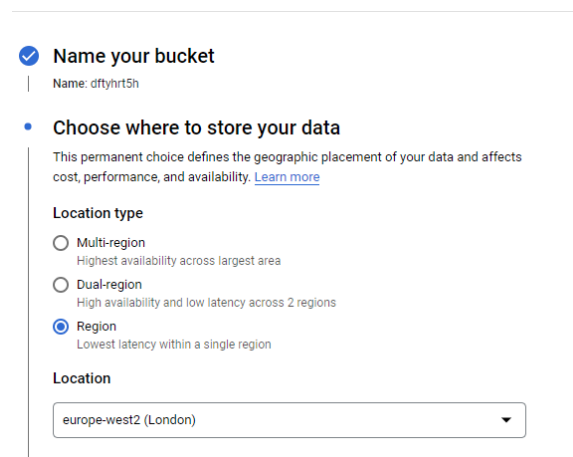


# Cloud Computing Hadoop Map Reduce Report:

## GCP Setup and Additional Features:

### Bucket Setup and Features:

In order to optimise performance for when I run the Dataproc cluster, I selected location type as region and the location as europe-west2. This means that I can select the same location for when I create the Dataproc cluster in order to keep the latency low and optimise the performance since communication and bandwidth will be much better.



The screenshot shows the 'Name your bucket' step of a Google Cloud Storage bucket creation process. The bucket name is 'dftyhrt5h'. Below this, the 'Choose where to store your data' section is active. It explains that the location type affects cost, performance, and availability. Three options are listed: 'Multi-region' (highest availability), 'Dual-region' (high availability and low latency), and 'Region' (lowest latency), with 'Region' selected. The location is set to 'europe-west2 (London)'.

✓ Name your bucket

Name: dftyhrt5h

• Choose where to store your data

This permanent choice defines the geographic placement of your data and affects cost, performance, and availability. [Learn more](#)

Location type

☐ Multi-region  
Highest availability across largest area

☐ Dual-region  
High availability and low latency across 2 regions

☒ Region  
Lowest latency within a single region

Location

europe-west2 (London)

Since it is not specified to keep any files within my bucket private, I have decided to create the bucket with Uniform control and have added the permission Storage Object Viewer for allUsers meaning that all users who access the bucket have read-only permissions to folders and files within the bucket – in this case being the input folder and the output folder. This saves me from having to change the access control of the output folder once generated as it will already be accessible to the public. In other cases if there were folders and files specified not to be public, I would’ve selected fine grained access control when creating the bucket and then specified “gsutil acl ch -u AllUsers:R -r [filepath to the input folder]” and “gsutil acl ch -u AllUsers:R -r [filepath to the output folder]” in order to set the access control as read-only for the input and output folder.



The screenshot shows a table of permissions for a Google Cloud Storage bucket. It has columns for Type, Principal, Name, Role, and Inheritance. There are two rows of permissions: one for 'allUsers' with the role 'Storage Object Viewer', and another for 'Editors of project: coc105-lboro-333910' with the role 'Storage Legacy Bucket Owner'.

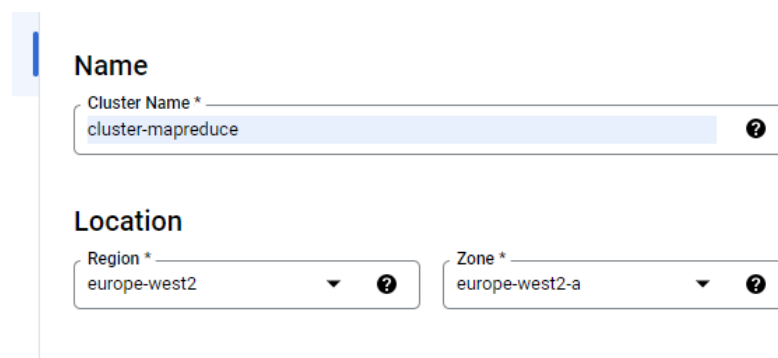
Type	Principal	Name	Role	Inheritance
<input type="checkbox"/>	allUsers		Storage Object Viewer	
<input type="checkbox"/>	Editors of project: coc105-lboro-333910		Storage Legacy Bucket Owner	

I had then copied the coc105-gutenberg-10000books directory into my Google Storage Bucket through cloudshell by running the command: “gsutil -m cp -r gs://coc105-gutenberg-10000books

gs://ngram-MapReduce-assignment/", so that it is set as the input directory for later when I run my MapReduce program on my dataproc cluster. Here it is also worth mentioning that using the -m command when copying this allowed for parallel multi-processing when copying thus speeding the process.

### Dataproc Cluster Setup and Features:

In order to keep latency low and optimise performance, I set the region of the cluster as europe-west2 as this is the same region as the Cloud Storage Bucket, so communication between the servers of the cluster and the bucket would be significantly better as compared to them being in different locations, improving the read/write speed drastically. Furthermore the payment rate of communicating between Google Cloud services within the same location is cheaper.



The screenshot shows a configuration form for a Dataproc cluster. Under the 'Name' section, there is a text input field for 'Cluster Name \*' with the value 'cluster-mapreduce' and a help icon. Under the 'Location' section, there are two dropdown menus: 'Region \*' set to 'europe-west2' and 'Zone \*' set to 'europe-west2-a', both with help icons.

The size of the input directory is roughly 3.6GB, and thus for block sizes of 64MB this equates to about 56 maps. Since there are, by default, 10-100 mappings per node, for this input size 2 worker nodes are sufficient in the Dataproc cluster.

Another feature I added when creating the cluster is scheduled deletion after a cluster idle time period without any submitted jobs. This is so that the cluster does not drain my credits if the situation arises where the map reduce program has finished and the cluster hasn't been deleted manually.

#### Scheduled deletion

Use Scheduled Deletion to help avoid incurring Google Cloud charges for an inactive cluster.  
[Learn more](#)

☒ Delete on a fixed time schedule

☐ Delete cluster at a specified future time

☐ Delete after elapsed time since creation

☒ Delete after a cluster idle time period without submitted jobs

Timeout \*  Minutes ▼

## MapReduce Code Setup and Additional Features:

### NGram part:

My Student ID is B626182, so in order to calculate my n in ngrams:  $(626182 \bmod 5) + 1 = 3$ ; therefore, I would have to create a map reduce program which counts the occurrences of trigrams from the 10,000 books dataset. I used a local VM instance running Ubuntu for testing my program on 1-2 books I uploaded in an input folder I created on the local VM. Once working I then would test it out on the Dataproc Cluster I created in order to save credits from running it on the Cluster each and every time during testing.

Before the MapReduce function will run, it is essential to get rid of any punctuation marks within the input files so that, for example, 'hello my name' and 'hello my name,' aren't counted as separate trigrams. In order to do this, I replaced all punctuation (using the regular expression `\\p{P}`) with whitespace, which then is set as the delimiter so that no punctuation is outputted.

To implement the trigram, I added conditional statements nested in the 'while String Tokenizer has more tokens' loop in the mapper class. The initial trigram would run through each if statement and the while loop 3 times, setting the value for the first word (based on the first token received), second word (second token on the second while loop run), and then the third word (third token on the third while loop run) respectively. For every loop after this, the second word will be set as the first and the third word set as the second, whilst the third word would be set as the new token word. The trigram would then be set as these words and passed through the reducer.

Not much needs to change in the reducer class, as it simply takes the trigrams from the mapper class and sums up the occurrences of each trigram.

```
34    );
35
36    while (itr.hasMoreTokens()) {
37        String str = itr.nextToken();
38        if ((!(str.equals("")) && (str != null) && (str.matches("[a-zA-Z]*")))) // if the string is a word
39        {
40            if ((prevWord1 != null) && (prevWord2 != null)) { // Setting the trigram, then setting the the first word as the second word and second word as the first
41                word.set(prevWord1 + " " + prevWord2 + " " + str);
42                prevWord1 = prevWord2;
43                prevWord2 = str;
44                context.write(word, one); //writing the occurrence of the trigram
45            } else if ((prevWord2 == null) && (prevWord1 != null)) { // For the initial trigram set the 2nd word as the 2nd token
46                prevWord2 = str;
47            } else if ((prevWord1 == null) && (prevWord2 == null)) { // For the initial trigram set 1st token as the first word
48                prevWord1 = str;
49            }
50        }
51    }
52 }
53 }
```

### Sorting the trigrams via WritableComparator:

Another feature that I have added to my map reduce program is sorting the trigrams by their length, in descending order. This will output in the order of the trigram with the largest size and ending with the trigram with the smallest size. To implement this feature I used Hadoop WritableComparator

with the optimization hook, which overrides the default sorting (`public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2)`) and allows for custom sorting.

```
74 }
75 }
76
77 public static class IntComparator extends WritableComparator { // this class compares the trigrams based on no. of bytes (ie. their length using Java's ByteBuffer)
78     public IntComparator() {
79         super(IntWritable.class);
80     }
81
82     @Override
83     public int compare (byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
84         Integer v1 = ByteBuffer.wrap(b1, s1, l1).getInt();
85         Integer v2 = ByteBuffer.wrap(b2, s2, l2).getInt();
86         return v1.compareTo(v2) * (-1);
87     }
88 }
89
```

The comparator compares the size of the byte representation of 2 objects and returns an integer value of the comparison. The reducer calls this class which has been set in the driver as `job.setSortComparatorClass(IntComparator.class)` before it sums up the number of occurrences.

```
job.setSortComparatorClass(IntComparator.class);
```

#### Combiner Class:

In order to optimise the efficiency of the map reduce task, I have implemented a combiner class. The combiner class aggregates all of the same keys from the mapper before passing on the key-value pairs over the network to the reducers. Thus, this reduces the load of key-value pairs sent over to the reducer, and in turn leads to more efficient processing of data. Therefore, the combiner class is the exact same as the reducer class, however it runs the reducer task on the output of each map task before being passed through to the reducers which aggregate all occurrences of the same word amongst all mappings. So in order to implement this I have defined “`job.setCombinerClass(WCReducer.class)`” which sets the combiner class as the same reducer class.

```
job.setCombinerClass(WCReducer.class);
```

#### Setting The Number of Reducers for the Task:

When running my MapReduce program on Google’s Dataproc cluster, I found that the default number of reducers was 1 when I was running 2 worker nodes each with 4 processors. In order to improve performance, I therefore set the number of reducers to  $0.95 \times 2 \times 4 =$  rounded to 7 reducers which seems to be the optimal number for these nodes and processors. I did this by setting `job.setNumReduceTasks(7)`. This created 7 files in my output directory, each for the output of each reducer.

```
job.setReducerClass(WCReducer.class);
job.setNumReduceTasks(7); // 2 nodes and with 4 cores per node in my cluster therefore 7 is optimal reducer number
job.setOutputKeyClass(Text.class);
```

### Hadoops small files problem:

I encountered an issue when running my MapReduce task in dataproc. There are 10,000 small files in the input directory, for which hadoop creates a map for each one, resulting in 10,000 maps. Even setting the number of mappers manually did not resolve this splitting of 10,000 maps. The issue with this was that it in-turn led to inefficient data processing, where mappers are handling much less data than the block size of 64mb. One resolution I found for resolving this issue was to pre-process the input data using the CombineTextInputFormat class available as a hadoop import. This function splits the input files into larger chunks that are less than the max size set (I chose a max size of 248mb resulting in 15 splits as this will not result in such a huge split that there's just 1 split of 3.6GB and so will still result in an adequate number of maps). This dramatically improved the efficiency, decreasing the elapsed time by half.

### Bug Fixing:

```
91
92 public static void main(String[] args) throws Exception {
93     Configuration conf = new Configuration();
94     conf.set("mapreduce.input.fileinputformat.split.maxsize", "268435456"); //concatenate the input files under a max size of 248mb
95     Job job = Job.getInstance(conf, "word count");
96     job.setJarByClass(WordCount.class);
97     job.setMapperClass(WCMapper.class);
98     job.setReducerClass(WCReducer.class);
99     job.setInputFormatClass(CombineTextInputFormat.class); //set the input format class to the imported CombineTextInputFormat
100    job.setNumReduceTasks(7); //2 nodes and with 4 cores per node in my cluster therefore 7 is optimal reducer number
101    job.setOutputKeyClass(Text.class);
102    job.setOutputValueClass(IntWritable.class);
103 }
```

However, one issue I noticed with the output when running this was that when combining files, the output doesn't distinguish trigrams between different files. I also noticed that the combiner doesn't set a delimiter between the last word and first word in a file. The output that has resulted in my Google Storage Bucket is in relation to this issue, however to undo it you can simply comment out the 2 lines in the screenshot above and run the code on the dataproc cluster again. Unfortunately due to running out of credits I was no able to output this in time.