QUESTIONS

QUESTION 1

1. Distributed file systems

A distributed file system for cloud is **a file system that allows many clients to have access to data and supports operations (create, delete, modify, read, write)** on that data. ... Typically, data is stored in files in a hierarchical tree, where the nodes represent directories.

1. Commodity clusters

A commodity cluster is **a distributed computing system** consisting of an integrated set of fully and independently operational and marketed computer subsystems (node) used together to perform a single application program or workload.

1. MapReduce

MapReduce is **a programming model or pattern within the Hadoop framework** that is used to access big data stored in the Hadoop File System (HDFS). ... MapReduce facilitates concurrent processing by splitting petabytes of data into smaller chunks, and processing them in parallel on Hadoop commodity servers

QUESTION 2

At a high level, MapReduce breaks input data into fragments and distributes them across different machines.

The input fragments consist of key-value pairs. Parallel map tasks process the chunked data on machines in a cluster. The mapping output then serves as input for the reduce stage. The reduce task combines the result into a particular key-value pair output and writes the data to HDFS.

The Hadoop Distributed File System usually runs on the same set of machines as the MapReduce software. When the framework executes a job on the nodes that also store the data, the time to complete the tasks is reduced significantly.

As the name suggests, MapReduce works by processing input data in two stages – **Map** and **Reduce**. To demonstrate this, we will use a simple example with counting the number of occurrences of words in each document.

The final output we are looking for is: *How many times the words Apache, Hadoop, Class, and Track appear in total in all documents*.

For illustration purposes, the example environment consists of three nodes. The input contains six documents distributed across the cluster. We will keep it simple here, but in real circumstances, there is no limit. You can have thousands of servers and billions of documents.

**Distributed Data Processing**

Distributed systems are often used to collect, access, and manipulate large data sets. For example, the database systems described earlier in the chapter can operate over datasets that are stored across multiple machines. No single machine may contain the data necessary to respond to a query, and so communication is required to service requests.

This section investigates a typical big data processing scenario in which a data set too large to be processed by a single machine is instead distributed among many machines, each of which process a portion of the dataset. The result of processing must often be aggregated across machines, so that results from one machine's computation can be combined with others. To coordinate this distributed data processing, we will discuss a programming framework called MapReduce.

Creating a distributed data processing application with MapReduce combines many of the ideas presented throughout this text. An application is expressed in terms of pure functions that are used to *map* over a large dataset and then to *reduce* the mapped sequences of values into a final result.

Familiar concepts from functional programming are used to maximal advantage in a MapReduce program. MapReduce requires that the functions used to map and reduce the data be pure functions. In general, a program expressed only in terms of pure functions has considerable flexibility in how it is executed. Sub-expressions can be computed in arbitrary order and in parallel without affecting the final result. A MapReduce application evaluates many pure functions in parallel, reordering computations to be executed efficiently in a distributed system.

The principal advantage of MapReduce is that it enforces a separation of concerns between two parts of a distributed data processing application:

1. The map and reduce functions that process data and combine results.
2. The communication and coordination between machines.

The coordination mechanism handles many issues that arise in distributed computing, such as machine failures, network failures, and progress monitoring. While managing these issues introduces some complexity in a MapReduce application, none of that complexity is exposed to the application developer. Instead, building a MapReduce application only requires specifying the map and reduce functions in (1) above; the challenges of distributed computation are hidden via abstraction.

**MapReduce**

The MapReduce framework assumes as input a large, unordered stream of input values of an arbitrary type. For instance, each input may be a line of text in some vast corpus. Computation proceeds in three steps.

1. A map function is applied to each input, which outputs zero or more intermediate key-value pairs of an arbitrary type.
2. All intermediate key-value pairs are grouped by key, so that pairs with the same key can be reduced together.
3. A reduce function combines the values for a given key k; it outputs zero or more values, which are each associated with k in the final output.

To perform this computation, the MapReduce framework creates tasks (perhaps on different machines) that perform various roles in the computation. A *map task* applies the map function to some subset of the input data and outputs intermediate key-value pairs. A *reduce* task sorts and groups key-value pairs by key, then applies the reduce function to the values for each key. All communication between map and reduce tasks is handled by the framework, as is the task of grouping intermediate key-value pairs by key.

In order to utilize multiple machines in a MapReduce application, multiple mappers run in parallel in a *map phase*, and multiple reducers run in parallel in a *reduce phase*. In between these phases, the *sort phase* groups together key-value pairs by sorting them, so that all key-value pairs with the same key are adjacent.

**Question 4 [15 marks]**. **Hands-On Exercise.**

Create a GCP Dataproc cluster and SSH connect to the master node. Complete the following hands-on exercises in a terminal of the master node. At each step, you are required to capture screenshots to show the commands and results.

**Step 1**. In the terminal of the master node, create a folder named “info323-assign2” in the home directory on the master node.

**Step 2**. Download a novel from https://www.gutenberg.org/browse/scores/top. Upload it to the folder “info323-assign2” as “a-novel.txt”.

**Step 3**. Create a directory /user/wc-input in HDFS and copy the file a-novel.txt to the HDFS folder:

hdfs dfs -put ~/info323-assign2/a-novel.txt /user/wc-input/

**Step 4**. List the file in the HDFS folder:

hdfs dfs -ls /user/wc-input

**Step 5**. Check the Hadoop MapReduce example programs: Hadoop comes with several example MapReduce applications. You can check the examples by running:

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar.

*We are interested in running WordCount*.

**Step 6.** Check the WordCount command line arguments: You can learn how to run WordCount by examining its command-line arguments:

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar wordcount

**Step 7.** Run WordCount on “a-novel.txt”. If the output folder /user/wc-output exists in HDFS, you need delete the folder first.

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar wordcount /user/wc-input/a-novel.txt /user/wc-output

**Step 8.** Look inside the output directory /user/wc-output**:** The directory created by WordCount contains several files. List the content of the directory:

hdfs dfs -ls /user/wc-output

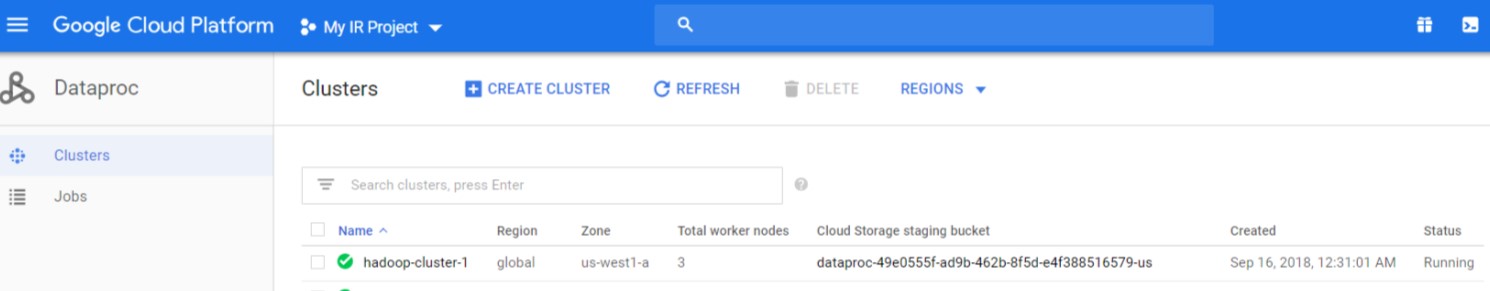
**Step 9.** Copy the output folder in the HDFS to a folder named output in the master node:

hdfs dfs -get /user/wc-output output

**Step 10**. Combine all the partitions in the output folder as a file named counts.txt:

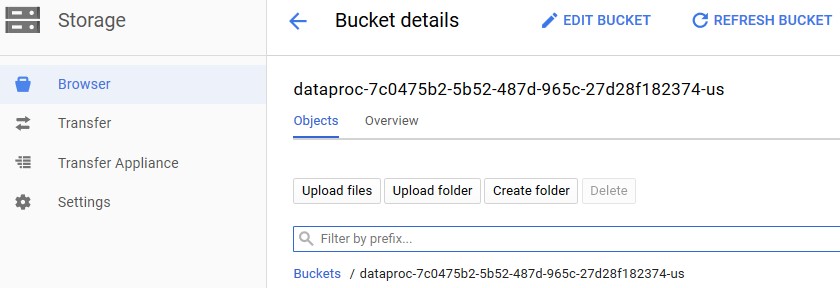
cat part-r-0000\* > counts.txt

1. Click on ‘Dataproc’ in the left navigation menu under . Next, locate the address of the default **Google cloud storage staging** bucket for your cluster in the Figure 1 below. If you’ve previously disabled billing, you need to re-enable it before you can upload the data. Refer to the **“Enable and Disable Billing account”** section to see how to do this. d.

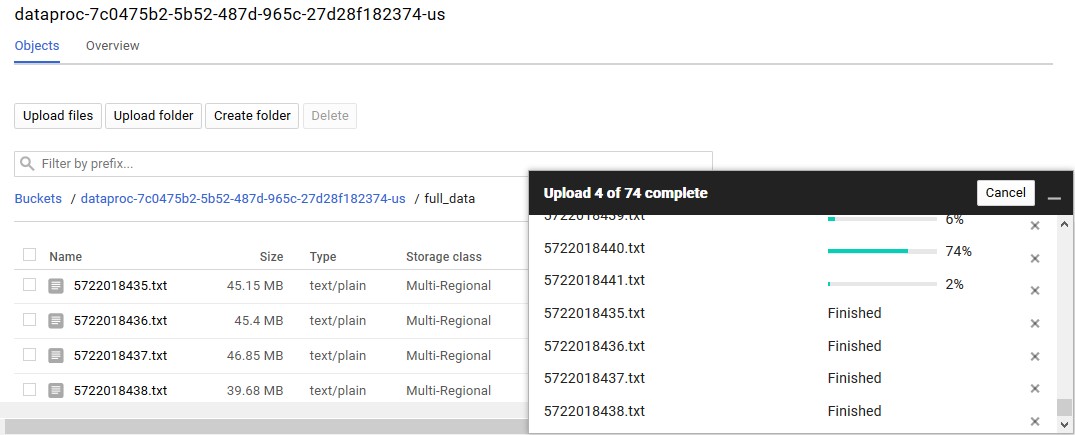


**Figure 1:** The default Cloud Storage bucket.

e. Go to the storage section in the left navigation bar and select your cluster’s default bucket from the list of buckets. At the top you should see menu items UPLOAD FILES, UPLOAD FOLDER, CREATE FOLDER, etc (Figure 2). Click on the UPLOAD FOLDER button and upload the dev\_data folder and full\_data folder individually. This will take a while, but there will be a progress bar (Figure 3). You may not see this progress bar as soon as you start the upload but, it will show up eventually.



**Figure 2:** Cloud Storage Bucket.



**Figure 3**: Progress of uploading

**Inverted Index Implementation using Map-Reduce**

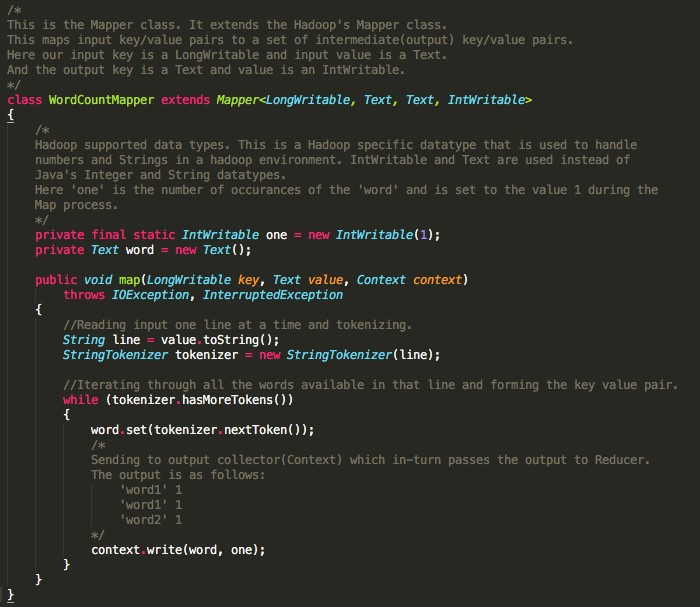
Now that you have the cluster and the files in place, you need to write the actual code for the job. As of now, Google Cloud allows us to submit jobs via the UI, only if they are packaged as a jar file. The following steps are focused on submitting a job written in Java via the Cloud console UI.

Refer to the examples below and write a Map-Reduce job in Java that creates an Inverted Index given a collection of text files. You can very easily tweak a **word-count example** to create an inverted index instead

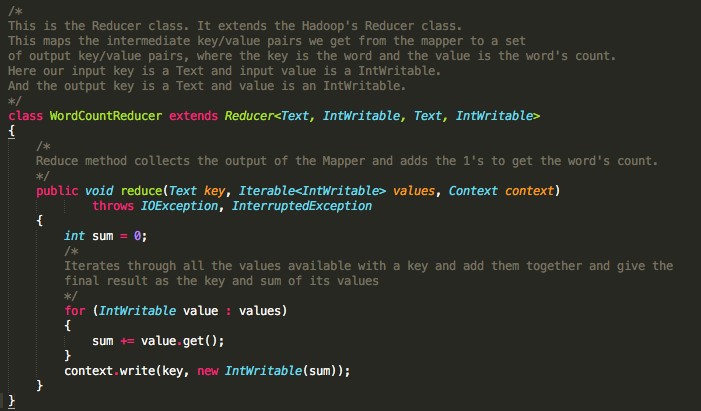
(**Hint**: Change the mapper to output word **docID** instead of word count and in the reducer use a **HashMap**).

The example in the following pages explains a Hadoop word count implementation in detail. It takes one text file as input and returns the word count for every word in the file. Refer to the comments in the code for explanation.

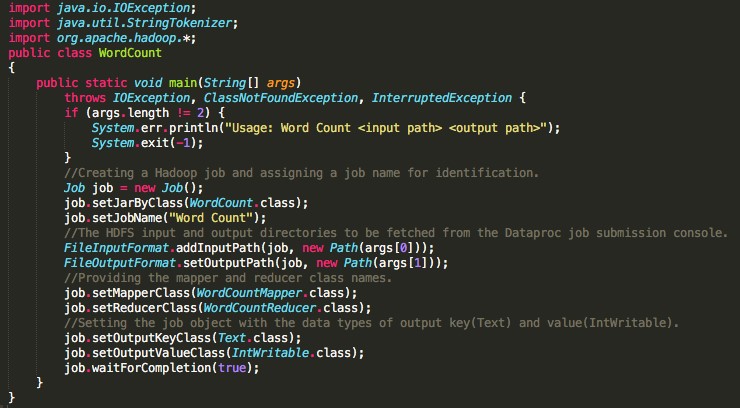
**The Mapper Class:**



**The Reducer Class:**



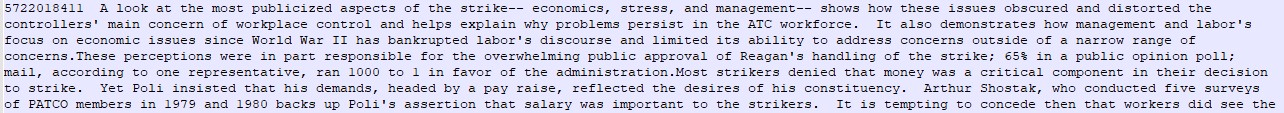
**Main Class**



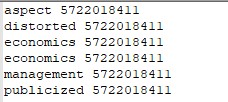
The input data is cleaned, that is all the \n\r s is removed but one or more \t might still be present (which needs to be handled). There will be punctuation and you are required to handle this in your code. Replace all the occurrences of special characters and numerals by space character, convert all the words to the lowercase. **Single ‘\t’ separates the key (Document ID) from the value (Document)**. The input files are in a key value format as below:

|  |  |
| --- | --- |
| DocumentID | document |

Sample document:



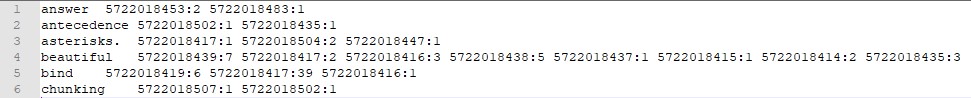
The mapper’s output is expected to be as follows:



The above example indicates that the word aspect occurred 1 time in the document with docID 5722018411 and economics 2 times.

The reducer takes this as input, aggregates the word counts using a HashMap and creates the Inverted index. The format of the index is as follows.

|  |  |  |
| --- | --- | --- |
| word docID:count | docID:count | docID:count... |



The above is just a sample and shows a portion of the inverted index created by the reducer.