1 Overview

Welcome to the Hadoop MapReduce programming assignment. You may choose to complete this machine problem in **either** Java or Python. Please choose the language that you prefer. It is highly recommended that you practice the **Tutorial: Docker installation** and **Tutorial: Introduction to Hadoop MapReduce** (under week 6) before beginning this assignment.

2 General Requirements

Please note that our grader runs on a docker container and is **NOT connected to the internet**. Therefore, **no additional libraries are allowed** for this assignment. Also, you will **NOT be allowed to create any file or folder outside the current folder** (that is, you can only create files and folders in the folder that your solutions are in).

3 Sorting

When you are to select top N items in a list, sorting is implicitly needed. Use the following steps to sort:

1. Sort the list ASCENDING based on **Firstly** count then **Secondly** on the value. If the value is string, sort lexicographically.

2. Select the bottom N items in the sorted list as Top items.

There is an implementation of this logic in the the third example of the Hadoop MapReduce Tutorial.

For example, to select top 5 items in the list {"A": 100, "B": 99, "C":98, "D": 97, "E": 96, "F": 96, "G":90}, first sort the items ASCENDING:

"G":90

"E": 96

"F": 96

"D": 97

"C":98

"B": 99

"A": 100

Then, the bottom 5 items are **A, B, C, D, F**.

Another example, to select 5 top items in the list {"43": 100, "12": 99, "44":98, "12": 97, "1": 96, "100": 96, "99":90}

"99":90

"1": 96

"100": 96

"12": 97

"44":98

"12": 99

"43": 100

Then, the bottom 5 items are **43, 12, 44, 12, 100**.

Java submission

If you choose to do this assignment in Python, skip this part and go to "Python submission"** part below.

1 Requirements

This assignment will be graded based on **JDK 8**

2 Procedures

Step 1: Run the provided sample Docker image (please follow "Tutorial: Docker installation" in week 6)

```
$ docker run -it sample_image.v1 bin/bash
root@f9922c3fe307:/#
```

Step 2: Download the project files

```
# git clone https://github.com/UIUC-CS498-Cloud/MP4_java.git
```

step 3: Change the current folder to

```
# cd MP4_java
```

step 4: Finish the exercises by editing the provided templates files. All you need to do is complete the parts marked with **TODO**. Please note that you are **NOT allowed to import any additional libraries.**

- Each exercise has a Java code template. All you must do is edit this file.
- Each exercise should be implemented in one file only. Multiple file implementation is not allowed.
- The code should be compiled and run on the sample Docker image.
- For partial credit: Only submit the files related to the exercise in a zip format (MP4.zip)
- If you need to create a new path in the **run** function, please always use **Path tmpPath = new Path("./tmp")**. Creating any other path may lead to Hadoop jar Error.

More information about these exercises is provided in the next section.

step 5: After you are done with the assignments, compress all your 5 java files (TopTitles.java, TopTitlesStatistics.java, OrphanPages.java, TopPolularLinks.java, PopularityLeague.java) into a .zip file named as "**MP4.zip**" (just like MP0, please don't put files into a folder and then compress this folder!!!). Submit your "**MP4.zip**".

Exercise A: **Top Titles**

In this exercise, you are going to implement a counter for words in Wikipedia titles and an application to find the top words used in these titles. To make the implementation easier, we have provided a boilerplate for this exercise in the following file: **TopTitles.java**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia titles (one in each line) as an input and first tokenizes them using provided delimiters, after that make the tokens lowercased, then removes common words from the provided stopwords. Next, your application selects top **10** words, and finally, saves the count for them in the output. **Use the method in section 3 Sorting to select top words.**

(optional) In TopTitles.java, you will find a class named as "TitleCountMap" and a class named as "TitleCountReduce". To complete these two classes, you may find the "**TitleCount.java**" template helpful. In this part, you should tokenize Wikipedia titles, make the tokens lowercased, remove common words, and save the count for all words in the output. You can test your TitleCount.java with:

```
# mkdir ./TitleCountClasses

# javac -cp $(hadoop classpath) TitleCount.java -d TitleCountClasses

# jar -cvf TitleCount.jar -C TitleCountClasses/ ./

# hadoop jar TitleCount.jar TitleCount -D stopwords=stopwords.txt -D

delimiters=delimiters.txt dataset/titles ./temp-output
```

The order of output is NOT important. Here is **a part** of the sample output:

```
list  1948
de    1684
new   1077
school      1065
county      1020
state 940
john  936
disambiguation    893
station     800
route 771
...
```

Because of the possible problems with special characters, we will not run TitleCount.java or check the output for this part. You don't need to submit this file. The template is there only to help you to understand TitleCountMap class and TitleCountReduce class in **TopTitles.java**.

The following is the sample command we will use to run the application:

```
# mkdir ./TopTitlesClasses

# javac -cp $(hadoop classpath) TopTitles.java -d TopTitlesClasses
```

```
# jar -cvf TopTitles.jar -C TopTitlesClasses/ ./

# hadoop jar TopTitles.jar TopTitles -D stopwords=stopwords.txt -D

delimiters=delimiters.txt dataset/titles ./A-output
```

If you want to check your output, run:

```
# cat A-output/part-r-00000
```

Here is the output of an application that selects top **5** words:

```
county  1020
school  1065
new     1077
de      1684
list    1948
```

The order of lines matters. TextArrayWritter and Pair classes are included in the template to optionally help you with your implementation.

Exercise B: **Top Title Statistics**

In this exercise, you are going to implement an application to find some statistics about the top words used in Wikipedia titles. To make the implementation easier, we have provided a boilerplate for this exercise in the following file: **TopTitleStatistics.java**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia titles (one in each line) as an input and first tokenizes them using provided delimiters, after that make the tokens lowercased, then removes common words from the provided stopwords list. Next, selects top **10** words. Use the method in section 3 Sorting to select top words.

Finally, the application saves the following statistics about the top words in the output: “Mean” count, “Sum” of all counts, “Minimum” and “Maximum” of counts, and “Variance” of the counts. All values should be **floored** to be an integer. For the sake of simplicity, simply use Integer in all calculations.

The following is the sample command we will use to run the application:

```
# mkdir ./TopTitleStatisticsClasses
```

```
# javac -cp $(hadoop classpath) TopTitleStatistics.java -d

TopTitleStatisticsClasses

# jar -cvf TopTitleStatistics.jar -C TopTitleStatisticsClasses/ ./

# hadoop jar TopTitleStatistics.jar TopTitleStatistics -D

stopwords=stopwords.txt -D delimiters=delimiters.txt dataset/titles

./B-output
```

If you want to check your output, run:

```
# cat B-output/part-r-00000
```

Here is the output of an application that selects top **5** words:

```
Mean    1358
Sum     6794
Min     1020
Max     1948
Var     146686
```

The order of lines matters. Var is calculated by this formula: Var(X)=E[(X−μ)^2]. TextArrayWritter and Pair classes are included in the template to optionally help you with your implementation.

Exercise C: **Orphan Pages**

In this exercise, you are going to implement an application to find orphan pages in Wikipedia. To make the implementation easier, we have provided a boilerplate for this exercise in the following file: **OrphanPages.java**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia links (not Wikipedia titles any more) as an input. All pages are represented by their ID numbers. Each line starts with a page ID, which is followed by a list of all the pages that the ID has a link to. The following is a sample line in the input:

```
2: 3 747213 1664968 1691047 4095634 5535664
```

In this sample, page 2 has links to page 3, 747213, and so on. Note that links are not necessarily two-way. The application should save the IDs of orphan pages in the output. Orphan pages are pages to which no other pages link.

The following is the sample command we will use to run the application:

```
# mkdir ./OrphanPagesClasses

# javac -cp $(hadoop classpath) OrphanPages.java -d OrphanPagesClasses

# jar -cvf OrphanPages.jar -C OrphanPagesClasses/ ./

# hadoop jar OrphanPages.jar OrphanPages dataset/links ./C-output
```

If you want to check your output, run:

```
# cat C-output/part-r-00000
```

If you want to check a **part** of your output, run:

```
# head C-output/part-r-00000
```

Here is a part of the output for this application:

```
2
14
24
51
68
83
103
107
149
158
```

The order of lines matters. All values are integers.

Exercise D: **Top Popular Links**

In this exercise, you are going to implement an application to find the most popular pages in Wikipedia. To make the implementation easier, we have provided a boilerplate for this exercise in the following file: **TopPopularLinks.java**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia links as an input. All pages are represented by their ID numbers. Each line starts with a page ID, which is followed by a list of all the pages that the ID has a link to. The following is a sample line in the input:

```
2: 3 747213 1664968 1691047 4095634 5535664
```

In this sample, page 2 has links to page 3, 747213, and so on. Note that links are not necessarily two-way. The application should save the IDs of top 10 popular pages as well as the number of links to them in the output. A page is popular if more pages are linked to it. **Use the method in section 3 Sorting to select top links.**

The following is the sample command we will use to run the application:

```
# mkdir ./TopPopularLinksClasses

# javac -cp $(hadoop classpath) TopPopularLinks.java -d

TopPopularLinksClasses

# jar -cvf TopPopularLinks.jar -C TopPopularLinksClasses/ ./

# hadoop jar TopPopularLinks.jar TopPopularLinks dataset/links

./D-output
```

If you want to check your output, run:

```
# cat D-output/part-r-00000
```

Here is the output of an application that selects top **5** popular links;

```
1921890 721
481424  729
84707   1060
5302153 1532
88822   1676
```

The order of lines matters. All values are integers. IntArrayWritter and Pair classes are included in the template to optionally help you with your implementation.

Exercise E: **Popularity League**

In this exercise, you are going to implement an application to find the most popular pages in Wikipedia. To make the implementation easier, we have provided a boilerplate for this exercise in the following file: **PopularityLeague.java**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia links as an input. All pages are represented by their ID numbers. Each line starts with a page ID, which is followed by a list of all the pages that the ID has a link to. The following is a sample line in the input:

```
2: 3 747213 1664968 1691047 4095634 5535664
```

In this sample, page 2 has links to page 3, 747213, and so on. Note that links are not necessarily two-way.

The **popularity** of a page is determined by the number of pages in the whole Wikipedia graph that link to that specific page. (Same number as **Exercise D**)

The application also takes a list of page IDs as an input (also called a **league list**). The goal of the application is to calculate the rank of pages in the league using their popularity.

The **rank** of the page is the number of pages in the**league** with strictly less (not equal) popularity than the original page.

The following is the sample command we will use to run the application:

```
# mkdir ./PopularityLeagueClasses

# javac -cp $(hadoop classpath) PopularityLeague.java -d

PopularityLeagueClasses

# jar -cvf PopularityLeague.jar -C PopularityLeagueClasses/ ./

# hadoop jar PopularityLeague.jar PopularityLeague -D

league=dataset/league.txt dataset/links ./E-output
```

If you want to check your output, run:

```
# cat E-output/part-r-00000
```

Here is the output with **League**={5300058,3294332,3078798,1804986,2370447,81615,3,1}):

```
5300058 2
3294332 4
3078798 4
2370447 1
1804986 6
81615   3
3       0
```

Here is the output
with **League**={88822,774931,4861926,1650573,66877,5115901,75323,4189215}):

```
5115901 4
4861926 0
4189215 6
1650573 2
774931  0
88822   7
75323   5
66877   3
```

The order matters. Note that we will use a different League file in our test. All values are integers. **IntArrayWriter** class is included in the template to optionally help you with your implementation.

Python submission

If you choose to do this assignment in Java, skip this part and go to **"Java submission" part above

1 Requirements

This assignment will be graded based on **Python 3**

2 Procedures

Step 1: Run the provided Docker image (please follow "Tutorial: Docker installation" in week 6)

```
$ docker run -it sample_image.v1 bin/bash
root@f9922c3fe307:/#
```

Step 2: Download the project files

```
# git clone https://github.com/UIUC-CS498-Cloud/MP4_python.git
```

step 3: Change the current folder to

```
# cd MP4_python/
```

step 4: Finish the exercises by editing the provided templates files. All you need to do is complete the parts marked with **TODO**. Please note that you are **NOT allowed to import any additional libraries.**

- Each exercise has one or more code template. All you must do is edit these files.
- The code will be run on the provided Docker image.
- For partial credit: Only submit the files related to the exercise in a zip format (MP4.zip). Example: For part A, only submit **TitleCountMapper.py, TitleCountReducer.py, TopTitlesMapper.py, TopTitlesReducer.py files to receive a partial credit.**

More information about these exercises is provided in the next section.

step 5: After you are done with the assignments, compress all your 14 python files (TitleCountMapper.py, TitleCountReducer.py, TopTitlesMapper.py, TopTitlesReducer.py, TopTitlesStatisticsMapper.py, TopTitlesStatisticsReducer.py, OrphanPagesMapper.py, OrphanPagesReducer.py, LinkCountMapper.py, LinkCountReducer.py, TopPolularLinksMapper.py, TopPolularLinksReducer.py, PopularityLeagueMapper.py, PopularityLeagueReducer.py) into "**MP4.zip**" (just like MP0, please don't put files into a folder and then compress this folder!!!).. Submit your "**MP4.zip**".

Exercise A: **Top Titles**

In this exercise, you are going to implement a counter for words in Wikipedia titles and an application to find the top words used in these titles. To make the implementation easier, we have provided a boilerplate for this exercise in the following files: **TitleCountMapper.py, TitleCountReducer.py, TopTitlesMapper.py, TopTitlesReducer.py**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia titles (one in each line) as an input and first tokenizes them using provided delimiters, after that make the tokens lowercased, then removes common words from the provided stopwords. Next, your application selects top **10** words, and finally, saves the count for them in the output. **Use the method in section 3 Sorting to select top words.**

You should first tokenize Wikipedia titles, make the tokens lowercased, remove common words, and save the count for all words in the output
with **TitleCountMapper.py** and **TitleCountReducer.py**.

You can test your output with:

```
# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2

.9.2.jar -files TitleCountMapper.py,TitleCountReducer.py -mapper

'TitleCountMapper.py stopwords.txt delimiters.txt' -reducer

'TitleCountReducer.py' -input dataset/titles/ -output

./preA-output_Python

# cat preA-output_Python/part-00000
```

The order of the output is NOT important. Here is **a part** of the output of this:

```
list  1948
de    1684
new   1077
school      1065
county      1020
state 940
john  936
disambiguation    893
station     800
route 771
...
```

Because of the possible problems with special characters, we will not check the output for this part. We will only check the top words and their counts.

The following is the sample command we will use to run the application:

```
# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2

.9.2.jar -files TitleCountMapper.py,TitleCountReducer.py -mapper
```

```
'TitleCountMapper.py stopwords.txt delimiters.txt' -reducer

'TitleCountReducer.py' -input dataset/titles/ -output

./preA-output_Python

# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2

.9.2.jar -files TopTitlesMapper.py,TopTitlesReducer.py -mapper

'TopTitlesMapper.py' -reducer 'TopTitlesReducer.py' -input

./preA-output_Python/ -output ./A-output_Python
```

If you want to check your output, run:

```
# cat A-output_Python/part-00000
```

Here is the output of an application that selects top **5** words:

```
county  1020
de      1684
list    1948
new     1077
school  1065
```

The order of lines matters. Please sort the output (key value) in alphabetic order. Also, make sure the key and value pair in final output are tab separated.

Exercise B: **Top Title Statistics**

In this exercise, you are going to implement an application to find some statistics about the top words used in Wikipedia titles. To make the implementation easier, we have provided a boilerplate for this exercise in the following files: **TopTitleStatisticsMapper.py, TopTitleStatisticsReducer.py**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your output from Exercise A will be used here. The application saves the following statistics about the top words in the output: “Mean” count, “Sum” of all counts, “Minimum” and “Maximum” of counts, and “Variance” of the counts. All values should be **floored** to be an integer. For the sake of simplicity, simply use Integer in all calculations.

The following is the sample command we will use to run the application:

```
# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2

.9.2.jar -files TopTitleStatisticsMapper.py,TopTitleStatisticsReducer.py

-mapper 'TopTitleStatisticsMapper.py' -reducer

'TopTitleStatisticsReducer.py' -input ./A-output_Python/ -output

./B-output_Python
```

If you want to check your output, run:

```
# cat B-output_Python/part-00000
```

Here is the output of an application that selects top **5** words:

```
Mean    1358
Sum     6794
Min     1020
Max     1948
Var     146686
```

Var is calculated by this formula: Var(X)=E[(X−μ)^2]. Make sure the stats and the corresponding results are tab separated.

Exercise C: **Orphan Pages**

In this exercise, you are going to implement an application to find orphan pages in Wikipedia. To make the implementation easier, we have provided a boilerplate for this exercise in the following files: **OrphanPagesMapper.py, OrphanPagesReducer.py**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia links (not Wikipedia titles anymore) as an input. All pages are represented by their ID numbers. Each line starts with a page ID, which is followed by a list of all the pages that the ID has a link to. The following is a sample line in the input:

```
2: 3 747213 1664968 1691047 4095634 5535664
```

In this sample, page 2 has links to page 3, 747213, and so on. Note that links are not necessarily two-way. The application should save the IDs of orphan pages in the output. Orphan pages are pages to which no other pages link.

The following is the sample command we will use to run the application:

```
# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2

.9.2.jar -files OrphanPagesMapper.py,OrphanPagesReducer.py -mapper

'OrphanPagesMapper.py' -reducer 'OrphanPagesReducer.py' -input

dataset/links/ -output ./C-output_Python
```

If you want to check your output, run:

```
# cat C-output_Python/part-00000
```

If you want to check a **part** of your output, run:

```
# head C-output_Python/part-00000
```

Here is a part of the output of this application:

```
100005
100019
100027
100028
100055
100072
100105
100110
100162
100219
```

The order of lines matters. Please sort your output (key value) in alphabetic order.

Exercise D: **Top Popular Links**

In this exercise, you are going to implement an application to find the most popular pages in Wikipedia. To make the implementation easier, we have provided a boilerplate for this exercise in the following files: **LinkCountMapper.py, LinkCountReducer.py, TopPopularLinksMapper.py, TopPopularLinksReducer.py**

If you have finished Exercise A, **LinkCountMapper.py, LinkCountReducer.py** should produce output similar to **TitleCountMapper.py, TitleCountReducer.py** in Exercise A. Instead of printing the count for each title, **LinkCountMapper.py** and **LinkCountReducer.py** should output the link count for each page P, or the number of pages are linked to P**.** Be careful about the format of output produced **by LinkCountMapper.py** and **LinkCountReducer.py.,** since the output of them is supposed to be the input of TopPopularLinksMapper.py . So if you use tab to separate the page ID and its link count, your TitleCountMapper.py should also consume its input in this way.

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia links as an input. All pages are represented by their ID numbers. Each line starts with a page ID, which is followed by a list of all the pages that the ID has a link to. The following is a sample line in the input:

```
2: 3 747213 1664968 1691047 4095634 5535664
```

In this sample, page 2 has links to page 3, 747213, and so on. Note that links are not necessarily two-way. The application should save the IDs of top **10** popular pages as well as the number of links to them in the output. A page is popular if more pages are linked to it. **Use the method in section 3 Sorting to select top links.**

The following is the sample command we will use to run the application:

```
# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2

.9.2.jar -files LinkCountMapper.py,LinkCountReducer.py -mapper

'LinkCountMapper.py' -reducer 'LinkCountReducer.py' -input

dataset/links/ -output ./linkCount-output_Python

# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2

.9.2.jar -files TopPopularLinksMapper.py,TopPopularLinksReducer.py

-mapper 'TopPopularLinksMapper.py' -reducer 'TopPopularLinksReducer.py'
```

```
-input ./linkCount-output_Python -output ./D-output_Python
```

If you want to check your output, run:

```
# cat D-output_Python/part-00000
```

Here is the output of an application that selects top **5** popular links:

```
1921890 721
481424  729
5302153 1532
84707   1060
88822   1676
```

The order of lines matters. Please sort your output (key value) in alphabetic order. Also, make sure the key and value pair in final output are tab separated.

Exercise E: **Popularity League**

In this exercise, you are going to implement an application to find the most popular pages in Wikipedia. To make the implementation easier, we have provided a boilerplate for this exercise in the following file: **PopularityLeagueMapper.py, PopularityLeagueReducer.py**

All you need to do is make the necessary changes to parts that are marked with **TODO**.

Your application takes a huge list of Wikipedia links as an input. All pages are represented by their ID numbers. Each line starts with a page ID, which is followed by a list of all the pages that the ID has a link to. The following is a sample line in the input:

```
2: 3 747213 1664968 1691047 4095634 5535664
```

In this sample, page 2 has links to page 3, 747213, and so on. Note that links are not necessarily two-way.

The **popularity** of a page is determined by the number of pages in the whole Wikipedia graph that link to that specific page. (Same number as **Exercise D**)

The application also takes a list of page IDs as an input (also called a **league list**). The goal of the application is to calculate the rank of pages in the league using their popularity.

The **rank** of the page is the number of pages in the **league** with strictly less (not equal) popularity than the original page.

The following is the sample command we will use to run the application:

```
# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2

.9.2.jar -files PopularityLeagueMapper.py,PopularityLeagueReducer.py

-mapper 'PopularityLeagueMapper.py dataset/league.txt' -reducer

'PopularityLeagueReducer.py' -input ./linkCount-output_Python -output

./E-output_Python
```

If you want to check your output, run:

```
# cat E-output_Python/part-00000
```

Here is the output with **League**={5300058,3294332,3078798,1804986,2370447,81615,3,1}):

```
81615   3
5300058 2
3294332 4
3078798 4
3       0
2370447 1
1804986 6
```

Here is the output
with **League**={88822,774931,4861926,1650573,66877,5115901,75323,4189215}):

```
88822   7
774931  0
75323   5
66877   3
5115901 4
4861926 0
4189215 6
1650573 2
```

The order matters. Please sort your output (key value) in **decreasing alphabetic order**.
Also, make sure the key and value pair in final output are tab separated.

Note that we will use a different League file in our test.