INTRODUCTION TO LAB#2

This MP will introduce the map/reduce computing paradigm. In essence, map/reduce breaks tasks down into a map phase and a reduce phase. Running our computation within this framework allows us to parallelize data processing, as we'll see in a bit.

For this week, we'll be doing all computation locally (on your laptop/computer/EWS). In the future, we can take this same code and run it on larger datasets on our Hadoop cluster.

SET UP

Download and install python3 if you haven't already. Next, install mrjob – a popular python library developed by YELP. For more information about mrjob or how to install it, follow the documentation below:

https://mrjob.readthedocs.io/en/latest/

MapReduce Paradigm

In the MapReduce paradigm, data is processed and manipulated in the form of <key, value> pairs. This key/value representation is applicable to a wide range of data types and structures. A classic example is the "word count" problem, where words are represented as keys, and the corresponding values indicate the number of occurrences of each word:

Example data:
(cat, 10)
(dog, 8)
(manhattan, 2)
(the, 16)
(a, 32)

It's important to note that both the key and value fields can be utilized selectively based on the requirements of the specific stage in a map/reduce job. In certain scenarios, it is common to disregard either the key or the value. For instance, in the "word count" map function, the value for each word is often set to 1 as a placeholder value, without significant meaning beyond indicating the existence of the word.

Map

The Map phase in MapReduce involves processing the input data by applying a procedure that transforms or filters the data. It operates on key/value tuples, iterating over them and generating a stream of output key/value tuples based on the applied map function.

For instance, let's consider some examples. We can have a map function that takes a list of integers as input and returns only the integers that are greater than 10:

Input: 23, 8, -2, 53, -7, 25, 35
Output: 23, 52, 25, 35

Alternatively, we can have a map function that selects words starting with the letter 'c':

Input: 'cat', 'dog', 'mouse', 'chicken', 'wolf'
Output: 'cat', 'chicken'

These examples demonstrate "filtering" maps, where the map function filters the input data based on certain conditions. However, maps can also perform transformation tasks. For instance, consider a map function that takes a sentence as input and yields its constituent words as output:

Input: 'the cat runs', 'the dog barks'
Output: 'the', 'cat', 'runs', 'the', 'dog', 'barks'

In this case, the Map function generates multiple outputs per call, as it splits the sentence into individual words. In pseudocode, the map function can be represented as follows, disregarding the value field:

```
def map(key, value):
    for word in split_sentence_to_words(key):
        yield (word, 1)
```

Here, the key represents the sentence, and the value field is disregarded. The map function splits the sentence into words and yields each word along with a placeholder value of 1, which is a common approach in the word count example.

Reduce

The Reduce phase collects keys and aggregates their values into some type of summary of the data. The standard example used to demonstrate this programming approach is a word count problem, where words (or tokens) are the keys and the number of occurrences of each word (or token) is the value.
The MapReduce framework structures the data in the reduce task such that each call to reduce is given a key, and an iterable (i.e. a list or iterator) of all the values associated with that key. Thus, the function signature of a reduce task looks like this:

```
def reduce(key, values):
    pass
```

In the computation stage between the map task and the reduce task, MapReduce performs a “shuffle and sort”. The “shuffle” is the process of transmitting map output data from mappers to reducers. Essentially, this process groups the map output tuples by keys, and carries all values

for a given key to a single reducer, so that reducer has access to all data for the given key. Then, within each key set, the values are sorted. Thus, the iterator given to the reducer represents a sorted stream of values for that key. This property can be extremely useful for some workloads.
As the MapReduce technique was popularized by large web search companies like Google and Yahoo who were processing large quantities of unstructured text data, this approach quickly became popular for a wide range of problems. The standard MapReduce approach uses Hadoop, which was built using Java. However, to introduce you to this topic without adding the extra overhead of learning Hadoop's idiosyncrasies, we will be simulating a map/reduce workload in pure Python using Yelp's mrjob library.

To test all these problems, use the following command on your shell:

```
python (name_of_your_file).py  (name_of_input_file).txt >  (name_of_output_file).txt
```

Problem#1: Unique WordCount

Write a MapReduce program that counts the number of unique words in a given text file.

Example Input:

Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Donec condimentum elit vel mauris varius, id laoreet tortor placerat.
Nulla scelerisque felis ac risus varius, sit amet luctus elit mattis.

Example Output:

"adipiscing" 1
"condimentum" 1
"consectetur" 1
"Donec" 1
"dolor" 1
"elit" 1
"felis" 1
"id" 1
"ipsum" 1
"laoreet" 1
"luctus" 1
"mattis" 1
"mauris" 1
"Nulla" 1
"placerat" 1
"risus" 1
"scelerisque" 1
"sit" 1

"tortor" 1
"vel" 1
"varius" 1

Problem#2: WordCount With Stopwords

Write a MapReduce program that only counts non-stop words. List of stopwords are: the, and, of, a, to, in, is, it.

Example Input:

This is a sample input text. It contains some common words such as the, and, of, a, and to. These stopwords should be removed in the output.

Example Output:

"common" 1
"contains" 1
"input" 1
"output" 1
"removed" 1
"sample" 1
"should" 1
"some" 1
"stopwords" 1
"text" 1

Problem#3:

Let's consider a scenario where we are interested in counting the occurrences of word bigrams instead of individual words. A word bigram refers to a pair of words that are adjacent to each other in the text (excluding bigrams that span across line breaks). For example, given the line of text "cat dog sheep horse," the corresponding bigrams would be ("cat", "dog"), ("dog", "sheep"), and ("sheep", "horse"). To achieve this goal, we need to construct a map function and a reduce function. The map function will emit each word bigram as a key-value pair, where the key represents the bigram separated by a comma (e.g., "cat,dog"), and the value is set to 1.
Please note that we will only consider bigrams that occur on the same line, and there is no need to handle bigrams that cross line breaks.

Here's an example illustrating the input and output format:

Example Input:

a man a plan a canal panama there was a plan to build a canal in panama in panama a canal was built

Example Output:

"a,canal" 3
"a,man" 1
"a,plan" 2
"build,a" 1
"canal,in" 1
"canal,panama" 1
"in,panama" 2
"man,a" 1
"panama,a" 1
"plan,a" 1
"plan,to" 1
"there,was" 1
"to,build" 1
"was,a" 1
"was,built" 1

Please note that the order of the output is not crucial for this exercise. However, it's important to format the output keys to match the given format. Additionally, remember to use the regular expression [\w']+ as defined in the word-count example to split the text into words.

Problem#4: Inverted Index (Search Engine Algorithm)

Write a MapReduce program that generates an inverted index for a given text file. An inverted index is a data structure that maps words to the documents (or locations) in which they appear.

Example Input:

Document 1: Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Document 2: Donec condimentum elit vel mauris varius, id laoreet tortor placerat.
Document 3: Nulla scelerisque felis ac risus varius, sit amet luctus elit mattis.

Example Output:

"ac" "Document 3"
"adipiscing" "Document 1"
"amet" "Document 1", "Document 3"
"condimentum" "Document 2"
"consectetur" "Document 1"
"Donec" "Document 2"
"dolor" "Document 1"
"elit" "Document 2", "Document 3"

"felis" "Document 3"
"id" "Document 2"
"ipsum" "Document 1"
"laoreet" "Document 2"
"luctus" "Document 3"
"mauris" "Document 2"
"mattis" "Document 3"
"Nulla" "Document 3"
"placerat" "Document 2"
"risus" "Document 3"
"scelerisque" "Document 3"
"sit" "Document 1", "Document 3"
"tortor" "Document 2"
"vel" "Document 2"
"varius" "Document 2", "Document 3"