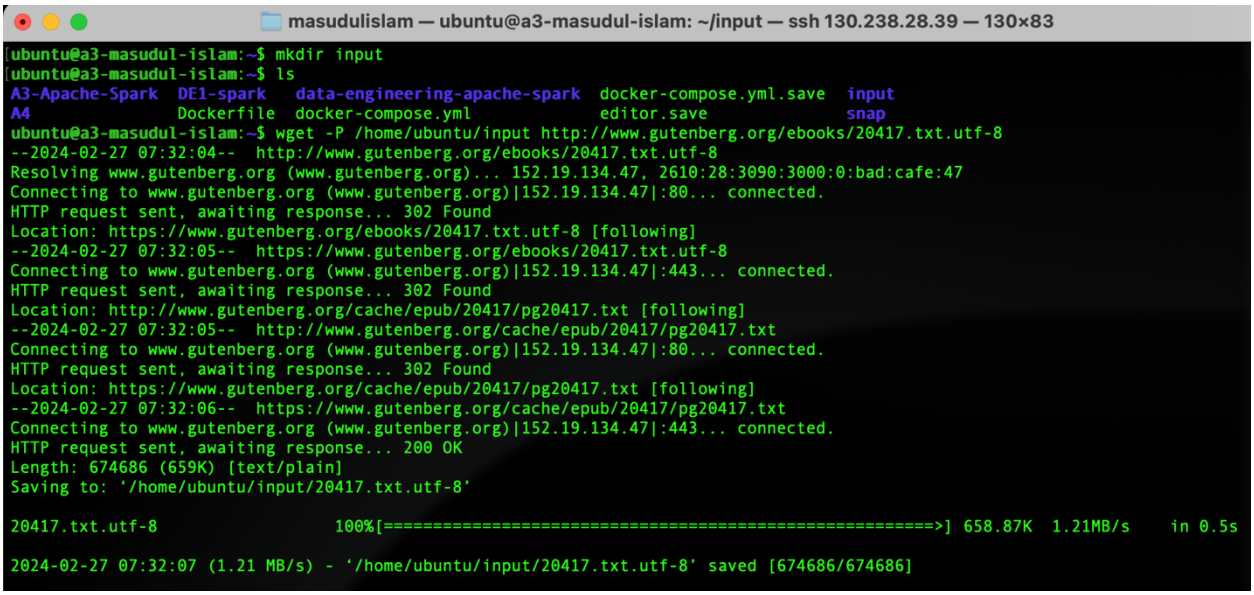
A2 - Hadoop and MapReduce

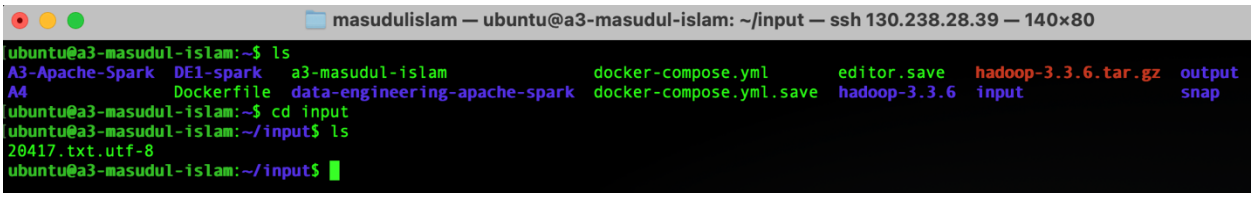
Task 1.1: Word count example in local (standalone) mode

1. Create a folder “input” in your home directory (/home/ubuntu) on your instance using command mkdir

2. Download the following data file and place it in that directory http://www.gutenberg.org/ebooks/20417.txt.utf-8

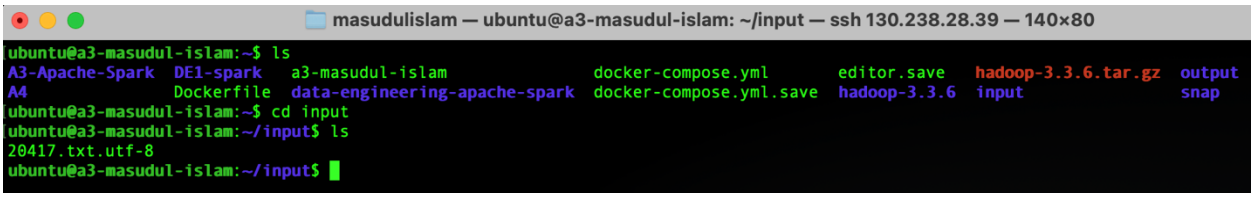


We see that the file '20417.txt.utf-8' has been downloaded to the input folder.

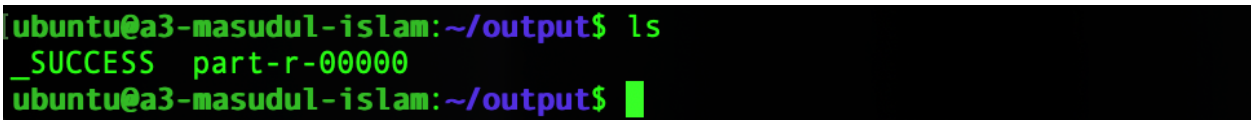


3.Then go back to your home directory to execute the following command to run the canonical MapReduce example: hadoop-3.3.6/bin/hadoop jar ./hadoop-3.3.6/share/hadoop/mapreduce/hadoop- mapreduce-examples-3.3.6.jar wordcount ./input ./output

After running the given command, a folder named 'output' is created. In the screenshot, I was supposed to show the command and output folder together, but I forgot to do that.



4. Answer the following questions:
a. Look at the contents of the folder “output”, what files are placed in there? What do they mean?



Inside **output**, we found two items: **SUCCESS** and **part-r-00000**. The **_SUCCESS** file indicates that a Hadoop or Spark job completed successfully. The **part-r-00000** file is likely the output of that job.

b. Looking at the output files, how many times did the word ‘Discovery’ (case-sensitive) appear? (hint: commands grep/cat could be useful)

The word “Discovery” appers 5 times.

c. In this example we used Hadoop in “Local (Standalone) Mode”. What is the difference between this mode and the Pseudo-distributed mode?

In Local (Standalone) Mode, Hadoop is configured to run in a non-distributed environment as a single java process. This mode is typically used for development, testing, and debugging. Pseudo-distributed Mode, on the other hand, runs each Hadoop daemon in a separate java process on the same machine, simulating a distributed cluster. This mode is useful for more realistic testing while still operating on a single node.

Task 1.2: Setup pseudo-distributed mode

4. Answer the following questions:
a. What are the roles of files core-site.xml and hdfs-site.xml?

Role of core-site.xml:

```
<!-- Put site-specific property overrides in this file. -->

<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

This file contains settings for the core of the Hadoop system. In my configuration, the *fs.defaultFS* property is set to *hdfs://localhost:9000*, which specifies the URI of the Hadoop file system (HDFS). This tells Hadoop that its file system is located at localhost on port 9000.

Role of hdfs-site.xml:

```
<!-- Put site-specific property overrides in this file. -->

<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

This file contains settings specific to the Hadoop Distributed File System (HDFS). In my configuration, the dfs.replication property is set to 1, which means that HDFS will store only one copy of each data block. This setting is crucial for data redundancy and reliability. In a single-node setup like mine, a replication factor of 1 is common since there are no other nodes to replicate the data to.

b. Describe the different services listed when executing ‘jps’. What are their functions in Hadoop and HDFS?

```
ubuntu@a4-masudul-islam:~$ jps
85538 Jps
84970 NameNode
85116 DataNode
ubuntu@a4-masudul-islam:~$
```

Jps (85538):

This is not a Hadoop service but a Java Virtual Machine Process Status Tool. It displays the status of Java processes (Java Virtual Machine processes) for the current user. In this

context, it's used to check whether the Hadoop daemons (NameNode and DataNode) are running.

NameNode (84970):

The NameNode is a important component of HDFS. It manages the directory tree of all files in the file system and tracks where across the cluster the file data is kept. It does not store the data of these files but manages the metadata (like the file system tree, metadata for each file, etc.). The NameNode responds to client requests for file system operations like opening, closing, and renaming files and directories.

DataNode (85116):

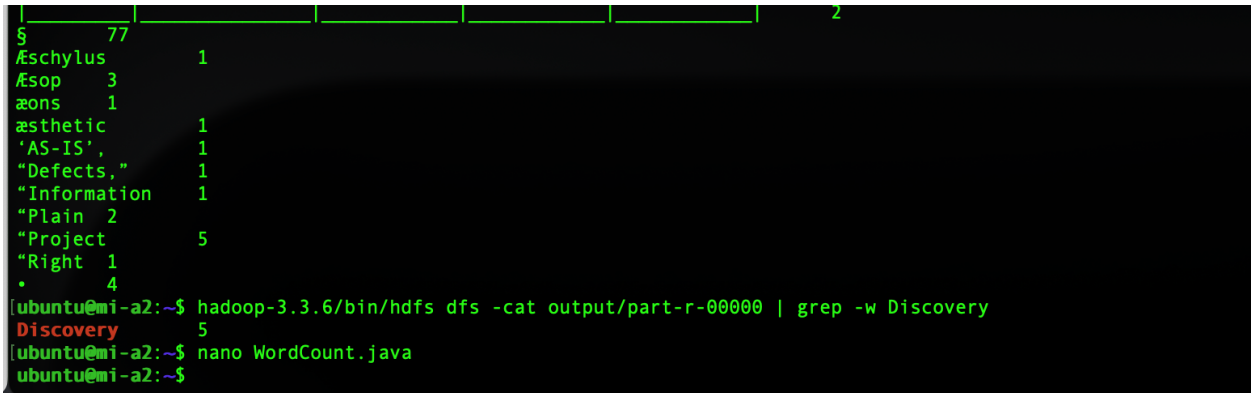
DataNodes are the workhorses of HDFS. They store and retrieve blocks when told to by clients or the NameNode. They report back to the NameNode periodically with lists of blocks that they are storing. Each DataNode serves up blocks of data over the network using a block protocol specific to HDFS.

Task 1.3: Word count in pseudo-distributed mode

Verify that it contains the same word counts you saw before when running the word count program locally.

```
hadoop-3.3.6/bin/hdfs dfs -cat output/part-r-00000
```

```
hadoop-3.3.6/bin/hdfs dfs -cat output/part-r-00000 | grep -w Discovery
```



6. Answer the following questions:

a. Explain the roles of the different classes in the file WordCount.java.

Roles of the different classes in WordCount.java

```
 GNU nano 4.8                                                    WordCount.java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

TokenizerMapper Class (Mapper):

Role: Breaks down the input text (Text value) into words (tokens).

Function: For each word in the input, it emits a key-value pair, with the word as the key (Text type) and 1 as the value (IntWritable type), indicating one occurrence of the word.

IntSumReducer Class (Reducer):

Role: Aggregates the counts for each word.

Function: It sums up the counts (IntWritable values) for each word (key). The output is the word (Text type) and its total count (IntWritable type).

WordCount Class (Main Class):

Role: Configures and sets up the MapReduce job.

Function: Defines the job's specifications, including setting the mapper, combiner, and reducer classes, as well as the input and output formats. It then runs the job and exits based on the job's success.

main Method:

Role: Entry point for the program.

Function: Sets up the job configuration, defines the job's name, input and output data paths, and triggers the execution of the MapReduce job.

b. What is HDFS, and how is it different from the local file system on your VM?

HDFS is a distributed file system designed to run on commodity hardware. It is highly fault-tolerant and designed to be deployed on low-cost hardware.

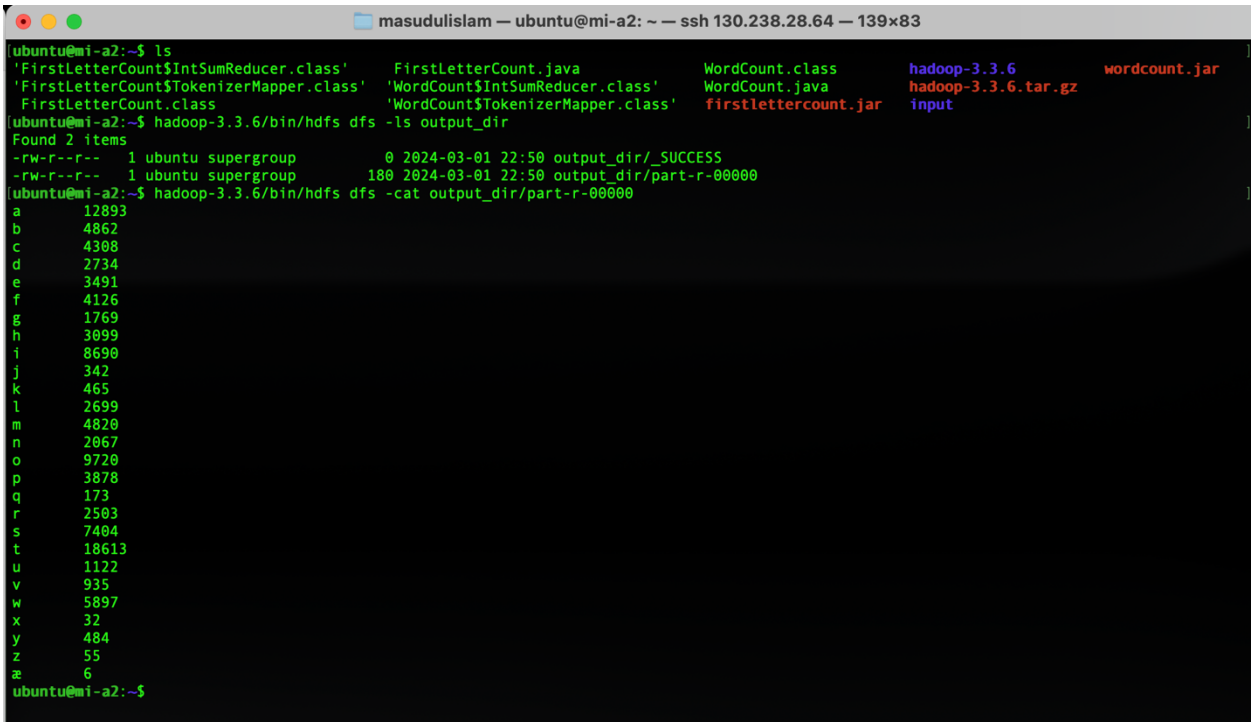
Differences from local file system:

- Storage mechanism:
  - Local file system: Stores data on the local hard drive of a single machine.
  - HDFS: Stores data across a network of machines, distributing data and computations.
- Fault tolerance:
  - Local file system: Limited fault tolerance; failure affects data availability.
  - HDFS: High fault tolerance through data replication across multiple nodes.
- Scalability:
  - Local file system: Limited by the capacity of the individual machine.
  - HDFS: Easily scalable by adding more nodes to the Hadoop cluster.

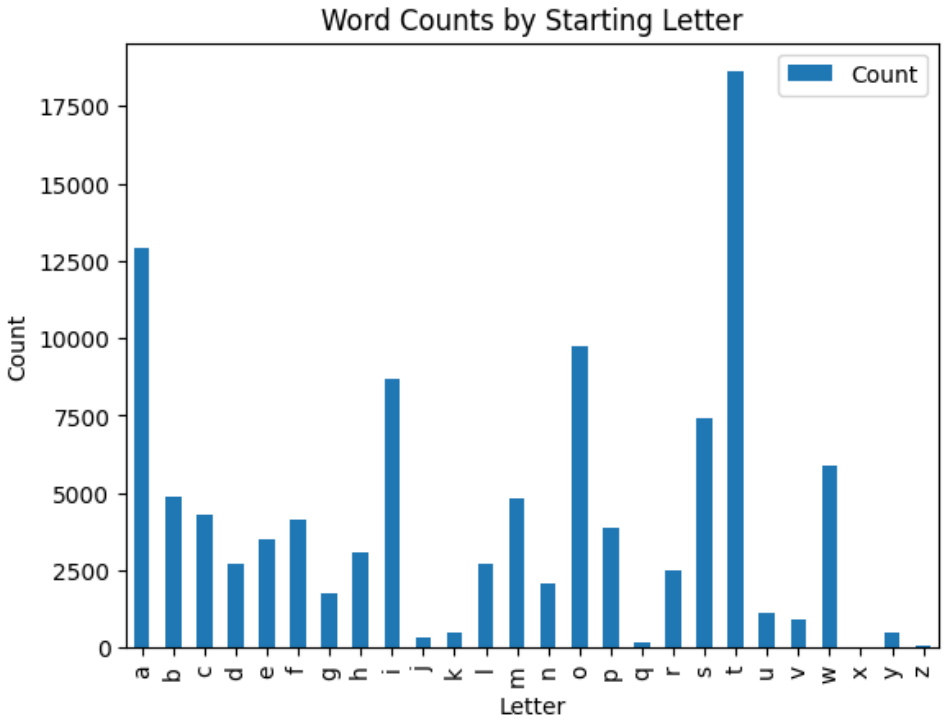
Task 1.4: Modified word count example

1. Modify the above example code in WordCount.java so that it, based on the same input files, counts the occurrences of words starting with the same first letter (ignoring case). The output of

the job should thus be a file containing lines like (if there were 1234 words beginning with A or a): a 1234 b... c...



2. Make a plot that shows the counts for each letter (i.e. excluding all non-letter symbols) and include that in your report. *Do not include the entire output file.*



FirstLetterCount.java file with modified codes and sufficient comments

```
  GNU nano 4.8                                  FirstLetterCount.java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

// Class FirstLetterCount contains the main method and is the entry point of the MapReduce job.
public class FirstLetterCount {

  // TokenizerMapper is a custom Mapper that processes the input text.
  public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    // The map method is called for every line in the input file.
    public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
          String token = itr.nextToken();
          char firstChar = Character.toLowerCase(token.charAt(0));
          // Check if the first character is a letter. If so, write it and its count (1) to the context.
          if (Character.isLetter(firstChar)) {
              word.set(String.valueOf(firstChar));
              context.write(word, one);
          }
      }
    }
  }

  // IntSumReducer is a custom Reducer that sums up the counts.
  public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    // The reduce method is called for each key (in this case, the first letter of words).
    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      // Summing all the occurrences of the particular key (letter).
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      // Writing the letter and its total count to the context.
      context.write(key, result);
    }
  }

  // The main method sets up the configuration and initiates the MapReduce job.
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(FirstLetterCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Task 1.5: NoSQL and MongoDB

1 - Classify the JSON-formatted tweets

Twitter tweets, as represented in JSON format according to the docuemntation, can be classified as **semi-structured data**. This type of data doesn't conform to a rigid structure like tables in relational databases but has some organizational properties that make it easier to analyze. JSON (JavaScript Object Notation) is a typical format for such data.

Characteristics in Tweets: JSON-formatted tweets have a flexible schema. They contain structured elements (like user IDs, timestamps, and tweet IDs) which are consistent across tweets. They also contain unstructured elements like the text of the tweet itself, which can vary greatly in content and format. Metadata, such as hashtags, user mentions, or URLs, further adds layers of semi-structure to the data.

2 - Elaborate on pros and cons for SQL and NoSQL solutions, respectively. Give some examples of particular data sets/scenarios that might be suitable for these types of databases.

SQL databases, traditionally used for structured data, excel in handling consistent, relational data models. They offer strong ACID (atomicity, consistency, isolation, durability) compliance, ensuring transaction reliability and data integrity. SQL databases support complex queries and join operations, beneficial for applications like financial systems or customer relationship management, where data relationships and integrity are crucial. However, they face limitations in horizontal scalability and struggle with rapid schema evolution, making them less suited for environments where data schema is fluid and scaling out is necessary.

On the other hand, **NoSQL databases** cater to a wide range of data models, including key-value, document, columnar, and graph formats. They are highly scalable, making them apt for large-scale applications and big data needs. NoSQL databases handle semi-structured and unstructured data well, offering flexibility in data modeling. This makes them suitable for use cases like social media analytics, content management, and IoT applications, where data comes in various formats and evolves rapidly. However, they may lack full ACID compliance and can present challenges in ensuring data consistency across distributed systems. NoSQL solutions are also a newer technology compared to SQL, which means they are in a continuous state of evolution and might lack the maturity and extensive tooling available in SQL ecosystems.

In conclusion, the choice between SQL and NoSQL databases is driven by the specific requirements of the application. SQL databases are preferred when dealing with structured data and a defined schema is paramount, and transactional integrity is a priority. NoSQL databases are chosen for their flexibility in handling various data types, scalability, and performance efficiency in dealing with large volumes of data or rapidly changing schemas.