Assignment 1: MapReduce and Apache Spark

Submission Instructions

1. Put all your results and observation in a neatly written PDF file with the outputs. You may take screenshots of your execution to show in the output.
2. Name the PDF file as <roll*number*> CSL7110 _ Assignment.pdf
3. Put all your code in a GitHub repo and attach the link in the PDF file (step 1).
4. The accepted languages for code: MapReduce code in Java, and Spark code in Python (PySpark).
5. Submit on Google Classroom
6. Deadline: February 14, 2026.

Note: We strongly discourage copying code from online resources. Any such activity will lead to a 0 score in this assignment.

Total: 120 Marks (10 marks for each problem)

Apache Hadoop & MapReduce

- For this section, you will need to install Apache Hadoop on your system.
- Follow the link for Single Node Cluster Setup.
- There are many YouTube videos for more help. We encourage you to use Linux based system for the same.
- Additionally, you'll need some basic HDFS commands for putting the data-sets onto HDFS. Something like:

```
$ hdfs fs -mkdir <path>
$ hdfs fs -copyFromLocal etc..
```

More Help: Apache HDFS Commands, HDFS Commands Basics

1. Run and show the working of the `WordCount` example as shown on the Apache Hadoop website as well as discussed in class earlier. (No need to send code for this. Just output to show that it is working is enough).
2. Suppose we use an input file that contains the following lyrics from a famous song:

```
We’re up all night till the sun
We’re up all night to get some
```

```
We’re up all night for good fun
We’re up all night to get lucky
```

The input pairs for the Map phase will be the following:

```
(0, "We’re up all night to the sun")
(31, "We’re up all night to get some")
(63, "We’re up all night for good fun")
(95, "We’re up all night to get lucky")
```

The key is the byte offset starting from the beginning of the file. While we won’t need this value in Word Count, it is always passed to the Mapper by the Hadoop framework.

The byte offset is a number that can be large if there are many lines in the file.

- What will the output pairs look like?
- What will be the types of keys and values of the input and output pairs in the Map phase?

Remember that instead of standard Java data types (String, Int, etc.), Hadoop uses data types from the `org.apache.hadoop.io` package. Check Hadoop's API.

3. For the Reduce phase, some of the output pairs will be the following:

```
("up", 4)
("to", 3)
("get", 2)
("lucky", 1)
```

- What will the input pairs look like?
- What will be the types of keys and values of the input and output pairs in the Reduce phase?

4. See the `WordCount.java` code from the MapReduce Tutorial, and copy its contents to the corresponding file in your project. Find the definitions of the `Map` class and the `map()` function:

```
public static class Map extends Mapper {
@Override
public void map(/*?*/ key, /*?*/ value, Context context)
throws IOException, InterruptedException {
...
```

Use the data types you found in Question 1 to replace the `/*?*/` placeholders.

Similarly, find the definitions of the Reduce class and the `reduce()` function and replace the `/*?*/` placeholders with the data types found in Question 2.

You will also have to find which arguments to pass to `job.setOutputKeyClass()` and `job.setOutputValueClass()`.

5. Write the `map()` function. We want to make sure to disregard punctuation: to this end, you can use `String.replaceAll()`. In order to split lines into words, you can use a `StringTokenizer`.
6. Write the `reduce()` function. When you're done, make sure that compiling the project doesn't produce any errors.

Note
For the following sections (both Hadoop and Spark), you will need to download a dataset, available as D184MB. It is a collection of books from Project Gutenberg.

- It's a zip file containing books in text files. Unzip it, this will be later re-used in Spark Problems as well.
- For questions 7 to 9, use file `200.txt`.

---

7. We will now run `WordCount` on the `200.txt` file.

First, we have to copy that file to the HDFS. To do so, run the following command on the cluster:

```
$ hadoop fs -copyFromLocal /user/iitj/200.txt
```

It will copy `200.txt` to `/user/` on the HDFS. It's now time to see if your `WordCount` class works. On the cluster, run the following command in your project directory (say CSL7110):

```
$ hadoop jar WordCount.jar output/
```

If the code did not work, then you can edit your `WordCount.java` file again, recompile it, copy it again to the cluster, remove the `output/` directory from the HDFS (`hadoop fs -rm -r output`) and launch the command above again. When everything works, you can merge the output from the HDFS to a local file:

```
$ hadoop fs -getmerge output/ output.txt
```

Open `output.txt`, the results should look like this:

```
A 182
AA 16
AAN 5 …
```

8. In order to copy data from your local file system to the HDFS you used the following command:

```
$ hadoop fs -copyFromLocal /user/iitj/200.txt
```

In addition to the `-copyFromLocal` and `-copyToLocal` operations that are pretty self-explanatory, you can use basic UNIX file system commands on HDFS by prefixing them with `hadoop fs -`. So for instance, instead of ls, you would type:

```
$ hadoop fs -ls The output should look like this:
drwx------ - dsb hdfs 0 2014-10-09 23:00 .Trash
drwx------ - dsb hdfs 0 2014-10-09 14:55 .staging
drwxr-xr-x - dsb hdfs 0 2014-10-09 14:55 output
-rw-r--r-- 3 dsb hdfs 104857600 2014-10-09 13:01 200.txt
```

The result is similar to what you would see for a standard ls operation on a UNIX file system. The only difference here is the second column that shows the replication factor of the file. In this case, the file `200.txt` is replicated three times.

Why don't we have a replication factor for directories?

9. Edit `WordCount.java` to make it measure and display the total execution time of the job. Experiment with the `mapreduce.input.fileinputformat.split.maxsize` parameter. You can change its value using:

```
job.getConfiguration().setLong("mapreduce.input.fileinputformat.split.maxs
ize",);
```

How does changing the value of that parameter impact performance? Why?

Apache Spark

- For this section, you will need to install Apache Spark on your system.
- Follow the download link: Download Spark
- Both Spark and Hadoop require Java, which must have been installed already for successful execution of the previous section.
- For a better perspective of how Spark works, look at some examples.

Note: For these Spark problems, first load the provided Project Gutenberg dataset into a Spark DataFrame named `books_df` with the schema:

- `file_name` (string): The name of the text file (e.g., "10.txt").
- `text` (string): The raw text content of the book.

10. **Book Metadata Extraction and Analysis**

Task:

1. **Metadata Extraction:** Extract the following metadata from the `text` column of each book in the `books_df` DataFrame:
   - `title`
   - `release_date`
   - `language`
   - `encoding`
2. **Analysis:**
   - Calculate the number of books released each year.
   - Find the most common language in the dataset.
   - Determine the average length of book titles (in characters).

- Explain the regular expressions you used to extract the title, release date, language, and encoding. Discuss any challenges or limitations in using regular expressions for this task.
- What are some potential issues with the extracted metadata (e.g., inconsistencies, missing values)? How would you handle these issues in a real-world scenario?

11. **TF-IDF and Book Similarity**

Task: Calculate TF-IDF scores for words in each book and use them to determine book similarity based on cosine similarity.

1. **Preprocessing:**
   - Clean the `text` column in `books_df`: remove Project Gutenberg header/footer, convert to lowercase, remove punctuation, tokenize into words, and remove stop words.
2. **TF-IDF Calculation:**
   - Calculate the Term Frequency (TF) of each word in each book.
   - Calculate the Inverse Document Frequency (IDF) for each word across all books.
   - Compute the TF-IDF score for each word in each book (TF * IDF).
3. **Book Similarity:**
   - Represent each book as a vector of its TF-IDF scores.
   - Calculate the cosine similarity between all pairs of book vectors.
   - For a given book (e.g., `file_name` "10.txt"), identify the top 5 most similar books based on cosine similarity.

- Explain TF and IDF. Why is TF-IDF useful for weighting the importance of words in a document?
- How is cosine similarity calculated? Why is it appropriate for comparing documents represented as TF-IDF vectors?

- Discuss scalability challenges when calculating pairwise cosine similarity for many documents. How could Spark help address these challenges?

12. **Author Influence Network**

Task:
Construct a simple author influence network based on whether authors' books were released within a certain time window of each other. This is a simplified representation, as true influence is much more complex, but it demonstrates graph-like analysis with Spark.

1. **Preprocessing:**
   - Extract the `author` and `release_date` from the `text` of each book, similar to what you did in the first question of the previous set. You might need to refine your regular expressions or use more advanced techniques if the format varies significantly.
2. **Influence Network Construction:**
   - Define an "influence" relationship between two authors if one author released a book within *X* years of another author's release (e.g., *X* = 5 years). You can adjust *X* to explore different influence ranges.
   - Create an RDD or DataFrame representing the edges of this influence network. Each edge should be a tuple or row of the form `(author1, author2)`, indicating that `author1` potentially influenced `author2`.
3. **Analysis:**
   - Calculate the "in-degree" and "out-degree" of each author in the network. In-degree represents the number of authors who potentially influenced them, and out-degree represents the number of authors they potentially influenced.
   - Identify the top 5 authors with the highest in-degree and the top 5 authors with the highest out-degree.

- Discuss how you chose to represent the influence network in Spark (e.g., RDD of tuples, DataFrame). What are the advantages and disadvantages of your chosen representation?
- How does the choice of the time window (*X*) affect the structure of the influence network? What are the limitations of this simplified definition of influence?
- How well would your approach scale to a much larger dataset with millions of books and authors? What optimizations could you consider?