorage(',') as (id,name,year,rating,duration);

The above statement is made up of two parts. The part to the left of “=” is called the relation or alias. It looks like a variable but you should note that this is not a variable. When this statement is executed, no MapReduce task is executed.

Since the dataset has records with fields separated by a comma we use the keyword USING PigStorage(‘,’).

Another thing we have done in the above statement is giving the names to the fields using the ‘as’ keyword.

Test to see if the alias has the data we loaded.

grunt> DUMP movies;

It is only after the DUMP statement that a MapReduce job is initiated.

Now we have data in Pig we can do some processing.

***List the movies that having a rating greater than 4***

grunt> movies\_greater\_than\_four = FILTER movies BY (float)rating>4.0;

grunt> DUMP movies\_greater\_than\_four;

The above statements filters the alias movies and store the results in a new alias *movies\_greater\_than\_four*. The *movies\_greater\_than\_four* alias will have only records of movies where the rating is greater than 4.

The DUMP command is only used to display information onto the standard output. If you need to store the data to a file you can use the following command:

grunt> store movies\_greater\_than\_four into '/user/cloudera/movies\_greater\_than\_four';

Check the data has been stored in the Hue file browser.

Apache Pig Tutorial – Part 2

Let’s have a quick look at the FILTER command from our [Part 1](http://www.rohitmenon.com/index.php/apache-pig-tutorial-part-1/):

grunt> movies\_greater\_than\_four = FILTER movies BY (float)rating>4.0;

Here, we see a (float) keyword placed before the column ‘rating’. This is done to tell Pig that the column we are working on is of type, float. Pig was not informed about the type of the column when the data was loaded.

Following is the command we used to load the data:

grunt> movies = LOAD '/user/cloudera/movies\_data.csv' USING PigStorage(',') as (id,name,year,rating,duration);

The load command specified only the column names. We can modify the statement as follows to include the data type of the columns:

grunt> movies = LOAD '/user/cloudera/movies\_data.csv' USING PigStorage(',') as (id:int,name:chararray,year:int,rating:double,duration:int);

In the above statement, name is chararray (string), rating is a double and fields id, year and duration are integers. If the data was loaded using the above statement we would not need to cast the column during filtering.  
The datatypes used in the above statement are called scalar data types. The other scalar types are long, double and bytearray.

To get better at using filters, let’s ask the data a few more questions:

***List the movies that were released between 1950 and 1960***

grunt> movies\_between\_50\_60 = FILTER movies by year>1950 and year<1960;

***List the movies that start with the Alpahbet A***

grunt> movies\_starting\_with\_A = FILTER movies by name matches 'A.\*';

***List the movies that have duration greater that 2 hours***

grunt> movies\_duration\_2\_hrs = FILTER movies by duration > 7200;

***List the movies that have rating between 3 and 4***

grunt> movies\_rating\_3\_4 = FILTER movies BY rating>3.0 and rating<4.0;

***DESCRIBE***

*The schema of a relation/alias can be viewed using the DESCRIBE command:*

grunt> DESCRIBE movies;

movies: {id: int,name: chararray,year: int,rating: double,duration: int}

**ILLUSTRATE**

To view the step-by-step execution of a sequence of statements you can use the ILLUSTRATE command:

grunt> ILLUSTRATE movies\_duration\_2\_hrs;

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

| movies | id:int | name:chararray | year:int | rating:double | duration:int |

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

| | 1567 | Barney: Sing & Dance with Barney | 2004 | 2.7 | 3244 |

| | 3045 | Strange Circus | 2005 | 2.8 | 6509 |

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

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

| movies\_duration\_2\_hrs | id:int | name:chararray | year:int | rating:double | duration:int |

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

| | 3045 | Strange Circus | 2005 | 2.8 | 6509 |

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

DESCRIBE and ILLUSTRATE are really useful for debugging.

**Complex Types**  
Pig supports three different complex types to handle data. It is important that you understand these types properly as they will be used very often when working with data.

**Tuples**  
A tuple is just like a row in a table. It is comma separated list of fields.

(49539,'The Magic Crystal',2013,3.7,4561)

The above tuple has five fields. A tuple is surrounded by brackets.

**Bags**  
A bag is an unordered collection of tuples.

{ (49382, 'Final Offer'), (49385, 'Delete') }

The above bag is has two tuples. Each tuple has two fields, Id and movie name.

**Maps**  
A map is a <key, value> store. The key and value are joined together using #.

['name'#'The Magic Crystal', 'year'#2013]

The above map has two keys and name and year and have values ‘The Magic Crystal’ and 2013. The first value is a chararray and the second one is an integer.

We will be using the above complex type quite often in our future examples.

**FOREACH**

FOREACH gives a simple way to apply transformations based on columns. Let’s understand this with an example.

***List the movie names and their duration in minutes***

grunt> movie\_duration = FOREACH movies GENERATE name, (double)(duration/60);

The above statement generates a new alias that has the list of movies and it duration in minutes.  
You can check the results using the DUMP command.

**GROUP**

The GROUP keyword is used to group fields in a relation.

List the years and the number of movies released each year.

grunt> grouped\_by\_year = group movies by year;

grunt> count\_by\_year = FOREACH grouped\_by\_year GENERATE group, COUNT(movies);

You can check the result by dumping the count\_by\_year relation on the screen.

We know in advance that the total number of movies in the dataset is 49590. We can check to see if our GROUP operation is correct by verify the total of the COUNT field. If he sum of of the count field is 49590 we can be confident that our grouping has worked correctly.

grunt> group\_all = GROUP count\_by\_year ALL;

grunt> sum\_all = FOREACH group\_all GENERATE SUM(count\_by\_year.$1);

grunt> DUMP sum\_all;

From the above three statements, the first statement, GROUP ALL, groups all the tuples to one group. This is very useful when we need to perform aggregation operations on the entire set.  
The next statement, performs a FOREACH on the grouped relation group\_all and applies the SUM function to the field in position 1 (positions start from 0). Here field in position 1, are the counts of movies for each year. One execution of the DUMP statement the MapReduce program kicks off and gives us the following result:

(49590)

The above value matches to our know fact that the dataset has 49590 movies. So we can conclude that our GROUP operation worked successfully.

**ORDER BY**

Let us question the data to illustrate the ORDER BY operation.

***List all the movies in the ascending order of year.***

grunt> asc\_movies\_by\_year = ORDER movies BY year ASC;

grunt> DUMP asc\_movies\_by\_year;

***List all the movies in the descending order of year.***

grunt> desc\_movies\_by\_year = ORDER movies by year DESC;

grunt> DUMP desc\_movies\_by\_year;

**DISTINCT**

The DISTINCT statement is used to remove duplicated records. It works only on entire records, not on individual fields.  
Let’s illustrate this with an example:

grunt> movies\_with\_dups = LOAD '/user/cloudera/movies\_with\_duplicates.csv' USING PigStorage(',') as (id:int,name:chararray,year:int,rating:double,duration:int);

grunt> DUMP movies\_with\_dups;

(1,The Nightmare Before Christmas,1993,3.9,4568)

(1,The Nightmare Before Christmas,1993,3.9,4568)

(1,The Nightmare Before Christmas,1993,3.9,4568)

(2,The Mummy,1932,3.5,4388)

(3,Orphans of the Storm,1921,3.2,9062)

(4,The Object of Beauty,1991,2.8,6150)

(5,Night Tide,1963,2.8,5126)

(5,Night Tide,1963,2.8,5126)

(5,Night Tide,1963,2.8,5126)

(6,One Magic Christmas,1985,3.8,5333)

(7,Muriel's Wedding,1994,3.5,6323)

(8,Mother's Boys,1994,3.4,5733)

(9,Nosferatu: Original Version,1929,3.5,5651)

(10,Nick of Time,1995,3.4,5333)

(9,Nosferatu: Original Version,1929,3.5,5651)

You see that there are are duplicates in this data set. Now let us***list the distinct records present movies\_with\_dups***:

grunt> no\_dups = DISTINCT movies\_with\_dups;

grunt> DUMP no\_dups;

(1,The Nightmare Before Christmas,1993,3.9,4568)

(2,The Mummy,1932,3.5,4388)

(3,Orphans of the Storm,1921,3.2,9062)

(4,The Object of Beauty,1991,2.8,6150)

(5,Night Tide,1963,2.8,5126)

(6,One Magic Christmas,1985,3.8,5333)

(7,Muriel's Wedding,1994,3.5,6323)

(8,Mother's Boys,1994,3.4,5733)

(9,Nosferatu: Original Version,1929,3.5,5651)

(10,Nick of Time,1995,3.4,5333)

**LIMIT**

Use the LIMIT keyword to get only a limited number for results from relation.

grunt> top\_10\_movies = LIMIT movies 10;

grunt> DUMP top\_10\_movies;

(1,The Nightmare Before Christmas,1993,3.9,4568)

(2,The Mummy,1932,3.5,4388)

(3,Orphans of the Storm,1921,3.2,9062)

(4,The Object of Beauty,1991,2.8,6150)

(5,Night Tide,1963,2.8,5126)

(6,One Magic Christmas,1985,3.8,5333)

(7,Muriel's Wedding,1994,3.5,6323)

(8,Mother's Boys,1994,3.4,5733)

(9,Nosferatu: Original Version,1929,3.5,5651)

(10,Nick of Time,1995,3.4,5333)

**SAMPLE**  
Use the sample keyword to get sample set from your data.

grunt> sample\_10\_percent = sample movies 0.1;

grunt> dump sample\_10\_percent;

Here, 0.1 = 10%  
As we already know that the file has 49590 records. We can check to see the count of records in the relation.

grunt> sample\_group\_all = GROUP sample\_10\_percent ALL;

grunt> sample\_count = FOREACH sample\_group\_all GENERATE COUNT(sample\_10\_percent.$1);

grunt> dump sample\_count;

The output is (4911) which is approximately 10% for 49590.