Lanzhou University

School of Information Sciences and Engineering

Cloud Computing and Big Data

Assignment 2

Module Code: 2043623

Name: Yuming Chen

Student Number: 320180939611

Set date: Thursday, March 18, 2021 Due Date: 11:59pm, Saturday, March 27, 2021

Table of Contents

A. Requirements

1. Provide full and detailed answer to each question.
2. Work individually. The Lanzhou University Academic Honesty Rules and Procedures (as stated in the student handbook) will be adhered to strictly.
3. There are 6 questions. The total marks are 100. This assignment contributes 15% to the overall assessment for this module.

B. Questions

Question 1 [15 marks]. What is MapReduce? Describe the main steps in a MapReduce application. How does MapReduce solve big data problem? Where does parallel computation happen in MapReduce?

1. The MapReduce is:

MapReduce is a scalable software programming paradigm\framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. MapReduce allows us to perform distributed and parallel processing on large data sets in a distributed environment with a Hadoop cluster which has hundreds or thousands of servers and these characters improve the speed of MapReduce. Moreover, as the processing component, MapReduce is the data processing layer of Hadoop, is also regarded as the heart of Hadoop ecosystem.

2. The Main steps:

Step 1 Mapping

Step 1.1 *Reading and Splitting*

After client start a *MapReduce* program, the *InputFormat* which is a factory for *RecordReader* objects of MapReduce will validate the input-specification of the job and the input data which is mostly in the form of file or directory residing in HDFS will be divided into several *InputSplit* which are the logical instances representing data by mapper. Each of these *InputSplit* will be assigned to individual Mapper. Therefore, the number of maps is usually driven by the total volume of the inputs that is the total number of blocks of the input files. Moreover, in the physical side, the *InputSplit* will be mapped to data blocks. Therefore,

Step 1.2 ***Processing***

After *InputSplit* are generated, the process *Job* of *Mapper*s which is used to process the input data will begin. Each *InputSplit* will be processed by calling the custom method written previously by users in the *Mapper* class and be converted in the form of *Key-Value Pair* records. Then, each *Key-Value Pair* record will be processed in turn, according to the custom method written previously by users in the *Mapper* class. To be specify, *InputSplit* and *Key-Value Pair* record are logical, they are not physically bound to a file.

Step 1.3 ***Outputting***

Once the *Job* is complete, Hadoop's *Mapper*s will store the intermediate outputs which are the *Key-Value Pair* records generated by the previous steps into the local disk on the respective node from where it is shuffled to reduce nodes and not to *HDFS*. Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of *HDFS* store operation. However, before the intermediate outputs really saved into the local disk, there are several *Shuffling* steps which will discuss later.

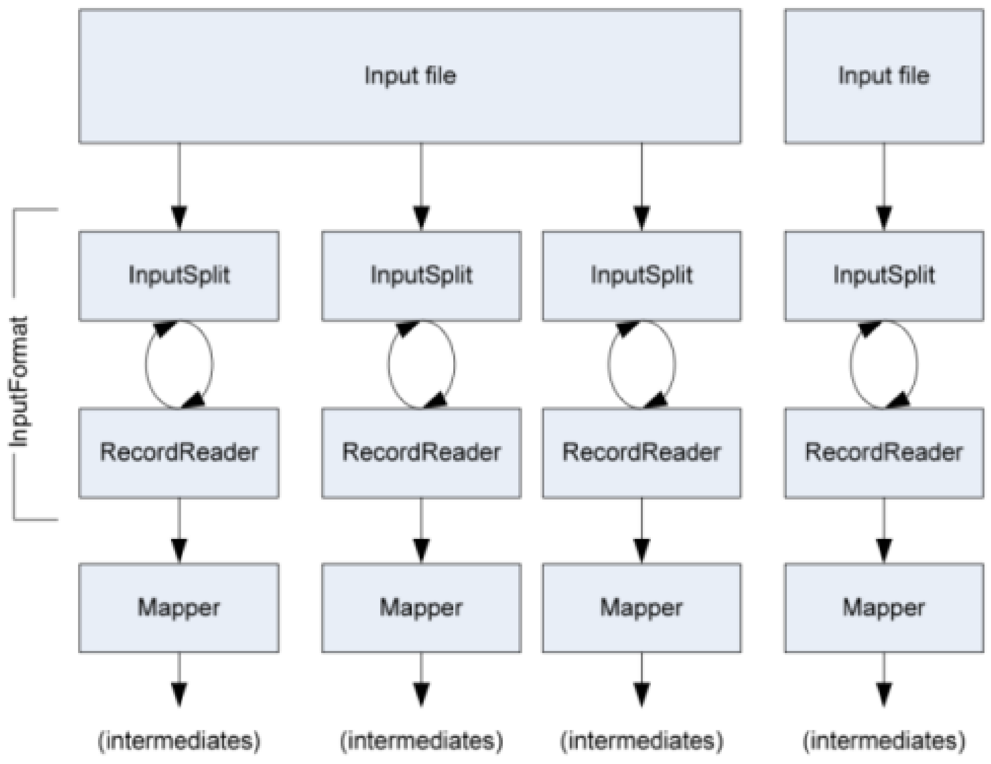


Figure 1 The diagram of Mapping

Step 2 Shuffling ----- Grouping by key

Shuffle is the most important part of the *MapReduce* framework. It is the whole process between the end of *Mapping* and the start of *Reducing*.

Step 2.1 *Map Side*

The intermediate outputs generated from *Mapping* stage will be stored in the cache first. Each *Mapper* has a ring memory buffer for storing these intermediate outputs. The default size is 100MB (the “io.sort.mb” attribute). Once the threshold of 0.8(“io.sort.spil l.percent”) is reached, a background thread writes the contents to a newly created overflow write file in a directory (“mapred.local.dir”) on the local disk.

Step 2.1.1 Combining (optional)

This is an optional optimization which achieved by *Combiner*. In this stage when the Reducer not really get involve in, the *Combiner* will perform as a Mini-reducers that run local aggregation in memory, before the “shuffle and sort” phase. To be specify, each combiner operates in isolation.

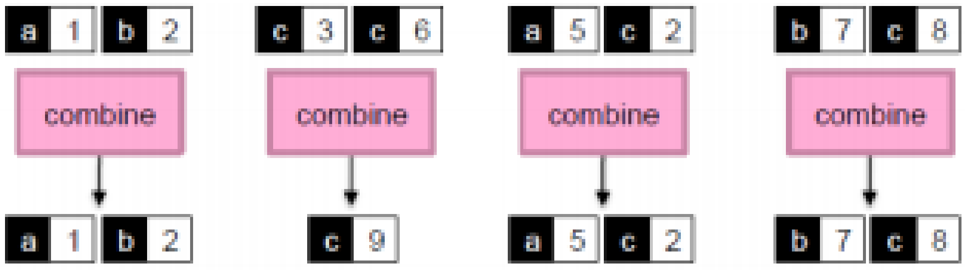


Figure 2 The diagram of Combining

Step 2.1.2 Partitioning

In this stage, the intermediate output *Key-Value Pair* produced by *Mapper* tasks will be divided into partition for parallel reduce operations by *Partitioner.* The total number of partitions is equal to the number of reducers where each partition is processed by the corresponding reducer. The partitioning is done using the hash function based on a single key or group of keys. The default *Partitioner* available in Hadoop is “*HashPartitioner*”.

Step 2.1.3 Sorting

The MapReduce framework guarantees the input to every reducer to be sorted by key. Therefore, in this stage, *Key-Value Pairs* will be sorted in each partition. Firstly, *Key-Value Pairs* will be sorted by key. Secondly, when key-value pairs with the same key, *Key-Value Pairs* will be sorted by value. For example, there are 3 *Key-Value Pairs* <2,2>, <1,3>, <2,1>. Then, the result of sorting is <1,3>, <2,1>, <2,2>.

Step 2.1.4 Storing

After above process, the intermediate outputs will be finally stored in the local disk. When the intermediate outputs written in the local disk successfully, *MapReduce* will merge all overflow files into one partitioned and sorted file in the local disk. Then, the data stored in the local disk will be sent to the *Reducer* for next process.

Step 2.2 *Reduce Side*

Step 2.2.1 Copying

The *Reducer* obtains the intermediate outputs generated by *Mapper* through HTTP. However, the time of these intermediate outputs arriving in *Reducer* is different. Therefore, when one of the *Mapper* finishes, the *Reducer* gets that information from the *JobTracker*. When the *Mapper* runs, the *TaskTracker* will get a message and report the message to the *JobTracker*. *Reducer* will get the information from the *JobTracker* on a regular basis. *Reducer* will take the initiative to copy the intermediate outputs. By default, there will be 5 data replication threads in Reduce to copy data from the Map.

Step 2.2.2 Combining, Partitioning and Sorting

The intermediate outputs generated by *Mapper* is first written to the cache of the *Reducer*. Similarly, after the cache utilization reaches a certain threshold, the data will be written to the disk. Then *Reducer* will do the same processes as the *Mapper* will do, such as *Combining*, *Partitioning* and *Sorting*.

Step 2.2.3 Merging

If multiple disk files are formed, they will be merged, and the result of the last merge will be used as input to *reduce* rather than written to disk.

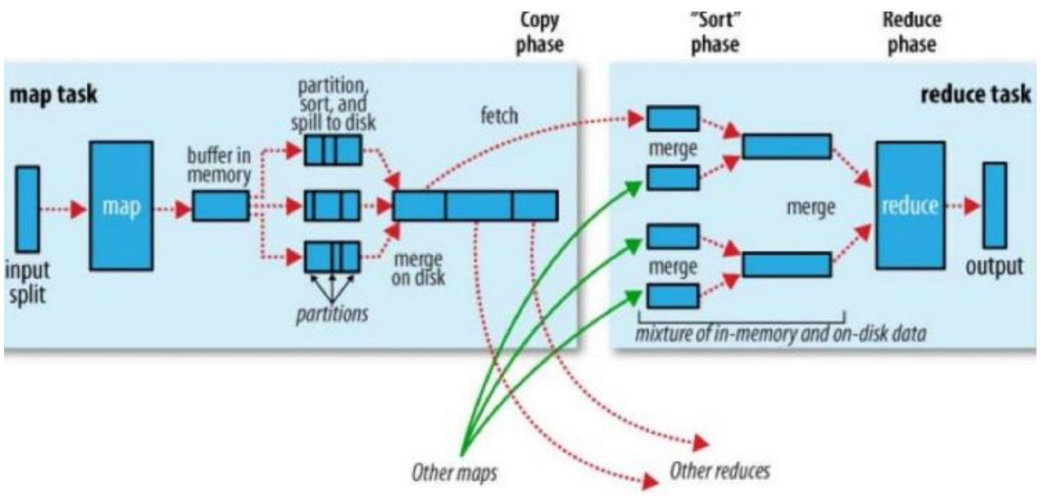


Figure 3 The diagram of Suffering

Step 3 Reducing

In this step, the Intermediate output generated from the mapper is fed to the reducer which processes it and generates the final output which is then saved in the HDFS. To be specify, the Reducer cannot start until all *Mapper*s have finished.

Step 3.1 *Processing*

Reducer calls the custom method in *Reducer* class written previously by the users for the sorted *Key-Value Pairs*, and call another method in *Reducer* class for *Key-Value Pairs* with equal keys. Each call of the method in *Reducer* class will produce zero or more *Key-Value Pair*s which is the final output of *MapReduce.*

Step 3.2 *Outputting*

Finally, the *Reducer* will call *RecordWriter* to store the output into the output files, mostly *HDFS.*

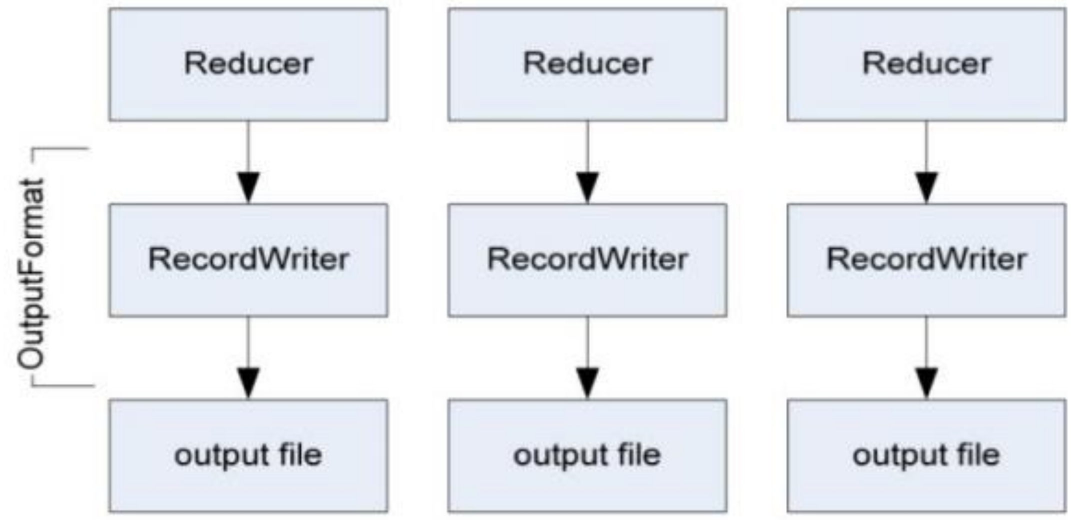


Figure 4 The diagram of Reducing

3. MapReduce solves big data problem by:

3.1 Parallel Processing

3.1.1 Problem

Due to the data volume which refers to the tremendous scale of big data and the data velocity refers which to big data begin generated fast and need to be processed fast, the processing of

Big Data is extremely difficult. In the traditional system, it is an impossible challenge to deal with the big data because of the poor ability to deal with large amount of data with a really fast speed.

3.1.2 Solution

In *MapReduce*, jobs are divided among multiple nodes and each node works with a part of the job simultaneously. Therefore, *MapReduce* is based on Divide and Conquer paradigm which helps users to process the data using multiple different machines. As the data is processed by multiple machines instead of a single machine in parallel, the time taken to process the data gets reduced by a tremendous amount as shown in the figure below which serves as a great solution to deal with the problem brought big data.

3.2 Data Locality

3.1.1 Problem

In the traditional system, users used to bring big data to the processing unit and process it. But, as the data grew and became very huge, bringing this huge amount of data to the processing unit posed the following problems:

① Moving huge data to processing is costly and deteriorates the network performance.
② Processing takes time as the data is processed by a single unit which becomes the bottleneck.
③ The master node can get over-burdened and may fail.

3.1.2 Solution

MapReduce allows users to overcome the above problems by bringing the processing unit to the data. So, as you can see in the above image that the data is distributed among multiple nodes where each node processes the part of the data residing on it. This allows us to have the following advantages:

① It is very cost-effective to move processing unit to the data.
② The processing time is reduced as all the nodes are working with their part of the data in parallel.
③ Every node gets a part of the data to process and therefore, there is no chance of a node getting overburdened.

4. Parallel computation happen in MapReduce:

A *MapReduce* program is composed of a *Mapping* procedure, which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a *Reducing* procedure, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). These two procedures are both in parallel.

4.1 Mapping

The *Mapping* procedure takes a series of *Key-Value Pairs*, processes each, and generates zero or more output *Key-Value Pairs*. The input and output types of the map can be (and often are) different from each other. During the *Mapping* procedure, each worker node applies the map function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of the redundant input data is processed. If the application is doing a word count, the map function would break the line into words and output a *Key-Value Pairs* for each word. Each output pair would contain the word as the key and the number of instances of that word in the line as the value.

4.2 Reducing

The framework calls the application's *Reducing* function once for each unique key in the sorted order. The *Reducing* can iterate through the values that are associated with that key and produce zero or more outputs. During the *Reducing* procedure, worker nodes now process each group of output data, per key, in parallel. In the word count example, the *Reduce* function takes the input values, sums them and generates a single output of the word and the final sum.

Question 2 [15 marks]. Do some research on the Internet to complete the following steps: (1) Describe the common characteristics of the problems suitable for MapReduce. (2) What types of problems are unsuitable for MapReduce? Use examples to illustrate your answers.

1. The common characteristics of problems suitable for MapReduce

characteristics	***Description***
Scale of data is changing a lot / **Scalability**	*MapReduce* as a distributed computing framework that is highly scalable and is largely because of its ability that it stores and distributes large data sets across lots of servers. The servers used here are quite inexpensive and can operate in parallel. The processing power of the system can be improved with the addition of more servers. When the servers are added, the performance of *MapReduce* will be increase which reflect that MapReduce Framework is a suitable solution to solve the problem whose data is large and increased by time quickly. However, the traditional relational database management systems or *RDBMS* were not able to scale to process huge data sets. ***Example*** Nowadays, due to the data velocity refers which to big data begin generated fast and need to be processed fast, most of companies choose *MapReduce* to deal with the increasing data. Google, the inventor of *MapReduce*, was trying to solve this problem by creating a computing framework with highly scalability that can adapt to the increasing scale of data.
Multiple requirements / **Flexibility**	*MapReduce* programming framework offers flexibility to process structure or unstructured data by various business organizations who can make use of the data and can operate on different types of data. Moreover, *MapReduce* libraries have been written in many programming languages, with different levels of optimization. Therefore, it offers support for a lot of languages used for data processing. Therefore, the *MapReduce* programming framework can support to solve the problem that has various types of data and the programming is combined with multiple area.

	Example With the flexibility to process structure or unstructured data, business organizations can deal with the situation with various data sources such as social media, clickstream, email, etc and generate various business value which are meaningful and useful for the business organizations to analyze. Along with the libraries written by various programming language, *MapReduce* framework can solve problems that need to combine many applications, such as marketing analysis which requires recommendation system, data warehouse, and fraud detection.
Error-prone —— **Resilient**	In the *MapReduce* framework, when data is sent to an individual node in the entire network, the very same set of data is also forwarded to the other numerous nodes that make up the network. Thus, if there is any failure that affects a particular node, there are always other copies that can still be accessed whenever the need may arise. This always assures the availability of data. *MapReduce* also has the ability to quickly recognize faults that occur and then apply a quick and automatic recovery solution. This makes it a game changer when it comes to big data processing. According to the above characteristics, the *MapReduce* framework can solve the problem which are error-prone. ***Example*** As the result of the highly resilient mechanism of *MapReduce* framework, many companies who needs high fault tolerance to deal with the big data will choose the *MapReduce* framework to solve the problem. For example, twitter, one of the biggest companies of the world, used the SummingBird which is based on the *MapReduce* to process the tweets. Rely on the high fault tolerance of *MapReduce*, they can avoid the catastrophic event on their customers' data which is of vitally important for Twitter to protect.
insufficient fund —— **Cost-effective**	*MapReduce*'s highly scalable structure also implies that it comes across as a very cost-effective solution for businesses that need to store ever growing data dictated by today's requirements. In the case of traditional relational database management systems, it becomes massively cost prohibitive to scale to the degrees possible with

	MapReduce, just to process data. As such, many of the businesses would have to downsize data and further implement classifications based on assumptions of how certain data could be more valuable that the other. In the process, raw data would have to be deleted. This basically serves short term priorities, and if a business happens to change its plans somewhere down the line, the complete set of raw data would be unavailable for later usage. *MapReduce*'s scale-out architecture with *MapReduce* programming, allows the storage and processing of data in a very affordable manner. It can also be used in later times. In fact, the cost savings are massive and costs can reduce from thousands and figures to hundred figures for every terabyte of data. Therefore, the *MapReduce* framework is really suitable for situation which is not enough fund providing.
Need Simple Operation **Simple**	Among the various advantages that *MapReduce* offers, one of the most important ones is that it is based on a simple programming model. This basically allows programmers to develop MapReduce programs that can handle tasks with more ease and efficiency. The programs for *MapReduce* can be written using Java, which is a language that isn't very hard to pick up and is also used widespread. Thus, it is easy for people to learn and write programs that meets their data processing needs sufficiently. ***Example*** As the result of the major advantage of *MapReduce* framework, people are easy to learn and use *MapReduce* framework to do some simple jobs which is hard to do with traditional framework. For example, college students can simply achieve the word count program for terabyte of text data.
Need Fast Process **Fast**	*MapReduce* uses a storage method known as distributed file system, which basically implements a mapping system to locate data in a cluster. The tools used for data processing, such as *MapReduce* programming, are also generally located in the very same servers, which allows for faster processing of data. Even if you happen to be dealing with large volumes of data that is unstructured, *MapReduce* takes minutes to process terabytes of data, and hours for petabytes of data.

	Example Due to the Velocity of data which refers to big data begin generated fast and need to be processed fast. Nowadays, data volume is increasing exponentially, because data is generated from various sources in different formats. From 2009 to 2020, data volume has increased from 0.8 zettabytes to 35zb. Therefore, the *MapReduce* framework is suitable in such situation by providing parallel computing. Google used the *MapReduce* framework to compute PageRank in an enormous network, which is a really time-consuming work in an early age. With the fast processing of *MapReduce* framework, the *PageRank* became one of the most important Algorithm in the world.

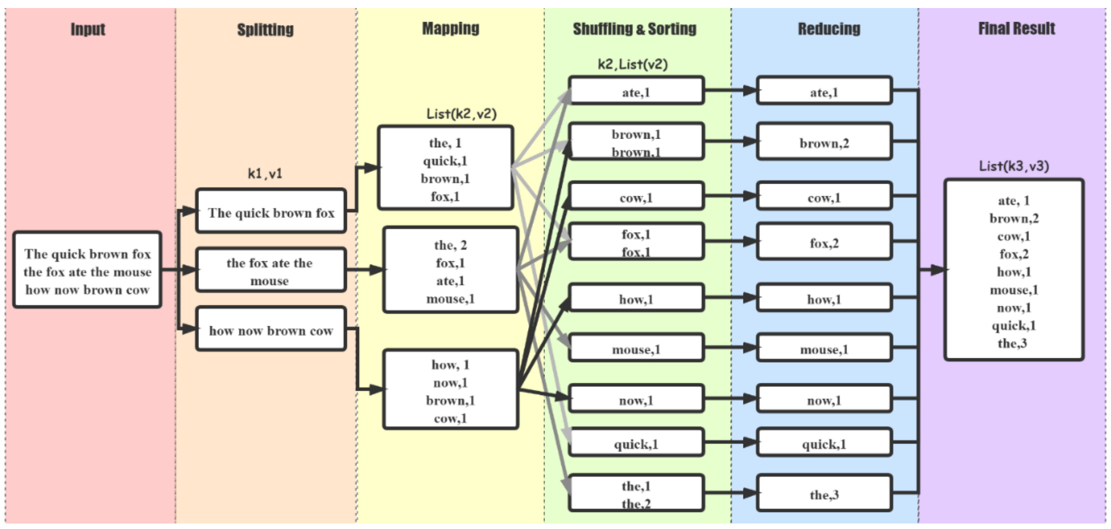
2. The problems are unsuitable for MapReduce

problems	***Description***
Needing flexible framework	The tasks running in the *MapReduce* framework must be written as acyclic dataflow programs, i.e. a stateless *Mapper* followed by a stateless *Reducer*, that are executed by a batch job scheduler. ***Example*** Due to the grid framework of *MapReduce*, it is really difficult to finish complex tasks. For example, repeated querying of datasets difficult and imposes limitations that are felt in fields such as machine learning, where iterative algorithms that revisit a single working set multiple times are the norm. Moreover, the graph computation is also a limitation of *MapReduce.*
Needing real-time data processing	At the core, *MapReduce* is a batch processing framework which is not efficient in stream processing. It cannot produce output in real-time with low latency. It only works on data which we collect and store in a file in advance before processing. ***Example*** Due to the *MapReduce*'s weakness, many situations which needs real-time data processing or getting the result as soon as possible is really not suitable for *MapReduce.* For example, in the analysis of some data

	generated by sensor, *MapReduce* does not work. This is because *MapReduce* processes data statically, while sensor data is dynamic, that is, constantly changing, *MapReduce* cannot process it in real time, which will cause data congestion.
Too much Small Files	*MapReduce* does not suit for the situation with too many small data. The reason is that Hadoop distributed file system (*HDFS*) which is the basis of *MapReduce* framework lacks the ability to efficiently support the random reading of small files because of its high capacity which is designed for working properly with a small number of large files for storing large data sets. ***Example*** Due to the weakness, it really hard to deal with the problem which has large amount of small data. If we are storing huge numbers of small files which is significantly smaller than the *HDFS* block size (default 128MB), the NameNode will get overload since it stores the namespace of *HDFS*. Therefore, *HDFS* can’t handle too much of small files.

Question 3 [15 marks]. Assume a text file containing 3 lines of text: “The quick brown fox”, “the fox ate the mouse”, “how now brown cow”. Draw a flow diagram to show a step-by-step MapReduce process (input, map, shuffle & sort, reduce, output) for computing the counts of each word in the file. Show the results of intermediate steps and the final count of each word.

Flow diagram:



Question 4 [25 marks]. Design a MapReduce algorithm to solve the following problem. Given a big data set containing files of Web metadata. Each file contains lines in the format (URL, size, date, …). For each host, find the total number of bytes. That is, the sum of the page sizes for all URLs from that host. Write your algorithm in pseudo code.

Pseudo code

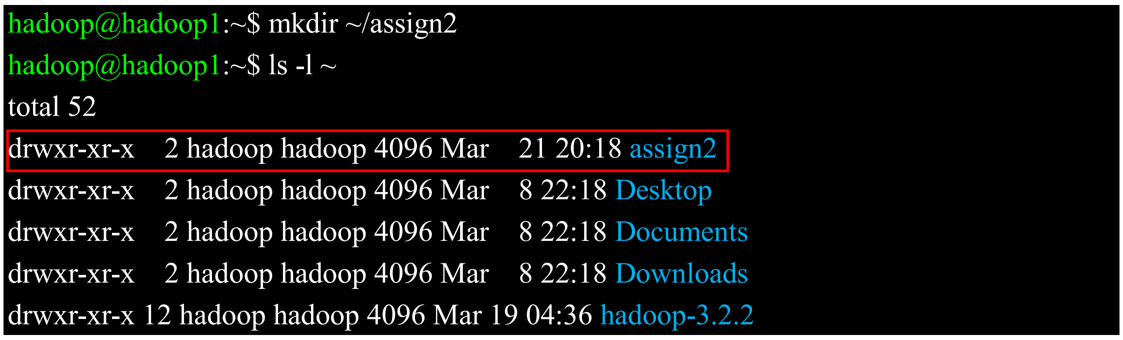
Algorithm Count page sizes for all URLs from that host

```
The mapper emits an intermediate key-value pair (host, size) for each host.
The reducer sums up the page sizes for each host.
 1: class MAPPER
 2:     method MAP (fileid fid, file f)
 3:         for all line l ∈ file f do
 4:             line_array ← Split(line l, mark ‘,’)   ▷ Split each line into array through the mark ‘,’
 5:             u ← GetFromIndex(array line_array, index 0)   ▷ Get “url” from line array’s 1st place
 6:             s ← GetFromIndex(array line_array, index 1)   ▷ Get “size” from line array’s 2nd place
 7:             h ← GetHost(url u)   ▷ Get the “host” from “url” by remove things after “/”
 8:             EMIT (host h, size s)   ▷ Emit size for each host

 1: class REDUCER
 2:     method REDUCE(host h, sizes [s1,s2,…])
 3:         sum ← 0
 4:         for all size s ∈ sizes [s1,s2,…] do
 6:             sum ← sum + s
 7:         EMIT(host h, size sum) ▷ Sum the sizes for each host
```

Question 5 [35 marks]. In this hands-on exercise, you will need to work in the Apache Hadoop environment installed in a Linux in the Virtual Machine (VM) (VirtualBox or VMware). Do the exercise using the following instructions. At each step, you are required to capture screenshots to show the results.

Step 1. Open a terminal and create a folder named “assign2” in your home directory.



```
hadoop@hadoop1:~$ mkdir ~/assign2
hadoop@hadoop1:~$ ls -l ~
total 52
drwxr-xr-x  2 hadoop hadoop 4096 Mar  21 20:18 assign2
drwxr-xr-x  2 hadoop hadoop 4096 Mar   8 22:18 Desktop
drwxr-xr-x  2 hadoop hadoop 4096 Mar   8 22:18 Documents
drwxr-xr-x  2 hadoop hadoop 4096 Mar   8 22:18 Downloads
drwxr-xr-x 12 hadoop hadoop 4096 Mar 19 04:36 hadoop-3.2.2
```

```
drwxrwxr-x   2 hadoop hadoop 4096 Mar 19 04:46 input
drwxr-xr-x   8 hadoop hadoop 4096 Dec   9 20:50 jdk1.8.0_281
drwxr-xr-x   2 hadoop hadoop 4096 Mar   8 22:18 Music
drwxr-xr-x   2 hadoop hadoop 4096 Mar   8 22:18 Pictures
drwxr-xr-x   2 hadoop hadoop 4096 Mar   8 22:18 Public
drwxr-xr-x   2 hadoop hadoop 4096 Mar   8 22:18 Templates
```

Step 2. Open a Web browser pointing at https://www.gutenberg.org/browse/scores/top. Download a novel in text format and store the text file in the "assign2" folder as "a novel.txt".

I download the book named *The Modern Prometheus\ Frankenstein* written by Mary Wollstonecraft Shelley.

```
hadoop@hadoop1:~$ cat assign2/a-novel.txt | head -n 5
The Project Gutenberg eBook of Frankenstein, by Mary Wollstonecraft (Godwin) Shelley

This eBook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
```

Step 3. Copy the file to HDFS. Run the command in terminal.

```
hadoop@hadoop1:~$ hdfs dfs -copyFromLocal assign2/a-novel.txt
```

Step 4. List the file in HDFS. Run the command:

```
hadoop@hadoop1:~$ hdfs dfs -ls
Found 1 items
-rw-r--r--   1 hadoop supergroup      441075 2021-03-21 22:00 a-novel.txt
```

Step 5. See example MapReduce programs. Hadoop comes with several example MapReduce applications.

```
hadoop@hadoop1:~$ hadoop jar hadoop-3.2.2/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar | head -n 10

An example program must be given as the first argument.
Valid program names are:
  aggregatewordcount: An Aggregate based map/reduce program that counts the words in the input files.
  aggregatewordhist: An Aggregate based map/reduce program that computes the histogram of the words in the input files.
  bbp: A map/reduce program that uses Bailey-Borwein-Plouffe to compute exact digits of Pi.
  dbcount: An example job that count the pageview counts from a database.
  distbbp: A map/reduce program that uses a BBP-type formula to compute exact bits of Pi.
  grep: A map/reduce program that counts the matches of a regex in the input.
  join: A job that effects a join over sorted, equally partitioned datasets
  multifilewc: A job that counts words from several files.
                                        ……
```

Step 6. See WordCount command line arguments. We can learn how to run WordCount by examining its command-line arguments.

```
hadoop@hadoop1:~$ hadoop jar hadoop-3.2.2/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar wordcount

Usage: wordcount <in> [<in>...] <out>
```

Step 7. Run WordCount. Run WordCount for a-novel.txt.

```
hadoop@hadoop1:~$ hadoop jar hadoop-3.2.2/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar wordcount a-novel.txt out

2021-03-21 22:30:15,204 INFO client.RMProxy: Connecting to ResourceManager at hadoop1/192.168.231.128:8032
2021-03-21 22:30:15,849 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/hadoop/.staging/job_1616099842115_0003
2021-03-21 22:30:15,971 INFO input.FileInputFormat: Total input files to process : 1
2021-03-21 22:30:16,004 INFO mapreduce.JobSubmitter: number of splits:1
2021-03-21 22:30:16,087 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1616099842115_0003
2021-03-21 22:30:16,088 INFO mapreduce.JobSubmitter: Executing with tokens: []
2021-03-21 22:30:16,189 INFO conf.Configuration: resource-types.xml not found
2021-03-21 22:30:16,190 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2021-03-21 22:30:16,228 INFO impl.YarnClientImpl: Submitted application application_1616099842115_0003
2021-03-21 22:30:16,251 INFO mapreduce.Job: The url to track the job: http://hadoop1:8088/proxy/application_1616099842115_0003/
2021-03-21 22:30:16,251 INFO mapreduce.Job: Running job: job_1616099842115_0003
2021-03-21 22:30:22,835 INFO mapreduce.Job: Job job_1616099842115_0003 running in uber mode : false
2021-03-21 22:30:22,836 INFO mapreduce.Job:  map 0% reduce 0%
2021-03-21 22:30:26,881 INFO mapreduce.Job:  map 100% reduce 0%
2021-03-21 22:30:31,907 INFO mapreduce.Job:  map 100% reduce 100%
2021-03-21 22:30:33,076 INFO mapreduce.Job: Job job_1616099842115_0003 completed successfully
2021-03-21 22:30:33,233 INFO mapreduce.Job: Counters: 54
	File System Counters
		FILE: Number of bytes read=177385
		FILE: Number of bytes written=824661
		FILE: Number of read operations=0
		FILE: Number of large read operations=0
		FILE: Number of write operations=0
		HDFS: Number of bytes read=441183
		HDFS: Number of bytes written=129593
		HDFS: Number of read operations=8
		HDFS: Number of large read operations=0
```

```
        HDFS: Number of write operations=2
        HDFS: Number of bytes read erasure-coded=0
    Job Counters
        Launched map tasks=1
        Launched reduce tasks=1
        Data-local map tasks=1
        Total time spent by all maps in occupied slots (ms)=2329
        Total time spent by all reduces in occupied slots (ms)=2655
        Total time spent by all map tasks (ms)=2329
        Total time spent by all reduce tasks (ms)=2655
        Total vcore-milliseconds taken by all map tasks=2329
        Total vcore-milliseconds taken by all reduce tasks=2655
        Total megabyte-milliseconds taken by all map tasks=2384896
        Total megabyte-milliseconds taken by all reduce tasks=2718720
    Map-Reduce Framework
        Map input records=7743
        Map output records=78122
        Map output bytes=752377
        Map output materialized bytes=177385
        Input split bytes=108
        Combine input records=78122
        Combine output records=12178
        Reduce input groups=12178
        Reduce shuffle bytes=177385
        Reduce input records=12178
        Reduce output records=12178
        Spilled Records=24356
        Shuffled Maps =1
        Failed Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=1329
        CPU time spent (ms)=5870
        Physical memory (bytes) snapshot=665731072
        Virtual memory (bytes) snapshot=5371150336
        Total committed heap usage (bytes)=748683264
        Peak Map Physical memory (bytes)=390840320
        Peak Map Virtual memory (bytes)=2688765952
        Peak Reduce Physical memory (bytes)=274890752
        Peak Reduce Virtual memory (bytes)=2682384384
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
```

```
		WRONG_MAP=0
		WRONG_REDUCE=0
	File Input Format Counters
		Bytes Read=441075
	File Output Format Counters
		Bytes Written=129593
```

Step 8. Look inside output directory. The directory created by WordCount contains several files. Look inside the directory by running

```
hadoop@hadoop1:~$ hdfs dfs -ls out
-rw-r--r--   1 hadoop supergroup          0 2021-03-21 22:30 out/_SUCCESS
-rw-r--r--   1 hadoop supergroup     129593 2021-03-21 22:30 out/part-r-00000
```

Step 9. Copy WordCount results to local file system. Copy part-r-00000 to the local file system by running

```
hadoop@hadoop1:~$ hdfs dfs -copyToLocal out/part-r-00000 counts.txt
```

Step 10. Check out how many lines in the file counts.txt.

```
hadoop@hadoop1:~$ wc -l counts.txt
12178 counts.txt
```

Step 11. View the WordCount results. Run a command to print out the 20 lines in the middle of the file counts.txt.

```
hadoop@hadoop1:~$ sed -n '6089,6109p' counts.txt
impatient	4
impatiently	1
impediment,	1
impediments	1
impelled	2
impend	1
impending	2
impenetrable	2
impenetrable.	1
imperatively	1
imperceptible	1
imperfect	1
imperfect,	1
imperial	1
imperious	1
impertinent	1
impervious	1
impetuous	1
implements	1
implied	1
implores	1
```

Reference:

[1] https://en.wikipedia.org/wiki/MapReduce#Partition_function

[2] https://www.zhihu.com/question/303101438

[3] https://www.quora.com/What-are-the-disadvantages-of-mapreduce

[4] https://blog.csdn.net/yangshaojun1992/article/details/85003668?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-4.control&dist_request_id=&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-4.control

[5] http://blog.sina.com.cn/s/blog_9f4f649d01015cas.html

[6] https://zh.wikipedia.org/wiki/PageRank

[7] http://tech.it168.com/a2016/0622/2731/000002731026.shtml

[8] https://data-flair.training/blogs/13-limitations-of-hadoop/

[9] https://www.softwaretestinghelp.com/hadoop-mapreduce-tutorial/

[10] https://vikram-bajaj.gitbook.io/cs-gy-9223-d-programming-for-big-data/hadoop/advantages-and-disadvantages-of-mapreduce

[11] https://www.researchgate.net/figure/The-advantages-and-disadvantage-of-MapReduce-applications_tbl1_303286828

[12] https://www.sciencedirect.com/topics/computer-science/big-data-problem

[13] Karloff H, Suri S, Vassilvitskii S. A model of computation for mapreduce[C]//Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2010: 938-948.

[14] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[15] Lee K H, Lee Y J, Choi H, et al. Parallel data processing with MapReduce: a survey[J]. AcM sIGMoD Record, 2012, 40(4): 11-20.

[16] Jiang D, Ooi B C, Shi L, et al. The performance of mapreduce: An in-depth study[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 472-483.

[17] Roy I, Setty S T V, Kilzer A, et al. Airavat: Security and privacy for MapReduce[C]//NSDI. 2010, 10: 297-312.

[18] Roy I, Setty S T V, Kilzer A, et al. Airavat: Security and privacy for MapReduce[C]//NSDI. 2010, 10: 297-312.

[19] Ekanayake J, Li H, Zhang B, et al. Twister: a runtime for iterative mapreduce[C]//Proceedings of the 19th ACM international symposium on high performance distributed computing. 2010: 810-818.