CHAPTER 3
Pig and Python

Pig is composed of two major parts: a high-level data flow language called Pig Latin, and an engine that parses, optimizes, and executes the Pig Latin scripts as a series of MapReduce jobs that are run on a Hadoop cluster. Compared to Java MapReduce, Pig is easier to write, understand, and maintain because it is a data transformation language that allows the processing of data to be described as a sequence of transformations. Pig is also highly extensible through the use of the User Defined Functions (UDFs) which allow custom processing to be written in many languages, such as Python.

An example of a Pig application is the Extract, Transform, Load (ETL) process that describes how an application extracts data from a data source, transforms the data for querying and analysis purposes, and loads the result onto a target data store. Once Pig loads the data, it can perform projections, iterations, and other transformations. UDFs enable more complex algorithms to be applied during the transformation phase. After the data is done being processed by Pig, it can be stored back in HDFS.

This chapter begins with an example Pig script. Pig and Pig Latin are then introduced and described in detail with examples. The chapter concludes with an explanation of how Pig's core features can be extended through the use of Python.

WordCount in Pig

Example 3-1 implements the WordCount algorithm in Pig. It assumes that a a data file, *input.txt*, is loaded in HDFS under */user/hduser/input*, and output will be placed in HDFS under */user/hduser/output*.

Example 3-1. pig/wordcount.pig

```
%default INPUT '/user/hduser/input/input.txt';
%default OUTPUT '/user/hduser/output';

-- Load the data from the file system into the relation records
records = LOAD '$INPUT';

-- Split each line of text and eliminate nesting
terms = FOREACH records GENERATE FLATTEN(TOKENIZE((chararray) $0))
AS word;

-- Group similar terms
grouped_terms = GROUP terms BY word;

-- Count the number of tuples in each group
word_counts = FOREACH grouped_terms GENERATE COUNT(terms), group;

-- Store the result
STORE word_counts INTO '$OUTPUT';
```

To execute the Pig script, simply call Pig from the command line and pass it the name of the script to run:

```
$ pig wordcount.pig
```

While the job is running, a lot of text will be printed to the console. Once the job is complete, a success message, similar to the one below, will be displayed:

```
2015-09-26  14:15:10,030  [main]  INFO    org.apache.pig.back-
end.hadoop.executionengine.mapReduceLayer.MapReduceLauncher  -
Success!
2015-09-26 14:15:10,049 [main] INFO  org.apache.pig.Main - Pig
script completed in 18 seconds and 514 milliseconds (18514 ms)
```

The results of the *wordcount.pig* script are displayed in Example 3-2 and can be found in HDFS under */user/hduser/output/pig_wordcount/part-r-00000*.

Example 3-2. /user/hduser/output/pig_wordcount/part-r-00000

```
2	be
1	the
3	jack
1	over
1	quick
1	jumped
1	nimble
1	candlestick
```

WordCount in Detail

This section describes each Pig Latin statement in the *wordcount.pig* script.

The first statement loads data from the filesystem and stores it in the relation `records`:

```
records = LOAD '/user/hduser/input/input.txt';
```

The second statement splits each line of text using the `TOKENIZE` function and eliminates nesting using the `FLATTEN` operator:

```
terms = FOREACH records GENERATE FLATTEN(TOKENIZE((chararray)
$0)) AS word;
```

The third statement uses the `GROUP` operator to group the tuples that have the same field:

```
grouped_terms = GROUP terms BY word;
```

The fourth statement iterates over all of the terms in each bag and uses the `COUNT` function to return the sum:

```
word_counts = FOREACH grouped_terms GENERATE COUNT(terms),
group;
```

The fifth and final statement stores the results in HDFS:

```
STORE word_counts INTO '/user/hduser/output/pig_wordcount'
```

Running Pig

Pig contains multiple modes that can be specified to configure how Pig scripts and Pig statements will be executed.

Execution Modes

Pig has two execution modes: local and MapReduce.

Running Pig in local mode only requires a single machine. Pig will run on the local host and access the local filesystem. To run Pig in local mode, use the `-x local` flag:

```
$ pig -x local ...
```

Running Pig in MapReduce mode requires access to a Hadoop cluster. MapReduce mode executes Pig statements and jobs on the cluster and accesses HDFS. To run Pig in MapReduce mode, simply call Pig from the command line or use the `-x mapreduce` flag:

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

Interactive Mode

Pig can be run interactively in the Grunt shell. To invoke the Grunt shell, simply call Pig from the command line and specify the desired execution mode. The following example starts the Grunt shell in local mode:

```
pig -x local
...
grunt>
```

Once the Grunt shell is initialized, Pig Latin statements can be entered and executed in an interactive manner. Running Pig interactively is a great way to learn Pig.

The following example reads */etc/passwd* and displays the usernames from within the Grunt shell:

```
grunt> A = LOAD '/etc/passwd' using PigStorage(':');
grunt> B = FOREACH A GENERATE $0 as username;
grunt> DUMP B;
```

Batch Mode

Batch mode allows Pig to execute Pig scripts in local or MapReduce mode.

The Pig Latin statements in Example 3-3 read a file named *passwd* and use the `STORE` operator to store the results in a directory called *user_id.out*. Before executing this script, ensure that */etc/passwd* is copied to the current working directory if Pig will be run in local mode, or to HDFS if Pig will be executed in MapReduce mode.

Example 3-3. pig/user_id.pig

```
A = LOAD 'passwd' using PigStorage(':');
B = FOREACH A GENERATE $0 as username;
STORE B INTO 'user_id.out';
```

Use the following command to execute the *user_id.pig* script on the local machine:

```
    $ pig -x local user_id.pig
```

Pig Latin

This section describes the basic concepts of the Pig Latin language, allowing those new to the language to understand and write basic Pig scripts. For a more comprehensive overview of the language, visit the Pig online documentation.

All of the examples in this section load and process data from the tab-delimited file, *resources/students* (Example 3-4).

Example 3-4. resources/students

```
john    21    3.89
sally    19    2.56
alice    22    3.76
doug    19    1.98
susan    26    3.25
```

Statements

Statements are the basic constructs used to process data in Pig. Each statement is an operator that takes a relation as an input, performs a transformation on that relation, and produces a relation as an output. Statements can span multiple lines, but all statements must end with a semicolon (;).

The general form of each Pig script is as follows:

1. A `LOAD` statement that reads the data from the filesystem
2. One or more statements to transform the data
3. A `DUMP` or `STORE` statement to view or store the results, respectively

Loading Data

The LOAD operator is used to load data from the system into Pig. The format of the LOAD operator is as follows:

```
LOAD 'data' [USING function] [AS schema];
```

Where *'data'* is the name of the file or directory, in quotes, to be loaded. If a directory name is not specified, all of the files within the directory are loaded.

The USING keyword is optional and is used to specify a function to parse the incoming data. If the USING keyword is omitted, the default loading function, PigStorage, is used. The default delimiter is the tab character ('\t').

The AS keyword allows a schema to be defined for the data being loaded. Schemas enable names and datatypes to be declared for individual fields. The following example defines a schema for the data being loaded from the file *input.txt*. If no schema is defined, the fields are not named and default to type bytearray.

```
A = LOAD 'students' AS (name:chararray, age:int);

DUMP A;
(john,21,3.89)
(sally,19,2.56)
(alice,22,3.76)
(doug,19,1.98)
(susan,26,3.25)
```

Transforming Data

Pig contains many operators that enable complex transforming of data. The most common operators are FILTER, FOREACH, and GROUP.

FILTER

The FILTER operator works on tuples or rows of data. It selects tuples from a relation based on a condition.

The following examples use the relation A that contains student data:

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

DUMP A;
(john,21,3.89)
(sally,19,2.56)
(alice,22,3.76)
```

```
    (doug,19,1.98)
    (susan,26,3.25)
```

The following example filters out any students under the age of 20, and stores the results in a relation R:

```
    R = FILTER A BY age >= 20;

    DUMP R;
    (john,21,3.89)
    (alice,22,3.76)
    (susan,26,3.25)
```

Condition statements can use the AND, OR, and NOT operators to create more complex FILTER statements. The following example filters out any students with an age less than 20 or a GPA less than or equal to 3.5, and stores the results in a relation R:

```
    R = FILTER A BY (age >= 20) AND (gpa > 3.5)

    DUMP R;
    (john,21,3.89)
    (alice,22,3.76)
```

FOREACH

While the FILTER operator works on rows of data, the FOREACH operator works on columns of data and is similar to the SELECT statement in SQL.

The following example uses the asterisk (*) to project all of the fields from relation A onto relation X:

```
    R = FOREACH A GENERATE *;

    DUMP R;
    (john,21,3.89)
    (sally,19,2.56)
    (alice,22,3.76)
    (doug,19,1.98)
    (susan,26,3.25)
```

The following example uses field names to project the age and gpa columns from relation A onto relation X:

```
    R = FOREACH A GENERATE age, gpa;

    DUMP R;
    (21,3.89)
    (19,2.56)
    (22,3.76)
```

```
(19,1.98)
(26,3.25)
```

GROUP

The GROUP operator groups together tuples that have the same group key into one or more relations.

The following example groups the student data by age and stores the result into relation B:

```
B = GROUP A BY age;

DUMP B;
(19,{(doug,19,1.98),(sally,19,2.56)})
(21,{(john,21,3.89)})
(22,{(alice,22,3.76)})
(26,{(susan,26,3.25)})
```

The result of a GROUP operation is a relation that has one tuple per group. This tuple has two fields: the first field is named group and is of the type of the grouped key; the second field is a bag that takes the name of the original relation. To clarify the structure of relation B, the DESCRIBE and ILLUSTRATE operations can be used:

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

ILLUSTRATE B;
------------------------------------------------------------
| B   | group:int   | A:bag{:tuple(name:chararray,         |
                      age:int,gpa:float)}                  |
-------------------------------------------------------------
|     | 19          | {(sally, 19, 2.56), (doug, 19, 1.98)}   |
-------------------------------------------------------------
```

Using the FOREACH operator, the fields in the previous relation, B, can be referred to by names group and A:

```
C = FOREACH B GENERATE group, A.name;

DUMP C;
(19,{(doug),(sally)})
(21,{(john)})
(22,{(alice)})
(26,{(susan)})
```

Storing Data

The STORE operator is used to execute previous Pig statements and store the results on the filesystem. The format of the STORE operator is as follows:

```
STORE alias INTO 'directory' [USING function];
```

Where alias is the name of the relation to store, and 'directory' is the name of the storage directory, in quotes. If the directory already exists, the STORE operation will fail. The output files will be named *part-nnnnn* and are written to the specified directory.

The USING keyword is optional and is used to specify a function to store the data. If the USING keyword is omitted, the default storage function, PigStorage, is used. The following example specifies the PigStorage function to store a file with pipe-delimited fields:

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

DUMP A;
(john,21,3.89)
(sally,19,2.56)
(alice,22,3.76)
(doug,19,1.98)
(susan,26,3.25)

STORE A INTO 'output' USING PigStorage('|');

CAT output;
john|21|3.89
sally|19|2.56
alice|22|3.76
doug|19|1.98
susan|26|3.25
```

The provided Pig Latin statements are great general-purpose computing constructs, but are not capable of expressing complex algorithms. The next section describes how to extend the functionality of Pig with Python.

Extending Pig with Python

Pig provides extensive support for custom processing through User Defined Functions (UDFs). Pig currently supports UDFs in six languages: Java, Jython, Python, JavaScript, Ruby, and Groovy.

When Pig executes, it automatically detects the usage of a UDF. To run Python UDFs, Pig invokes the Python command line and streams data in and out of it.

Registering a UDF

Before a Python UDF can be used in a Pig script, it must be registered so Pig knows where to look when the UDF is called. To register a Python UDF file, use Pig's `REGISTER` statement:

```
REGISTER 'udfs/myudf.py' USING streaming_python AS my_udf;
```

Once the UDF is registered, it can be called from within the Pig script:

```
relation = FOREACH data GENERATE my_udf.function(field);
```

In this example the UDF, referenced as `my_udf`, contains a function called `function`.

A Simple Python UDF

A simple Python UDF, located in *pig/udfs/my_first_udf.py*, that returns the integer value 1 each time it is called, is shown in Example 3-5.

Example 3-5. pig/udfs/my_first_udf.py

```
from pig_util import outputSchema

@outputSchema('value:int')
def return_one():
   """
   Return the integer value 1
   """
   return 1
```

Some important things to note in this Python script are the `from` statement on the first line, and the output decorator, `@outputSchema decorator`, on the third line. These lines enable the Python UDF to define an alias and datatype for the data being returned from the UDF.

The Pig script in Example 3-6 registers the Python UDF and calls the `return_one()` function in a `FOREACH` statement.

Example 3-6. pig/simple_udf.pig

```
REGISTER 'udfs/my_first_udf.py' USING streaming_python AS pyudfs;

A = LOAD '../resources/input.txt';
B = FOREACH A GENERATE pyudfs.return_one();
DUMP B;
```

When the Pig script is executed, it will generate an integer value 1 for each line in the input file. Use the following command to execute the script (sample output is shown as well):

```
$ pig -x local simple_udf.pig
...
(1)
(1)
(1)
```

String Manipulation

Python UDFs are an easy way of extending Pig's functionality and an easy way to transform and process data.

The Python UDF in Example 3-7 contains two functions: `reverse()` and `num_chars()`. The `reverse()` function takes in a chararray and returns the chararray in reverse order. The `num_chars()` function takes in a chararray and returns the number of characters in the chararray.

Example 3-7. pig/udfs/string_funcs.py

```
from pig_util import outputSchema

@outputSchema('word:chararray')
def reverse(word):
   """
   Return the reverse text of the provided word
   """
   return word[::-1]

@outputSchema('length:int')
def num_chars(word):
   """
   Return the length of the provided word
   """
   return len(word)
```

The Pig script in Example 3-8 loads a text file and applies the `reverse()` and `num_chars()` Python functions to each unique word.

Example 3-8. pig/playing_with_words.pig

```
REGISTER   'udfs/string_funcs.py'   USING   streaming_python   AS
string_udf;

-- Load the data from the file system
records = LOAD '../resources/input.txt';

-- Split each line of text and eliminate nesting
terms = FOREACH records GENERATE FLATTEN(TOKENIZE((chararray) $0))
AS word;

-- Group similar terms
grouped_terms = GROUP terms BY word;

-- Count the number of tuples in each group
unique_terms = FOREACH grouped_terms GENERATE group as word;

-- Calculate the number of characters in each term
term_length    =    FOREACH    unique_terms    GENERATE    word,
string_udf.num_chars(word) as length;

-- Display the terms and their length
DUMP term_length;

-- Reverse each word
reverse_terms    =    FOREACH    unique_terms    GENERATE    word,
string_udf.reverse(word) as reverse_word;

-- Display the terms and the reverse terms
DUMP reverse_terms;
```

Use the following command to execute the script (sample output shown):

```
$ pig -x local playing_with_words.pig
...
(be,2)
(the,3)
(jack,4)
(over,4)
(quick,5)
(jumped,6)
(nimble,6)
(candlestick,11)
...
(be,eb)
```

```
(the,eht)
(jack,kcaj)
(over,revo)
(quick,kciuq)
(jumped,depmuj)
(nimble,elbmin)
(candlestick,kcitseldnac)
```

Most Recent Movies

The following example uses movie data from the groupLens datasets and external libraries to calculate the 10 most recent movies.

The Python UDF in Example 3-9 contains two functions: `parse_title()` and `days_since_release()`. The `parse_title()` function uses Python's regular expression module to remove the release year from a movie's title. The `days_since_release()` function uses the `datetime` module to calculate the number of days between the current day and a movie's release date.

Example 3-9. pig/udfs/movies_udf.py

```
from pig_util import outputSchema
from datetime import datetime
import re


@outputSchema('title:chararray')
def parse_title(title):
    """
    Return the title without the year
    """
    return re.sub(r'\s*\(\d{4}\)','', title)

@outputSchema('days_since_release:int')
def days_since_release(date):
    """
    Calculate the number of days since the titles release
    """
    if date is None:
        return None

    today = datetime.today()
    release_date = datetime.strptime(date, '%d-%b-%Y')
    delta = today - release_date
    return delta.days
```

The Pig script in Example 3-10 uses the Python UDFs to determine the 10 most recent movies.

Example 3-10. pig/recent_movies.pig

```
REGISTER 'udfs/movies_udf.py' USING streaming_python AS movies_udf;

-- Load the data from the file system
records = LOAD '../resources/movies' USING PigStorage('|')
   AS (id:int, title:chararray, release_date:chararray);

-- Parse the titles and determine how many days since the release
date
titles = FOREACH records GENERATE movies_udf.parse_title(title),
movies_udf.days_since_release(release_date);

-- Order the movies by the time since release
most_recent = ORDER titles BY days_since_release ASC;

-- Get the ten most recent movies
top_ten = LIMIT most_recent 10;

-- Display the top ten most recent movies
DUMP top_ten;
```

The following command is used to execute the script (sample output shown):

```
$ pig -x local recent_movies.pig
...
(unknown,)
(Apt Pupil,6183)
(Mighty, The,6197)
(City of Angels,6386)
(Big One, The,6393)
(Lost in Space,6393)
(Mercury Rising,6393)
(Spanish Prisoner, The,6393)
(Hana-bi,6400)
(Object of My Affection, The,6400)
```

Chapter Summary

This chapter introduced and Pig and Pig Latin. It described the basic concepts of Pig Latin, allowing simple Pig scripts to be created and executed. It also introduced how to extend the functionality of Pig Latin with Python UDFs.