Practical 3

In this practical we will repeat the word count example that we used in the Practical 2 computer practical. You can make a copy of that file, create it again using gedit, or just call the file from the **prac2** folder if you still have it.

Here are the commands that create our new practical directory structure and copy our text file from prac2 to prac3.

```
$ mkdir /home/prac/prac3 /home/prac/prac3/input
```

```
$ cp /home/prac/prac2/input/file01.txt
/home/prac/prac3/input/test.txt
```

In the commands that follow, I'll be loading the file from the **prac3** folder, change this line accordingly if you are doing otherwise. Hopefully you'll notice how much more simple it is to solve the word count problem in pig, compared to Hadoop MapReduce.

For reference the contents of the text file are below.

```
We're the pride of South Australia
We're the mighty Adelaide Crows
We're courageous stronger faster
And respected by our foes
Admiration of the nation our determination shows
We're the pride of South Australia
We're the mighty Adelaide Crows
```

The word count problem in Pig, from local mode (interactive mode)

To get pig started we could simply type **pig** in the terminal. By default this runs Pig on HDFS with MapReduce execution. However, for our first test we'll run pig locally. To run Pig locally we need to specify '**pig -x local**' in the terminal, where the 'x' stands for 'execution'.

Run the below commands one after the other in the terminal. The first command will produce a number of lines of output and start the grunt shell. After that, each line should work without giving any output or error.

```
lines = LOAD '/home/prac/prac3/input/test.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN (TOKENIZE(line, ' ')) AS word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);
DUMP wordcount;
```

The output should be as we would expect from a word count program.

Have a look at the code and try to understand what each line is doing. If you're curious about any commands, find them in the documentation, we'll also cover some of them later in this practical.

The word count problem in Pig, from local mode, from a script (batch mode)

Type **quit** in the command line to exit the grunt shell. Make a prac3 directory with a src directory inside it as we normally would. Use gedit to create a file called wordcount.pig in the src directory. In the wordcount.pig file, type the above lines of code, then save and exit.

```
$ mkdir /home/prac/prac3/src
```

Start pig again using '**pig -x local**'. Then use the **exec** command to run your script as below.

```
grunt> exec /home/prac/prac3/src/wordcount.pig
```

Finally, there's one more way to run a pig script locally. Type **quit** to exit the grunt shell. We can run the script directly by calling it at the same time as we call pig.

```
$ pig -x local /home/prac/prac3/src/wordcount.pig
```

Again, this should give us the same result. And as a final aside, we can pipe the output from DUMP to a sort command just as we have already been doing if we choose to.

The word count problem in Pig, from MapReduce mode

It's pretty straight forward to complete the above two tasks in MapReduce mode. Do everything exactly the same, just exempt the '**-x local**' part of the call to pig. Repeat the above two tasks in MapReduce mode now. You might notice that this time the program output includes some 'Job Stats' relating to MapReduce.

In order to run Pig in MapReduce mode, you need to upload the test.txt file to HDFS:

```
$ hadoop fs -mkdir -p /user/prac/prac3/input
```

```
$ hadoop fs -put /home/prac/prac3/input/test.txt
/user/prac/prac3/input
```

And change the path to point to the file using the HDFS path:

```
lines = LOAD '/user/prac/prac3/input/test.txt' AS (line:chararray);
```

Getting the hang of Pig Latin

Now we can try other operators in Pig Latin.

For the rest of the practical we'll run pig in local mode just to get some practice writing in Pig Latin. Using gedit save the following text file in the **input** folder in **prac3** as **team.txt**.

```
Taylor,Walker,Fwd,13,Broken Hill,30
Rory,Sloane,Mid,9,Melbourne,30
Daniel,Talia,Def,12,Kilmore,28
Brad,Crouch,Mid,2,Ballarat,26
Matthew,Crouch,Mid,5,Ballarat,25
```

In the terminal load the **team.txt** file into a relation with alias **team** in Pig local mode.

```
team = LOAD '/home/prac/prac3/input/team.txt' USING PigStorage(',') AS
(firstName:chararray,lastName:chararray,position:chararray,playerNum:int,hometown:charar
ray,age:int);
```

Try using the four diagnostic operators to inspect the relation. The operators are **DUMP**, **DESCRIBE**, **EXPLAIN** and **ILLUSTRATE**. Use them directly in the terminal, one at a time.

Next, we will generate a relation for the first names of players.

```
first_name = FOREACH team GENERATE firstName;
```

As we're typing these commands into the terminal, it would be a good idea to use one of the diagnostic operators to check that we're getting the correct results. **DUMP** is the operator most often used for error checking.

This time we'll generate a relation for the first name and last name of each player. Note that we call the columns in two different ways.

```
first_and_last_name = FOREACH team GENERATE firstName,$1;
```

Now on your own, create a new relation, similar to **first_and_last_name**, except it should also include the players age.

We can use the **GROUP** command to group our data by a certain field. Let's say we want to group together all players of the same age. Try the following command and dump the output.

```
group_ages = GROUP team BY age;
```

You can see why the output looks this way. You can see it even better using the **DESCRIBE** or **ILLUSTRATE** operator. But it's not really what we were hoping for. Try grouping the relation that you just made previously by age, that should look a bit nicer.

You can apply functions like **AVG**, **MIN**, **MAX**, **SUM**, **COUNT**. For example, you can get the sum of each group.

```
player_sums = FOREACH group_ages GENERATE SUM(team.playerNum);
```

You can even group by multiple fields.

```
group_hometown_and_position = GROUP team BY (hometown,position);
```

Create another text file, this time called **power.txt** and save it in your **input** folder.

```
Tom,Rockliff,Mid,11
Travis,Boak,Mid,10
Charlie,Dixon,Fwd,22
Tom,Jonas,Def,1
```

We can use the **COGROUP** operator to group two relations. Assuming we already have the team relation, create a new relation called **power_team** as follows.

```
power_team = LOAD '/home/prac/prac3/input/power.txt' USING PigStorage(',') AS
(firstName:chararray,lastName:chararray,position:chararray,playerNum:int);
```

And join the two relations by the position field.

```
cogroup_data = COGROUP team BY position, power_team BY position;
```

We'll continue by introducing a new operator that works with two or more relations, the **JOIN** operator. If you've used SQL before you might be familiar with the term join, it's used to *join* fields from two or more relations.

There are three main types of joins in Pig. They are **self join**, **inner join** and **outer join**.

To perform self joins in Pig load the same data multiple times, under different aliases, to avoid naming conflicts. To do this we generally load the same data two or more times and join the new relations together. We already have a **team** relation, so lets create a copy and name it **team2**.

```
team2 = LOAD '/home/prac/prac3/input/team.txt' USING PigStorage(',') AS
(firstName:chararray,lastName:chararray,position:chararray,playerNum:int,hometown:charar
ray,age:int);
```

When we use the **JOIN** operator we need to specify which relations are being joined, and by which keys are they being joined.

```
joined_team = JOIN team BY playerNum, team2 BY playerNum;
```

Use the **DUMP** command to look at the new **joined_team** relation. In general, we use self-joins for instances where tables reference themselves, can you think of how this might be useful?

Inner join is probably the most frequently used join operator, it returns rows when there is a match in both tables. The output is a new relation where columns are based upon the keys that we chose for joining. You'll notice that the syntax for both self and inner joins is the same, they simply differ by what input we choose to provide.

Open the **team.txt** input file and add the following line to the bottom of the table, save the file and exit.

```
Jordan,Gallucci,Fwd,7,Melbourne,22
```

Try the following command and view the results.

```
crow_power = JOIN team BY age, power_team BY playerNum;
```

Can you tell what this command has achieved? Also, notice that we didn't have to re-define our **team** relation after altering the .txt file? Pig is pretty powerful huh!

There are three types of outer join; **left**, **right** and **full**. The thing that separates outer join from the previous two types is that outer join will always return all rows from at least one of the specified relations. Try all three types of joins as specified below. See if you can figure out why the joins are named as such, and get an idea of what their outputs will generally consist of.

Left outer join

```
outer_left = JOIN team BY age LEFT OUTER, power_team BY playerNum;
```

Right outer join

```
outer_right = JOIN team BY age RIGHT OUTER, power_team BY playerNum;
```

Full outer join

```
outer_full = JOIN team BY age FULL OUTER, power_team BY playerNum;
```

There are two more important operators that involve two or more relations. The **UNION** operator is used to merge the contents of two relations and the **SPLIT** operator is used to split the content of a relation into two or more new relations.

The **SPLIT** operator takes one relation as input and outputs two or more new relations, using an **IF** statement. For example, lets split our **team** relation into two new relations based on player age.

```
SPLIT team INTO team_young IF age<28, team_old IF age>27;
```

Dump both of the new relations to make sure that the operator behaved as you expected it to.

The **UNION** operator is used to join the contents of two relations. However, unlike the **JOIN** operator, the **UNION** operator won't produce empty fields. As a result, it requires that the two input relations share exactly the same fields.

Let's use the union operator to merge our young and old teams back into one team.

```
original_team = UNION team_young, team_old;
```

You'll notice that the reconstructed team is now essentially sorted by age, due to the merging of the two age-defined relations. We can sort the contents of a relation however we choose to by using the

ORDER operator. Remember that in Pig it's generally not good practice to simply overwrite old relations with new ones.

```
original_team_name_sort = ORDER original_team BY firstName;
```

There are a number of other useful operators in Pig, a full list of which can be found in the Pig documentation:

https://pig.apache.org/docs/r0.17.0/basic.html

We'll quickly introduce three more useful operators before finishing this practical.

SAMPLE

Used to take a random sample from a relation. The inputs are the alias of the relation and the proportion of the relation to be sampled (i.e. 0.01 = 1%).

```
sampled = SAMPLE team 0.5;
```

FILTER

Used to select specific tuples from a relation based on a given condition. For example:

```
midfielders = FILTER team BY position == 'Mid';
```

DISTINCT

Used to remove duplicate rows from relations i.e. only keep distinct rows.

```
cleaned_team = DISTINCT team;
```

Lastly, Pig has an extensive collection of built in functions, such as **tokenize**, **flatten** and **count**. You can find a full list of in-built Pig functions at the following link:

https://pig.apache.org/docs/r0.17.0/func.html