A guide to Install and Configure PIG on Hadoop-2.6.0
Sardar Vallabhbhai National Institute of Technology, Surat

3. Installing Pig on Hadoop-2.2.0

3.1 Download Pig-0.12.0

Use the below link in your Ubuntu Vm's Mozilla browser to download pig tar file.

https://drive.google.com/file/d/0B-Cl0IfLnRozdTBoclBZd1BVX2s/edit?usp=sharing

By default the downloaded file will go to Downloads folder, so move it to home directory.

$sudo mv Downloads/pig-0.12.0.tar.gz /home/user/

$ls

```
user@ubuntu:~$ sudo mv Downloads/pig-0.12.0.tar.gz /home/user/
user@ubuntu:~$ ls
Desktop    examples.desktop     hello   Pictures           pig_1396538123754.log  Videos
Documents  hadoop-2.2.0         hello~  pig-0.12.0         Public
Downloads  hadoop-2.2.0.tar.gz  Music   pig-0.12.0.tar.gz  Templates
user@ubuntu:~$
```

3.2 Untar the package

$tar -xvf pig-0.12.0.tar.gz

$ls

```
user@ubuntu:~$ ls
Desktop    examples.desktop     Music         pig-0.12.0.tar.gz  Videos
Documents  hadoop-2.2.0         Pictures      Public
Downloads  hadoop-2.2.0.tar.gz  pig-0.12.0    Templates
user@ubuntu:~$
```

3.3 Edit .bashrc file

We need to set pig home and path in .bashrc file.

a) Change directory to home.

$ cd

b) Edit the file

$ gedit .bashrc

export PIG_INSTALL=/home/user/pig-0.12.0

export PATH=$PATH:$PIG_INSTALL/bin

export PIG_CLASSPATH=/home/user/hadoop-2.6.0/sbin

```
#pig

export PIG_INSTALL=/home/user/pig-0.12.0
export PATH=$PATH:$PIG_INSTALL/bin
export PIG_CLASSPATH=/home/user/hadoop-2.2.0/sbin
```

3.4 Configure JAVA_HOME

Configure JAVA_HOME in '.bashrc'.

Update the JAVA_HOME to:

export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-i386

```
# The java implementation to use.
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-i386
```

Source the .bashrc file to set the pig environment variables without having to invoke a

new shell:

$source .bashrc

3.5 Start Grunt Shell

To confirm that pig has installed correctly, type pig in the terminal and press enter, it should give you grunt shell.

```
user@ubuntu:~$ pig
2014-04-04 02:55:34,778 [main] INFO  org.apache.pig.Main - Apache Pig version 0.12.1-SNAPSHOT (rexp
 22 2014, 07:41:29
2014-04-04 02:55:34,779 [main] INFO  org.apache.pig.Main - Logging error messages to: /home/user/pi
2014-04-04 02:55:34,831 [main] INFO  org.apache.pig.impl.util.Utils - Default bootup file /home/use
und
2014-04-04 02:55:35,148 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - mapred.job.
ed. Instead, use mapreduce.jobtracker.address
2014-04-04 02:55:35,148 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.
 Instead, use fs.defaultFS
2014-04-04 02:55:35,148 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine
doop file system at: hdfs://localhost:9000
2014-04-04 02:55:35,151 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - mapred.used
er is deprecated. Instead, use mapreduce.client.genericoptionsparser.used
2014-04-04 02:55:36,002 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.
 Instead, use fs.defaultFS
grunt>
```

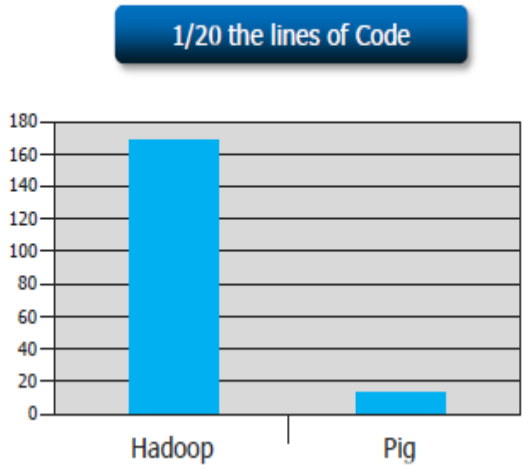
What is Pig and Pig Latin?

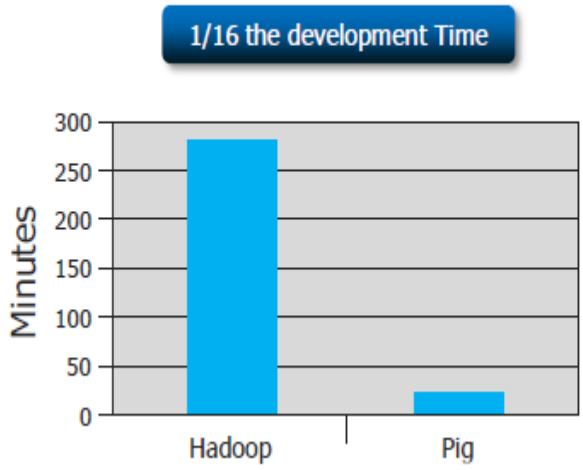
Pig is an open-source high level data flow system. It provides a simple language called Pig Latin, for queries and data manipulation, which are then compiled in to MapReduce jobs that run on Hadoop.

Pig is important as companies like Yahoo, Google and Microsoft are collecting huge amounts of data sets in the form of click streams, search logs and web crawls. Pig is also used in some form of ad-hoc processing and analysis of all the information.

Why Do you Need Pig?

- It's easy to learn, especially if you're familiar with SQL.
- Pig's multi-query approach reduces the number of times data is scanned. This means 1/20th the lines of code and 1/16th the development time when compared to writing raw MapReduce.





- Performance of Pig is in par with raw MapReduce
- Pig provides data operations like filters, joins, ordering, etc. and nested data types like tuples, bags, and maps, that are missing from MapReduce.
- Pig Latin is easy to write and read.

Why was Pig Created?

Pig was originally developed by Yahoo in 2006, for researchers to have an ad-hoc way of creating and executing MapReduce jobs on very large data sets. It was created to reduce the development time through its multi-query approach. Pig is also created for professionals from non-Java background, to make their job easier.

Where Should Pig be Used?

Pig can be used under following scenarios:

- When data loads are time sensitive.
- When processing various data sources.
- When analytical insights are required through sampling.

Where Not to Use Pig?

- In places where the data is completely unstructured, like video, audio and readable text.
- In places where time constraints exist, as Pig is slower than MapReduce jobs.
- In places where more power is required to optimize the codes.

Applications of Apache Pig:

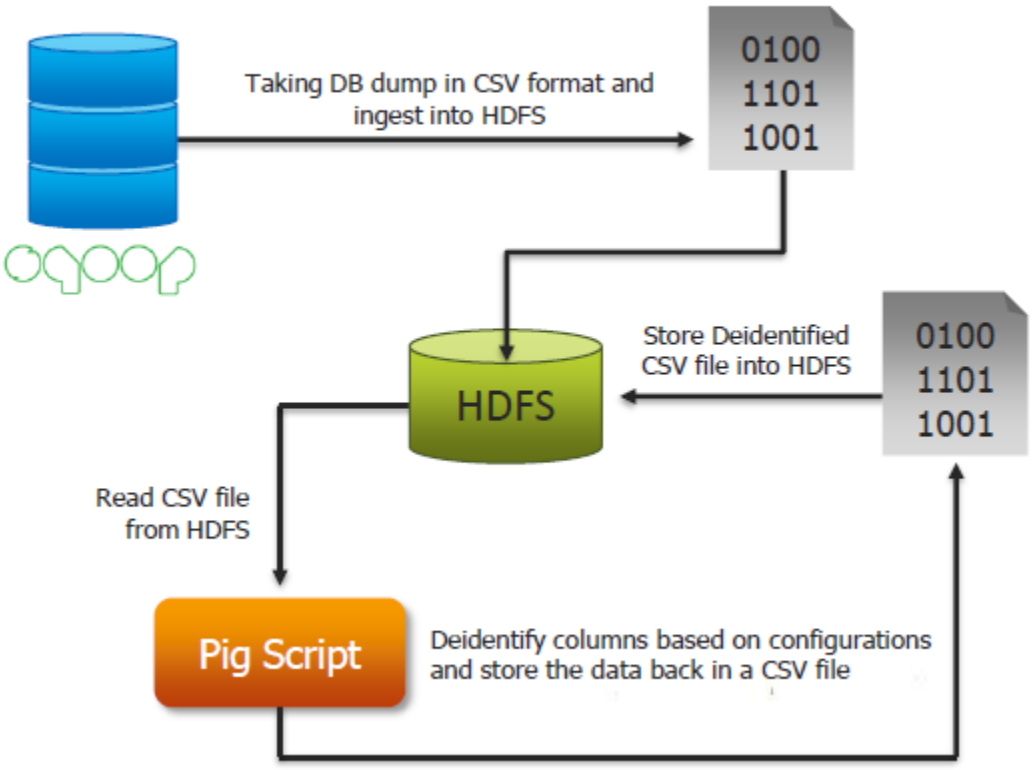
- Processing of web logs.
- Data processing for search platforms.
- Support for Ad-hoc queries across large data sets.
- Quick prototyping of algorithms for processing large data sets.

How Yahoo! Uses Pig:

Yahoo uses Pig for the following purpose:

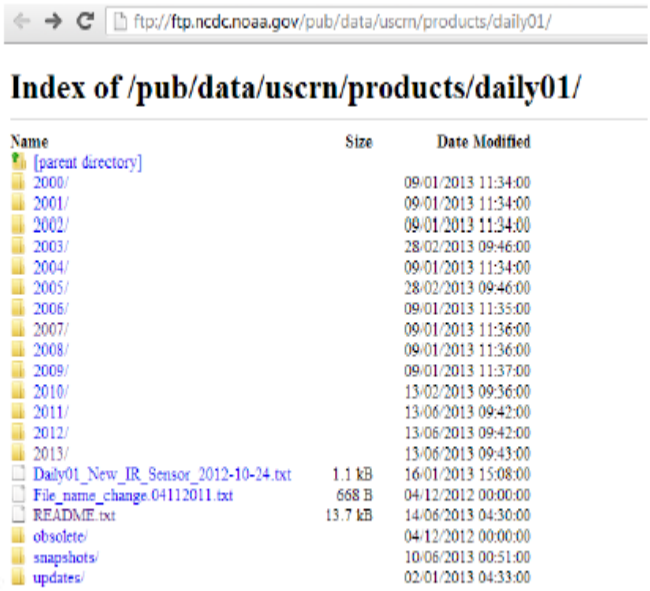
- **In Pipelines** – To bring logs from its web servers, where these logs undergo a cleaning step to remove bots, company interval views and clicks.
- **In Research** – To quickly write a script to test a theory. Pig Integration makes it easy for the researchers to take a Perl or Python script and run it against a huge data set.

Use Case of Pig in Healthcare Domain:



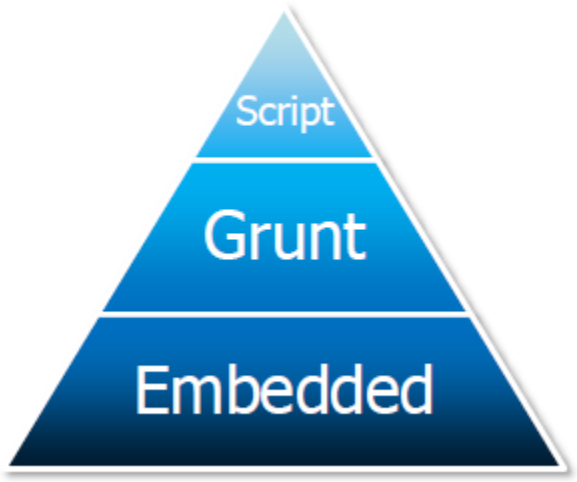
The above diagram gives a clear, step by step explanation of how the data flows through Sqoop, HDFS and Pig script.

Comparing MapReduce and Pig using Weather Data:



Source: ftp://ftp.ncdc.noaa.gov/pub/data/uscrn/products/daily01/

Basic Program Structure of Pig:



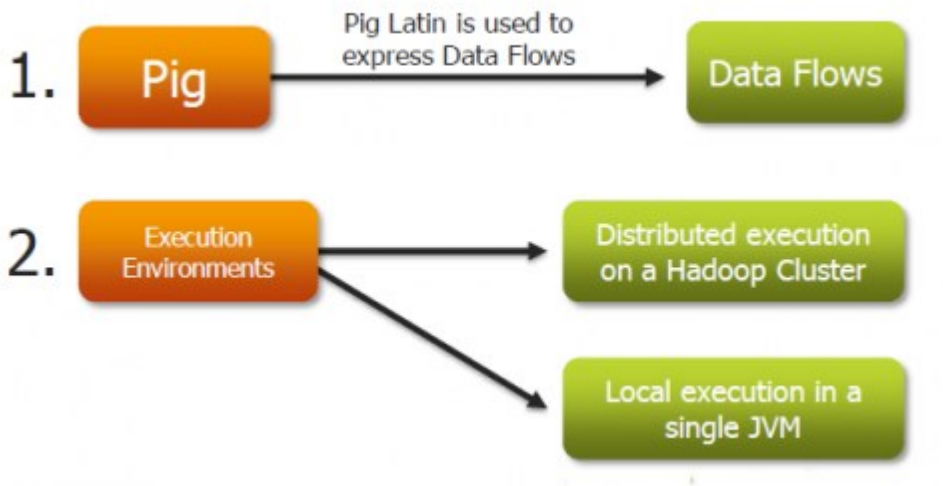
Source: Pig Wiki

Here's the hierarchy of Pig's program structure:

- **Script** – Pig Can run a file script that contains Pig Commands. Eg: pig script .pig runs the command in the local file script.pig

- **Grunt** – It is an interactive shell for running Pig commands. It is also possible to run pig scripts from within Grunts using run and exec.
- **Embedded** – Can run Pig programs from Java, much like you can use JDBC to run SQL programs from Java.

Components of Pig:



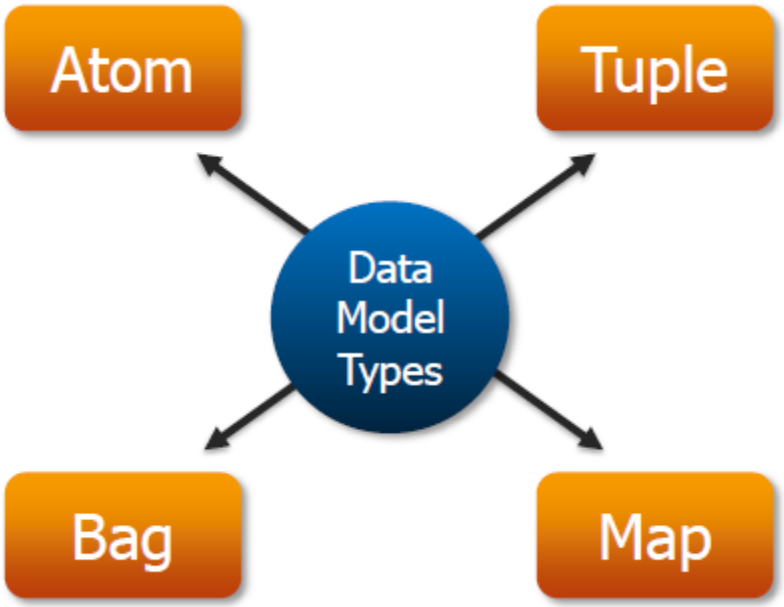
What is Pig Latin Program?

Pig Latin program is made up of a series of operations or transformations that are applied to the input data to produce output. The job of Pig is to convert the transformations in to a series of MapReduce jobs.

Basic Types of Data Models in Pig:

Pig comprises of 4 basic types of data models. They are as follows:

- Atom – It is a simple atomic data value. It is stored as a string but can be used as either a string or a number
- Tuple – An ordered set of fields
- Bag – An collection of tuples.
- Map – set of key value pairs.



operators-in-apache-pig

Operators in Apache Pig: Part 1- Relational Operators

This post is about the operators in Apache Pig. Let's take a quick look at what Pig and Pig Latin is and the different modes in which they can be operated, before heading on to Operators.

What is Apache Pig?

Apache Pig is a high-level procedural language for querying large data sets using Hadoop and the Map Reduce Platform. It is a Java package, where the scripts can be executed from any language implementation running on the JVM. This is greatly used in iterative processes.

Apache Pig simplifies the use of Hadoop by allowing SQL-like queries to a distributed dataset and makes it possible to create complex tasks to process large volumes of data quickly and effectively. The best feature of Pig is that, it backs many relational features like Join, Group and Aggregate.

I know Pig sounds a lot more like an ETL tool and it does have many features common with ETL tools. But the advantage of Pig over ETL tools is that it can run on many servers simultaneously.

What is Apache Pig Latin?

Apache Pig create a simpler procedural language abstraction over Map Reduce to expose a more Structured Query Language (SQL)-like interface for Hadoop applications called Apache Pig Latin, So instead of writing a separate Map Reduce application, you can write a single script in Apache Pig Latin that is automatically parallelized and distributed across a cluster. In simple words, Pig Latin, is a sequence of simple statements taking an input and producing an output. The input and output data are composed of bags, maps, tuples and scalar.

Apache Pig Execution Modes:

Apache Pig has two execution modes:

- **Local Mode**

In 'Local Mode', the source data would be picked from the local directory in your computer system. The MapReduce mode can be specified using 'pig –x local' command.

```
cloudera@cloudera-vm:~$ pig -x local
2013-11-21 23:08:28,088 [main] INFO  org.apache.pig.Main - Logging error messages to: /home/cloudera/pig_1385104108087.log
2013-11-21 23:08:28,208 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop fil
e system at: file:///
grunt>
```

- **MapReduce Mode:**

To run Pig in MapReduce mode, you need access to Hadoop cluster and HDFS installation. The MapReduce mode can be specified using the 'pig' command.

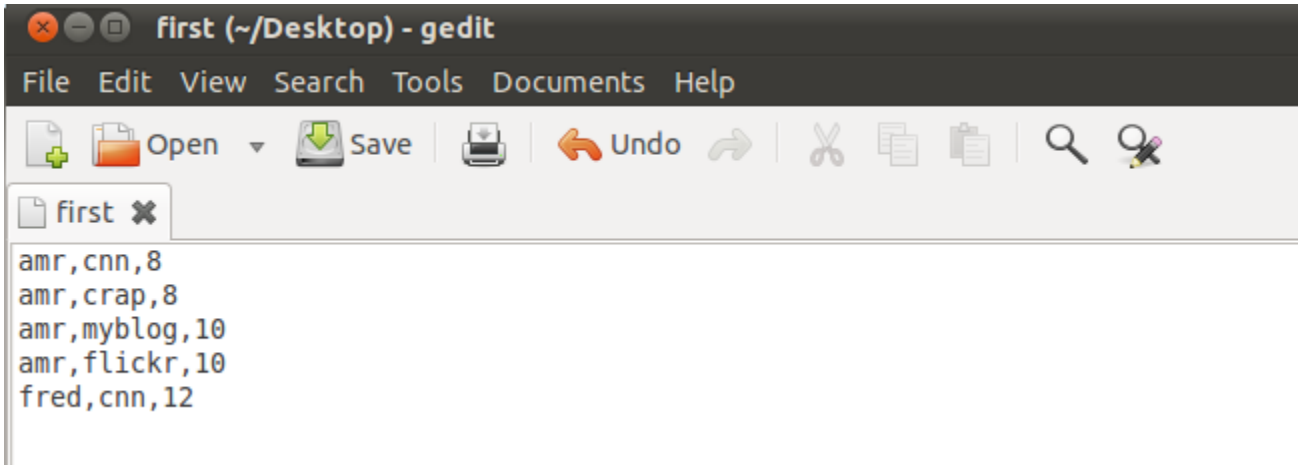
```
cloudera@cloudera-vm:~$ pig
2013-11-21 22:54:24,964 [main] INFO  org.apache.pig.Main - Logging error messages to: /home/cloudera/pig_1385103264962.log
2013-11-21 22:54:25,154 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop fil
e system at: hdfs://localhost:8020
2013-11-21 22:54:25,405 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to map-reduce
 job tracker at: localhost:8021
grunt>
```

Apache Pig Operators:

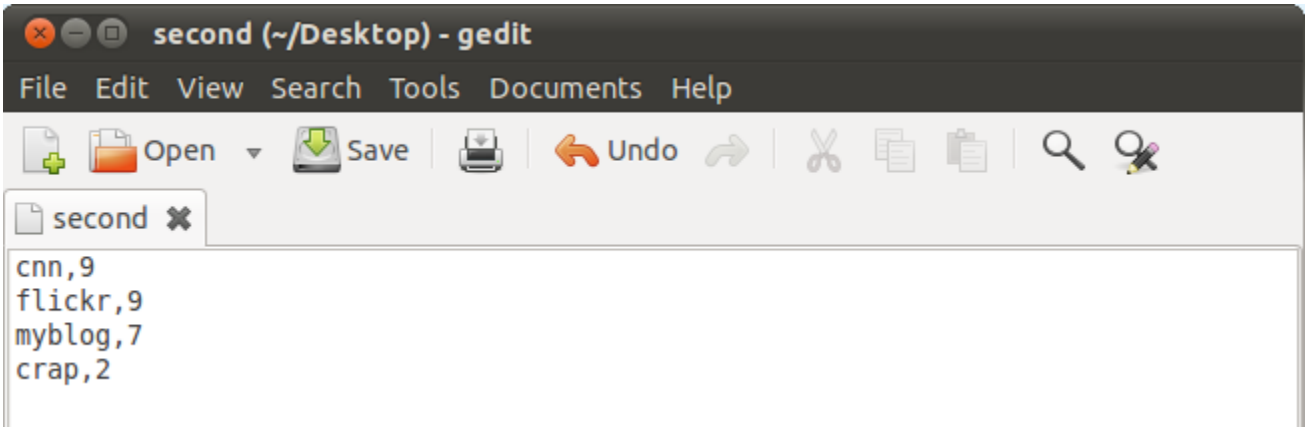
The Apache Pig Operators is a high-level procedural language for querying large data sets using Hadoop and the Map Reduce Platform. A Pig Latin statement is an operator that takes a relation as input and produces another relation as output. These operators are the main tools for Pig Latin provides to operate on the data. They allow you to transform it by sorting, grouping, joining, projecting, and filtering.

Let's create two files to run the commands:

We have two files with name 'first' and 'second.' The first file contain three fields: user, url & id.



The second file contain two fields: url & rating. These two files are CSV files.



The Apache Pig operators can be classified as: *Relational and Diagnostic.*

Relational Operators:

Relational operators are the main tools Pig Latin provides to operate on the data. It allows you to transform the data by sorting, grouping, joining, projecting and filtering. This section covers the basic relational operators.

LOAD:

LOAD operator is used to load data from the file system or HDFS storage into a Pig relation.

In this example, the Load operator loads data from file 'first' to form relation 'loading1'. The field names are user, url, id.

```
grunt> loading1 = load '/first' using PigStorage(',') as(user:chararray,url:chararray,id:int);
grunt>
```

```
grunt> loading2 = load '/second' using PigStorage(',') as(url:chararray,rating:int);
grunt>
```

FOREACH:

This operator generates data transformations based on columns of data. It is used to add or remove fields from a relation. Use FOREACH-GENERATE operation to work with columns of data.

```
grunt> for_each = foreach loading1 generate url,id;
grunt> dump for_each;
```

FOREACH Result:

```
(cnn,8)
(crap,8)
(myblog,10)
(flickr,10)
(cnn,12)
grunt>
```

FILTER:

This operator selects tuples from a relation based on a condition.

In this example, we are filtering the record from 'loading1' when the condition 'id' is greater than 8.

```
grunt> filter_command = filter loading1 by id>8;
grunt> dump filter_command;
```

FILTER Result:

```
(amr,myblog,10)
(amr,flickr,10)
(fred,cnn,12)
grunt>
```

JOIN:

JOIN operator is used to perform an inner, equijoin join of two or more relations based on common field values. The JOIN operator always performs an inner join. Inner joins ignore null keys, so it makes sense to filter them out before the join.

In this example, join the two relations based on the column 'url' from 'loading1' and 'loading2'.

```
grunt> join_command = join loading1 by url,loading2 by url;
grunt> dump join_command;
```

JOIN Result:

```
(amr,cnn,8,cnn,9)
(fred,cnn,12,cnn,9)
(amr,crap,8,crap,2)
(amr,flickr,10,flickr,9)
(amr,myblog,10,myblog,7)
grunt>
```

ORDER BY:

Order By is used to sort a relation based on one or more fields. You can do sorting in ascending or descending order using ASC and DESC keywords.

In below example, we are sorting data in loading2 in ascending order on ratings field.

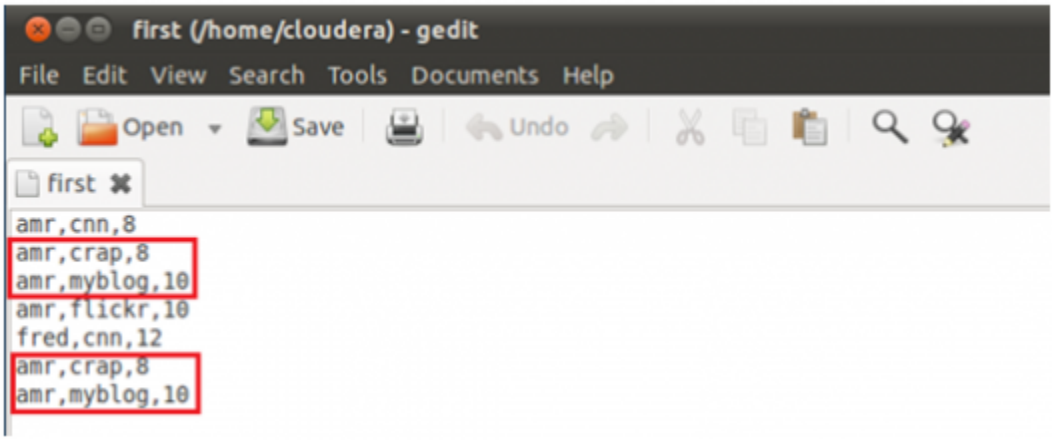
```
grunt> loading4 = ORDER loading2 by rating ASC;
grunt> dump loading4;
```

ORDER BY Result:

```
(crap,2)
(myblog,7)
(cnn,9)
(flickr,9)
grunt>
```

DISTINCT:

Distinct removes duplicate tuples in a relation.Lets take an input file as below, which has **amr,crap,8** and **amr,myblog,10** twice in the file. When we apply distinct on the data in this file, duplicate entries are removed.




```
grunt> loading1 = load '/first' using PigStorage(',') as (user:chararray,url:chararray,id:int);
grunt> loading3 = DISTINCT loading1;
grunt> dump loading3;
```

DISTINCT Result:

```
(amr,cnn,8)
(amr,crap,8)
(amr,flickr,10)
(amr,myblog,10)
(fred,cnn,12)
grunt>
```

STORE:

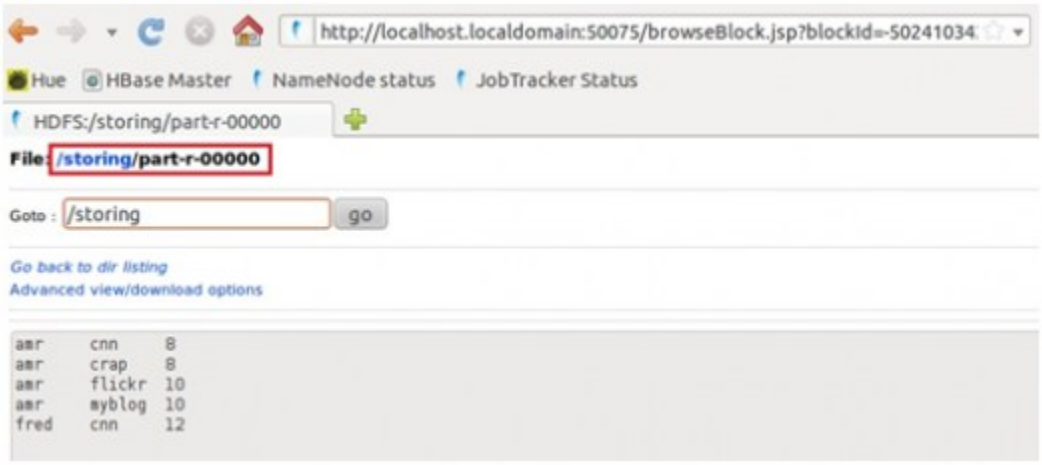
Store is used to save results to the file system.

Here we are saving **loading3** data into a file named **storing** on HDFS.

```
grunt> store loading3 into '/storing';
```

STORE Result:

```
Input(s):
Successfully read 7 records (426 bytes) from: "/first"

Output(s):
Successfully stored 5 records (61 bytes) in: "/storing"
```



GROUP:

The GROUP operator groups together the tuples with the same group key (key field). The key field will be a tuple if the group key has more than one field, otherwise it will be the same type as that of the group key. The result of a GROUP operation is a relation that includes one tuple per group.

In this example, group th

```
ɑɪnuʃ> qnwb ɑɪonb_cowwɜuq;
ɑɪnuʃ> ɑɪonb_cowwɜuq = ɑɪonb ʃoɜqɪuɑɪ pʎ nɪʃ;
```

e relation 'loading1' by column url.

GROUP Result:

```
(cnn,{(amr,cnn,8),(fred,cnn,12)})
(crap,{(amr,crap,8)})
(flickr,{(amr,flickr,10)})
(myblog,{(amr,myblog,10)})
grunt>
```

COGROUP:

COGROUP is same as GROUP operator. For readability, programmers usually use GROUP when only one relation is involved and COGROUP when multiple relations re involved.

In this example group the 'loading1' and 'loading2' by url field in both relations.

```
grunt> cogroup_command = cogroup loading1 by url,loading2 by url;
grunt> dump cogroup_command;
```

COGROUP Result:

```
(cnn,{(amr,cnn,8),(fred,cnn,12)},{(cnn,9)})
(crap,{(amr,crap,8)},{(crap,2)})
(flickr,{(amr,flickr,10)},{(flickr,9)})
(myblog,{(amr,myblog,10)},{(myblog,7)})
grunt>
```

CROSS:

The CROSS operator is used to compute the cross product (Cartesian product) of two or more relations.

Applying cross product on loading1 and loading2.

```
grunt> cross_command = cross loading1,loading2;
grunt> dump cross_command;
```

CROSS Result:

```
(fred,cnn,12,crap,2)
(amr,flickr,10,crap,2)
(fred,cnn,12,myblog,7)
(amr,flickr,10,myblog,7)
(amr,flickr,10,cnn,9)
(amr,flickr,10,flickr,9)
(fred,cnn,12,cnn,9)
(fred,cnn,12,flickr,9)
(amr,myblog,10,crap,2)
(amr,myblog,10,myblog,7)
(amr,myblog,10,cnn,9)
(amr,myblog,10,flickr,9)
(amr,cnn,8,crap,2)
(amr,crap,8,crap,2)
(amr,cnn,8,myblog,7)
(amr,crap,8,myblog,7)
(amr,crap,8,cnn,9)
(amr,crap,8,flickr,9)
(amr,cnn,8,cnn,9)
(amr,cnn,8,flickr,9)
grunt>
```

LIMIT:

LIMIT operator is used to limit the number of output tuples. If the specified number of output tuples is equal to or exceeds the number of tuples in the relation, the output will include all tuples in the relation.

```
grunt> limit_command = limit loading1 3;
grunt> dump limit_command;
```

LIMIT Result:

```
(amr,cnn,8)
(amr,crap,8)
(amr,myblog,10)
grunt>
```

SPLIT:

SPLIT operator is used to partition the contents of a relation into two or more relations based on some expression. Depending on the conditions stated in the expression.

Split the loading2 into two relations x and y. x relation created by loading2 contain the fields that the rating is greater than 8 and y relation contain fields that rating is less than or equal to 8.

```
grunt> split loading2 into x if rating>8, y if rating<=8;
grunt> dump x;
```

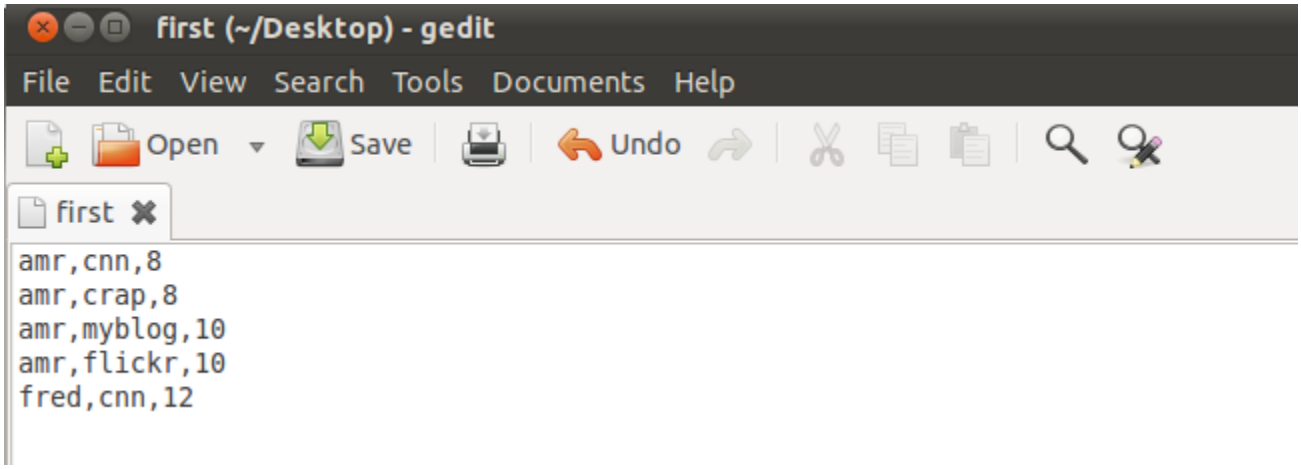
```
(cnn,9)
(flickr,9)
grunt>
```

```
(myblog,7)
(crap,2)
grunt>
```

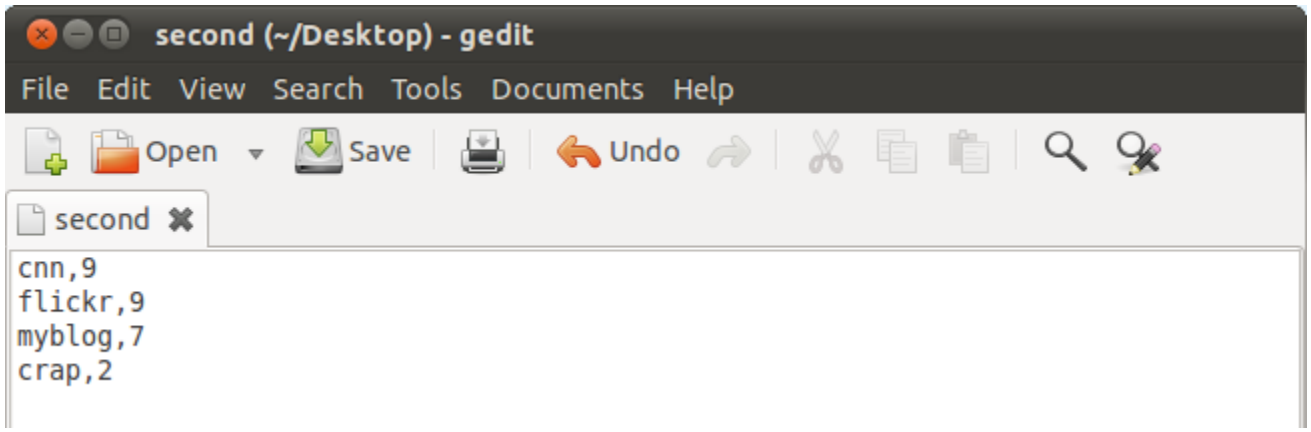
operators-in-apache-pig-diagnostic-operators

This is the 2nd post in series of Apache Pig Operators. This post is about the '**Diagnostic Operators**' in Apache Pig. You can also refer to our previous post on Relational Operators for more information.

Let's create two files to run the commands. We have two files with name 'first' and 'second.' The first file contain three fields: user, url & id.



The second file contain two fields: url & rating. These two files are CSV files.



Diagnostic Operators:

DUMP:

The DUMP operator is used to run Pig Latin statements and display the results on the screen. *In this example,* the operator prints 'loading1' on to the screen.

```
grunt> dump loading1;
2013-11-15 22:55:36,601 [main] INFO  org.apache.pig.tools.pigstats.ScriptState - Pig features u
2013-11-15 22:55:36,601 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEn
 set to true. New logical plan will be used.
```

DUMP Result:

```
(cnn,9)
(flickr,9)
(myblog,7)
(crap,2)
grunt>
```

DESCRIBE:

Use the DESCRIBE operator to review the schema of a particular relation. The DESCRIBE operator is best used for debugging a script.

```
grunt> describe loading1;
loading1: {user: chararray,url: chararray,id: int}
grunt>
```

ILLUSTRATE:

ILLUSTRATE operator is used to review how data is transformed through a sequence of Pig Latin statements. ILLUSTRATE command is your best friend when it comes to debugging a script. This command alone might be a good reason for choosing Pig over something else.

```
grunt> illustrate loading1;
2013-11-16 00:37:16,037 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEn
e system at: hdfs://localhost:8020
2013-11-16 00:37:16,037 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEn
 job tracker at: localhost:8021
2013-11-16 00:37:16,093 [main] INFO  org.apache.hadoop.mapreduce.lib.input.FileInputFormat - To
2013-11-16 00:37:16,093 [main] INFO  org.apache.pig.backend.hadoop.executionengine.util.MapRedU
ess : 1
-------------------------------------------------------------------
| loading1     | user: bytearray | url: bytearray | id: bytearray |
-------------------------------------------------------------------
|              | amr             | myblog         | 10            |
-------------------------------------------------------------------
-------------------------------------------------------------
| loading1     | user: chararray | url: chararray | id: int |
-------------------------------------------------------------
|              | amr             | myblog         | 10      |
-------------------------------------------------------------

grunt>
```

EXPLAIN:

The EXPLAIN operator prints the logical and physical plane.

```
grunt> explain loading1;
2013-11-16 00:37:59,268 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEn
 set to true. New logical plan will be used.
#-----------------------------------------------
# Logical Plan:
#-----------------------------------------------
fake: Store 1-432 Schema: {user: chararray,url: chararray,id: int} Type: Unknown
|
|---loading1: Load 1-402 Schema: {user: chararray,url: chararray,id: int} Type: bag
```

```
#-----------------------------------------------
# Physical Plan:
#-----------------------------------------------
loading1: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-235
|
|---loading1: New For Each(false,false,false)[bag] - scope-234
    |   |
    |   Cast[chararray] - scope-226
    |   |
    |   |---Project[bytearray][0] - scope-225
    |   |
    |   Cast[chararray] - scope-229
    |   |
    |   |---Project[bytearray][1] - scope-228
    |   |
    |   Cast[int] - scope-232
    |   |
    |   |---Project[bytearray][2] - scope-231
    |
    |---loading1: Load(/first:PigStorage(',')) - scope-224
```

Improvements in Apache Pig 0.12.0

0.12.0 is the current version of Apache Pig available. This release include several new features such as ASSERT operator, IN operator, CASE operator.

Assert Operator:

An Assert operator can be used for data validation. *For example,* the following script will fail if any value is a negative integer:

a = load ‘something’ as (a0: int, a1: int);

assert a by a0 > 0, ‘a can’t be negative for reasons’;

IN Operator:

Previously, Pig had no support for IN operators. To imitate an IN operation, users had to concatenate several OR operators, as shown in below *example*:

a = LOAD ’1.txt’ USING PigStorage (‘,’) AS (i:int);

b = FILTER a BY

(i == 1) OR

(i == 22) OR

(i == 333) OR

(i == 4444) OR

(i == 55555)

Now, this type of expression can be re-written in a more compressed manner using an IN operator:

a = LOAD ’1.txt’ USING PigStorage (‘,’) AS (i:int);

b = FILTER a BY i IN (1, 22, 333, 4444, 55555);

CASE Expression:

Earlier, Pig had no support for a CASE statement. To mimic it, users often use nested bincond operators. Those could become unreadable when there were multiple levels of nesting. Following is an *example* of the type of CASE expression that Pig currently supports:

Case_operator = FOREACH foo GENERATE (

CASE i % 3

WHEN 0 THEN ’3n’

WHEN 1 THEN ’3n+1′

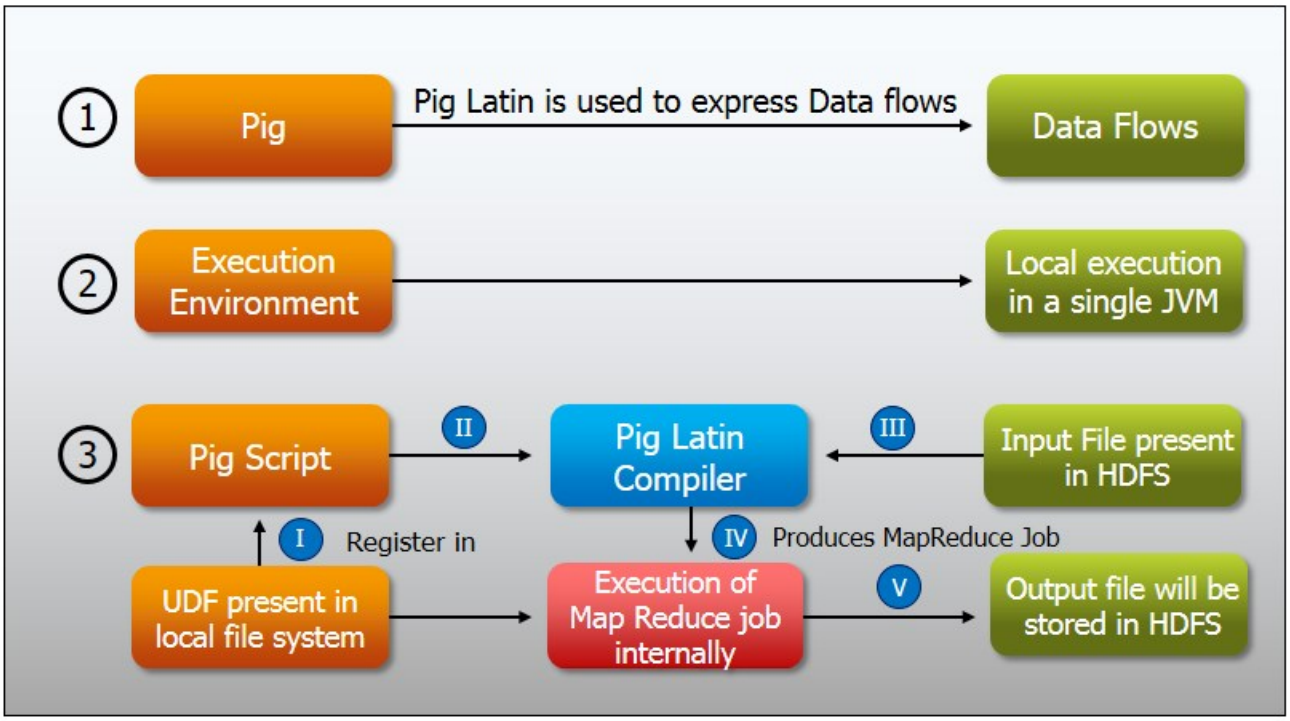
ELSE ’3n+2′

END

);

Got a question for us? Please mention them in the comments section and we will get back to you.

Pig Programming: Apache Pig Script with UDF in HDFS Mode



In the previous blog posts we saw how to start with ***Pig Programming and Scripting***. We have seen the steps to write a **Pig Script in HDFS Mode** and **Pig Script Local Mode** without UDF. In the third part of this series we will review the steps to write a Pig script with ***UDF in HDFS Mode***.

We have explained how to implement Pig UDF by creating built-in functions to explain the functionality of Pig built-in function. For better explanation, we have taken two built-in functions. We have done this with the help of a pig script.

Here, we have taken one example and we have used both the UDF (user defined functions) i.e. making a string in upper case and taking a value & raising its power.

The dataset is depicted below which we are going to use in this example:

Name	Marks	Multiplier
sean	25	3
katy	15	4
zac	16	2
adam	22	5

Our aim is to make 1st column letter in upper case and raising the power of the 2nd column with the value of 3rd column.

Let's start with writing the java code for each UDF. Also we have to configure 4 JARs in our java project to avoid the compilation errors.
First, we will create java programs, both are given below:

Upper.java

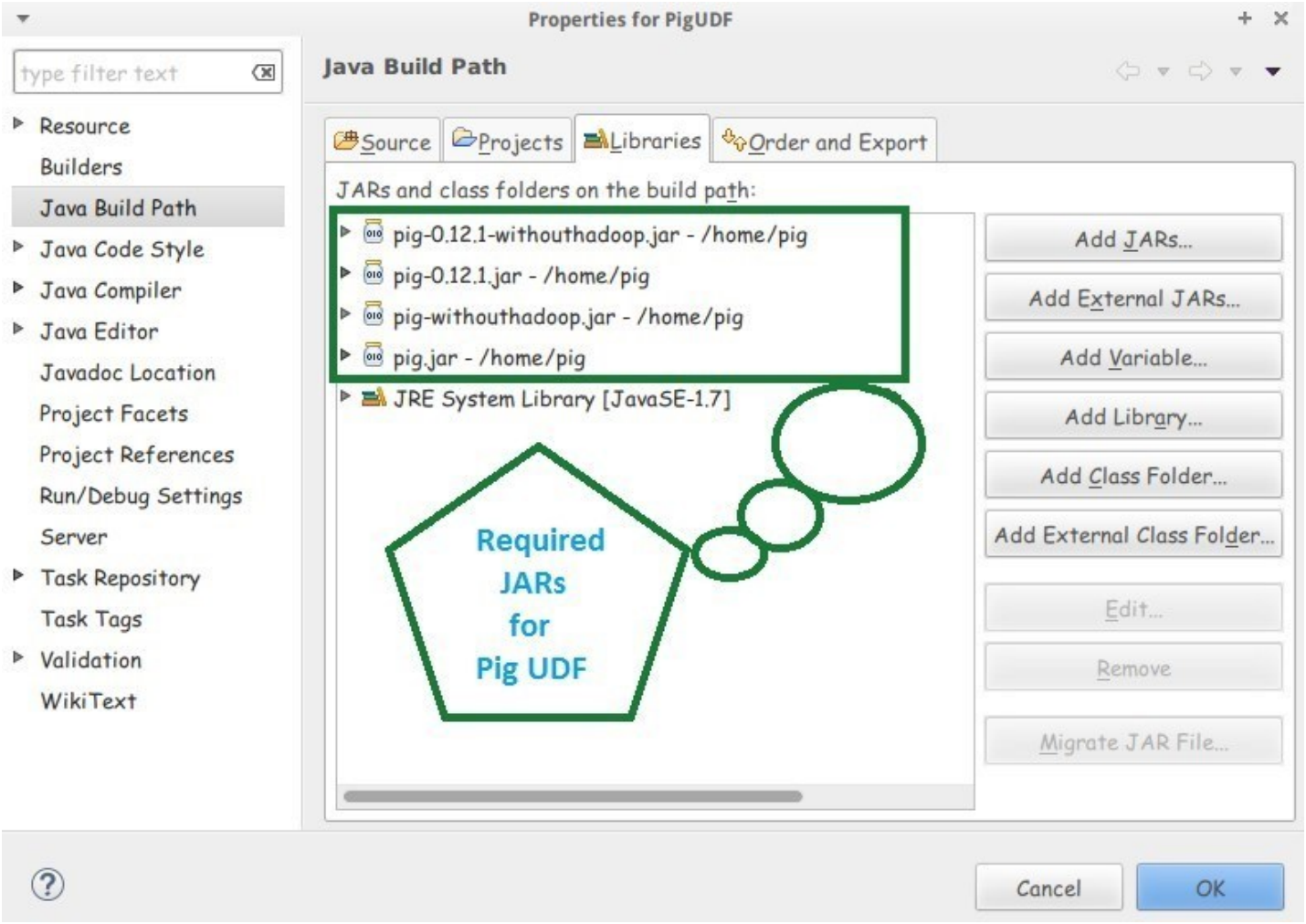
```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

@SuppressWarnings("deprecation")
public class Upper extends EvalFunc<String> {

public String exec(Tuple input) throws IOException {

if (input == null || input.size() == 0)

return null;

try {

String str = (String) input.get(0);

str=str.toUpperCase();

return str;

}
catch (Exception e) {

throw WrappedIOException.wrap("Caught exception processing input row ", e);

}
}
}
```

Power.java

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

public class Pow extends EvalFunc<Long> {
```

```
public Long exec(Tuple input) throws IOException {
try {

int base = (Integer)input.get(0);
int exponent = (Integer)input.get(1);
long result = 1;

/* Probably not the most efficient method...*/
for (int i = 0; i < exponent; i++) {
long preresult = result;
result *= base;
if (preresult > result) {
// We overflowed. Give a warning, but do not throw an
// exception.
warn("Overflow!", PigWarning.TOO_LARGE_FOR_INT);
// Returning null will indicate to Pig that we failed but
// we want to continue execution.
return null;
}
}
return result;
} catch (Exception e) {
// Throwing an exception will cause the task to fail.
throw new IOException("Something bad happened!", e);
}
}
}
```

To remove compilation errors, we have to configure **4 JARs** in our java project.

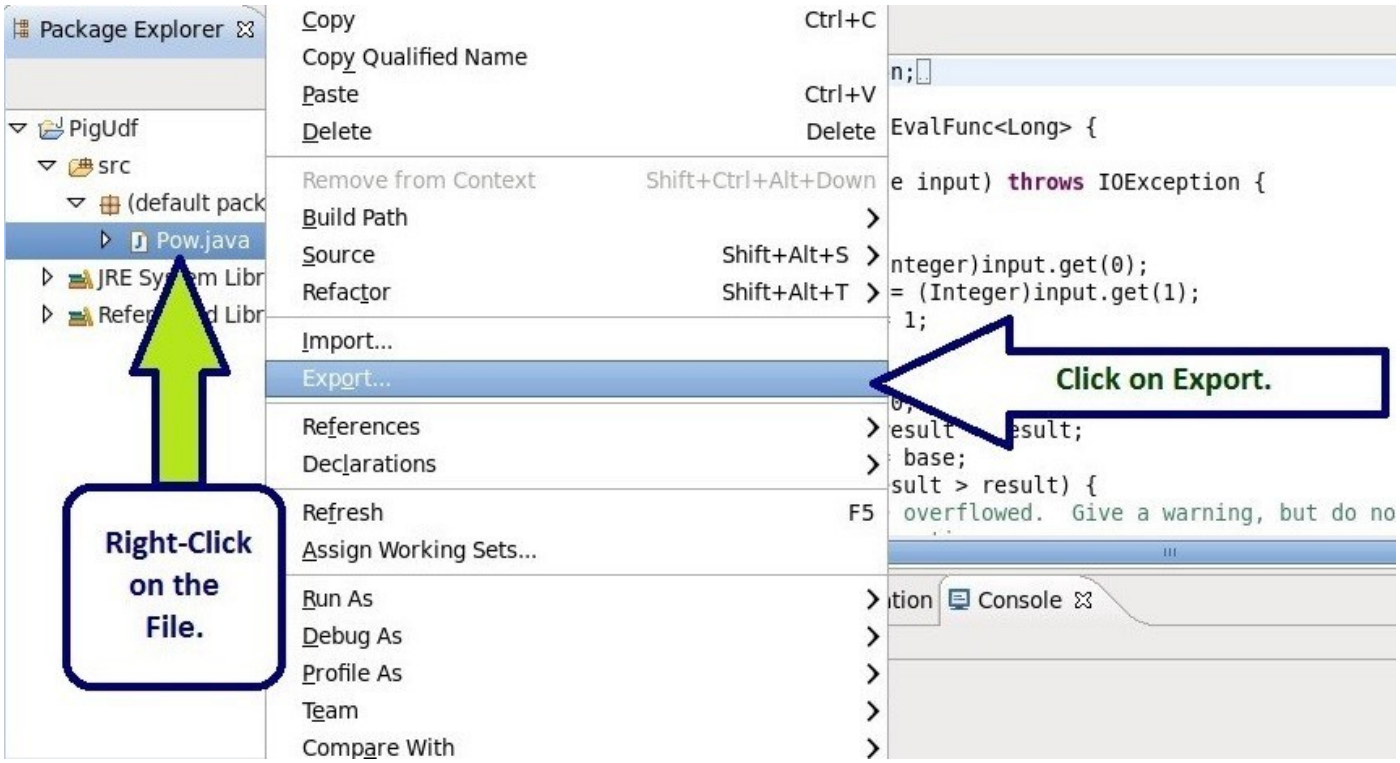


Click on the Download button to download the JARs

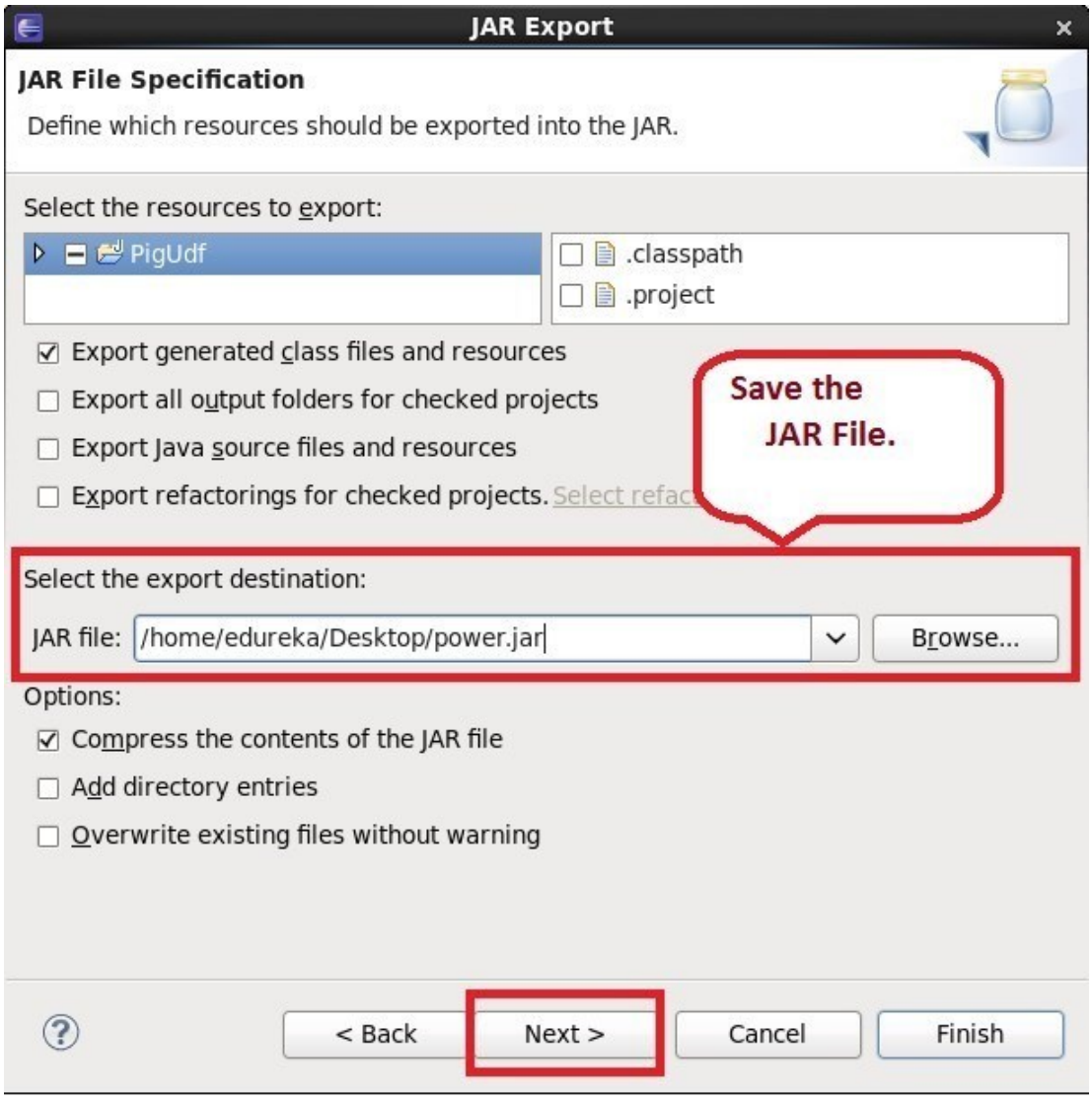
[buttonleads form_title="Download Code" redirect_url=https://edureka.wistia.com/medias/wtboe1hmkr/download?media_file_id=76900193 course_id=32 button_text="Download JARs"]

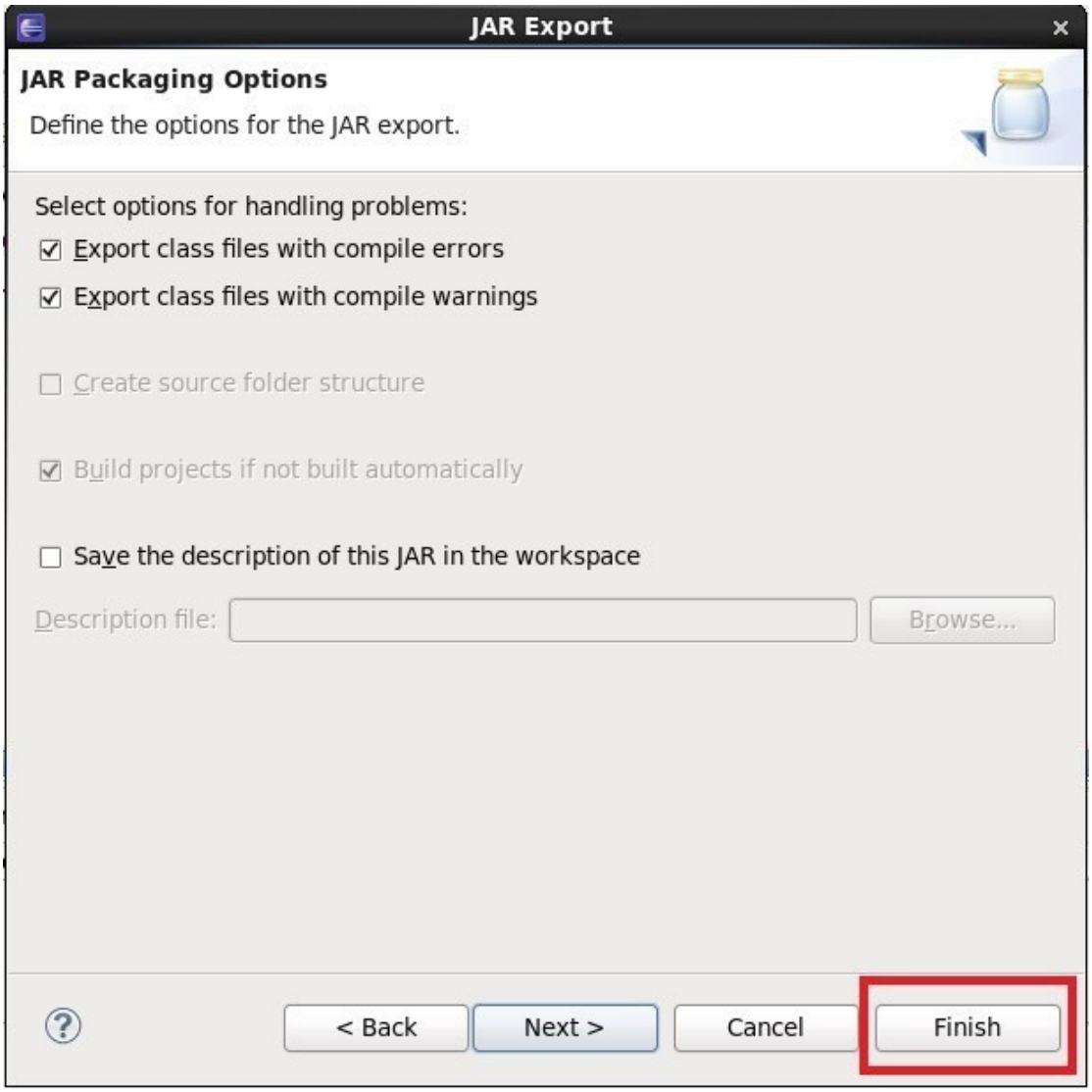
Now, we export JAR files for both the java codes. Please check the below steps for JAR creation.

Here, we have shown for one program, proceed in the same way in the next program as well.


Package Explorer
PigUdf
src
(default pack
Pow.java
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Shift+Ctrl+Alt+Down
Build Path
Source Shift+Alt+S
Refactor Shift+Alt+T
Import...
Export...
References
Declarations
Refresh F5
Assign Working Sets...
Run As
Debug As
Profile As
Team
Compare With
Right-Click on the File.
Click on Export.
Console


Export
Select
Export resources into a JAR file on the local file system.
Select an export destination:
type filter text
Select JAR file
General
C/C++
Java
JAR file
Javadoc
Runnable JAR file
Plug-in Development
Remote Systems
RPM
Run/Debug
Tasks
< Back
Next >
Cancel


JAR Export
JAR File Specification
Define which resources should be exported into the JAR.
Select the resources to export:
PigUdf
.classpath
.project
Export generated class files and resources
Export all output folders for checked projects
Export Java source files and resources
Export refactorings for checked projects.
Save the JAR File.
Select the export destination:
JAR file: /home/edureka/Desktop/power.jar
Browse...
Options:
Compress the contents of the JAR file
Add directory entries
Overwrite existing files without warning
< Back
Next >
Cancel
Finish



After creating the JARs and text files, we have moved all the data to HDFS cluster, which is depicted by the following images:

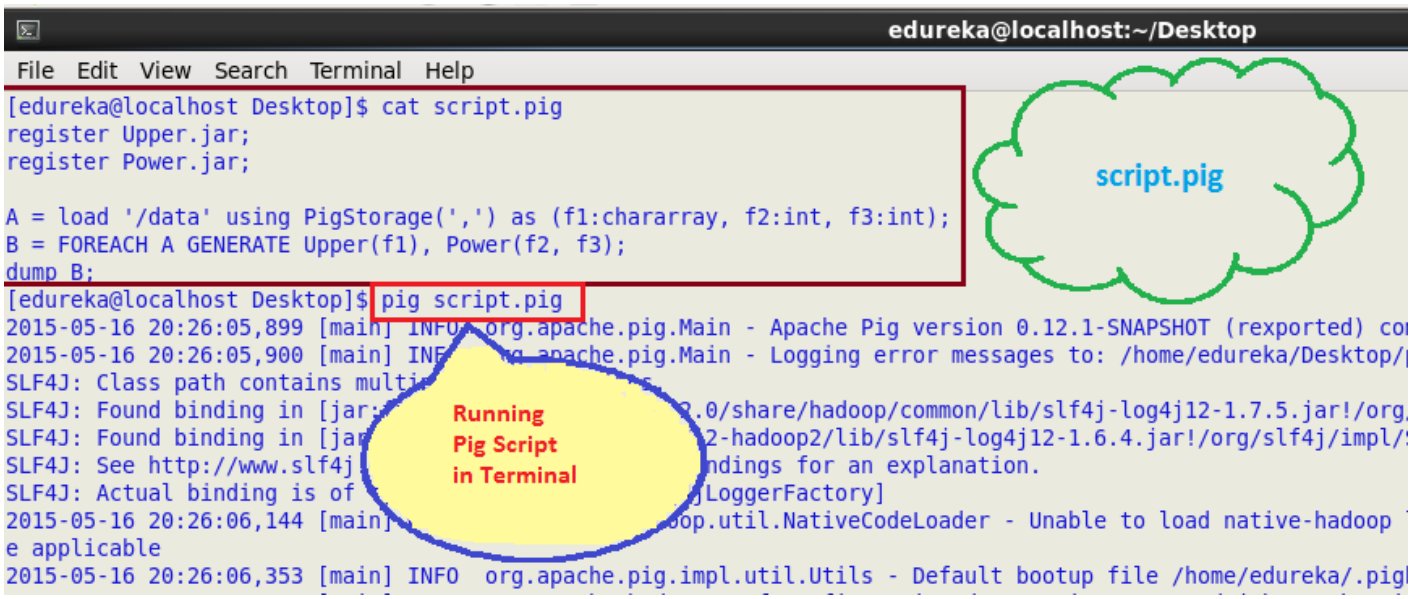
```
[edureka@localhost Desktop]$ hadoop dfs -ls /
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

15/05/16 19:58:11 WARN util.NativeCodeLoader: Unable to load native-hadoop libr
Found 10 items
-rw-r--r--   1 edureka supergroup     200818 2015-05-13 16:22 /CBTTickets.csv
-rw-r--r--   1 edureka supergroup       1160 2015-05-16 19:58 /Power.jar
-rw-r--r--   1 edureka supergroup        951 2015-05-16 19:57 /Upper.jar
-rw-r--r--   1 edureka supergroup         39 2015-05-16 19:51 /data
drwxr-xr-x   - edureka supergroup          0 2015-05-08 15:28 /reviewoutput
```

In our dataset, fields are comma (,) separated.

```
[edureka@localhost Desktop]$ hadoop dfs -cat /data
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

15/05/16 20:02:44 WARN util.NativeCodeLoader: Unable to load native-had
sean,25,3
katy,15,4
zac,16,2
adam,22,5
[edureka@localhost Desktop]$
```

After moving the file, we have created script with .pig extension and put all the commands in that script file.

Now in terminal, type PIG followed by the name of the script file which is shown in the following image:



Here, this is the output for running the pig script.

```
(SEAN,15625)
(KATY,50625)
(ZAC,256)
(ADAM,5153632)
```

Got a question for us? Please mention them in the comments section and we will get back to you.

apache-pig-udf-part-1-eval-aggregate-filter-functions

Apache Pig provides extensive support for user defined functions (UDFs) as a way to specify custom processing. Pig UDFs can currently be executed in three languages: *Java, Python, JavaScript and Ruby.* The most extensive support is provided for Java functions.

Java UDFs can be invoked through multiple ways. The simplest UDF can just extend EvalFunc, which requires only the exec function to be implemented. Every Eval UDF must implement this. Additionally, if a function is algebraic, it can implement Algebraic interface to significantly improve query performance.

Importance of UDFs in Pig:

Pig allows users to combine existing operators with their own or others' code via UDFs. The advantage of Pig is its ability to let users combine its operators with their own or others' code via UDFs. Up through version 0.7, all UDFs must be written in Java and are implemented as Java classes. This makes it easier to add new UDFs to Pig by writing a Java class and informing Pig about the JAR file.

Pig itself comes with some UDFs. Prior to version 0.8, it was a very limited set with only the standard SQL aggregate functions and a few others. In 0.8, a large number of standard string-processing, math, and complex-type UDFs were added.

What is a Piggybank?

Piggybank is a collection of user-contributed UDFs that is released along with Pig. Piggybank UDFs are not included in the Pig JAR, so you have to register them manually in your script. You can also write your own UDFs or use those written by other users.

Eval Functions

The UDF class extends the EvalFunc class which is the base for all Eval functions. All Evaluation functions extend the Java class 'org.apache.pig.EvalFunc. 'It is parameterized with the return type of the UDF which is a Java String in this case. The core method in this class is 'exec.' The 1st line of the code indicates that the function is a part of myudfs package.

It takes one record and returns one result, which will be invoked for every record that passes through the execution pipeline. It takes a tuple , which contains all of the fields the script passes to your UDF as a input. It then returns the type by which you have parameterized EvalFunc.

This function is invoked on every input tuple. The input into the function is a tuple with input parameters in the order they are passed to the function in the Pig script. In the example shown below, the function takes string as input. The following function converts the string from lowercase to uppercase. Now that the function is implemented, it needs to be compiled and included in a JAR.

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

```
public class UPPER extends EvalFunc<String>
{
public String exec(Tuple input) throws IOException {
if (input == null || input.size() == 0)
return null;
try{
String str = (String)input.get(0);
return str.toUpperCase();
}catch(Exception e){
throw new IOException("Caught exception processing input row ", e);
}
}
}
```

Aggregate Functions:

Aggregate functions are another common type of Eval function. Aggregate functions are usually applied to grouped data. The Aggregate function takes a bag and returns a scalar value. An interesting and valuable feature of many Aggregate functions is that they can be computed incrementally in a distributed manner. In Hadoop world, this means that the partial computations can be done by the Map and Combiner and the final result can be computed by the Reducer.

It is very important to make sure that Aggregate functions that are algebraic are implemented as such. Examples of this type include the built-in COUNT, MIN, MAX and AVERAGE.

COUNT is an example of an algebraic function where we can count the number of elements in a subset of the data and then sum the counts to produce a final output. Let's look at the implementation of the COUNT function:

```
public class COUNT extends EvalFunc<Long> implements Algebraic{
public Long exec(Tuple input) throws IOException {return count(input);}
public String getInitial() {return Initial.class.getName();}
public String getIntermed() {return Intermed.class.getName();}
public String getFinal() {return Final.class.getName();}
static public class Initial extends EvalFunc<Tuple> {
public Tuple exec(Tuple input) throws IOException {return
TupleFactory.getInstance().newTuple(count(input));}
}
static public class Intermed extends EvalFunc<Tuple> {
public Tuple exec(Tuple input) throws IOException {return
TupleFactory.getInstance().newTuple(sum(input));}
}
static public class Final extends EvalFunc<Long> {
public Tuple exec(Tuple input) throws IOException {return sum(input);}
}
static protected Long count(Tuple input) throws ExecException {
```

```
Object values = input.get(0);
if (values instanceof DataBag) return ((DataBag)values).size();
else if (values instanceof Map) return new Long(((Map)values).size());
}
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;
}
}
```

COUNT implements Algebraic interface which looks like this:

```
public interface Algebraic{
public String getInitial();
public String getIntermed();
public String getFinal();
}
```

For a function to be algebraic, it needs to implement Algebraic interface that consist of definition of three classes derived from EvalFunc. The contract is that the execfunction of the Initial class is called once and is passed to the original input tuple. Its output is a tuple that contains partial results.
The exec function of the Intermed class can be called zero or more times and takes as its input a tuple that contains partial results produced by the Initial class or by prior invocations of the Intermed class and produces a tuple with another partial result. Finally, the exec function of the Final class is called and gives the final result as a scalar type.

Filter Functions:

Filter functions are Eval functions that returns a Boolean value. It can be used anywhere a Boolean expression is appropriate, including the FILTER operator or Bincond expression. Apache Pig does not support Boolean totally, so Filter functions cannot appear in statements such as 'Foreach', where the results are output to another operator. However, Filter functions can be used in filter statements.

The example below implements IsEmpty function:

```
import java.io.IOException;
import java.util.Map;
import org.apache.pig.FilterFunc;
import org.apache.pig.PigException;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataBag;
```

```
import org.apache.pig.data.Tuple;
import org.apache.pig.data.DataType;
/**
* Determine whether a bag or map is empty.
*/
public class IsEmpty extends FilterFunc {
@Override
public Boolean exec(Tuple input) throws IOException {
try {
Object values = input.get(0);
if (values instanceof DataBag)
return ((DataBag)values).size() == 0;
else if (values instanceof Map)
return ((Map)values).size() == 0;
else {
int errCode = 2102;
String msg = "Cannot test a " +
DataType.findTypeName(values) + " for emptiness.";
throw new ExecException(msg, errCode, PigException.BUG);
}
} catch (ExecException ee) {
throw ee;
}
}
}
```

apache-pig-udf-part-2-load-functions/

Today's post is about the Load functions in Apache Pig. This is the sequel to the first post which covered UDF functions like Eval, Filter and Aggregate. Please refter to them for more information on other functions of Pig UDF.

Pig's load function is built on top of a Hadoop's InputFormat, the class that Hadoop uses to read data. InputFormat has two purposes: It determines how input will be fragmented between map tasks and provides a RecordReader that results in key-value pairs as input to those map tasks. The base class for the load function is LoadFunc.

Load Function – Classification:

LoadFunc abstract class has three main methods for loading data and in most use cases it would suffice to extend it. There are three other optional interfaces which can be implemented to achieve extended functionality:

- **LoadMetadata:**

LoadMetadata has methods to deal with metadata. Most execution of loaders don't need to implement this unless they interact with a metadata system. The getSchema() method in this interface offers a way for the loader implementations to communicate about the schema of the data back to Pig. If a loader implementation returns data comprised of fields of real types, it should provide the schema describing the data returned through the getSchema() method. The other methods deal with other types of metadata like partition keys and statistics. Implementations can return null return values for these methods if they are not valid for the other implementation.

- **LoadPushDown:**

LoadPushDown has different methods to push operations from Pig runtime into loader implementations. Currently, only the pushProjection() method is called by Pig to communicate to the loader, the exact fields that are required in the Pig script. The loader implementation can choose to abide or not abide the request. If the loader implementation decides to abide the request, it should implement LoadPushDown to improve query performance.

- **pushProjection():**

This method informs LoadFunc, which fields are required in the Pig script. Thus enabling LoadFunc to enhance performance by loading only the fields that are required. pushProjection() takes a 'requiredFieldList.' 'requiredFieldList' is read only and cannot be changed by LoadFunc. 'requiredFieldList' includes a list of 'requiredField', where each 'requiredField' indicates a field required by the Pig script and is comprised of index, alias, type and subFields. Pig uses the column index requiredField.index to communicate with the LoadFunc about the fields required by the Pig script. If the required field is a map, Pig will pass 'requiredField.subFields' which contains a list of keys required by Pig scripts for the map.

- **LoadCaster:**

LoadCaster has techniques to convert byte arrays in to specific types. A loader implementation should implement this when implicit or explicit casts from DataByteArray fields to other types needs to be supported.

The LoadFunc abstract class is the main class to extend for implementing a loader. The methods which is required to be overridden are explained below:

- **getInputFormat():**

  This method is called by Pig to get the InputFormat utilized by the loader. The methods in the InputFormat are called by Pig in the same fashion as Hadoop in a MapReduce Java program. If the InputFormat is a Hadoop packaged one, the implementation should use the new API based one, under org.apache.hadoop.mapreduce. If it is a custom InputFormat, it is better to be implemented using the new API in org.apache.hadoop.mapreduce.

- **setLocation():**

This method is called by Pig to communicate the load location to the loader. The loader needs to use this method to communicate the same information to the core InputFormat. This method is called multiple times by pig.

- **prepareToRead():**

  In this method, the RecordReader related to the InputFormat provided by the LoadFunc is passed to the LoadFunc. The RecordReader can now be used by the implementation in getNext() to return a tuple representing a record of data back to Pig.

- **getNext():**

  The meaning of getNext() has not changed and is called by Pig runtime to acquire the next tuple in the data. In this method, the implementation should use the underlying RecordReader and construct the tuple to return.

Default Implementations in LoadFunc:

Take note that the default implementations in LoadFunc should be overridden only when needed.

- **setUdfContextSignature():**

  This method will be called by Pig, both in the front end and back end to pass a unique signature to the Loader. The signature can be utilized to store any information in to the UDFContext which the Loader needs to store between various method invocations in the front end and back end. A use case is to store RequiredFieldList passed to it in LoadPushDown.pushProjection(RequiredFieldList) for use in the back end before returning tuples in getNext(). The default implementation in LoadFunc has an empty body. This method will be called before other methods.

- **relativeToAbsolutePath():**

  Pig runtime will call this method to permit the Loader to convert a relative load location to an absolute location. The default implementation provided in LoadFunc handles this for FileSystem locations. If the load source is something else, loader implementation may choose to override this.

The loader implementation in the example is a loader for text data with line delimiter as '\n' and '\t' as default field delimiter similar to current PigStorage loader in Pig. The implementation uses an existing Hadoop supported Inputformat – TextInputFormat – as the underlying InputFormat.

```
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() {
}
/**
* Constructs a Pig loader that uses specified character as a field delimiter.
*
* @param delimiter
*          the single byte character that is used to separate fields.
*          ("\t" is the default.)
*/
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 delimeter must be a single character");
}
}
@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;
}
}
// pick up the last field
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) {
// NULL value
mProtoTuple.add(null);
} else {
mProtoTuple.add(new DataByteArray(buf, start, end));
}
}
@Override
public InputFormat getInputFormat() {
return new TextInputFormat();
}
@Override
public void prepareToRead(RecordReader reader, PigSplit split) {
in = reader;
}
@Override
public void setLocation(String location, Job job)
throws IOException {
FileInputFormat.setInputPaths(job, location);
}
}
```

Got a question for us? Please mention it in the comments section and we will get back to you.

Related Posts:

apache-pig-udf-store-functions part3

StoreFunc abstract class has the main methods for storing data and for most use cases it should suffice to extend it. There is an optional interface which can be implemented to achieve extended functionality:

StoreMetadata

This interface has methods to interact with metadata systems to store schema and statistics. This interface is optional and should be implemented only if metadata needs to stored.

The methods which need to be overridden in StoreFunc are explained below:

- **getOutputFormat():**

  This method will be called by Pig to get the OutputFormat used by the Storer. The methods in the OutputFormat will be called by Pig in the same manner and in the same context as by Hadoop in a map-reduce Java program. If the OutputFormat is a Hadoop packaged one, the implementation should use the new API based one under org.apache.hadoop.mapreduce. If it is a custom OutputFormat, it should be implemented using the new API under org.apache.hadoop.mapreduce. The checkOutputSpecs() method of the OutputFormat will be called by pig to check the output location up-front. This method will also be called as part of the Hadoop call sequence when the job is launched. So implementations should ensure that this method can be called multiple times without inconsistent side effects.

- **setStoreLocation():**

  This method is called by Pig to communicate the store location to the storer. The storer should use this method to communicate the same information to the underlying OutputFormat. This method is called multiple times by Pig. Implementations should take note that this method is called multiple times and should ensure there are no inconsistent side effects due to the multiple calls.

- **prepareToWrite():**

  In the new API, writing of the data is through the OutputFormat provided by the StoreFunc. In prepareToWrite() the RecordWriter associated with the OutputFormat provided by the StoreFunc is passed to the StoreFunc. The RecordWriter can then be used by the implementation in putNext() to write a tuple representing a record of data in a manner expected by the RecordWriter.

- **putNext():**

The meaning of putNext() has not changed and is called by Pig runtime to write the next tuple of data – in the new API, this is the method wherein the implementation will use the underlying RecordWriter to write the Tuple out.

Default Implementations in StoreFunc:

- **setStoreFuncUDFContextSignature():**

  This method will be called by Pig both in the front end and back end to pass a unique signature to the Storer. The signature can be used to store any information in to the UDFContext which the Storer needs to store between various method invocations in the front end and back end. The default implementation in StoreFunc has an empty body. This method will be called before any other methods.

- **relToAbsPathForStoreLocation():**

  Pig runtime will call this method to allow the Storer to convert a relative store location to an absolute location. An implementation is provided in StoreFunc which handles this for FileSystem based locations.

- **checkSchema():**

  A Store function should implement this function to check that a given schema describing the data to be written is acceptable to it. The default implementation in StoreFunc has an empty body. This method will be called before any calls to setStoreLocation().

Example Implementation:

The storer implementation in the example, is a storer for text data with line delimiter as '\n' and '\t' as default field delimiter (which can be overridden by passing a different field delimiter in the constructor) – this is similar to current PigStorage storer in Pig. The implementation uses an existing Hadoop supported OutputFormat – TextOutputFormat as the underlying OutputFormat.

```
public class SimpleTextStorer extends StoreFunc {
protected RecordWriter writer = null;
private byte fieldDel = '\t';
private static final int BUFFER_SIZE = 1024;
private static final String UTF8 = "UTF-8";
public PigStorage() {
}
public PigStorage(String delimiter) {
this();
if (delimiter.length() == 1) {
this.fieldDel = (byte)delimiter.charAt(0);
} else if (delimiter.length() > 1delimiter.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 delimeter must be a single character");
}
}
ByteArrayOutputStream mOut = new ByteArrayOutputStream(BUFFER_SIZE);
@Override
public void putNext(Tuple f) throws IOException {
int sz = f.size();
for (int i = 0; i < sz; i++) {
Object field;
try {
field = f.get(i);
} catch (ExecException ee) {
throw ee;
}
putField(field);
if (i != sz - 1) {
mOut.write(fieldDel);
}
}
Text text = new Text(mOut.toByteArray());
try {
writer.write(null, text);
mOut.reset();
} catch (InterruptedException e) {
throw new IOException(e);
}
}
@SuppressWarnings("unchecked")
private void putField(Object field) throws IOException {
//string constants for each delimiter
String tupleBeginDelim = "(";
```

```
String tupleEndDelim = ")";
String bagBeginDelim = "{";
String bagEndDelim = "}";
String mapBeginDelim = "[";
String mapEndDelim = "]";
String fieldDelim = ",";
String mapKeyValueDelim = "#";
switch (DataType.findType(field)) {
case DataType.NULL:
break; // just leave it empty
case DataType.BOOLEAN:
mOut.write(((Boolean)field).toString().getBytes());
break;
case DataType.INTEGER:
mOut.write(((Integer)field).toString().getBytes());
break;
case DataType.LONG:
mOut.write(((Long)field).toString().getBytes());
break;
case DataType.FLOAT:
mOut.write(((Float)field).toString().getBytes());
break;
case DataType.DOUBLE:
mOut.write(((Double)field).toString().getBytes());
break;
case DataType.BYTEARRAY: {
byte[] b = ((DataByteArray)field).get();
mOut.write(b, 0, b.length);
break;
}
case DataType.CHARARRAY:
// oddly enough, writeBytes writes a string
mOut.write(((String)field).getBytes(UTF8));
break;
case DataType.MAP:
boolean mapHasNext = false;
Map<String, Object> m = (Map<String, Object>)field;
mOut.write(mapBeginDelim.getBytes(UTF8));
for(Map.Entry<String, Object> e: m.entrySet()) {
if(mapHasNext) {
mOut.write(fieldDelim.getBytes(UTF8));
} else {
mapHasNext = true;
}
putField(e.getKey());
mOut.write(mapKeyValueDelim.getBytes(UTF8));
```

```
putField(e.getValue());
}
mOut.write(mapEndDelim.getBytes(UTF8));
break;
case DataType.TUPLE:
boolean tupleHasNext = false;
Tuple t = (Tuple)field;
mOut.write(tupleBeginDelim.getBytes(UTF8));
for(int i = 0; i < t.size(); ++i) {
if(tupleHasNext) {
mOut.write(fieldDelim.getBytes(UTF8));
} else {
tupleHasNext = true;
}
try {
putField(t.get(i));
} catch (ExecException ee) {
throw ee;
}
}
mOut.write(tupleEndDelim.getBytes(UTF8));
break;
case DataType.BAG:
boolean bagHasNext = false;
mOut.write(bagBeginDelim.getBytes(UTF8));
Iterator<Tuple> tupleIter = ((DataBag)field).iterator();
while(tupleIter.hasNext()) {
if(bagHasNext) {
mOut.write(fieldDelim.getBytes(UTF8));
} else {
bagHasNext = true;
}
putField((Object)tupleIter.next());
}
mOut.write(bagEndDelim.getBytes(UTF8));
break;
default: {
int errCode = 2108;
String msg = "Could not determine data type of field: " + field;
throw new ExecException(msg, errCode, PigException.BUG);
}
}
}
@Override
public OutputFormat getOutputFormat() {
return new TextOutputFormat<WritableComparable, Text>();
```

```
}
@Override
public void prepareToWrite(RecordWriter writer) {
this.writer = writer;
}
@Override
public void setStoreLocation(String location, Job job) throws IOException {
job.getConfiguration().set("mapred.textoutputformat.separator", "");
FileOutputFormat.setOutputPath(job, new Path(location));
if (location.endsWith(".bz2")) {
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job,  BZip2Codec.class);
}  else if (location.endsWith(".gz")) {
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
}
}
}
```

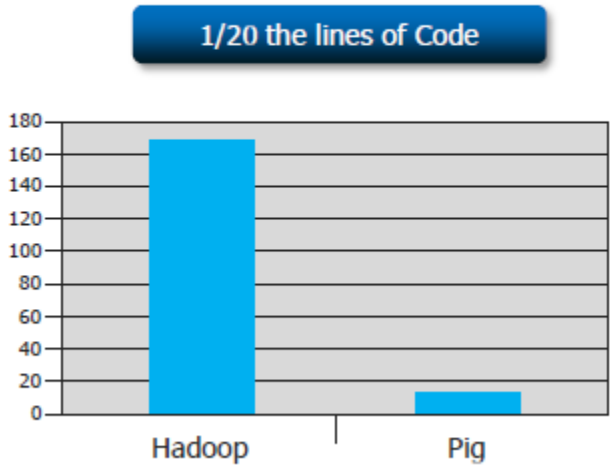
What is Pig and Pig Latin?

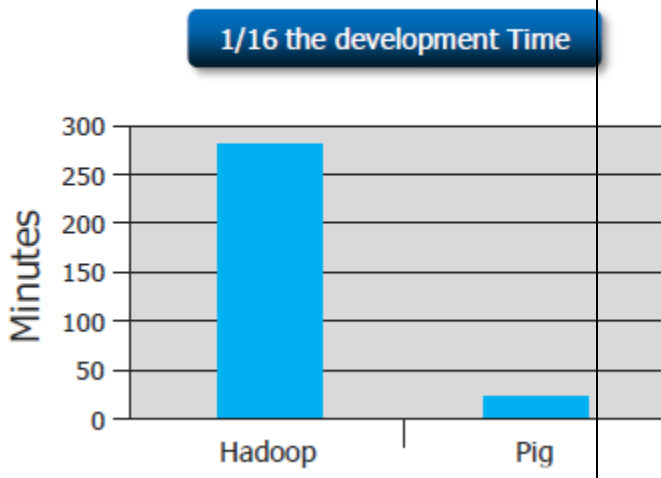
Pig is an open-source high level data flow system. It provides a simple language called Pig Latin, for queries and data manipulation, which are then compiled in to MapReduce jobs that run on Hadoop.

Pig is important as companies like Yahoo, Google and Microsoft are collecting huge amounts of data sets in the form of click streams, search logs and web crawls. Pig is also used in some form of ad-hoc processing and analysis of all the information.

Why Do you Need Pig?

- It's easy to learn, especially if you're familiar with SQL.
- Pig's multi-query approach reduces the number of times data is scanned. This means 1/20th the lines of code and 1/16th the development time when compared to writing raw MapReduce.





- Performance of Pig is in par with raw MapReduce
- Pig provides data operations like filters, joins, ordering, etc. and nested data types like tuples, bags, and maps, that are missing from MapReduce.
- Pig Latin is easy to write and read.

Why was Pig Created?

Pig was originally developed by Yahoo in 2006, for researchers to have an ad-hoc way of creating and executing MapReduce jobs on very large data sets. It was created to reduce the development time through its multi-query approach. Pig is also created for professionals from non-Java background, to make their job easier.

Where Should Pig be Used?

Pig can be used under following scenarios:

- When data loads are time sensitive.
- When processing various data sources.
- When analytical insights are required through sampling.

Where Not to Use Pig?

- In places where the data is completely unstructured, like video, audio and readable text.
- In places where time constraints exist, as Pig is slower than MapReduce jobs.
- In places where more power is required to optimize the codes.

Applications of Apache Pig:

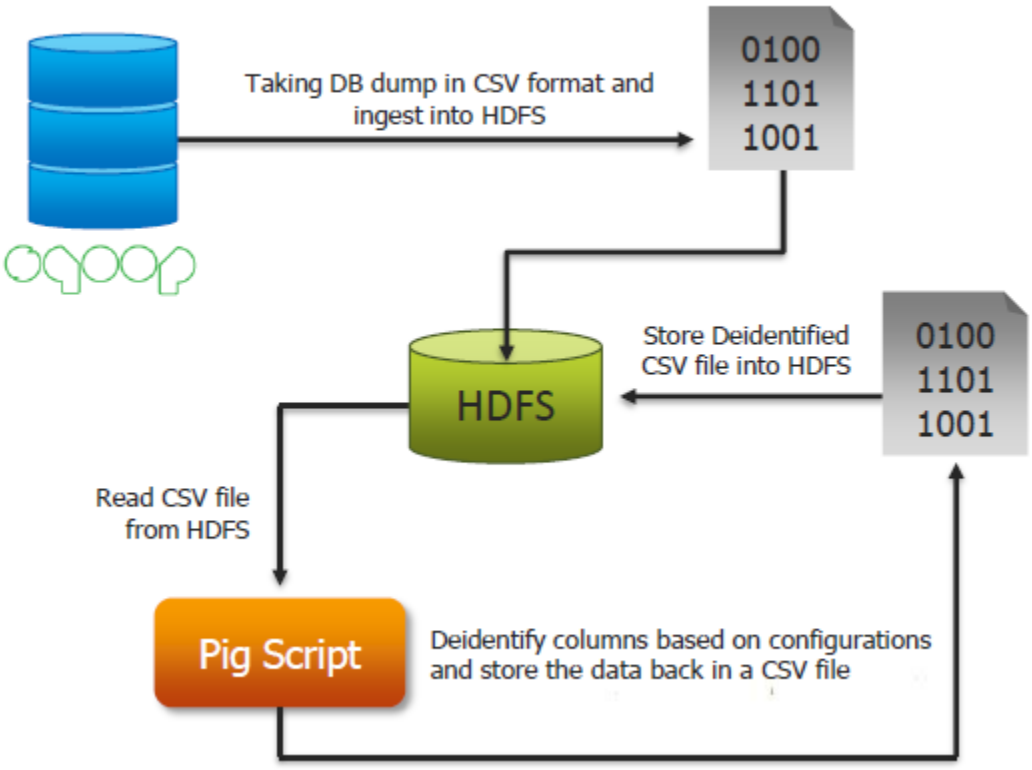
- Processing of web logs.
- Data processing for search platforms.
- Support for Ad-hoc queries across large data sets.
- Quick prototyping of algorithms for processing large data sets.

How Yahoo! Uses Pig:

Yahoo uses Pig for the following purpose:

- **In Pipelines** – To bring logs from its web servers, where these logs undergo a cleaning step to remove bots, company interval views and clicks.
- **In Research** – To quickly write a script to test a theory. Pig Integration makes it easy for the researchers to take a Perl or Python script and run it against a huge data set.

Use Case of Pig in Healthcare Domain:



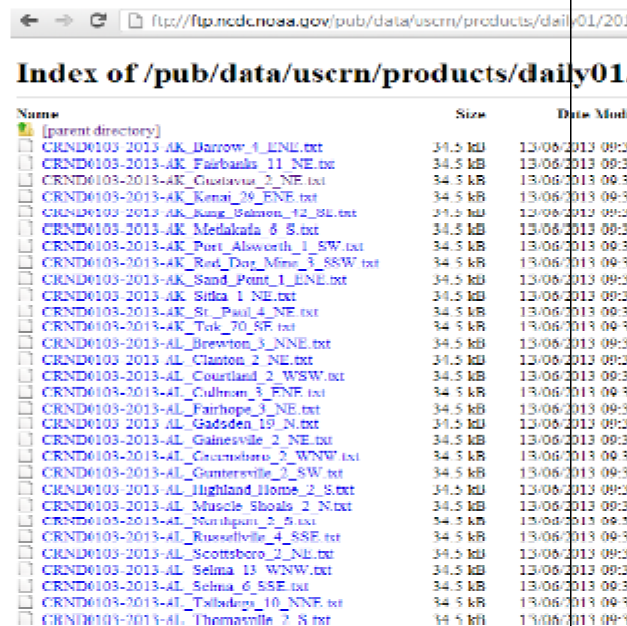
The above diagram gives a clear, step by step explanation of how the data flows through Sqoop, HDFS and Pig script.

Comparing MapReduce and Pig using Weather Data:

ftp://ftp.ncdc.noaa.gov/pub/data/uscrn/products/daily01/

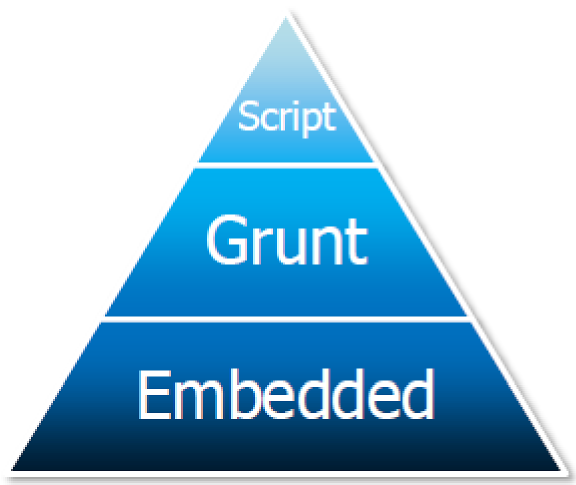
Index of /pub/data/uscrn/products/daily01/

Name	Size	Date Modified
[parent directory]		
2000/		09/01/2013 11:34:00
2001/		09/01/2013 11:34:00
2002/		09/01/2013 11:34:00
2003/		28/02/2013 09:46:00
2004/		09/01/2013 11:34:00
2005/		28/02/2013 09:46:00
2006/		09/01/2013 11:35:00
2007/		09/01/2013 11:36:00
2008/		09/01/2013 11:36:00
2009/		09/01/2013 11:37:00
2010/		13/02/2013 09:36:00
2011/		13/06/2013 09:42:00
2012/		13/06/2013 09:42:00
2013/		13/06/2013 09:43:00
Daily01_New_IR_Sensor_2012-10-24.txt	1.1 kB	16/01/2013 15:08:00
File_name_change.04112011.txt	668 B	04/12/2012 00:00:00
README.txt	13.7 kB	14/06/2013 04:30:00
obsolete/		04/12/2012 00:00:00
snapshots/		10/06/2013 00:51:00
updates/		02/01/2013 04:33:00



Source: ftp://ftp.ncdc.noaa.gov/pub/data/uscrn/products/daily01/

Basic Program Structure of Pig:



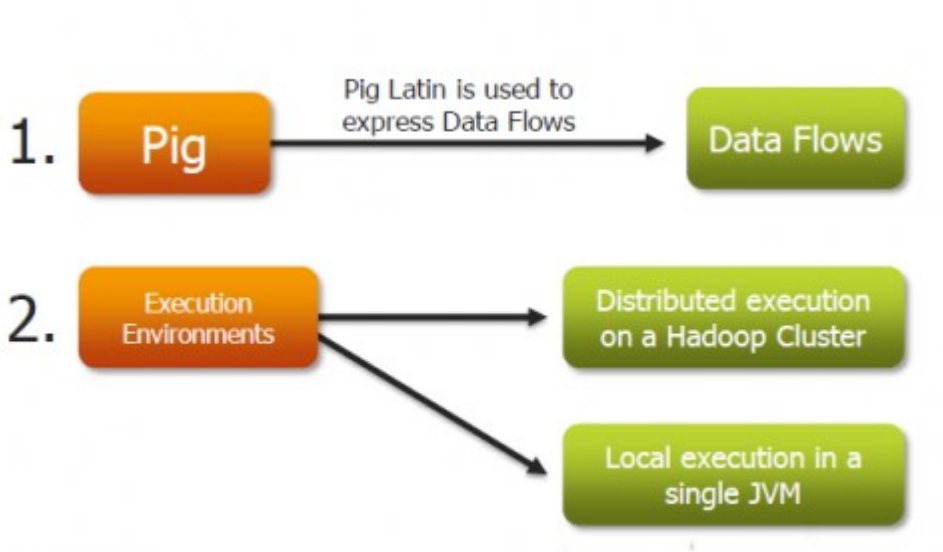
Source: Pig Wiki

Here's the hierarchy of Pig's program structure:

- **Script** – Pig Can run a file script that contains Pig Commands. Eg: pig script .pig runs the command in the local file script.pig

- **Grunt** – It is an interactive shell for running Pig commands. It is also possible to run pig scripts from within Grunts using run and exec.
- **Embedded** – Can run Pig programs from Java, much like you can use JDBC to run SQL programs from Java.

Components of Pig:



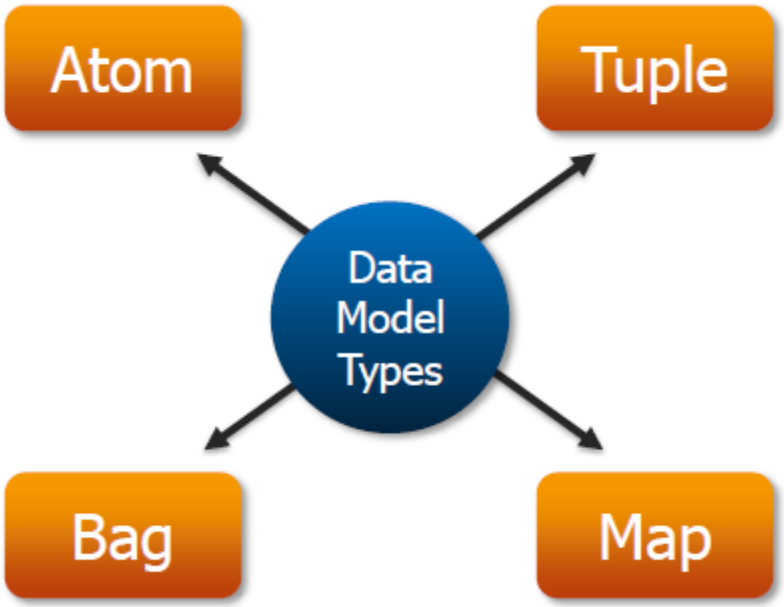
What is Pig Latin Program?

Pig Latin program is made up of a series of operations or transformations that are applied to the input data to produce output. The job of Pig is to convert the transformations in to a series of MapReduce jobs.

Basic Types of Data Models in Pig:

Pig comprises of 4 basic types of data models. They are as follows:

- Atom – It is a simple atomic data value. It is stored as a string but can be used as either a string or a number
- Tuple – An ordered set of fields
- Bag – An collection of tuples.
- Map – set of key value pairs.


Atom
Tuple
Data
Model
Types
Bag
Map