Assignment 4: Pig, Hive and Spark

Jean-Pierre Lozi

March 15, 2015

Provided files An archive that contains all files you will need for this assignment can be found at the following URL:
http://sfu.ca/~jlozi/cmpt732/assignment4.tar.gz
Download it and extract it (using "`tar -xvzf assignment4.tar.gz`", for instance).

1 Pig

Apache Pig makes it possible to write complex queries over big datasets using a language named *Pig Latin*. The queries are transparently compiled into MapReduce jobs and run with Hadoop.

While doing this exercise,[1] you are advised to read Chapter 11 ("Pig") from the book *Hadoop: The Definitive Guide*. The book is available in SFU's online library (http://www.lib.sfu.ca/). You will need more information than what is contained in that single chapter, though, don't hesitate to look for other resources online. In particular, you are invited to check the official documentation (http://pig.apache.org/docs/r0.12.0/). For questions in which a sample output is provided, make sure your output matches it exactly.

You will find two files named `movies_en.json` and `artists_en.json` in `assignment4.tar.gz`. They contain a small movie database in the JSON format. Here is an example record from `movies_en.json` (both files actually have one record per line, newlines are added here for readability):

```
{
"id": "movie:14",
"title": "Se7en",
"year": 1995,
"genre": "Crime",
"summary": " Two detectives, a rookie and a veteran, hunt a serial killer who uses the seven
deadly sins as his modus operandi.",
"country": "USA",
"director":     {
    "id": "artist:31",
    "last_name": "Fincher",
    "first_name": "David",
    "year_of_birth": "1962"
    },
"actors": [
```

[1]The first questions of this exercise are based on Philippe Rigaux' *Travaux Pratiques* in his course on Distributed Computing.

```
    {
    "id": "artist:18",
    "role": "Doe"
    },
    {
    "id": "artist:22",
    "role": "Somerset"
    },
    {
    "id": "artist:32",
    "role": "Mills"
    }
    ]
}
```

And here is an example record from `artists_en.json`:

```
{
"id": "artist:18",
"last_name": "Spacey",
"first_name": "Kevin",
"year_of_birth": "1959"
}
```

As you can see, `movies_en.json` contains the full names and years of birth of movie directors, but only the identifiers of actors. The full names and years of birth of all artists, as well as their identifier, are listed in `artists_en.json`.

Question 1 Connect to the Hadoop cluster and copy the two files `movies_en.json` and `artists_en.json` to the HDFS. Run Grunt, Pig's interactive shell, using:

```
$ pig
```

We want to load the two files into two relations named `movies` and `artists`, respectively. How can we load a JSON file into a relation? Start with the easier case: `artists_en.json`. We want a relation with four fields: `id`, `firstName`, `lastName`, and `yearOfBirth`. All fields will be of type `chararray`. Use `DESCRIBE` and `DUMP` to ensure your relation was created properly. When it is, load `movies_en.json` into a relation with the following fields (all fields except `director` and `actors` are of type `chararray`):

```
id, title, year, genre, summary, country,
director: (id, lastName, firstName, yearOfBirth),
actors: {(id, role)}
```

Question 2 We want to get rid of the descriptions of the movies to simplify the output of the next questions. How can we modify the `movies` relation to remove descriptions?

Question 3 Create a relation named `mUS_year` that groups the titles of American movies by year. `DUMP` it to ensure it's correct. Your results should look like the following:

```
[...]
(1988,{(Rain Man),(Die Hard)})
(1990,{(The Godfather: Part III),(Die Hard 2),(The Silence of the Lambs),(King of New York)})
(1992,{(Unforgiven),(Bad Lieutenant),(Reservoir Dogs)})
(1994,{(Pulp Fiction)})
[...]
```

Question 4 Create a relation named `mUS_director` that groups the titles of American movies by director. Your results should look like this:

```
((artist:1,Coppola,Sofia,1971),{(Marie Antoinette)})
((artist:3,Hitchcock,Alfred,1899),{(Psycho),(North by Northwest),(Rear Window),(Marnie),(The B
irds),(Vertigo)})
((artist:4,Scott,Ridley,1937),{(Alien),(Gladiator),(Blade Runner)})
((artist:6,Cameron,James,1954),{(The Terminator),(Titanic)})
[...]
```

Question 5 Create a relation named `mUS_actors` that contains `(movieId, actorId, role)` tuples. Each movie will appear in as many tuples as there are actors listed for that movie. Again, we only want to take American movies into account. Your results should look like the following:

```
(movie:1,artist:15,John Ferguson)
(movie:1,artist:16,Madeleine Elster)
(movie:2,artist:5,Ripley)
(movie:3,artist:109,Rose DeWitt Bukater)
[...]
```

Question 6 Create a relation named `mUS_actors_full` that associates the identifier of each American movie to the full description of each of its actors. Your results should look like the following:

```
(movie:67,artist:2,Marie Antoinette,Dunst,Kirsten,)
(movie:2,artist:5,Ripley,Weaver,Sigourney,1949)
(movie:5,artist:11,Sean Archer/Castor Troy,Travolta,John,1954)
(movie:17,artist:11,Vincent Vega,Travolta,John,1954)
[...]
```

Question 7 Create a relation named `mUS_full` that associates the full description of each American movie with the full description of all of its actors. First, do it using `JOIN`, which will work but will repeat all information about a movie for each of its actors:

```
(movie:1,
    {(movie:1,artist:16,Madeleine Elster,Novak,Kim,1925,
      movie:1,Vertigo,1958,Drama,USA,(artist:3,Hitchcock,Alfred,1899)),
     (movie:1,artist:15,John Ferguson,Stewart,James,1908,
      movie:1,Vertigo,1958,Drama,USA,(artist:3,Hitchcock,Alfred,1899))})
[...]
```

Do it again using `COGROUP`. You should obtain cleaner results:

```
(movie:1,
    {(movie:1,Vertigo,1958,Drama,USA,(artist:3,Hitchcock,Alfred,1899))},
    {(movie:1,artist:15,John Ferguson,Stewart,James,1908),
     (movie:1,artist:16,Madeleine Elster,Novak,Kim,1925)})
[...]
```

For both versions, clean up the results even more by avoiding to needlessly repeat the movie identifier.

Question 8 Create a relation named `mUS_artists_as_actors_directors` that lists, for each artist, the list of American movies which he directed and in which he played. The full name of the artist as well as their year of birth should be made available. You should get, for instance:

```
[...]
(artist:20,Eastwood,Clint,1930,artist:20,
    {(movie:8,Unforgiven,artist:20,William Munny),
     (movie:63,Million Dollar Baby,artist:20,Frankie Dunn),
     (movie:26,Absolute Power,artist:20,Luther Whitney)},
    {(movie:26,Absolute Power,artist:20),
     (movie:63,Million Dollar Baby,artist:20),
     (movie:8,Unforgiven,artist:20)})
[...]
```

Question 9 — *Filter UDF* We now want to list the title and director of all movies for which their director's first name and last name start with the same letter. To do so, you will need to write a *Filter User-Defined Function* (*Filter UDF*).

- Create a project in Eclipse with one class for your Filter UDF. What JAR(s) will you have to add? Which version? Where do you get them? What will you put in your `build.xml` file to automatically create and upload a JAR to the hadoop cluster?
- How do you make it so that Pig finds your JAR? How do you refer to your Filter UDF to use it?

You should get the following output:

```
(Jaws,(artist:45,Spielberg,Steven,1946))
(The Lost World: Jurassic Park,(artist:45,Spielberg,Steven,1946))
(Sound and Fury,(artist:138,Chabrol,Claude,1930))
```

Question 10 — *Eval UDF* We now want to list the title and the `imdb.com` URL of the three movies found in the previous question. To do so, you will write an *Eval User-Defined Function* (*Eval UDF*) that queries IMDB's search function for a movie. For the movie "The Matrix Reloaded", for instance, it will access the following webpage:

`http://www.imdb.com/find?q=The+Matrix+Reloaded`

Warning: make sure that you add a call to `Thread.sleep(500)` after you access this URL to ensure that you don't query the IMDB servers too often, which could result in `rcg-hadoop`'s IP getting blacklisted!

IMDB movie identifiers start with `tt`, followed by 7 digits. You can look for the first occurrence of such a string in the page (using a regular expression, for instance), and consider that it is the movie identifier of the first result. You can then construct the URL of the movie by using this identifier. For instance, for "The Matrix Reloaded", the corresponding IMDB URL will be:

`http://www.imdb.com/title/tt0234215/`

Your objective will be to obtain the following output:

```
(Jaws,http://www.imdb.com/title/tt0073195/)
(The Lost World: Jurassic Park,http://www.imdb.com/title/tt0119567/)
(Sound and Fury,http://www.imdb.com/title/tt0240912/)
```

Question 11 — *Load UDF* The file `artists_en.txt` from the `assignment4.tar.gz` archive contains the same information as the `artists_en.json` file, in a different format:

```
id=artist:1
last_name=Coppola
first_name=Sofia
year_of_birth=1971
--
id=artist:2
last_name=Dunst
first_name=Kirsten
year_of_birth=null
--
id=artist:3
last_name=Hitchcock
first_name=Alfred
year_of_birth=1899
...
```

Write a *Load UDF* that loads the file into a new relation. Make sure that this relation contains the same data as `artists` (you can just `DUMP` the two relations and compare the results manually, no need to write a program that will compare the relations).

Question 12 — *Store UDF* We would like to export our tables in the XML format. Could Pig easily produce a well-formed XML file? What is the issue? (You are not asked to write a Store UDF, just answer the question ;-))

Question 13 — *Aggregate/Algebraic UDF* We now want to find the description of the movie with the longest description. To this end, write an *Algebraic UDF* that finds the longest `chararray` from a bag of `chararray`s. Since we got rid of the movie descriptions in Question 2, you may need to reload the `movies` relation. The objective is to obtain the following result:

```
(The Birds, A wealthy San Francisco socialite pursues a potential boyfriend to a small Northern
California town that slowly takes a turn for the bizarre when birds of all kinds suddenly begin
to attack people there in increasing numbers and with increasing viciousness.)
```

Bonus points if you use an `Accumulator`!

2 Hive

Like Pig, *Apache Hive* makes it possible to write complex queries over big datasets, but it has the advantage of using *HiveQL*, a language that is very easy to learn for people who are used to work with databases since it is very similar to SQL.

While doing this exercise, you are advised to read Chapter 12 ("Hive") from the book *Hadoop: The Definitive Guide*. The book is available in SFU's online library (`http://www.lib.sfu.ca/`). Again, you are encouraged to look up resources online and to use your Google-fu to figure out how to solve questions.

Question 1 Similarly to what we did in the previous exercise, we want to create two tables, named `artists` and `movies`, that contain the data from files `artists_en.json` and `movies_en.json`, respectively. Unfortunately, Hive doesn't support JSON natively. Using the following *Serializer/Deserializer* (*SerDe*), create two tables with similar structures as the Pig relations from the first exercise that you will populate with the data from the JSON files:

`https://github.com/rcongiu/Hive-JSON-Serde`

Hint: you can keep the column names from the JSON files if it makes things easier.

Question 2 Take questions 3 to 11 from Exercise 1, and solve them again, using Hive instead of Pig this time. You may have to adapt things a bit, but always try to make it so that your output is as close as possible as what you got with Pig. If you don't manage to solve some of the questions with Hive, try to explain why. Good luck!

3 Spark

In this exercise,[2] we will experiment with *Apache Spark*. According to Wikipedia:

[2]This exercise is based on code written by Justin Fuston.

"Apache Spark is an open-source cluster computing framework originally developed in the AMPLab at UC Berkeley. In contrast to Hadoop's two-stage disk-based MapReduce paradigm, Spark's in-memory primitives provide performance up to 100 times faster for certain applications. By allowing user programs to load data into a cluster's memory and query it repeatedly, Spark is well suited to machine learning algorithms."

Have a look at the official documentation (`https://spark.apache.org/documentation.html`) and examples (`https://spark.apache.org/examples.html`). Again, you're encouraged to use Google!

We will use the WDC tick data from `http://www.kibot.com/Buy.aspx#free_historical_data`. You will find the 1.1GB `WDC_tickbidask.txt` file in `/cs/bigdata/datasets`. You will also find a 4.6MB file named `WDC_tickbidask_short.txt` that only contains data for the first few days; working with this file is much faster. Both files contain comma-separated values:

```
09/28/2009,09:10:37,35.6,35.29,35.75,150
09/28/2009,09:30:06,35.48,35.34,35.64,14900
09/28/2009,09:30:06,35.49,35.37,35.61,100
09/28/2009,09:30:10,35.4,35.4,35.57,100
09/28/2009,09:30:10,35.48,35.43,35.53,200
[...]
```

According to `http://www.kibot.com/Buy.aspx#free_historical_data`:

"The order of fields in the tick files is: Date,Time,Price,Bid,Ask,Size. Bid and Ask values are omitted in regular tick files and are aggregated from multiple exchanges and ECNs. For each trade, current best bid/ask values are recorded together with the transaction price and volume. Trade records are not aggregated and all transactions are included."

In `assignment4.tar.gz`, you will find a file named `stocktick.py` that contains the declaration of a Python class named `StockTick` that will be used to represent a record:

Contents of `stocktick.py`:

```python
class StockTick:
   def __init__(self, text_line=None, date="", time="", price=0.0, bid=0.0, ask=0.0, units=0):
      if text_line != None:
         tokens = text_line.split(",")
         self.date = tokens[0]
         self.time = tokens[1]
         try:
            self.price = float(tokens[2])
            self.bid = float(tokens[3])
            self.ask = float(tokens[4])
            self.units = int(tokens[5])
         except:
            self.price = 0.0
            self.bid = 0.0
            self.ask = 0.0
            self.units = 0
      else:
         self.date = date
         self.time = time
```

```
        self.price = price
        self.bid = bid
        self.ask = ask
        self.units = units
```

Question You will also find a file named `stock.py` that you are asked to complete. The objective is to compute various statistics on the tick file. Read all comments in the file carefully:

Contents of `stock.py`:

```
import sys
from stocktick import StockTick
from pyspark import SparkContext

def maxValuesReduce(a, b):
   ### TODO: Return a StockTick object with the maximum value between a and b for each one of its
   ### four fields (price, bid, ask, units)

def minValuesReduce(a, b):
   ### TODO: Return a StockTick object with the minimum value between a and b for each one of its
   ### four fields (price, bid, ask, units)

def generateSpreadsDailyKeys(tick):  ### TODO: Write Me (see below)
def generateSpreadsHourlyKeys(tick): ### TODO: Write Me (see below)
def spreadsSumReduce(a, b):          ### TODO: Write Me (see below)


if __name__ == "__main__":
   """
   Usage: stock
   """
   sc = SparkContext(appName="StockTick")

   # rawTickData is a Resilient Distributed Dataset (RDD)
   rawTickData = sc.textFile("/cs/bigdata/datasets/WDC_tickbidask.txt")

   tickData =  ### TODO: use map to convert each line into a StockTick object
   goodTicks = ### TODO: use filter to only keep records for which all fields are > 0

   ### TODO: store goodTicks in the in-memory cache

   numTicks =  ### TODO: count the number of lines in this RDD

   sumValues = ### TODO: use goodTicks.reduce(lambda a,b: StockTick(???)) to sum the price, bid,
               ### ask and unit fields
   maxValuesReduce = goodTicks.reduce(maxValuesReduce) ### TODO: write the maxValuesReduce function
   minValuesReduce = goodTicks.reduce(minValuesReduce) ### TODO: write the minValuesReduce function
```

```
avgUnits = sumValues.units / float(numTicks)
avgPrice = sumValues.price / float(numTicks)

print "Max units %i, avg units %f\n" % (maxValuesReduce.units, avgUnits)
print "Max price %f, min price %f, avg price %f\n"
       % (maxValuesReduce.price, minValuesReduce.price, avgPrice)

### Daily and hourly spreads
# Here is how the daily spread is computed. For each data point, the spread can be calculated
# using the following formula : (ask - bid) / 2 * (ask + bid)
# 1) We have a MapReduce phase that uses the generateSpreadsDailyKeys() function as an argument
#    to map(), and the spreadsSumReduce() function as an argument to reduce()
#    - The keys will be a unique date in the ISO 8601 format (so that sorting dates
#      alphabetically will sort them chronologically)
#    - The values will be tuples that contain adequates values to (1) only take one value into
#      account per second (which value is picked doesn’t matter), (2) sum the spreads for the
#      day, and (3) count the number of spread values that have been added.
# 2) We have a Map phase that computes thee average spread using (b) and (c)
# 3) A final Map phase formats the output by producing a string with the following format:
#    "<key (date)>, <average_spread>"
# 4) The output is written using .saveAsTextFile("WDC_daily")

avgDailySpreads = goodTicks.map(generateSpreadsDailyKeys).reduceByKey(spreadsSumReduce);   # (1)
#avgDailySpreads = avgDailySpreads.map(lambda a: ???)                                       # (2)
#avgDailySpreads = avgDailySpreads.sortByKey().map(lambda a: ???)                           # (3)
avgDailySpreads = avgDailySpreads.saveAsTextFile("WDC_daily")                               # (4)


# For the hourly spread you only need to change the key. How?

avgHourlySpreads = goodTicks.map(generateSpreadsHourlyKeys).reduceByKey(spreadsSumReduce); # (1)
#avgHourlySpreads = avgHourlySpreads.map(lambda a: ???)                                     # (2)
#avgHourlySpreads = avgHourlySpreads.sortByKey().map(lambda a: ???)                         # (3)
avgHourlySpreads = avgHourlySpreads.saveAsTextFile("WDC_hourly")                            # (4)

sc.stop()
```

When you are done editing the file, you can try running your program. To this end, you must first load Spark:

```
$ module load natlang
$ module load NL/HADOOP/SPARK/1.0.0
```

If you get an error message saying that `load` was not found, you can try to run `source /etc/profile` and run the command again. You can then run your script using:

```
$ spark-submit --py-files stocktick.py stock.py
```

With the `WDC_tickbidask_short.txt`, the output should be:

```
[...]
Max units 394500, avg units 189.727741
Max price 37.120000, min price 34.520000, avg price 36.110827
[...]
```

With the following contents for the files that contain the daily and hourly spread:

```
$ cat WDC_daily/part-00000
2009-09-28, 8.7461956441e-05
2009-09-29, 9.62263125455e-05
2009-09-30, 0.000103149255577
2009-10-01, 9.09907962315e-05
2009-10-02, 0.000109917689058

$ head WDC_hourly/part-00000
2009-09-28:09, 0.000133175829341
2009-09-28:10, 9.21066196523e-05
2009-09-28:11, 7.37038962707e-05
2009-09-28:12, 7.80151688106e-05
2009-09-28:13, 6.903254691e-05
2009-09-28:14, 7.01609887022e-05
2009-09-28:15, 6.81955681382e-05
2009-09-28:16, 0.00329163416378
2009-09-29:09, 0.000168793690811
2009-09-29:10, 0.000124472420975
```

What results do you get for `WDC_tickbidask.txt`?