Pig

Data and Analytics Training

Big Data – Self-Study Guide (205)

This study guide serves as a basis for your self-study on the basics of Pig. The study guide gives you an overview on the main topics and presents a number of internal and external resources you should use to get a deeper understanding of Pig, the related terms and technical details.

Please especially use the external resources provided. They are a main part of this study guide and are required to get a good understanding of the discussed topics.

Estimated Completion Time

3 hours (including time for external resources)

Objectives

After completing this study guide, you will be able to understand

- What and Why of Pig
- How to work with Pig Latin
    - Interactively
    - PowerShell
- Pig Commands
- Debugging and Run modes

Contents

Introduction 4
Understanding Pig 4
Grunt 6
Pig Relations 7
Tuple 7
Bag 8
Map 8
Pig Latin in HDInsight 8
Pig Commands 12
LOAD 12
PigStorage() 12
FILTER 13
GROUP 14
STORE 15
FOREACH 16
ORDER BY 17
LIMIT 18
JOIN 19
Debugging Pig 20
Illustrate 20
Explain 22
Describe 23
Run Modes 24
Local mode 24
Mapreduce (Hadoop) mode 25
Submit Pig jobs using PowerShell 26
Conclusion 28

Introduction

Apache Pig is a high-level platform that allows you to perform complex MapReduce transformations on very large data sets using a simple scripting language called Pig Latin. Pig translates the Pig Latin scripts so they'll run within Hadoop. You can create User Defined Functions (UDFs) to extend Pig Latin.[1]

Working with big data is difficult using relational databases and statistics/visualization packages. Due to the large amounts of data and the computation of this data, parallel software running on tens, hundreds, or even thousands of servers is often required to compute this data in a reasonable time. Hadoop provides a MapReduce framework for writing applications that processes large amounts of structured and unstructured data in parallel across large clusters of machines in a very reliable and fault-tolerant manner.[2]

Apache Pig provides a layer of abstraction over the Java-based MapReduce framework, enabling users to analyze data without knowledge of Java or MapReduce. Pig is a platform for analyzing large data sets using Pig Latin which is an easy-to-use data-flow language. Pig reduces the time needed to write mapper and reducer programs. It does not require any knowledge of Java. You also have the flexibility to combine Java code with Pig. Many complex algorithms can be written in less than five lines of human-readable Pig code.

Pig Latin statements follow this general flow:

- Load: Read data to be manipulated from the file system
- Transform: Manipulate the data
- Dump or store: Output data to the screen or store for processing

In this module you learn how to run Apache Pig jobs on HDInsight to analyze large data files. You will use Pig Latin to execute MapReduce jobs as an alternative to writing Java code.

Understanding Pig

When writing MapReduce code in Java or in a language like C# or Python or JavaScript or any of these languages, you get to write a map function, a reduce function which fundamentally comes down to manipulating key-value pairs as really what you are doing at that level. This can lead to some fairly complex programming that you might have to do. Pig is an abstraction over MapReduce that lets us decide what we want to do to our data. We can specify the transformations that we want to

[1] http://azure.microsoft.com/en-us/documentation/articles/hdinsight-hadoop-introduction

[2] http://azure.microsoft.com/en-us/documentation/articles/hdinsight-use-pig/

make to our data as a series of operations. Effectively we transform, we take our initial source data, we apply some sort of schema to that data, we use that to then transform that schema, and gradually filter and reshape the data to the "what we want" in order to do our analysis with.

The principal is basically we load our data into something called a *relation*. A relation is effectively a schema that we project on to the data. The data isn't validated when we first upload it to HDFS, we just upload any data that we want to work on. The name Pig, is quite often used, as people talk about "Pigs eat anything". That's the idea, we can give any sort of data to Pig, and it will impose the schema on it when it reads it.

This is what is called the *schema on read* approach to processing the data: To apply a schema as the data is read, and then perform operations that need to do to transformations to that schema, filtering out rows, reshaping it, grouping data, whatever needs to happen to get to the results that are needed.

Pig Latin runs either interactively in the Grunt shell, which really is the name of the shell environment because it is Pig, or we can run them as a script file. You can run them interactively by typing each command and viewing the changes you make to data as you go. You can just run the script file and have the whole series of transformations done as a script.

Pig is great for scenarios where you can easily articulate the changes you want to make to your data as a sequence of operations.

Pig Latin has two interesting operations: DUMP and STORE. Looking at a Pig Latin script, you might think it being sequential processing sounds really inefficient. You define what you want to happen to the data. Nothing actually happens until you issue either a DUMP which displays the data on the screen or a STORE which stores the results into a folder. And what happens at that point, is that Pig engine generates MapReduce code. It goes and looks what you are trying to do with data, figures out the correct MapReduce that it needs for that and actually goes and generates a jar that is deployed to the cluster.

Pig is not performing these things like an interpretive language. It is not internally taking the data and then doing the next thing as you enter the commands, it is bundling all those commands up and compiling them into a jar and running that as a MapReduce job.

Please watch this 15 minute video with more details on Apache Pig:
http://www.youtube.com/watch?v=PQb9I-8986s

Grunt

Grunt is Pig's interactive shell. It enables users to enter Pig Latin interactively and provides a shell for users to interact with HDFS.[3]

To enter Grunt, invoke Pig with no script or command to run. Typing:

```
pig -x local
```

will result in the prompt:

```
grunt>
```

You will find details on how to open grunt in HDInsight below.

This gives you a shell to interact with your local filesystem. If you omit the -x local and have a cluster configuration set in PIG_CLASSPATH, this will put you in a Grunt shell that will interact with HDFS on your cluster.

As you would expect with a shell, Grunt provides command-line history and editing, as well as Tab completion. It does not provide filename completion via the Tab key. That is, if you type `kil` and then press the Tab key, it will complete the command as `kill`. But if you have a file `foo` in your local directory and type `ls fo`, and then hit Tab, it will not complete it as `ls foo`. This is because the response time from HDFS to connect and find whether the file exists is too slow to be useful.

Although Grunt is a useful shell, remember that it is not a full-featured shell. It does not provide a number of commands found in standard Unix shells, such as pipes, redirection, and background execution.

To exit Grunt you can type `quit` or enter Ctrl-D.

Grunt also provides commands for controlling Pig and MapReduce:

`kill` *`jobid`*

Kill the MapReduce job associated with *`jobid`*. The output of the `pig` command that spawned the job will list the ID of each job it spawns. You can also find the job's ID by looking at Hadoop's JobTracker GUI, which lists all jobs currently running on the cluster. Note that this command kills a particular MapReduce job. If your Pig job contains other MapReduce jobs that do not depend on the

[3] http://chimera.labs.oreilly.com/books/1234000001811/ch03.html#grunt_and_hdfs

killed MapReduce job, these jobs will still continue. If you want to kill all of the MapReduce jobs associated with a particular Pig job, it is best to terminate the process running Pig, and then use this command to kill any MapReduce jobs that are still running. Make sure to terminate the Pig process with a Ctrl-C or a Unix `kill`, not a Unix `kill -9`.

You can find more details on the parameter substitution here:

http://chimera.labs.oreilly.com/books/1234000001811/ch06.html#param_sub

Pig Relations

Pig Latin statements work with relations. A relation can be defined as follows:

- A relation is a bag (more specifically, an outer bag).
- A bag is a collection of tuples.
- A tuple is an ordered set of fields.
- A field is a piece of data.

A Pig relation is a bag of tuples. A Pig relation is similar to a table in a relational database, where the tuples in the bag correspond to the rows in a table. Unlike a relational table, however, Pig relations don't require that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

Also note that relations are unordered which means there is no guarantee that tuples are processed in any particular order. Furthermore, processing may be parallelized in which case tuples are not processed according to any total ordering.[4]

Tuple

A tuple is an ordered set of fields.[5]

You can think of a tuple as a row with one or more fields, where each field can be any data type and any field may or may not have data. If a field has no data, then the following happens:

In a load statement, the loader will inject null into the tuple. The actual value that is substituted for null is loader specific; for example, PigStorage substitutes an empty field for null.

In a non-load statement, if a requested field is missing from a tuple, Pig will inject null.

[4] https://pig.apache.org/docs/r0.7.0/piglatin_ref2.html#Relations%2C+Bags%2C+Tuples%2C+Fields

[5] https://pig.apache.org/docs/r0.7.0/piglatin_ref2.html#Relations%2C+Bags%2C+Tuples%2C+Fields

In this example the tuple contains three fields.

```
(John,18,4.0F)
```

Bag

A bag is a collection of tuples.[6]

- A bag can have duplicate tuples.
- A bag can have tuples with differing numbers of fields. However, if Pig tries to access a field that does not exist, a null value is substituted.
- A bag can have tuples with fields that have different data types. However, for Pig to effectively process bags, the schemas of the tuples within those bags should be the same. For example, if half of the tuples include chararray fields and while the other half include float fields, only half of the tuples will participate in any kind of computation because the chararray fields will be converted to null.

Bags have two forms: outer bag (or relation) and inner bag.

In this example A is a relation or bag of tuples. You can think of this bag as an outer bag.

```
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
```

Map

A map is a set of key value pairs. Key values within a relation must be unique.[7]

In this example the map includes two key value pairs.

```
[name#John,phone#5551212]
```

Pig Latin in HDInsight

You can use PigLatin statement in your HDInsight environment or in the HDInsight emulator. In case you would like to use the Hortonworx sandbox, you can find a link to a referenced tutorial below.

In this section, you will review some Pig Latin statements individually, and their results after running the statements.

[6] https://pig.apache.org/docs/r0.7.0/piglatin_ref2.html#Relations%2C+Bags%2C+Tuples%2C+Fields

[7] https://pig.apache.org/docs/r0.7.0/piglatin_ref2.html#Relations%2C+Bags%2C+Tuples%2C+Fields

1. Let's assume that we are analysing some weather data that is in the below format:

```
01-01-2000 00:00:00, 44.2, 4

01-01-2000 00:00:00, 48.3, 2

01-01-2000 00:00:00, 47.1, 2

02-01-2000 00:00:00, 39.7, 3

02-01-2000 00:00:00, 40.7, 1

02-01-2000 00:00:00, 42.3, 2
```

   This data contains date, temperature and wind speed from a weather station at different times of the day.

2. Upload this file to HDFS.

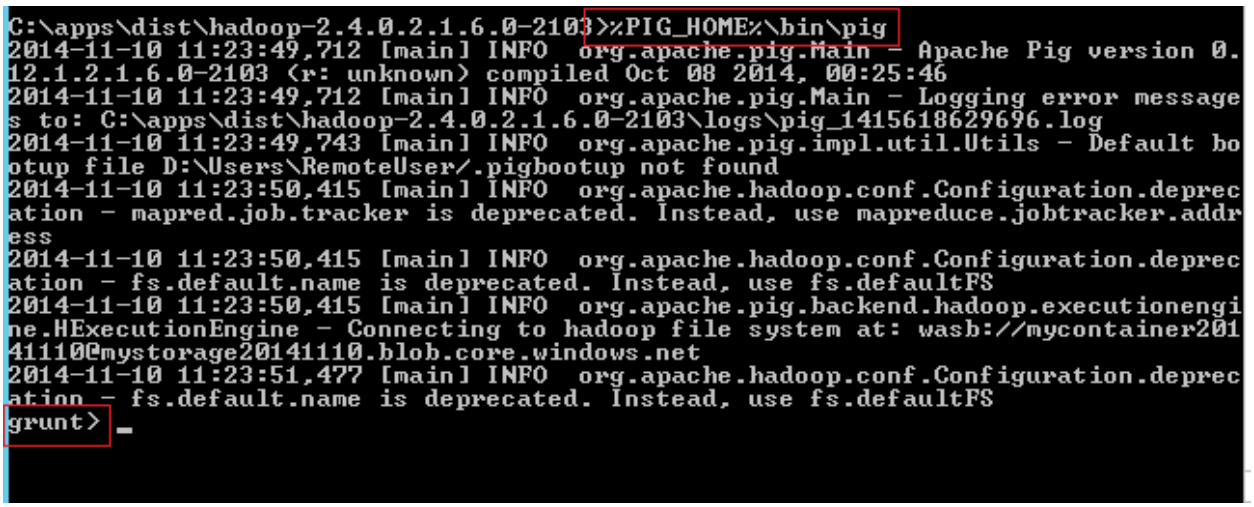
```
C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -mkdir -p /data/weather

C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -copyFromLocal c:\weather.txt /
data/weather

C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -ls /data/weather
Found 1 items
-rw-r--r--   1 RemoteUser supergroup       8937 2014-11-10 11:00 /data/weather/
weather.txt
```

3. Now we would like to go and open up our Pig environment in the console. Now to do that there are a bunch of built-in system variables that you can see by running SET command. One of the variables it shows is PIG_HOME=<path to where the Pig installation is>

```
PIG_HOME=C:\apps\dist\pig-0.12.1.2.1.6.0-2103
```

   And the reason we have the system variable is, each of these folder has a version specific name for when it was installed. And of course, if we move to different version of HDInsight next time, we get a different version of Pig. We don't have to rely on remembering the version number. So we use these system variables to make it easy to navigate around the file system.

4. Let us just go and launch the Pig environment, we'll launch Grunt as it is called by using command: %PIG_HOME%\bin\pig



And here's the Pig environment. This is what the grunt shell looks like. It is the console window where we can type our Pig Latin commands.

5. Run the Pig Latin command:

/data/weather folder contains the weather.txt file. We are using PigStorage with the comma as the delimiter. The fields are obs_date, by not specifically saying its type, it is assumed to be chararray (which is the default data type if not specified). Temp and windspeed have floating values rather than long because we want to average them out and want the result to be float as well. The line ends with a semicolon and when we press return, it just projects the schema over the data because we need something about it. And important thing is that we are projecting this schema on to the data, it is not inherited in the data. So we could have it a way, project each row, just have each row as one column. And read each row as a single column and filter out specific years or something. We can do that in the first pass, and in the second pass we can project a different schema on the data. Which is the flexibility of using a system like this, as much you know about the data, you can use. If you don't, it will still work and you can still get some insights out of it. So, we've created a relation called Weather. We haven't done any processing yet, we have just projected that schema.

6. Let's now group the Weather by obs_date. Again, we are using the schema that we have projected, grouping everything by obs_date. What we end up with is a group that contains just the obs_date and then all of the fields that relate to that date from the original weather relation.

```
grunt> GroupedWeather = GROUP Weather BY obs_date;
grunt>
```

7. Now we will aggregate the weather.

```
grunt> AggWeather = FOREACH GroupedWeather GENERATE group, AVG(Weather.windspeed
) AS avg_windspeed, MAX(Weather.temp) AS high_temp;
grunt>
```

   For each of the grouped weather (tuples that we have got), it will generate the group, which is obviously the date. It could be grouped using multiple fields, of course. But in this case it is just the date. It will also generate the average wind speed as avg_windspeed and maximum temperature as high_temp. This will show us if this weather station could be a viable holiday destination.

8. All we have been doing here is defining the sequence of operations we want to be performed. In this case, we are going through each of the aggregated weather, flatten out the groups (if it had multiple fields, it would separate out on those fields) as the date and then take the average windspeed and the high temperature.

```
grunt> DailyWeather = FOREACH AggWeather GENERATE FLATTEN(group) AS obs_date, av
g_windspeed, high_temp;
grunt>
```

   So, we end up after this with a schema that has three columns.

9. We don't have a guarantee in what order the data is going to be. So, let's sort it by date in ascending order:

```
grunt> SortedWeather = ORDER DailyWeather BY obs_date ASC;
grunt>
```

   The syntax is fairly intuitive. Once you start using it, it is pretty straight-forward and easy to understand.

10. And here's where it gets more interesting. Here we actually go and generate some MapReduce code. When we issue this DUMP command, what Pig is going to do is generate the appropriate jar files that it needs to do the MapReduce work.

```
grunt> DUMP SortedWeather;
```

Suddenly, it all comes to life. There's a whole bunch of things going on. You start to see some job ids being generated and it enters the MapReduce layer. Fundamentally everything you do in HDInsight cluster ends up being MapReduce. So, Pig allows us to make it easier to generate that MapReduce. Instead of writing the Java code to do this, it will write the jar file for us. Fundamentally it boils down to MapReduce.

HDInsight and MapReduce does really-really well with massive amount of data. If you are running this example on a small amount of data, it doesn't necessarily equate to the benefits. But the power of what we did is evident in the example.

Pig Commands

LOAD

Suppose we have a data file called myfile.txt. The fields are tab-delimited. The records are newline-separated.

```
1 2 3
4 2 1
8 3 4
```

In this example the default load function, PigStorage, loads data from myfile.txt to form relation A. The two LOAD statements are equivalent. Note that, because no schema is specified, the fields are not named and all fields default to type bytearray.

```
A = LOAD 'myfile.txt';

A = LOAD 'myfile.txt' USING PigStorage('\t');

DUMP A;

(1,2,3)
(4,2,1)
(8,3,4)
```

PigStorage()

PigStorage is probably the most frequently used Load/Store utility. It parses input records based on a delimiter and the fields thereafter can be positionally referenced or referenced via alias. Starting version 0.10, Pig has a couple of options that could be very useful.[8]

If no argument is provided, PigStorage will assume tab-delimited format. If a delimiter argument is provided, it must be a single-byte character; any literal (eg: 'a', '|'), known escape character (eg: '\t', '\r') is a valid delimiter. For example, to load a space-separated file:[9]

```
data = LOAD 's3n://input-bucket/input-folder' USING PigStorage(' ')
```

[8] https://hadoopified.wordpress.com/2012/04/22/pigstorage-options-schema-and-source-tagging/

[9] https://help.mortardata.com/technologies/pig/pigstorage

```
AS (field0:chararray, field1:int);
```

The schema must be provided in the AS clause.

PigStorage is an extremely simple loader that does not handle special cases such as embedded delimiters or escaped control characters; it will split on every instance of the delimiter regardless of context.

FILTER

Use the FILTER operator to work with tuples or rows of data (if you want to work with columns of data, use the FOREACH…GENERATE operation).[10]

FILTER is commonly used to select the data that you want; or, conversely, to filter out (remove) the data you don't want.

Suppose we have relation A.

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)
```

In this example the condition states that if the third field equals 3, then include the tuple with relation X.

```
X = FILTER A BY f3 == 3;

DUMP X;
(1,2,3)
(4,3,3)
(8,4,3)
```

In this example the condition states that if the first field equals 8 or if the sum of fields f2 and f3 is not greater than first field, then include the tuple relation X.

```
X = FILTER A BY (f1 == 8) OR (NOT (f2+f3 > f1));
```

[10] https://pig.apache.org/docs/r0.11.1/basic.html#filter

```
DUMP X;
(4,2,1)
(8,3,4)
(7,2,5)
(8,4,3)
```

GROUP

The GROUP operator groups together tuples that have 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. This tuple contains two fields:[11]

- The first field is named "group" (do not confuse this with the GROUP operator) and is the same type as the group key.
- The second field takes the name of the original relation and is type bag.
- The names of both fields are generated by the system as shown in the example below.

Note the following about the GROUP/COGROUP and JOIN operators:

- The GROUP and JOIN operators perform similar functions. GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples
- The GROUP/COGROUP and JOIN operators handle null values differently

Suppose we have relation A.

```
A = load 'student' AS (name:chararray,age:int,gpa:float);

DESCRIBE A;
A: {name: chararray,age: int,gpa: float}

DUMP A;
(John,18,4.0F)
(Mary,19,3.8F)
(Bill,20,3.9F)
(Joe,18,3.8F)
```

Now, suppose we group relation A on field "age" for form relation B. We can use the DESCRIBE and ILLUSTRATE operators to examine the structure of relation B. Relation B has two fields. The first field is

[11] https://pig.apache.org/docs/r0.11.1/basic.html#GROUP

named "group" and is type int, the same as field "age" in relation A. The second field is name "A" after relation A and is type bag.

```
B = GROUP A BY age;

DESCRIBE B;
B: {group: int, A: {name: chararray,age: int,gpa: float}}

ILLUSTRATE B;
etc ...
-------------------------------------------------------------------------
| B     | group: int | A: bag({name: chararray,age: int,gpa: float}) |
-------------------------------------------------------------------------
|       | 18         | {(John, 18, 4.0), (Joe, 18, 3.8)}             |
|       | 20         | {(Bill, 20, 3.9)}                             |
-------------------------------------------------------------------------

DUMP B;
(18,{(John,18,4.0F),(Joe,18,3.8F)})
(19,{(Mary,19,3.8F)})
(20,{(Bill,20,3.9F)})
```

Continuing on, as shown in these FOREACH statements, we can refer to the fields in relation B by names "group" and "A" or by positional notation.

```
C = FOREACH B GENERATE group, COUNT(A);

DUMP C;
(18,2L)
(19,1L)
(20,1L)

C = FOREACH B GENERATE $0, $1.name;

DUMP C;
(18,{(John),(Joe)})
(19,{(Mary)})
(20,{(Bill)})
```

STORE

Use the STORE operator to run (execute) Pig Latin statements and save (persist) results to the file system. Use STORE for production scripts and batch mode processing.[12]

To debug scripts during development, you can use DUMP to check intermediate results.

[12] https://pig.apache.org/docs/r0.11.1/basic.html#store

In this example data is stored using PigStorage and the asterisk character (*) as the field delimiter.

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

STORE A INTO 'myoutput' USING PigStorage ('*');

CAT myoutput;
1*2*3
4*2*1
8*3*4
4*3*3
7*2*5
8*4*3
```

FOREACH

Use the FOREACH…GENERATE operation to work with columns of data (if you want to work with tuples or rows of data, use the FILTER operation).[13]

FOREACH…GENERATE works with relations (outer bags) as well as inner bags:

If A is a relation (outer bag), a FOREACH statement could look like this.

```
X = FOREACH A GENERATE f1;
```

If A is an inner bag, a FOREACH statement could look like this.

```
X = FOREACH B {
        S = FILTER A BY 'xyz';
        GENERATE COUNT (S.$0);
}
```

In this example the asterisk (*) is used to project all fields from relation A to relation X. Relation A and X are identical.

```
X = FOREACH A GENERATE *;

DUMP X;
```

[13] https://pig.apache.org/docs/r0.11.1/basic.html#foreach

```
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)
```

In this example two fields from relation A are projected to form relation X.

```
X = FOREACH A GENERATE a1, a2;

DUMP X;
(1,2)
(4,2)
(8,3)
(4,3)
(7,2)
(8,4)
```

ORDER BY

In Pig, relations are unordered:[14]

- If you order relation A to produce relation X (X = ORDER A BY * DESC;) relations A and X still contain the same data.
- If you retrieve relation X (DUMP X;) the data is guaranteed to be in the order you specified (descending).
- However, if you further process relation X (Y = FILTER X BY $0 > 1;) there is no guarantee that the data will be processed in the order you originally specified (descending).

Pig currently supports ordering on fields with simple types or by tuple designator (*). You cannot order on fields with complex types or by expressions.

```
A = LOAD 'mydata' AS (x: int, y: map[]);
B = ORDER A BY x; -- this is allowed because x is a simple type
B = ORDER A BY y; -- this is not allowed because y is a complex type
B = ORDER A BY y#'id'; -- this is not allowed because y#'id' is an expression
```

Suppose we have relation A.

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;
```

[14] https://pig.apache.org/docs/r0.11.1/basic.html#order-by

```
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)
```

In this example relation A is sorted by the third field, f3 in descending order. Note that the order of the three tuples ending in 3 can vary.

```
X = ORDER A BY a3 DESC;

DUMP X;
(7,2,5)
(8,3,4)
(1,2,3)
(4,3,3)
(8,4,3)
(4,2,1)
```

LIMIT

Use the LIMIT operator to limit the number of output tuples.[15]

If the specified number of output tuples is equal to or exceeds the number of tuples in the relation, all tuples in the relation are returned.

If the specified number of output tuples is less than the number of tuples in the relation, then n tuples are returned. There is no guarantee which n tuples will be returned, and the tuples that are returned can change from one run to the next. A particular set of tuples can be requested using the ORDER operator followed by LIMIT.

Note: The LIMIT operator allows Pig to avoid processing all tuples in a relation. In most cases a query that uses LIMIT will run more efficiently than an identical query that does not use LIMIT. It is always a good idea to use limit if you can.

In this example the limit is expressed as a scalar.

```
a = load 'a.txt';
b = group a all;
c = foreach b generate COUNT(a) as sum;
d = order a by $0;
e = limit d c.sum/100;
```

Suppose we have relation A.

[15] https://pig.apache.org/docs/r0.11.1/basic.html#limit

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)
```

In this example output is limited to 3 tuples. Note that there is no guarantee which three tuples will be output.

```
X = LIMIT A 3;

DUMP X;
(1,2,3)
(4,3,3)
(7,2,5)
```

JOIN

Use the JOIN operator to perform an inner, equijoin join of two or more relations based on common field values. Inner joins ignore null keys, so it makes sense to filter them out before the join.[16]

Note the following about the GROUP/COGROUP and JOIN operators:

- The GROUP and JOIN operators perform similar functions. GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples.
- The GROUP/COGROUP and JOIN operators handle null values differently

Suppose we have relations A and B.

```
A = LOAD 'data1' AS (a1:int,a2:int,a3:int);

DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

B = LOAD 'data2' AS (b1:int,b2:int);

DUMP B;
(2,4)
(8,9)
```

[16] https://pig.apache.org/docs/r0.11.1/basic.html#join-inner

```
(1,3)
(2,7)
(2,9)
(4,6)
(4,9)
```

In this example relations A and B are joined by their first fields.

```
X = JOIN A BY a1, B BY b1;

DUMP X;
(1,2,3,1,3)
(4,2,1,4,6)
(4,3,3,4,6)
(4,2,1,4,9)
(4,3,3,4,9)
(8,3,4,8,9)
(8,4,3,8,9)
```

You can find more details on outer joins here:

https://pig.apache.org/docs/r0.11.1/basic.html#join-outer

and a list of all the Pig Latin operators here:

https://pig.apache.org/docs/r0.11.1/basic.html

Additionally you may want to follow the steps in this Hortonworks tutorial giving an introduction on the basic Pig commands by using the Hortonworks sandbox environment:

http://hortonworks.com/hadoop-tutorial/how-to-use-basic-pig-commands/

Debugging Pig

Illustrate

Use the ILLUSTRATE operator to review how data is transformed through a sequence of Pig Latin statements. ILLUSTRATE allows you to test your programs on small datasets and get faster turnaround times.[17]

[17] http://pig.apache.org/docs/r0.9.1/test.html#illustrate

ILLUSTRATE is based on an example generator. The algorithm works by retrieving a small sample of the input data and then propagating this data through the pipeline. However, some operators, such as JOIN and FILTER, can eliminate tuples from the data - and this could result in no data following through the pipeline. To address this issue, the algorithm will automatically generate example data, in near real-time. Thus, you might see data propagating through the pipeline that was not found in the original input data, but this data changes nothing and ensures that you will be able to examine the semantics of your Pig Latin statements.

As shown in the examples below, you can use ILLUSTRATE to review a relation or an entire Pig script.

This example demonstrates how to use ILLUSTRATE with a relation. Note that the LOAD statement must include a schema (the AS clause).

```
grunt> visits = LOAD 'visits.txt' AS (user:chararray, url:chararray,
timestamp:chararray);
grunt> DUMP visits;

(Amy,yahoo.com,19990421)
(Fred,harvard.edu,19991104)
(Amy,cnn.com,20070218)
(Frank,nba.com,20070305)
(Fred,berkeley.edu,20071204)
(Fred,stanford.edu,20071206)

grunt> recent_visits = FILTER visits BY timestamp >= '20071201';
grunt> user_visits = GROUP recent_visits BY user;
grunt> num_user_visits = FOREACH user_visits GENERATE group,
COUNT(recent_visits);
grunt> DUMP num_user_visits;

(Fred,2)

grunt> ILLUSTRATE num_user_visits;
-------------------------------------------------------------------------
| visits     | user: chararray | url: chararray | timestamp: chararray |
-------------------------------------------------------------------------
|            | Fred            | berkeley.edu   | 20071204             |
|            | Fred            | stanford.edu   | 20071206             |
|            | Frank           | nba.com        | 20070305             |
-------------------------------------------------------------------------
-------------------------------------------------------------------------
--
| recent_visits     | user: chararray | url: chararray | timestamp: chararray
|
-------------------------------------------------------------------------
--
|                   | Fred            | berkeley.edu   | 20071204             |
|                   | Fred            | stanford.edu   | 20071206             |
-------------------------------------------------------------------------
--
```

```
-------------------------------------------------------------------------------
-----------------------------------
| user_visits     | group: chararray | recent_visits: bag({user: chararray,url:
chararray,timestamp: chararray}) |
-------------------------------------------------------------------------------
-----------------------------------
|                 | Fred             | {(Fred, berkeley.edu, 20071204), (Fred,
stanford.edu, 20071206)}          |
-------------------------------------------------------------------------------
-----------------------------------
-------------------------------------------------
| num_user_visits     | group: chararray | long  |
-------------------------------------------------
|                     | Fred             | 2     |
-------------------------------------------------
```

Explain

Use the EXPLAIN operator to review the logical, physical, and map reduce execution plans that are used to compute the specified relationship.[18]

If no script is given:

The logical plan shows a pipeline of operators to be executed to build the relation. Type checking and backend-independent optimizations (such as applying filters early on) also apply.

The physical plan shows how the logical operators are translated to backend-specific physical operators. Some backend optimizations also apply.

The mapreduce plan shows how the physical operators are grouped into map reduce jobs.

If a script without an alias is specified, it will output the entire execution graph (logical, physical, or map reduce).

If a script with a alias is specified, it will output the plan for the given alias.

In this example the EXPLAIN operator produces all three plans. (Note that only a portion of the output is shown in this example.)

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);

B = GROUP A BY name;

C = FOREACH B GENERATE COUNT(A.age);
```

[18] http://pig.apache.org/docs/r0.9.1/test.html#explain

```
EXPLAIN C;
-----------------------------------------------
Logical Plan:
-----------------------------------------------
Store xxx-Fri Dec 05 19:42:29 UTC 2008-23 Schema: {long} Type: Unknown
|
|---ForEach xxx-Fri Dec 05 19:42:29 UTC 2008-15 Schema: {long} Type: bag
 etc ...

-----------------------------------------------
Physical Plan:
-----------------------------------------------
Store(fakefile:org.apache.pig.builtin.PigStorage) - xxx-Fri Dec 05 19:42:29 UTC
2008-40
|
|---New For Each(false)[bag] - xxx-Fri Dec 05 19:42:29 UTC 2008-39
    |   |
    |   POUserFunc(org.apache.pig.builtin.COUNT)[long] - xxx-Fri Dec 05
 etc ...

------------------------------------------------
| Map Reduce Plan
------------------------------------------------
MapReduce node xxx-Fri Dec 05 19:42:29 UTC 2008-41
Map Plan
Local Rearrange[tuple]{chararray}(false) - xxx-Fri Dec 05 19:42:29 UTC 2008-34
|   |
|   Project[chararray][0] - xxx-Fri Dec 05 19:42:29 UTC 2008-35
 etc ...
```

Describe

Use the DESCRIBE operator to view the schema of a relation. You can view outer relations as well as relations defined in a nested FOREACH statement.

In this example a schema is specified using the AS clause. If all data conforms to the schema, Pig will use the assigned types.

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);

B = FILTER A BY name matches 'J.+';

C = GROUP B BY name;

D = FOREACH B GENERATE COUNT(B.age);

DESCRIBE A;
A: {group, B: (name: chararray,age: int,gpa: float}

DESCRIBE B;
B: {group, B: (name: chararray,age: int,gpa: float}
```

```
DESCRIBE C;
C: {group, chararry,B: (name: chararray,age: int,gpa: float}

DESCRIBE D;
D: {long}
```

Of course you can also use the DUMP function that you have already seen above to show results on the screen.

http://pig.apache.org/docs/r0.9.1/test.html#dump

Run Modes

Pig has two run modes or exectypes, local and hadoop (currently called mapreduce).[19]

- Local Mode: To run Pig in local mode, you need access to a single machine.
- Hadoop (mapreduce) Mode: To run Pig in hadoop (mapreduce) mode, you need access to a Hadoop cluster and HDFS installation.

You should use the local mode to test and debug your scripts.

Local mode

To run Pig in local mode, you only need access to a single machine. To make things simple, copy these files to your current working directory (you may want to create a temp directory and move to it):

- The /etc/passwd file
- The pig.jar file, created when you build Pig
- The sample code files (id.pig and idlocal.java) located here:
    - https://wiki.apache.org/pig/RunPig?action=AttachFile&do=view&target=id.pig
    - https://wiki.apache.org/pig/RunPig?action=AttachFile&do=view&target=idhadoop.java

To run Pig's Grunt shell in local mode, follow these instructions.

First, point $PIG_CLASSPATH to the pig.jar file (in your current working directory):

[19] https://wiki.apache.org/pig/RunPig

```
$ export PIG_CLASSPATH=./pig.jar
```

From your current working directory, run:

```
$ pig -x local
```

The Grunt shell is invoked and you can enter commands at the prompt.

```
grunt> A = load 'passwd' using PigStorage(':');

grunt> B = foreach A generate $0 as id;

grunt> dump B;
```

Mapreduce (Hadoop) mode

This section shows you how to run Pig in hadoop (mapreduce) mode, using the Grunt shell, a Pig script, and an embedded program. [20]

To run Pig in hadoop (mapreduce) mode, you need access to a Hadoop cluster. You also need to copy these files to your home or current working directory.

- The /etc/passwd file
- The pig.jar file, created when you build Pig
- The sample code files (id.pig and idhadoop.java) located here:
    - https://wiki.apache.org/pig/RunPig?action=AttachFile&do=view&target=id.pig
    - https://wiki.apache.org/pig/RunPig?action=AttachFile&do=view&target=idhadoop.java

To run Pig's Grunt shell in hadoop (mapreduce) mode, follow these instructions. When you begin the session, Pig will allocate a 15-node cluster. When you quit the session, Pig will deallocate the nodes.

From your current working directory, run:

```
$ pig
```

or

```
$ pig -x mapreduce
```

The Grunt shell is invoked and you can enter commands at the prompt.

```
grunt> A = load 'passwd' using PigStorage(':');
```

[20] https://wiki.apache.org/pig/RunPig

```
grunt> B = foreach A generate $0 as id;

grunt> dump B;
```

Submit Pig jobs using PowerShell

In the previous section the working with Pig Latin in the command line was described. If you want to create some sort of script file and run that script file, just as and when you need it. We are going to look at doing that from within PowerShell. Because then you don't always have to remote desktop to the cluster every time to do this.

For Pid there is a NewAzureHDInsightPigJobDefinition commands that allows us to define what that job is going to consist of. It is going to consist of either a query with some explicit Pig Latin statements or a file that is going to contain the Pig Latin statements. Typically we're going to save this script in a file, upload the file to HDFS and then we use this commandlet to go and execute that file on-demand when we need it.

We don't need to be in our remote desktop anymore. We can perform this demo our client side, on premise machine on which we regularly work.

1. Create the Pig Latin script file with the following content:

```
Weather = LOAD '/data/weather' USING PigStorage(',') AS (obs_date, temp:float,
windspeed:float);

GroupedWeather = GROUP Weather BY obs_date;

AggWeather = FOREACH GroupedWeather GENERATE group, AVG(Weather.windspeed) AS
avg_windspeed, MAX(Weather.temp) AS high_temp;

DailyWeather = FOREACH AggWeather GENERATE FLATTEN(group) AS obs_date,
avg_windspeed, high_temp;

SortedWeather = ORDER DailyWeather BY obs_date ASC;

STORE SortedWeather INTO '/data/summarisedweather';
```

It is the same script that we have just seen, except that instead of DUMP to bring it to the screen, we are using STORE and we are storing the results into /data/summarisedweather

It is going to run this script, it will run asynchronously and it will generate the results, store them in that location, and as and when we are ready we can go and get the results and look at them.

Let's save this file as SummariseWeather.pig

2. We'll now look at the Pig Latin PowerShell script.

```
$clusterName = "<your_cluster_name>"
$storageAccountName = "<your_storage_account_name>"
$containerName = "<your_container_name>"
$localFolder = "<path_where_the_above_pig_file_resides>"
$destFolder = "data"
$scriptFile = "SummariseWeather.pig"
$outputFolder = "summarisedweather"
$outputFile = "part-r-00000"
```

We specify the name of the cluster, name of the storage account, container name and folder path where we saved the Pig file from previous step.

3. Uploading the Pig file has exact same code we saw in previous modules for uploading the data files. It uses Set-AzureStorageBlobContent to write the script file up there.

```
# Upload Pig Latin script to HDFS
$storageAccountKey = (Get-AzureStorageKey `
                        -StorageAccountName $storageAccountName).Primary
$blobContext = New-AzureStorageContext `
                    -StorageAccountName $storageAccountName `
                    -StorageAccountKey $storageAccountKey
$blobName = "$destFolder/$scriptFile"
$filename = "$localFolder\$scriptFile"
Set-AzureStorageBlobContent `
            -File $filename `
            -Container $containerName `
            -Blob $blobName `
            -Context $blobContext
Write-Host "$scriptFile uploaded to $containerName!"
```

4. Running the Pig Latin script is more interesting. We create a New-AzureHDInsightPigJobDefinition. And that consists of a parameter that says that you are going to execute a file. And that file would be under $destFolder as $scriptFile. We then start the job, and off it goes. We then wait on the job getting completed. And we then get the output, which is not the result. It is just the information about what happened while it was executing. It is all that stuff that appeared on the screen that gave us the breakdown of all the things that were going on.

```
# Run the Pig Latin script
$jobDef = New-AzureHDInsightPigJobDefinition `
                -File "wasb:///$destFolder/$scriptFile"
$pigJob = Start-AzureHDInsightJob `
                -Cluster $clusterName `
                -JobDefinition $jobDef
Write-Host "Pig job submitted..."
Wait-AzureHDInsightJob `
        -Job $pigJob `
        -WaitTimeoutInSeconds 3600
Get-AzureHDInsightJobOutput `
        -Cluster $clusterName `
        -JobId $pigJob.JobId `
        -StandardError
```

5. And then when it is finished, when we've got the job done, we connect to that remote blob store, we download it, and we put it into our local file system, and then just to show the results on the screen we use the cat command. This is actually going to go, run the script, get the results and bring the results down, all in one script.

```
# Get the job output
$remoteblob = "$destFolder/$outputFolder/$outputFile"
Write-Host "Downloading $remoteBlob..."
Get-AzureStorageBlobContent `
        -Container $containerName `
        -Blob $remoteblob `
        -Context $blobContext `
        -Destination "C:\HDInsight\Mod04"
cat $localFolder\$destFolder\$outputFolder\$outputFile
```

Conclusion

You've seen that we can use Pig to create a MapReduce job. If the operations that have to be performed are supported by Pig, that's a much simpler way to do than writing MapReduce code. If we wanted to do something more complicated in terms of our processing, Pig provides us with a syntax that's actually pretty intuitive to define our job as a series of steps and then generate the MapReduce code to give us the data in the shape we needed.

Pig Latin is a pretty extensive language. We have covered some of the main syntactical elements in this module and shown you some of the key things that you can do. But it is really much more powerful than what we've covered here. We can do all sorts of things with multiple relations, joining them, merging them, and all that type of thing. And pretty much anything you can imagine you can process your data with Pig.

The other interesting thing you can do with Pig is, that if you need to combine Pig with some lower level MapReduce code, you can create jars that contains Java functions and call those functions from Pig. There is some pretty powerful stuff you can do to be able to create very sophisticated solution using just Pig Latin and any additional Java code that you want to write.

It suits scenarios where data is processed sequentially, the scenario where you can define a series of transformations you want to apply to the data in order to get to the shape you need it to be. That really where Pig shines.

You can run the Pig Latin statements either interactively in the shell or you can put them in a script and you can run them from a PowerShell, you can run them remotely from a client machine and have your Pig Latin executed.

You can find more training resources for Apache Pig here:
https://cwiki.apache.org/confluence/display/PIG/Pig+Training