Cloud Computing, Part II CST 75%, 2018/19
Coursework Specifications

The aim of the coursework is to deploy and manage a prototypical Cloud cluster running batch processing applications. To complete the project, you are required to execute the following steps. The marks for each step are also shown. You are required to:

1. Configure and launch a Kubernetes cluster on Amazon AWS. (5%)

2. Develop a WordLetterCount application using Spark APIs and execute this on a Spark cluster running on the AWS Kubernetes cluster as described and configured in Step 1. (10%)

3. Develop your own WordLetterCount batch processing application without using any Spark APIs. Execute your code on the Kubernetes cluster from Step 1. (15%)

4. Compare the performance of these two applications running concurrently on the Cloud cluster across various scenarios described below. (20%)

5. Design, build and evaluate your own resource scheduler to match certain performance goals given below. (40%)

6. Write an **individual** report to describe the development process of your project, the design of the scheduler, your individual contribution to the project and to present and discuss the performance evaluation results. (10%)

The coursework is structured in a way to emulate the process of producing a systems research paper. You will first deploy the execution environment (Step 1). You will then deploy an application using existing APIs (Step 2). You then develop an application using your own design (Step 3). In Step 4 you measure the performance of the two applications and compare their execution times. In Step 5a you propose a new resource scheduler based on the evaluation result from Step 4. In Step 5b you propose a new online scheduler based on your experiences of the whole project. Finally, in Step 6 you produce the coursework report.

Budget: Each student is given an Amazon AWS classroom account with $80.00 to spend for the coursework. Use the following link to get an estimate of your costings and closely monitor your balance during the coursework development: https://calculator.s3.amazonaws.com/index.html. For example, the monthly cost of using 10 Linux t2.small instances, on-demand, with 40 hours/week usage is $44.80. Such estimates should give you a good idea on how to spend your budget. Note that you are allowed to spent as much of your available budget as you need for the project. You will not be marked based on how much of this budget you spend.

Step 1: Kubernetes Cluster (5%)

Write a script to automatically configure and launch a Kubernetes cluster on Amazon AWS. The **reference architecture** must include 1 master node of instance type c4.large and 10 worker nodes of instance type t2.small.

For this step, you should provide a CLI interface with the following options:

```
0: Exit
1: Define a Kubernetes Cluster
     11: Review the cluster Definition
2: Launch the cluster on AWS
     21: Validate the cluster
     22: Deploy the Kubernetes web-dashboard
     23: Access the Kubernetes web-dashboard
3: View the cluster
     31: Get the admin password
     32: Get the admin service account token
4: Delete the cluster
Please enter your choice: [#]
```

Your script must be able to automatically perform all the configurations required in order to create the above described Kubernetes cluster on top of AWS. The different CLI (note: the term CLI here should not be mistaken with AWS CLI) options should present information on the following cluster operations:

1: Description of the Kubernetes cluster by specifying the number of nodes (11 nodes in total, 1 master and 10 workers), instance type (t2.small for the workers and c4.large for the master), AWS zone, etc.

- **11:** Review and edit of the Kubernetes cluster definition/configuration created in 1

2: Creation of the cluster described in 1 on AWS

- **2.1:** Verification that the cluster is created successfully

2.2: Deployment of a dashboard to access the cluster by the web-based dashboard

2.3: Connect to the web-based dashboard of the cluster

3: Browse the cluster information, including status of all nodes

3.1 and **3.2:** Extraction of the credentials required for 2.3

4: Deletion of the cluster and its information

Testing: This step will be tested using the provided CLI. Clusters of various sizes will be created and verified using the CLI provided.

Step 2: Spark Implementation of a WordLetterCount application on Kubernetes and AWS (10%)

Using the Spark APIs, develop an application that implements an extended version of the popular WordCount batch processing example. The application should count the number of occurrences of words and letters in the input document and categorise words/letters in three groups of *popular*, *rare* and *common* words/letters.

A word/letter is popular when it is among the highest 5% of words when sorted by their number of occurrences (i.e., top 5% of the words with the maximum frequencies) in the data file. A word/letter is rare when it is ranked amongst the lowest 5% of words by their frequency (i.e., bottom 5% of the words with minimum frequencies). A word/letter is common when it can be ranked amongst the middle 5% of words, sorted in a descending order by their word frequency (i.e., ranked in range [47.5%-52.5%] of the list of words, sorted by descending order by the number of occurrences for each word).

Assume that your code takes as input a very large text file, available in a given URL. You can write your code using Python or Java. Please note that PySpark applications (for Spark version 2.3.2) are currently not supported directly on Kubernetes. You can directly submit your Java-Spark application to AWS Kubernetes cluster by using the spark-submit script. But, if you prefer to write your code in Python, then you need to containerise you Python/Spark application, by creating a (Docker) image, publishing the image to a registry, and finally pull the image and deploy your application to Kubernetes.

CLI interface: Extend the CLI interface above to run your Spark WordLetterCount against an input file found in a given URL.

For example, the following options can be added to the menu created in step 1:

```
5: Run Spark WordLetterCount App URL
     51: View Spark App
     52: Show Output
```

Output: The output of your code should be stored in two database tables. In particular, create a database named: **Your-CRSid_CloudComputingCoursework** (e.g., jc325_CloudComputingCourrsework) on AWS (e.g using Amazon RDS) with the following tables:

Table1: **words_spark**
Table2: **letters_spark**

Every time you run your application, it should re-create the above tables, and store the results. You do not need to re-create the database from scratch every time you run your application.

The schemas of the tables are:

words_spark

rank	word	category	frequency

letters_spark

rank	letter	category	frequency

where,

rank: is an unique primary key auto-increasing index (i.e., 0,1,2, …) to words/letters found. Words/Letters are ranked by decreasing order by their frequency.

word, letter: is the word or letter found in the text

category: is one of the three *rare, popular,* or *common*

frequency: is the frequency of the words/letters in the total population

Notes:

- Words/letters must be stored in lowercase and in ascending order by their frequency.
- Identify words using punctuation marks.
- In your analysis ignore all words/letters with non-letter characters.
- The analysis for both words and letters should not be case sensitive.
- Tables **words** and **letters** only store results relevant to words/letters which fall into the **rare, popular** and **common** categories. You can ignore the rest of the words/letters. The order of categories stored in the tables should be: first **popular**, then **common** and finally **rare**.

Testing*: This step will be tested against various input files of different sizes. The output of your code as stored in the database tables will be compared against a reference Spark/MapReduce implementation.*

Step 3: Custom-built WordLetterCount on Cloud Cluster (15%)

Without using any Spark APIs develop your own implementation of the WordLetterCount application as described in **Step 2**. Run your code on the AWS Kubernetes cluster created in **Step 1**.

The implementation must include approaches for efficiently distributing and orchestrating containers and pods within cluster nodes. In particular, your code must be able to dynamically use less or more nodes.[1] You must follow the MapReduce programming model.[2] Your code should not implement any fault-tolerance features. You should only implement the following:

1. The map() and reduce() functions.
2. The input file should be read in fixed sized chunks. The chunk size should be given as an input parameter. If this is omitted then your program should use a default value. You can choose this value.
3. Your code should implement all the necessary I/O operations for reading from the input file, sending intermediate values from the mappers to reducers, and finally writing the results to output files.

CLI interface: Extend the CLI interface above to run your custom-built WordLetterCount against an input file found in a given URL. For example:

```
6: Run Custom-built WordLetterCount App URL chunk-size
     61: View Custom-built App
     62: Show Output
```

Output: The output of your code should be exactly as in the case of the Spark application. However, the tables should now be named as: **words_custom** and **letters_custom** within the database created in step2**.**

Note that, you can proceed to the next steps without implementing Step 3. In this case, you will need to run your Spark WordLetterCount application twice to replace the custom-built application. If you decide to proceed in this way you will lose 15% of the total coursework mark. You must still use the above database table names **words_spark** and **letters_spark** for one instance of the Spark application and **words_custom** and **letters_custom** for the second instance of the same application.

[1] The dynamic addition/removal of containers/pods will be needed for Step 5. However, you could start your implementation without this feature.

[2] Note that you are allowed to get inspiration from the Spark or other MapReduce implementations but you are not allowed to copy and paste any existing code.

***Testing**: This step will be tested against various input files of different sizes. The output of your code as stored in the database tables will be compared against a reference Spark/MapReduce implementation.*

Step 4: Performance Experiments (20%)

In this step you need to compare the performance of the previous two applications sharing the cluster of the 10 worker nodes according to the three scenarios described below. In this Step the resource allocation for the cluster is static and is given below. In all cases run your experiments for input file sizes of 200MB, 400MB and 500MB. The files will be given to you online at the course web page.

- `scenarioA:` The custom application uses 2 nodes and the Spark application 8 nodes.
- `scenarioB:` The custom application uses 5 nodes and the Spark application 5 nodes.
- `scenarioC:` The custom application uses 8 nodes and the Spark application 2 nodes.

Report the performance results of your experiments. Produce two graphs to show the performance of the applications (in terms of execution time) as a subject of the input parameters i.e., input file size and number of nodes allocated per application. In particular, for each application produce a grouped bar chart where on the x axis you have three points, one for each scenario. For each scenario you should report three bars, one for each input file size showing the execution time of the application (y axis).

Discuss your results on the report. Your discussion should comment on how the performance of the applications is affected given the different input sizes and the nodes' configurations. Compare the performance of the two applications. Discuss your conclusions. Record your results in the **step4_performance_results** table as shown below.

Table: **step4_performance_results** (18 rows in total)

exp_id	application	nodes	data	execution_time (in secs)
1	Spark	8	200	?
2	Spark	5	200	?
3	Spark	2	200	?
4	Spark	8	400	?
5	Spark	5	400	?
6	Spark	2	400	?
...	...	...	...	...
16	Custom	2	500	?
17	Custom	5	500	?
18	Custom	8	500	?

Extend the CLI interface above to run both applications against the three scenarios and different input sizes. Your interface (and hence code) should give the option to the user to select how many nodes to use for each application. The sum of the total nodes used should not be larger than 10.

Output: The output of this step should be reported in the project report. Detailed instructions are given below.

Step 5: Design, Build and Evaluate a New Resource Scheduler (40%)

In this final step you are required to design, build and evaluate a new resource scheduler to dynamically allocate worker nodes to the two applications for unknown sizes of input files. **The goal of the scheduler is to minimise the execution time for both and each one of the applications**. You should assume that the two applications begin their execution simultaneously. **The scheduler is allowed to control the execution time of the two applications only by adding or removing containers/pods/nodes to each of the applications.** Your setup should only use the Kubernetes cluster you created in Step 1. You are asked to provide two versions of the scheduler using the approaches described below.

Step 5a: Static Allocation. (20%) In this step, the scheduler should use a performance model derived from Step 4. The decisions of the scheduler (how many nodes to allocate for each application) should solely depend on the performance results from **Step 4.** Devise a performance model using the data from **Step 4**. Using this model, the scheduler should perform a **static allocation** of nodes to the two applications in order to minimise the execution time for both and each one of the applications.

Hint: the derived performance model should predict the execution time of an application given an input file size and the number of allocated nodes. Note that in Step 3 you measured the execution time of an application for three different file sizes and scenarios. For this step you might explore additional input values to derive a more accurate performance model. Should you proceed in this direction then, then clearly describe any additional steps you are taking in your report.

Step 5b: Dynamic Allocation. (20%) In this step you should design a scheduler to dynamically allocate nodes **at run-time** to the two applications in order to minimise the execution time for both and each one of the applications. In this case the scheduler should not consider the performance results from Step 4. Rather, it should **dynamically learn** the way the addition/removal of nodes affects the applications' performance and adjust their allocation at run-time.

CLI interface: Extend the CLI interface above to run both schedulers against different input sizes. At the end of this final step you must have a program/script with an appropriate CLI to automatically regenerate all the above steps, configurations and deployments as well as reproducing your results. Do not forget that your script must be able destroy all resources and clean the environment at the end. For all tasks in the project you are required to use CLI, APIs and scripting. All parts of the required configurations, developments, and evaluations for the project must be easily reproduceable through full automation of scripts and proper command line interfaces.

Output: The output of this step should be reported in the project report. Detailed instructions are given below.

Step 6: Conclusion and Report (10%)

Write an individual (per group member) project report. You should work independently on the report. However the coursework is designed to be a team-work project and both group members should equally work for each step of the coursework. The project report must contain the following sections:

1. Step 1: Spark, Kubernetes and AWS (up to 1 page)
   Outline your approach to Step 1.
2. Step 2: Spark Application (up to 1 page)
   Outline the application implementation. Which Spark APIs did you use?
3. Step 3: Custom-built Application (up to 3 pages)
   Clearly describe how you designed and implemented the application.
4. Step 4: Evaluation Results (up to 2 pages)
   Provide the graphs showing the performance results. Clearly describe these and discuss your conclusions on how the input file sizes and the number of allocated nodes affect the performance of the applications.
5. Step 5: New Resource Scheduler (up to 4 pages)
   a. *Describe the derived performance model and outline the design of the new scheduler. Present evaluation results to clearly show the performance of the scheduler according to the specifications.*
   b. *Describe the design of the new scheduler. Present evaluation results to clearly show the performance of the scheduler according to the specifications.*
6. Per Student Contribution (up to 1 page)
   a. *Clearly discuss your own, individual contribution in the project.*
   b. *Report the total budget spent for the project from both students' accounts.*
7. Conclusions (up to 1 page)
   Discuss the conclusions of your work. Discuss future work on the resource scheduler based on your experience.

Good luck!