HW: Parallel Data Processing

Abstract

The objective in this homework is to develop practical skills in parallel data processing for computational and data science. The focus is scaling data-intensive computations using functional parallel programming over distributed architectures.

Table of Contents

Guidelines

- The files needed to do the exercises are available for download from the course github repository.
- **AWS**
  - **First you should have followed the Guide "First Access to AWS"**. It is assumed you already have an AWS account and a key pair, and you are familiar with the AWS EC2 environment.
- **MapRed/Hadoop**
  - Both the mapper and the reducer should be python executable scripts that read the input from stdin (line by line) and emit the output to stdout.
  - Codes can be easily developed and debugged locally by executing the following linux command, before running them on AWS

    ```
    $ mapper.py < input.txt | sort | reducer.py
    ```

  - For the execution on a Hadoop cluster we will use Hadoop streaming on EMR AWS, which is a utility that comes with the Hadoop distribution that allows you to create and run MapReduce jobs with any executable or script as the mapper and/or the reducer. Its is strongly recommended to firstly follow the Guide "Hadoop Cluster on AWS" in order to get familiar with the Hadoop environment and know how to set up a cluster. Ensure you terminate the cluster after the homework.
- **Spark**
  - Programs should be developed in Python. Alternatively, you can implement Spark experiments with Scala.
  - Install a local version of Spark, by following the Guide "Install Spark Cluster in Local Mode", to test your programs on your local computer before running on a Spark cluster. As Spark's local mode is fully compatible with the cluster mode; programs written and tested locally can be run on a cluster with just a few additional steps.
  - For the execution on a Spark cluster we will use EMR on AWS. See the Guide "Spark Cluster on AWS" on how to set up a cluster, to login, to use the HDFS and to run Spark scripts. Ensure you terminate the cluster after the homework.
- **Submission**
  - Upload on Canvas de files specified in each assignment.
  - Exercises 3 and 4 are optional. You can choose to work on one of them and make a submission and only the best two will be taken for final grading.

1. MapReduce Programming

MapReduce is more of a framework than a tool. You have to fit your application into the execution pattern of map and reduce, which in some situations might be challenging. Design patterns can make application design and development easier allowing to solve problems in a reusable and general way. In these exercises we will practice some of the most frequent MapReduce programming patterns that have been described in class. Exercises from 1.1 to 1.5 can be tried locally, 1.6 requires a EMR AWS cluster.

1.1. Distributed Grep

Develop a distributed version of the grep tool to search words in very large documents. The output should be the numbers of the lines that match a given pattern. You can use as input file the input text used in the word count example described in class (eBook of Moby Dick).

Submission
- `P11_mapper.py`: Mapper script
- `P11_reducer.py`: Reducer script

1.2. Count URL Access Frequency

Develop a MapReduce job to find the frequency of each URL in a web server log. The output should be the URLs and their frequency. You can use as input file a sample Apache log file available at

http://www.monitorware.com/en/logsamples/apache.php

Submission
- `P12_mapper.py`: Mapper script
- `P12_reducer.py`: Reducer script

1.3. Stock Summary

Write a MapReduce job to calculate the average daily stock price at close of Alphabet Inc. (GOOG) per year since 2009. The output should be the year and the average price. You can download the daily historical data from Yahoo Finance at:

https://finance.yahoo.com/quote/GOOG/history?ltr=1

Submission
- `P13_mapper.py`: Mapper script
- `P13_reducer.py`: Reducer script

1.4. Movie Rating Data

GroupLens Research has collected and made available rating data sets from the MovieLens web site (http://movielens.org). The data sets were collected over various periods of time, depending on the size of the set. Before using these data sets, please review their README files for the usage licenses and other details.

https://grouplens.org/datasets/movielens/

Develop a MapReduce job to show movies with an average rating in the ranges:

(1) 1 or lower
(2) 2 or lower (but higher than 1)
(3) 3 or lower (but higher than 2)
(4) 4 or lower (but higher than 3)
(5) 5 or lower (but higher than 4)

The job should have two MapReduce phases. The output of the first phase should be the movies and their average rating. The output of the second phase should be ranges and the title of the movies.

Submission
- `P14a_mapper.py`: Mapper script first phase
- `P14a_reducer.py`: Reducer script first phase
- `P14b_mapper.py`: Mapper script second phase
- `P14b_reducer.py`: Reducer script second phase

1.5. Pig and Hive

Pig and Hive are higher-level abstractions of MapReduce. They provide an interface that has nothing to do with "map" or "reduce," but the systems interpret the higher-level language into a series of MapReduce jobs. Much like how a query planner in an RDBMS translates SQL into actual operations on data, Hive and Pig translate their respective languages into MapReduce operations.

Discuss in detail how these two tools could be used to simplify the development of the "Movie Rating Data" exercise.

Submission
- `P15.pdf`: Discussion

1.6. Parallel Execution

Choose one of the previous exercises, and using a large enough input file, execute the code on two EMR clusters with different number of nodes in order to evaluate the speed-up.

Submission
- `P16.pdf`: Description of the experiment and discussion

2. Spark Programming

In these exercises we will practice Spark programming with special emphasis on data parallel processing. Exercises from 2.1 to 2.5 can be tried locally, 2.6 requires a EMR AWS cluster.

2.1. Distributed Grep

Develop a Spark version of the grep tool to search words in very large documents. The output should be the numbers of the lines that match a given pattern. You can use as input file the input text used in the word count example described in class (eBook of Moby Dick).

Submission
- `P21_spark.py`: Spark script

2.2. Count URL Access Frequency

Develop a Spark script to find the frequency of each URL in a web server log. The output should be the URLs and their frequency. You can use as input file a sample Apache log file available at

http://www.monitorware.com/en/logsamples/apache.php

Submission
- `P22_spark.py`: Spark script

2.3. Stock Summary

Write a Spark script to calculate the average daily stock price at close of Alphabet Inc. (GOOG) per year since 2009. The output should be the year and the average price. You can download the daily historical data from Yahoo Finance at:

https://finance.yahoo.com/quote/GOOG/history?ltr=1

Submission

- `P23_spark.py`: Spark script

2.4. Movie Rating Data

GroupLens Research has collected and made available rating data sets from the MovieLens web site (http://movielens.org). The data sets were collected over various periods of time, depending on the size of the set. Before using these data sets, please review their README files for the usage licenses and other details.

https://grouplens.org/datasets/movielens/

Develop a Spark script to show movies with an average rating in the ranges:

(1) 1 or lower
(2) 2 or lower (but higher than 1)
(3) 3 or lower (but higher than 2)
(4) 4 or lower (but higher than 3)
(5) 5 or lower (but higher than 4)

Submission
- `P24_spark.py`: Spark script

2.5. Further Analysis of Movie Rating

Develop a Spark script to answer the following questions:

- What is the average rating given by each user?.
- What is the overall average rating?
- What is the average rating of each movie?.
- What is the average rating of each genres?.
- Which are the top 10?. Which are the top 10 each month?

Submission
- `P25.pdf`: Spark script

2.6. Parallel Execution

Choose one of the previous exercises, and using a large enough input file, execute the code on two different number of workers in order to evaluate the speed-up.

Submission
- `P26.pdf`: Description of the experiment and discussion

3. Meteorite Landing

The NASA's Open Data Portal hosts a comprehensive data set from The Meteoritical Society that contains information on all of the known meteorite landings. The Table consists of 34,513 meteorites and includes fields like the type of meteorite, the mass and the year.

https://data.nasa.gov/Space-Science/Meteorite-Landings/gh4g-9sfh

Write a MapReduce job and a Spark script to calculate the average mass per year of a type of meteorite specified as an argument.

Submission
- `P3_mapper.py`: Mapper script
- `P3_reducer.py`: Reducer script
- `P3_spark.py`: Spark script

4. Distributed Markov Shakespeare

In this exercise we will explore the application of function compositions, narrow and wide dependencies and stages in the DAG parallelism for a slightly involved distributed computation to gain further insights into this programming approach.

The task is to build a simple statistical language model for the writing style of Shakespeare. Your model should be a simple Markov Chain of order 2. You will use that model to generate novel sentences based on Shakespeare's original texts.

Proceed with the implementation in two stages, generating test cases in each, as follows which will receive equal credit:

1. Construct the Triple

- Works of Shakespeare is available as: s3://Harvard-CS205/Shakespeare/Shakespeare.txt
- Parse the text file into words
- Filter out any words that:
  – contain only numbers,
  – contain only letters which are capitalized,
  – contain only letters which are capitalized and end with a period
- Build an RDD for the words that should look like:
  ((*Word1*,*Word2*), [(*Word3a*,*Count3a*), (*Word3b*,*Count3b*), ...]) where Count3a is the number of times the phrase "*Word1 Word2 Word3*" appears in order in the original data. For example, for the phrase "*Now is...*", where *Word1* == "*Now*" and *Word2* == "*is*", you should get the following values:
  [("*a*", 1),
  ("*be*", 1),

("*his*", 1),
("*that*", 1),
("*this*", 1),
("*it*", 3),
("*Mortimer*", 1),
("*the*", 9),
("*my*", 2),
("*your*", 1),
("*he*", 1)]

- You will need `flatMap`, `zipWithIndex`, `join` and `groupByKey` in this part of the Implementation. test this part by printing 10 triples.

2. Generate the Text

- Choose a random (*Word1*,*Word2*) from the RDD to start the phrase, and then choose *Word3* based on the counts in the model. Your choice should be biased by the counts. So, for example, when the model sees a phrase starting with "*Now is...*" it should usually continue with "the", given the data above.
- Generate 10 random phrases from the model, each with 20 words. Include these sentences in your writeup.

Submission

- `P4_spark.py`: Completed python script to build the model and generate new phrases
- `P4_spark.pdf`: Random phrases generated by your model