Distributed Systems S24 - Assignment 2

February 14, 2024

1 Instructions and Important Notes

The assignment consists of three problems. However, the first one (Inverted Index) is an **optional problem** meaning that it will **not be graded**. We've added it because we think it's a nice example to get you started with Hadoop MapReduce. Among the remaining two problems, you are required to solve **both** the problems.

The assignment uses the Hadoop streaming utility which is based on the Map-Reduce framework. So the following links to the documentation will help you get started:

- Hadoop Map-Reduce
- Hadoop Streaming

To install Hadoop Map-Reduce on your system, you can choose to follow the guide present here. However, we recommend installing it on Docker using a repo like this.

1.1 Submission Instruction

You may use as many mappers and reducers as your solution to each questions requires. You must submit (at least) four files per question.

- `mapper`
- `reducer` - Make sure to use **3 reducer** tasks for every MapReduce job run (you can do this by specifying the parameter `mapred.reduce.tasks`). This will create three files named part-00000 through part-00002 in the output directory. Leave them be. We'll handle their aggregation.
- `Makefile` - A Makefile with the following command,
  - `make build` - Should perform any necessary compilation if you have written your mapper, reducer or runner-script in a language that requires compilation. We will run this command before the `runner-script` is invoked.
- `runner-script` - This will be the only file run by us to evaluate the assignment. The script should take the following parameters as command line arguments in this exact order
  - `<hadoop streaming jar location>` This is the location of the streaming JAR to be used in the hadoop streaming command (Refer to the Hadoop Streaming documentation linked above to understand how this is used).

- `<input directory location on local FS>` This entire directory needs to be copied over to the location specified by the next parameter. This directory will contain the input file(s).
- `<input directory location on HDFS>` This is the HDFS directory where the input file (on local FS) must be copied to. In our testing, this directory is guaranteed to not exist before the script is run.
- `<output directory location on HDFS>` The final output of the Map-Reduce task must be placed in this directory. It is guaranteed that when we test your assignment, this output directory does not exist prior to running the script. Since we want you to use multiple reduce tasks, the order of lines in your output does not matter and you are not expected to merge the outputs of different reduce tasks. Your outputs would be merged using

  ```
  hdfs dfs -cat <output_dir>/* | sort
  ```

  Further, if you want to run multiple MapReduce jobs, feel free to name your intermediate output directories as `<output directory location on HDFS>0`, `<output directory location on HDFS>1` and so on. For example, if the given parameter was "`out`" then, you could create intermediate output directories called "`out0`", "`out1`", and so on. However, your final output should be in the directory specified by the parameter (in this case, "`out`") As an aside, you can specify a directory of files as input to the MapReduce Job, and the framework will ensure that the contents of each file is sent to a mapper.

For example, your script may be run with the following arguments:

```
./runner-script /opt/hadoop-3.2.1/share/hadoop/tools/lib/hadoop-streaming-3.2.1.jar
./input/ in out
```

The script must handle the following tasks:

- Copying over the local input directory onto HDFS using `hdfs dfs -put ...` command.
- Calling the Streaming MapReduce Job with the mapper and reducer as required your solution. You may assume that the mapper(s) and reducer(s) are in the current directory from which the script is run.

If you use a language that requires compilation (like C/C++) for the mapper/reducer then name your source code as `<file_name>.c` and the executables produced after running `make build` as the ones specified in the list above).

The mapper/reducer can be written in any language of your choice. So it is mandatory to add a shebang line at the beginning of the file. In the case of Python, add `#!/usr/bin/env python3`, for example.

1.1.1 Multiple Mappers/Reducers and/or Combiners

A typical way to handle complex problems in the Map-Reduce framework is to use multiple mappers and/or reducers. For example, we could break a problem down into two simpler stages A and B and write a mapper and reducer for each stage. Then the input would be run through the mapper and reducer for stage A, followed by the mapper and reducer for stage B.

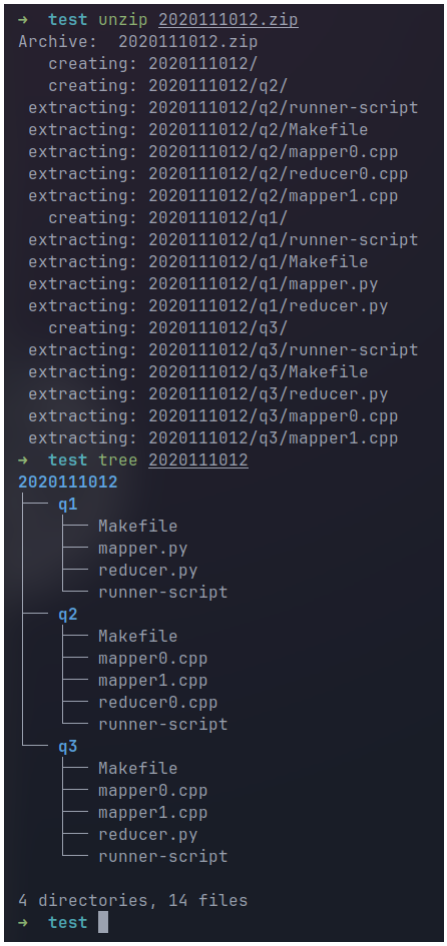
For the problems in this assignment, you are allowed (but not required) to use multiple mappers or reducers. In that case, the Makefile should perform any necessary compilation steps for those files as well. In such a case, please name the executables as `mapper0`, `mapper1` and so on (`reducer0` and onwards similarly) and correspondingly for the source file names.

It is optional to use combiners in this assignment and won't play a role in its evaluation. Generally, however, combiners can offer speed-up for your program by reducing the volume of data shuffled, so do consider incorporating them.

1.1.2 Example directory structures

You must submit a zip file named `<roll-number>.zip` which contains a directory named `<roll-number>`. This directory must contain the code for each question separated into directories `q1`, `q2`, and `q3` for questions 1, 2 and 3 respectively. However, the code inside the folder `q1` will not be graded.

The following image denotes the structure of an example zip file you are expected to submit. Please ensure that you stick to this directory structure since the evaluations for this assignment will be automated. Deviating from this submission format may incur a penalty.



2 Problems

2.1 Inverted Index

Given a set of documents, create an inverted index, which is a dictionary associating each word with a list of document identifiers where that word appears. The input is a 2-element list: [document_id, text], where document_id is a string representing a document identifier, and text is a string representing the content of the document. The text consists of space-separated words. The objective is to output the inverted index, where each word is associated with all its unique document identifiers.

Input Format

- Each line contains two tab-separated elements.
- The first element is a string representing the document id.
- The second element represents the text in that document, consisting of space-separated words.

```
doc1    I love pasta
doc2    Italy is a great country having great Pasta
doc3    I love Italy because Italy has good Pasta
```

Output format

In the output, each line starts with a word followed by a tab and a list of space-separated document ids associated with that word. The output needs to be sorted by the key (which is the word here)

```
a   doc2
because doc3
country doc2
good    doc3
great   doc2
has doc3
having  doc2
I doc1  doc3
is  doc2
Italy   doc2 doc3
love    doc1 doc3
Pasta   doc2 doc3
pasta   doc1
```

2.2 Friendships

An ideal friendship is commutative. If I consider you as my friend, then I know that you consider me as your friend.

There are **N** people in IIITH currently. You are given information about the existing friendships. You wonder for each pair of people, who are the mutual friends.

Formally, you are given input where each line contains two space-separated integers u v, such that $1 \leq u < v \leq N$, indicating that u, v are friends. This means that u is a friend of v and vice-versa. It is guaranteed that there won't be more than $\binom{N}{2}$ number of input lines.

For each pair of people x, y (with $x < y$) having atleast one mutual friend, you are required to output one line of the form `x y<tab-character><space-separated list of one or more mutual friends of x and y in sorted order>`.

Note that the order of lines in the output doesn't matter (since there are multiple reducer tasks). However, in a given line, the value field (i.e. the list of mutual friends) must be sorted.

2.2.1 Sample test-case

Input:

```
1 2
1 3
2 3
1 5
3 5
1 6
2 6
3 4
2 5
```

Output

```
1 2    3 5 6
1 3    2 5
1 4    3
2 6    1
3 5    1 2
3 6    1 2
4 5    3
5 6    1 2
1 5    2 3
1 6    2
2 3    1 5
2 4    3
2 5    1 3
```

2.3 Finding the Shortest Path to All Nodes

Consider a large-scale graph represented as an adjacency list, where each node has a unique identifier in the form of a positive integer and is connected to other nodes with undirected, unweighted edges (that is, each edge has a weight of 1). Your task is to output the shortest distance from node 1 to each node that is at most a distance of 100 from the node 1 using Map-Reduce.

The input will be an adjacency list of the graph. So for each node x in the graph, there will be a line of input of the format `x<tab-character><list of neighbours of node x>`.

Note that the graph is undirected; this means that if node A appears as a neighbour of B, then it is guaranteed that B also appears as a neighbour of A.

For each node u that is at a distance of at most 100 from node 1, output a line of the form `u<tab-character><shortest-distance-from-node-1>`

2.3.1 Sample test-case

Input:

```
1	2 3 6
2	1 3 4
3	1 2 5
4	2 5
5	3 4
6	1
```

Output

```
1	0
2	1
4	2
5	2
6	1
3	1
```