Hadoop MapReduce (Streaming) Tutorial ft. an example

Wordcount

Problem statement

We are given a bunch of lines that reads `<document_ID><tab character><space separate string of words representing the document text>`. We want to output the frequency count of each word in the document. That is, we want to output lines that read `<word><tab character><count of number of times the word occurs across all unique documents>`

Notice that here, we'll assume that document IDs may repeat and that different lines having the same document ID indicate that the corresponding texts belong to the same document. In such a case, if a word appears twice in the same document, then we'll count it a single time.

MapReduce idea

We now want to try to solve this problem using the MapReduce framework. For this, all we have to do is think about how a mapper would process the input and how its output would be processed by a reducer.

The output of the reducer is clear; we want it to output the frequency count for each word. To distinguish between a given word coming from two different documents or the same document, we'll also have to keep track of the document ID from which the word was obtained. Using this, we can remove duplicates, if any, because duplicate document IDs for a given word would indicate that the word occurred more than once in the same document, and we want to count such a word only once per document.

So, we want to keep track of `<word> <set of document IDs it appeared in>` so that, in the end, we can output `<word><tab character><size of the set>` (Note that a set by definition contains only unique elements).

Since a reducer is simply given all values for a given key, the mapper output should be one that allows us to aggregate the above info. It can then be deduced that the mapper should output `<word><tab character><document ID>` pairs. Given such pairs for the same word, the reducer simply inserts all of the document IDs into a set and outputs the size.

One thing to keep in mind is that usually (and almost always) multiple mapper/reducer tasks are launched. This means that each mapper and reducer would only get a fraction of what is supposed to be processed. The framework does guarantee that a line in its entirety would be given to one mapper/reducer. An implication of this is that the **mapper/reducer should be written in such a way that it functions correctly even when given a subset of the lines of input.**

Implementation

> The assignment requires you to submit for each question, a folder containing a runner script, Makefile, mapper(s) and reducer(s). The exact specifications are explained in the assignment document.

We'll start with the mapper and reducer since we already know what they should contain.

Mapper

```
#!/usr/bin/env python3
"""mapper.py"""
import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    doc_id, text = line.split('\t')
    # split the text into words
    words = text.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited;
        print('%s\t%s' % (word, doc_id))
```

Since we're working with Hadoop streaming, the input is read from stdin, and the output is written to stdout. It is important to note that the first line contains a shebang, allowing the file to run appropriately as an executable.

Notice how even if all lines go to different mappers, they still perform correctly with the reduced context.

Reducer

```
#!/usr/bin/env python3
"""reducer.py"""

import sys

current_word = None
current_set = set()
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, doc_id = line.split('\t', 1)

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_set.add(doc_id)
    else:
        if current_word:
```

```
            # write result to STDOUT
            print('%s\t%s' % (current_word, len(current_set)))
        current_set = set([doc_id])
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print('%s\t%s' % (current_word, len(current_set)))
```

The `reducer` has a shebang line similar to the mapper and also works with the standard input and outputs. The framework guarantees that the output to the reducer would be sorted and that all values that belong to one key would be processed by the same reducer. So, we process the input as discussed above and output the final set size for each word.

Since the input to the reducer is sorted, the program would also output the words in a sorted manner. However, if there are multiple reducer tasks, it is only guaranteed that each reducer will produce its own output to be sorted. Across reducers, the output would not be sorted.

Runner script

The runner script below is written as a shell script. We now need to call the MapReduce job on the input, which needs to be put on `hdfs` .

```
#!/usr/bin/bash

STREAM_JAR=$1
LOCAL_INP=$2
HDFS_INP=$3
HDFS_OUT=$4
FILES=$5

# put files on hdfs
hdfs dfs -put ${LOCAL_INP} ${HDFS_INP}q2_input
hadoop jar $STREAM_JAR -input ${HDFS_INP}q2_input -output $HDFS_OUT -mapper
${FILES}mapper -file ${FILES}mapper -reducer ${FILES}reducer -file
${FILES}reducer
```

Note that the arguments in this program are different from the ones specified in the assignment. You need to implement the program to take in the ones specified in the assignment.

The two main commands run are `hdfs dfs -put <local address> <hdfs address>` which copies over file/folder in the local directory and "puts" it onto the HDFS. Then, the streaming job is called with the appropriate arguments. This file, like the others contains a shebang.

Running the application

> For this example, since the our programs don't require a compilation step, the `Makefile` present in the current directory would have no actions to perform for the build command. But, it is important **to have the Makefile in the directory with the appropriate command (even if it doesn't actually perform any actions)**

The following would be steps followed to run the MapReduce job.

1. Exec into the `namenode` container
2. Either copy over the files into the container or bind a directory.
3. Run `make build`
4. Run the `runner_script` with the appropriate arguments. Here, that would look like `./runner_script /opt/hadoop-3.2.1/share/hadoop/tools/lib/hadoop-streaming-3.2.1.jar input.txt input/ output/ ./` . But, the arguments specified in the assignment is different to this. Note that if you are using the docker installation, then the streaming jar would be present in the location mentioned above. If not, supply the appropriate location in order to run the application.

Output

Say that `input.txt` contained the following input (Note that tab characters may not display accurately over Notion pages/pdf)

```
doc1    This ebook is for the use of anyone anywhere in the United States and
doc2    most other parts of the world at no cost and with almost no restrictions
doc1    whatsoever. You may copy it, give it away or re-use it under the terms
doc2    of the Project Gutenberg License included with this ebook or online
doc3    at www.gutenberg.org. If you are not located in the United States,
doc4    you will have to check the laws of the country where you are located
doc5    before using this eBook.
```

Upon running the script, the MapReduce job is called, and if all goes well, you should see no errors being output. You'll also see the number of map and reduce tasks launched under the section "Job Counters".

We instructed the output to be stored in a directory called `output` . So, a directory with that name would be created on HDFS and the output files placed in it.

Output of `hdfs dfs -ls`

```
Found 2 items
drwxr-xr-x   - root supergroup          0 2024-02-07 08:00 input
drwxr-xr-x   - root supergroup          0 2024-02-07 08:00 output
```

Output of `hdfs dfs -ls output`

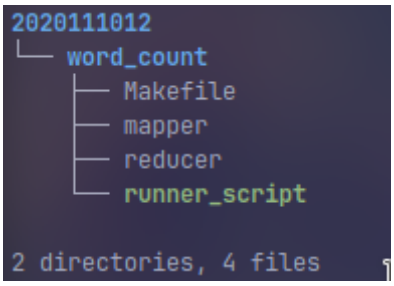
```
Found 2 items
-rw-r--r--   3 root supergroup          0 2024-02-07 08:00 output/_SUCCESS
-rw-r--r--   3 root supergroup        460 2024-02-07 08:00 output/part-00000
```

Since there was only one reduce job called, there is a single output file named `part-00000` . For the assignment, your program is expected to work with multiple reduce jobs as well. But, in such a case, it is not expected to concatenate all the outputs nor to sort them across files.

Output of `hdfs dfs -cat output/part-00000`

```
Gutenberg   1
If  1
License 1
Project 1
States  1
States, 1
This    1
United  2
You 1
almost  1
and 2
... 47 more lines ...
```

The submission directory, with just this problem in place would be:



The parent directory with your roll number as the name is the one that should be zipped and submitted (using `zip <roll>.zip <roll> -r`)