ARTIFICIAL INTELLIGENCE AND DATA ENGINEERING
project report for the examination
of
Cloud Computing

Student:
Alessandro Diana

AA. 2024/2025

Contents

List of Figures

1 Introduction

1.1 Project description

This report outlines the design of the project, its various components, key decisions, and implementation steps.
The project consist in writing a MapReduce program in Hadoop that computes word co-occurrences. The CoOccurrence idea is to count the number of times each pair of $word_i$ and $word_j$ occurs in a collection of text files.
For the project you have to do:

- Implement a CoOccurrence MapReduce algorithm using the Hadoop framework and Pairs design pattern;
- Test your implementation in the cluster;
- Write a short project report detailing the implementation and experimental results.

The other optional features are:

- Consider the Stripes design pattern (instead of the Pairs one);
- Consider a wider definition of a neighbor (window) of a term;
- Use combiners and/or more than one reducer;
- Use the Mapper and Reducer classes setup() and/or cleanup() methods.

In the version discussed in this paper, the project has all the points listed.

1.2 Code and Files organization

All the developed code and documentation can be found at this link:
https://github.com/Arzazrel/Project_CC_25.
The readme file in the repository contains all the instructions and explanations concerning the code, documentation, and results.

1.3 Hadoop Mode and VM Architecture

Hadoop can run in three modes:

- Local (or standalone): it runs as a single Java process and doesn't use daemons. Hadoop uses the local file system as a substitute for the HDFS file system, and the jobs will run as if there is 1 mapper and 1 reducer;
- Pseudo-distributed (single-node cluster): all the daemons run on a single machine locally, and there is a separate JVM for every Hadoop component (mimics the behaviour of a cluster). Hadoop uses the HDFS protocol, and there can be multiple mappers and reducers;
- Fully-distributed: all the daemons run on a subset of the cluster's machines using the HDFS protocol, and there are multiple mappers and reducers.

For this project, I used Hadoop in pseudo-distributed mode running on a VM in a cluster provided by the University of Pisa.
The assigned VM has this architecture:

- Single-core Virtual CPU;
- 7 GB of RAM;
- 1 Hard Disks of 40 GB for the OS (/dev/sda)
- 1 Network Adapters

1.4 Document collection

The collection of documents used for this project occupies 9.068 MB and contains 782 text files with names lineXXX, where XXX runs from 000 to 781. Each input file contains a text on multiple lines, including numbers, punctuation, and so on.
Each line will be stripped of special characters and punctuation and divided into tokens. These tokens will be the words that will be counted. The algorithm and cleaning configuration used found 41,166 distinct words. The total occurrence of these words (total number of tokens in the collection) reaches 1,734,299, in which only the three most frequent words exceed 50,000 occurrences ("the", "and", "of"), and the tenth most frequent word already has fewer than 20,000 occurrences.

2 Design patterns used

2.1 General

Designing patterns for Hadoop are solutions to common problems in developing distributed applications using the MapReduce paradigm.
These patterns are principles/guidelines to facilitate programming for certain types of general problems (classes of problems with certain characteristics) that help to write more efficient, readable, maintainable code and make the most of the power of distributed computing.

2.2 In-Mapper Combining

Intermediate data can be much larger than the input size, so aggregating locally for each mapper can help greatly by reducing the size of intermediate outputs. The normal combiner, which is usually the same as the reducer but done locally, is optional, meaning that Hadoop can choose to run it or not.
The in-mapper combiner is always executed because it is part of the mapper logic, which is mandatory (via the setup and cleanup functions).
Advantages:

- provides complete control over the local aggregation process;
- reduces both the amount of intermediate data that will be transferred over the network and the data emitted by the mapper and saved on the local disk.

Disadvantages:

- Memory management is more complex; no automatic spilling is performed;
- Increased memory usage, all keys must be kept in RAM until the end of the mapper (risk an OutOfMemoryError);
- More complex code, you need to manage persistent data structures, initialization, and memory release.

2.3 Pairs

For the problem of matrix generation is to generate a square output of size NxN given an input of size N. An example of this problem is the the one addressed in this project, CoOccurrence. Using Pair methodology, the key is formed by the pair of words that make up the bigram. This method is straightforward to implement, requires low memory usage, and yields a high number of mapper outputs, thereby increasing network utilization.

2.4 Stripes

For the problem of matrix generation, given an input of size N, generate a square output of size NxN. For this type of problem stripes method can be used. This method works on lines, creating an associative array for each term in a line, which will contain the neighbour in the bigram as the key and the number of occurrences for that bigram in that record as the value. Then, an associative array (stripe) is created to keep track of co-occurrences with its neighbours each time a new term is found. This method is more complex to implement than the pairs method and leads to fewer mapper outputs and less network traffic at the cost of greater memory usage.

3 Code

3.1 General

This section shows images of the code, focusing on the parts that implement the logic of the mapper and reducer. The complete programmes are available in the project's GitHub folder.
All the codes for the various approaches to solving the problem are divided as follows:

- basic version of the methodology (e.g., CoOccurrencePairs or CoOccurrenceStripes) which executes the basic methodology and allows to use of the external combiner (if activated);
- InMap version of the methodology (e.g., CoOccurrencePairsInMap or CoOccurrenceStripesInMap), which executes the version with in-map combining.

Each code has a main that receives the various parameters as input (number of reducers, use of combiners, number of runs), checks their values, and executes one of the methodologies, with a specific configuration, several times (jobs) decided by the user.
The various jobs performed will have the same configuration and will be executed in the same way. The various iterations are needed to collect more data for the same configuration and obtain a more accurate and statistically significant trend.
The main contains a section for reading the most important statistics, calculating averages for certain statistics, and saving them and the job information in a file. The file is saved locally where the job execution command is launched and not on HDFS, as this is only for testing purposes for the project, which is executed in pseudo-distributed mode.
The setup and clean-up methods are used to implement the various InMap-combining methods and are not used for the basic versions, except for the N-Gram case, where the setup method is used to pass the window size.

```java
/**
 * Reducer class to implement the reduce logic for the word count
 */

public static class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    private final IntWritable result = new IntWritable();   // var to set the value to associate with the found words

    // reduce function

    public void reduce(final Text key, final Iterable<IntWritable> values, final Context context)
            throws IOException, InterruptedException {
        int sum = 0;                            // var to the sum of the values associated to keys
        for (final IntWritable val : values) {
            sum += val.get();                   // update sum
        }
        result.set(sum);                // set the result with sum
        context.write(key, result);     // emit
    }
}
```

Figure 1: Code of the reducer and combiner of the case word count base.

3.2 WordCount

Code images related to the implementation of the simplest case, in which words are counted individually.

3.2.1 Base

Basic implementation in which every time a word is encountered, a key-value pair composed in this way will be placed: word-1. No optimisation except for the possibility of using the normal combiner, which will be the same as the reducer.
These images show the map function 2, the reduce function 1, and the configuration and job startup section in the main 3.

```
public static class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{

    private final static IntWritable one = new IntWritable(1);  // var to set the value to associate with the found words

    private final Text word = new Text();       // var which will contain the word that will be used as the key

    // map function

    public void map(final LongWritable key, final Text value, final Context context)
            throws IOException, InterruptedException {
        String[] words = value.toString().toLowerCase()
                .replaceAll("[^\\p{L}0-9\\s'-]", " ") // removes unwanted characters, keeps only letters, numbers and ',-
                .replaceAll("(?<=\\s)['-]+|[\'-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
                .replaceAll("(^|\\s)['-]+", " ")                // removes '-' or ''' at the beginning of the word
                .replaceAll("[-']+(\\s|$)", " ")                // removes '-' or ''' at the end of the word
                .trim()                                         // removes leading and trailing whitespace
                .split("\\s+");// splits the string into an array of words, using one or more consecutive whitespace as separators.

        for (int i = 0; i < words.length; i++)      // iterate over each word obtained from the line
        {
            if (!words[i].isEmpty())    // check if the current word is not empty
            {
                word.set(words[i]);         // set the var text with the current word
                context.write(word, one);   // emit the pair (key,value)
            }
        }
    }
}
```

Figure 2: Code of the mapper of the case word count base or with external combiner.

```
final Configuration conf = new Configuration(); // create configuration object
final Job job = Job.getInstance(conf, jobName + "_run_" + successfulRuns + "_comb_" + useCombiner + "_red_" + numReducer);
job.setJarByClass(WordCount.class);

job.setOutputKeyClass(Text.class);              // set the typer for the output key for reducer
job.setOutputValueClass(IntWritable.class);     // set the typer for the output value for reducer

job.setMapperClass(WordCountMapper.class);          // set mapper
if (useCombiner)
    job.setCombinerClass(WordCountReducer.class);   // set combiner -> See NOTE 1
job.setReducerClass(WordCountReducer.class);        // set reducer
job.setNumReduceTasks(numReducer);                  // to set the number of the reducer task

FileInputFormat.addInputPath(job, new Path(inputPath));     // first argument is the input folder

// Output folder specific for each successful run
String outputPath = outputBasePath + "/output_run_" + successfulRuns;   // set sub-path for the output
FileOutputFormat.setOutputPath(job, new Path(outputPath));              // give the output path to the job
System.out.println("\n--- Starting Job Attempt " + attempt + " ---");

startTime = System.currentTimeMillis();         // start time
boolean success = job.waitForCompletion(true);  // wait the end of the job
endTime = System.currentTimeMillis();           // end time
```

Figure 3: Code of the main case word count base or with external combiner.

3.2.2 InMap combining

In the version with in-map combining, the mapper (shown in image 4) doesn't emit every time a word is encountered, but an associative array is kept to store the occurrence of each word found, updating it each time it is encountered.
This associative array isn't for a single call of the map function, but for all. It is initialised at the beginning with the setup function and then cleaned up at the end with the cleanup function. This should significantly reduce the number of values emitted at the cost of more complicated logic and bigger memory usage. Memory usage must also be managed to prevent the associative array from growing too large and exceeding the available memory. When a certain usage threshold (80%) is reached, the data collected so far will be emitted and the associative array will be freed. The reducer will remain the same as in the base case, as will the main, except that there is no option to use the external combiner.

```
public static class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{

    private HashMap<String, Integer> wordCountMap;             // hash map to contain word occurrences

    private static final double MEMORY_THRESHOLD = 0.8;        // maximum usable memory threshold (80%)

    private final IntWritable countWritable = new IntWritable();// var to set the value to associate with the found words

    private final Text word = new Text();           // var which will contain the word that will be used as the key

    // to initialize the data structures used fo in mapping

    protected void setup(Context context) throws IOException, InterruptedException {
        wordCountMap = new HashMap<>();     // set the hash map
    }

    // map function

    public void map(final LongWritable key, final Text value, final Context context)
            throws IOException, InterruptedException {
        String[] words = value.toString().toLowerCase()
                .replaceAll("[^\\p{L}0-9\\s'-]", " ")          // removes unwanted characters, keeps only letters, numbers and ',-
                .replaceAll("(?<=\\s)['-]+|[\'-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
                .replaceAll("^[\\s]['-]+", " ")                // removes '-' or ''' at the beginning of the word
                .replaceAll("[-']+(\\s|$)", " ")               // removes '-' or ''' at the end of the word
                .trim()                                        // removes leading and trailing whitespace
                .split("\\s+");                                // splits the string into an array of words, using one or more consecut

        for (int i = 0; i < words.length; i++)      // iterate over each word obtained from the line
        {
            if (!words[i].isEmpty())    // check if the current word is not empty
            {
                wordCountMap.put(words[i], wordCountMap.getOrDefault(words[i], 0) + 1); // set or update the value per the current word
            }
        }

        if (isMemoryThresholdExceeded())    // check the used memory
        {
            flush(context);     // emit and flush memory
        }
    }
// to close the data structures used fo in mapping

protected void cleanup(Context context) throws IOException, InterruptedException {
    flush(context);         // emit and flush memory
}

// ---------------------- start: utility functions for in-combining ----------------------
// function to check the current used memory

private boolean isMemoryThresholdExceeded() {
    long maxMemory = Runtime.getRuntime().maxMemory();      // get maximum memory the JVM can use (imposed limit).
    long totalMemory = Runtime.getRuntime().totalMemory();  // get current total memory allocated to the JVM.
    long freeMemory = Runtime.getRuntime().freeMemory();    // get Free memory inside totalMemory()

    long usedMemory = totalMemory - freeMemory;                // calculate the current used memory
    long memoryLimit = (long) (maxMemory * MEMORY_THRESHOLD);   // calculate the threshold

    return usedMemory >= memoryLimit;       // check if the current used memory exceeds the threshold
}

// function to emit the data collected and flush the memory

private void flush(Context context) throws IOException, InterruptedException {
    // loop through the entire contents of the hasmap and output the key-values (word-occurrence) stored
    // inside and then free the hashmap.
    for (Map.Entry<String, Integer> entry : wordCountMap.entrySet()) {
        word.set(entry.getKey());                   // set the word
        countWritable.set(entry.getValue());        // set the occurrence
        context.write(word, countWritable);         // emit
    }
    wordCountMap.clear();                           // clean hash map
}
// ---------------------- end: utility functions for in-combining ----------------------
```

Figure 4: Code of the mapper of the word count in-map combining case.

3.3 CoOccurrence

Code images related to implementations for co-occurrence counting. In this section, we will examine both the "Pairs" and "Stripes" techniques, presented in both basic and in-map combining versions.

3.3.1 Pairs base

Basic implementation in which every time a couple of words are encountered, a key-value pair composed in this way will be emitted: $word_i$,$word_{i+1}$. No optimisation except for the possibility of using the normal combiner, which is the same as the reducer.
These images show the map function 5, the reduce function 6, and the configuration and job startup section in the main 7.

```java
public static class CoOccurrenceMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{

    private final static IntWritable ONE = new IntWritable(1);  // the value for each pairs

    private Text pair = new Text();                              // var to contain the key for each pair (word1,word2

    // map function

    protected void map(final LongWritable key,final Text value,final Context context)
            throws IOException, InterruptedException {
        String[] words = value.toString().toLowerCase()
                .replaceAll("[^\\p{L}0-9\\s'-]", " ")           // removes unwanted characters, keeps only letters,
                .replaceAll("(?<=\\s)['-]+|[\'-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
                .replaceAll("(^|\\s)['-]+", " ")                // removes '-' or ''' at the beginning of the word
                .replaceAll("[-']+(\\s|$)", " ")                // removes '-' or ''' at the end of the word
                .trim()                                         // removes leading and trailing whitespace
                .split("\\s+");                                 // splits the string into an array of words, using o

        if (words.length < 2)   // check for record withs size less than the window of N-gram (in this case 2-gram)
            return;

        for (int i = 0; i < words.length - 1; i++)         // iterate over each word obtained from the line
        {
            String w1 = words[i];           // take the current word
            String w2 = words[i + 1];       // take the next word
            if (w1.isEmpty() || w2.isEmpty())   // control check for the key
                continue;
            pair.set(w1 + "," + w2);        // create the key = (current word,next word)
            context.write(pair, ONE);       // emit key-value pairs
        }
    }
}
```

Figure 5: Code of the mapper of the case Pairs CoOccurrence count base.

```java
/**
 * Reducer class to implement the reduce logic for the CoOccurrence count
 */

public static class CoOccurrenceReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    private final IntWritable result = new IntWritable();   // var to set the value to associate with the found words

    // reduce function

    public void reduce(final Text key, final Iterable<IntWritable> values, final Context context)
            throws IOException, InterruptedException {
        int sum = 0;                            // var to the sum of the values associated to keys
        for (final IntWritable val : values) {
            sum += val.get();                   // update sum
        }
        result.set(sum);                // set the result with sum
        context.write(key, result);     // emit
    }
}
```

Figure 6: Code of the reducer and combiner for the case Pairs CoOccurrence count base.

```
final Configuration conf = new Configuration(); // create configuration object
final Job job = Job.getInstance(conf, jobName + "_run_" + successfulRuns + "_comb_" + useCombiner + "_red_" + numReducer);
job.setJarByClass(CoOccurrencePairs.class);

job.setOutputKeyClass(Text.class);             // set the typer for the output key for reducer
job.setOutputValueClass(IntWritable.class);    // set the typer for the output value for reducer

job.setMapperClass(CoOccurrenceMapper.class);          // set mapper
if (useCombiner)
    job.setCombinerClass(CoOccurrenceReducer.class);   // set combiner -> See NOTE 1
job.setReducerClass(CoOccurrenceReducer.class);        // set reducer
job.setNumReduceTasks(numReducer);                     // to set the number of the reducer task

FileInputFormat.addInputPath(job, new Path(inputPath));    // first argument is the input folder

// Output folder specific for each successful run
String outputPath = outputBasePath + "/output_run_" + successfulRuns;   // set sub-path for the output
FileOutputFormat.setOutputPath(job, new Path(outputPath));             // give the output path to the job
System.out.println("\n--- Starting Job Attempt " + attempt + " ---");

startTime = System.currentTimeMillis();        // start time
boolean success = job.waitForCompletion(true);  // wait the end of the job
endTime = System.currentTimeMillis();          // end time
```

Figure 7: Code of the main for the word Pairs CoOccurrence count or with external combiner.

```
// to initialize the data structures used for in mapping

protected void setup(Context context) throws IOException, InterruptedException {
    wordCountMap = new HashMap<>();     // set the hash map
}

// map function

protected void map(final LongWritable key,final Text value,final Context context)
        throws IOException, InterruptedException {
    String[] words = value.toString().toLowerCase()
            .replaceAll("[^\\p{L}0-9\\s'-]", " ")          // removes unwanted characters, keeps only letters, numbers
            .replaceAll("(?<=\\s)['-]+|['-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
            .replaceAll("(^|\\s)['-]+", " ")               // removes '-' or ''' at the beginning of the word
            .replaceAll("[-']+(\\s|$)", " ")               // removes '-' or ''' at the end of the word
            .trim()                                        // removes leading and trailing whitespace
            .split("\\s+");                                // splits the string into an array of words, using one or mo

    if (words.length < 2)   // check for record withs size less than the window of N-gram (in this case 2-gram)
        return;

    for (int i = 0; i < words.length - 1; i++)         // iterate over each word obtained from the line
    {
        String w1 = words[i];           // take the current word
        String w2 = words[i + 1];       // take the next word
        if (w1.isEmpty() || w2.isEmpty())   // control check for the key
            continue;
        String pair = w1 + "," + w2;    // create the key = (current word,next word)
        wordCountMap.put(pair, wordCountMap.getOrDefault(pair, 0) + 1); // set or update the value per the current word
    }

    if (isMemoryThresholdExceeded())    // check the used memory
    {
        flush(context);     // emit and flush memory
    }
}

// to close the data structures used fo in mapping

protected void cleanup(Context context) throws IOException, InterruptedException {
    flush(context);         // emit and flush memory
}
```

Figure 8: Code of the mapper of the Pairs CoOccurrence count with in-map combining.

3.3.2 Pairs InMap Combining

In the version with in-map combining, the mapper (shown in image 8) doesn't emit every time a word pair is encountered, but an associative array will be kept that stores the occurrence of each word pair found, updating it each time it is encountered.
This associative array isn't for a single call of the map function, but for all. It is initialised at

the beginning with the setup function and then cleaned up at the end with the cleanup function. This should significantly reduce the number of values emitted at the cost of more complicated logic and bigger memory usage. Memory usage must also be managed to prevent the associative array from growing too large and exceeding the available memory. When a certain usage threshold (80%) is reached, the data collected so far will be emitted and the associative array will be freed. The reducer will remain the same as in the base case, as will the main, except that there is no option to use the external combiner.
This image 8 shows only the map, setup, and cleanup functions. The variables, memory management logic, and emit are identical to the word count case with in-map combining 4, so they will not be shown here in order to limit the number of pages in the documentation.

```
protected void map(final LongWritable key,final Text value,final Context context)
        throws IOException, InterruptedException {
    HashMap<String, MapWritable> wordSeenMap = new HashMap<>(); // hash map to contain the words already seen
    String[] words = value.toString().toLowerCase()
            .replaceAll("[^\\p{L}0-9\\s'-]", " ")           // removes unwanted characters, keeps only letters, n
            .replaceAll("(?<=\\s)['-]+|['-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
            .replaceAll("(^|\\s)['-]+", " ")                // removes '-' or ''' at the beginning of the word
            .replaceAll("['-]+(\\s|$)", " ")                // removes '-' or ''' at the end of the word
            .trim()                                         // removes leading and trailing whitespace
            .split("\\s+"); // splits the string into an array of words, using one or more consecutive whitespace

    if (words.length < 2)   // check for record withs size less than the window of N-gram (in this case 2-gram)
        return;

    for (int i = 0; i < words.length - 1; i++)      // iterate over each word obtained from the line
    {
        String w1 = words[i];                           // take the current word
        String w2 = words[i + 1];                       // take the next word
        if (w1.isEmpty() || w2.isEmpty())               // control check for the key
            continue;

        if (!wordSeenMap.containsKey(w1))   // check if the current word has it already been seen or not
            wordSeenMap.put(w1, new MapWritable());     // set the stripes for the new word

        // update the value for the co-occurrence
        MapWritable stripe = wordSeenMap.get(w1);       // get stripe for the current word
        Text neighbor = new Text(w2);                   // neighbour of the current word

        if (stripe.containsKey(neighbor))       // neighbor already seen, value must be updated
        {
            IntWritable countWritable = (IntWritable) stripe.get(neighbor); // get old occurrence
            int count = countWritable.get();
            stripe.put(neighbor, new IntWritable(count + 1));              // update occurrence
        }
        else                                    // new neighbor
        {
            stripe.put(neighbor, new IntWritable(1));
        }
    }

    // now emit all the word and associated stripes -> format emit: (word,stripes) -> all word have a stripes
    //                                             (hash map) formatted in this way: (word[i+1], occurrence)
    for (Map.Entry<String, MapWritable> entry : wordSeenMap.entrySet()) {
        Text word = new Text(entry.getKey());             // get key (word)
        MapWritable stripe = entry.getValue();            // get stripe
        context.write(word, new MapWritable(stripe));     // emit (word, stripe)
    }
}
```

Figure 9: Code of the mapper of the Stripes CoOccurrence count.

3.3.3 Stripes base

Basic implementation in which for each distinct word in the line, an associative array is created, which will contain the neighbour in the bigram as the key and the number of occurrences for that bigram, in that record, as the value.
These images show the map function 9, the combiner 11, the reduce function 12 and the configuration and job startup section in the main 10.
In this case, the combiner must be different from the reducer because if the reducer logic were used as a combiner, the output types would not match the input types required by the reducer. The reducer requests a pair of (Text, Iterable<MapWritable>) as input and returns pairs (Text, Text). Therefore, the reducer must accept pairs (Text, Iterable<MapWritable>) as input and return them as output. Then it will merge the different stripes for the same word (key).

```
final Configuration conf = new Configuration(); // create configuration object
final Job job = Job.getInstance(conf, jobName + "_run_" + successfulRuns + "_comb_" + useCombiner + "_red_" + numReducer);
job.setJarByClass(CoOccurrenceStripes.class);

job.setMapOutputKeyClass(Text.class);          // set the typer for the output key for mapper
job.setMapOutputValueClass(MapWritable.class);  // set the typer for the output value for mapper

job.setOutputKeyClass(Text.class);             // set the typer for the output key for reducer
job.setOutputValueClass(Text.class);           // set the typer for the output value for reducer

job.setMapperClass(CoOccurrenceMapper.class);          // set mapper
if (useCombiner)
    job.setCombinerClass(CoOccurrenceCombiner.class);   // set combiner -> See NOTE 1
job.setReducerClass(CoOccurrenceReducer.class);        // set reducer
job.setNumReduceTasks(numReducer);                     // to set the number of the reducer task

FileInputFormat.addInputPath(job, new Path(inputPath));     // first argument is the input folder

// Output folder specific for each successful run
String outputPath = outputBasePath + "/output_run_" + successfulRuns;   // set sub-path for the output
FileOutputFormat.setOutputPath(job, new Path(outputPath));              // give the output path to the job
System.out.println("\n--- Starting Job Attempt " + attempt + " ---");

startTime = System.currentTimeMillis();         // start time
boolean success = job.waitForCompletion(true);  // wait the end of the job
endTime = System.currentTimeMillis();           // end time
```

Figure 10: Code of the main in case word Stripes CoOccurrence count or with external combiner.

```
public static class CoOccurrenceCombiner extends Reducer<Text, MapWritable, Text, MapWritable> {

    protected void reduce(Text key, Iterable<MapWritable> values, Context context)
            throws IOException, InterruptedException {
        MapWritable combinedStripe = new MapWritable(); // define aggregate map for

        for (MapWritable map : values)          // for each stripes associated to a word
        {
            for (Writable k : map.keySet())     // for each neighbour in the stripe
            {
                IntWritable count = (IntWritable) map.get(k);              // get local occurrence
                IntWritable existing = (IntWritable) combinedStripe.get(k); // get total occurrence

                if (existing != null)       // this pair of words has been found before
                {
                    int updatedCount = existing.get() + count.get();       // create updated count
                    combinedStripe.put(k, new IntWritable(updatedCount));   // update total occurrence
                } else                      // this pair of words has not been found before
                    combinedStripe.put(k, new IntWritable(count.get()));    // set (first time) the total occurrence
            }
        }

        context.write(key, combinedStripe);        // emit word and aggregate stripe associated
    }
}
```

Figure 11: Code of the combiner in word Stripes CoOccurrence count.

```java
public static class CoOccurrenceReducer extends Reducer<Text, MapWritable, Text, Text> {
    // reduce function

    protected void reduce(Text key, Iterable<MapWritable> values, Context context)
            throws IOException, InterruptedException {
        MapWritable aggregateMap = new MapWritable();       // define aggregate map

        for (MapWritable map : values)              // for each stripes associated to a word
        {
            for (Writable k : map.keySet())         // for each neighbour in the stripe
            {
                IntWritable count = (IntWritable) map.get(k);               // get local occurrence
                IntWritable existing = (IntWritable) aggregateMap.get(k);   // get total occurrence

                if (existing != null)           // this pair of words has been found before
                {
                    int updatedCount = existing.get() + count.get();        // create updated count
                    aggregateMap.put(k, new IntWritable(updatedCount));     // update total occurrence
                }
                else                            // this pair of words has not been found before
                    aggregateMap.put(k, new IntWritable(count.get()));      // set (first time) the total occurrence
            }
        }

        context.write(key, new Text(mapWritableToString(aggregateMap)));    // emit word and aggregate stripe associated
    }
```

Figure 12: Code of the reducer in word Stripes CoOccurrence count.

```java
protected void setup(Context context) throws IOException, InterruptedException {
    wordSeenMap = new HashMap<>();          // set the hash map
}
// map function

protected void map(final LongWritable key,final Text value,final Context context)
        throws IOException, InterruptedException {
    String[] words = value.toString().toLowerCase()
            .replaceAll("[^\\p{L}0-9\\s'-]", " ")           // removes unwanted characters, keeps only letters,
            .replaceAll("(?<=\\s)['-]+|['-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
            .replaceAll("(^|\\s)['-]+", " ")                // removes '-' or ''' at the beginning of the word
            .replaceAll("[-']+(\\s|$)", " ")                // removes '-' or ''' at the end of the word
            .trim()                                         // removes leading and trailing whitespace
            .split("\\s+");                                 // splits the string into an array of words, using

    if (words.length < 2)   // check for record withs size less than the window of N-gram (in this case 2-gram)
        return;

    for (int i = 0; i < words.length - 1; i++)      // iterate over each word obtained from the line
    {
        String w1 = words[i];                           // take the current word
        String w2 = words[i + 1];                       // take the next word
        if (w1.isEmpty() || w2.isEmpty())               // control check for the key
            continue;
        if (!wordSeenMap.containsKey(w1))   // check if the current word has it already been seen or not
            wordSeenMap.put(w1, new MapWritable());     // set the stripes for the new word
        // update the value for the co-occurrence
        MapWritable stripe = wordSeenMap.get(w1);       // get stripe for the current word
        Text neighbor = new Text(w2);                   // neighbour of the current word

        if (stripe.containsKey(neighbor))       // neighbor already seen, value must be updated
        {
            IntWritable countWritable = (IntWritable) stripe.get(neighbor); // get old occurrence
            int count = countWritable.get();
            stripe.put(neighbor, new IntWritable(count + 1));               // update occurrence
        }
        else                                    // new neighbor
            stripe.put(neighbor, new IntWritable(1));
    }
    if (isMemoryThresholdExceeded())    // check the used memory
        flush(context);     // emit and flush memory
}
// to close the data structures used fo in mapping

protected void cleanup(Context context) throws IOException, InterruptedException {
    flush(context);         // emit and flush memory
}
```

Figure 13: Code of the mapper in Word Stripes CoOccurrence count with in-map combining.

3.3.4 Stripes InMap Combining

In the version with in-map combining, the mapper (shown in image 13) doesn't create an associative array for each distinct word in the record but works on the whole input split. Hence, each words in the input split assigned to the mapper, theoretically, has a single stripe. It is initialised at the beginning with the setup function and then cleaned up at the end with the cleanup function. This should significantly reduce the number of values emitted at the cost of more complicated logic and bigger memory usage. Memory usage must also be managed to prevent the associative array from growing too large and exceeding the available memory. When a certain usage threshold (80%) is reached, the data collected so far will be emitted and the associative array will be freed. The reducer will remain the same as in the base case, as will the main, except that there is no option to use the external combiner.
This image 13 shows only the map, setup, and cleanup functions. The variables, memory management logic, and emit are identical to the word count case with in-map combining 4, so they will not be shown here in order to limit the number of pages in the documentation.

```
protected void setup(Context context) throws IOException, InterruptedException {
    Configuration conf = context.getConfiguration();    // retrieve configuration object
    window = conf.getInt("window",2);                   // get the configuration to contain the N of N-Gram (defa
}

// map function

protected void map(final LongWritable key,final Text value,final Context context)
        throws IOException, InterruptedException {
    String[] words = value.toString().toLowerCase()
            .replaceAll("[^\\p{L}0-9\\s'-]", " ")           // removes unwanted characters, keeps only letters, n
            .replaceAll("(?<=\\s)['-]+|[\'-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
            .replaceAll("(^|\\s)['-]+", " ")                // removes '-' or ''' at the beginning of the word
            .replaceAll("[-']+(\\s|$)", " ")                // removes '-' or ''' at the end of the word
            .trim()                                         // removes leading and trailing whitespace
            .split("\\s+");                                 // splits the string into an array of words, using or
    boolean errorWord = false;      // indicates if empty words were found in the current window, you should move

    if (words.length < window)      // check to see if the record has length less than the window
        return;

    for (int i = 0; i < (words.length - (window - 1)); i++)         // iterate over each word obtained from the
    {
        StringBuilder wordsKey = new StringBuilder();   // the builder for the key formed by the union of the var
        errorWord = false;                              // reset var

        for (int j = 0; j < window; j++)        // iteration to retrieve all the word in the current window
        {
            if (words[i + j].isEmpty())     // check if at least one word is empty
            {
                errorWord = true;               // set var
                break;                          // exit to the creation of the key string
            }
            wordsKey.append(words[i + j]);      // add word
            if (j != (window-1))            // check if is not the last iteration
                wordsKey.append(",");  // add ','
        }

        if (errorWord)   // control check for the key
            continue;                   // go to next iteration

        nGram.set(wordsKey.toString());      // create the key = (current word,next word,...,wordn-1)
        context.write(nGram, ONE);      // emit key-value pairs
    }
}
```

Figure 14: Code of the map in case n-gram pairs count.

3.4 N-Gram

Code images related to implementations for N-gram word counting. In this section, we will examine both the “Pairs” and “Stripes” techniques, presented in both basic and in-map combining versions.
An n-gram is the generalized case of the problem addressed in this project, with a window of size N. The previously discussed cases are specific N-grams with window sizes of 1 and 2. The code for this problem is similar to the co-occurrence versions, with the difference that the window management has been generalized: it can now take any positive size provided as input to the program. For the sake of brevity, only the core logic of the mapper will be shown in the figure, since this is the only part that changes and is handled differently compared to the previously discussed versions. The window size provided as input in the main is passed to the mapper through an integer parameter called 'window' within the configuration object.

3.4.1 Pairs base

Basic implementation in which every time an n-window of words is encountered, a key-value pair composed in this way will be emitted: $word_i$,...,$word_{i+n-1}$. No optimisation except for the possibility of using the normal combiner, which is the same as the reducer.
These images show the map function 14.

```
protected void setup(Context context) throws IOException, InterruptedException {
    Configuration conf = context.getConfiguration();    // retrieve configuration object
    window = conf.getInt("window",2);                   // get the configuration to contain the N of N-Gram (default
    wordCountMap = new HashMap<>();                     // set the hash map
}
// map function

protected void map(final LongWritable key,final Text value,final Context context)
        throws IOException, InterruptedException {
    String[] words = value.toString().toLowerCase()
            .replaceAll("[^\\p{L}0-9\\s'-]", " ")          // removes unwanted characters, keeps only letters, numbe
            .replaceAll("(?<=\\s)['-]+|['-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
            .replaceAll("(^|\\s)['-]+", " ")               // removes '-' or ''' at the beginning of the word
            .replaceAll("['-]+(\\s|$)", " ")               // removes '-' or ''' at the end of the word
            .trim()                                        // removes leading and trailing whitespace
            .split("\\s+");                                // splits the string into an array of words, using one or
    boolean errorWord = false;      // indicates if empty words were found in the current window, you should move to

    if (words.length < window)      // check to see if the record has length less than the window
        return;

    for (int i = 0; i < (words.length - (window - 1)); i++)         // iterate over each word obtained from the line
    {
        StringBuilder wordsKey = new StringBuilder();   // the builder for the key formed by the union of the various
        errorWord = false;                              // reset var

        for (int j = 0; j < window; j++)        // iteration to retrieve all the word in the current window
        {
            if (words[i + j].isEmpty())     // check if at least one word is empty
            {
                errorWord = true;               // set var
                break;                          // exit to the creation of the key string
            }
            wordsKey.append(words[i + j]);      // add word
            if (j != (window-1))           // check if is not the last iteration
                wordsKey.append(",");  // add ','
        }

        if (errorWord)   // control check for the key
            continue;                   // go to next iteration

        wordCountMap.put(wordsKey.toString(), wordCountMap.getOrDefault(wordsKey.toString(), 0) + 1); // set or updat
    }

    if (isMemoryThresholdExceeded())    // check the used memory
        flush(context);     // emit and flush memory
}
```

Figure 15: Code of the map in case n-gram pairs with in-map combining.

3.4.2 Pairs InMap Combining

In the version with in-map combining, the mapper (shown in image 15) keeps an associative array that stores the occurrence of each word window found, updating it each time it is encountered. This associative array isn't for a single call of the map function, but for all. It is initialised at the beginning with the setup function and then cleaned up at the end with the cleanup function. Memory usage is monitored as in the previous inmap case.

```
protected void map(final LongWritable key,final Text value,final Context context)
        throws IOException, InterruptedException {
    HashMap<String, MapWritable> wordSeenMap = new HashMap<>(); // hash map to contain the words already seen
    String[] words = value.toString().toLowerCase()
            .replaceAll("[^\\p{L}0-9\\s'-]", " ")          // removes unwanted characters, keeps only letters, number
            .replaceAll("(?<=\\s)['-]+|[\'-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
            .replaceAll("(^|\\s)['-]+", " ")               // removes '-' or ''' at the beginning of the word
            .replaceAll("[-']+(\\s|$)", " ")               // removes '-' or ''' at the end of the word
            .trim()                                        // removes leading and trailing whitespace
            .split("\\s+");                                // splits the string into an array of words, using one or

    boolean errorWord = false;      // indicates if empty words were found in the current window, you should move to t
    if (words.length < window)      // check for record withs size less than the window of N-gram
        return;

    for (int i = 0; i < words.length - (window - 1); i++)      // iterate over each word obtained from the line
    {
        String w1 = words[i];                          // take the current word
        if (w1.isEmpty())                              // control check for the key
            continue;
        StringBuilder wordsKey = new StringBuilder();  // the builder for the key formed by the union of the various
        errorWord = false;                             // reset var

        for (int j = 1; j < window; j++)        // iteration to retrieve all the word in the current window
        {
            if (words[i + j].isEmpty())         // check if at least one word is empty
            {
                errorWord = true;               // set var
                break;                          // exit to the creation of the key string
            }
            wordsKey.append(words[i + j]);      // add word
            if (j != (window-1))                // check if is not the last iteration
                wordsKey.append(",");           // add ','
        }
        if (errorWord)   // control check for the key
            continue;                   // go to next iteration

        if (!wordSeenMap.containsKey(w1))   // check if the current word has it already been seen or not
            wordSeenMap.put(w1, new MapWritable());     // set the stripes for the new word

        // update the value for the n-gram occurrence
        MapWritable stripe = wordSeenMap.get(w1);       // get stripe for the current word
        Text neighbor = new Text(wordsKey.toString());  // neighbours of the current word (word[i+1],...,word[i+w-1])

        if (stripe.containsKey(neighbor))       // neighbor already seen, value must be updated
        {
            IntWritable countWritable = (IntWritable) stripe.get(neighbor); // get old occurrence
            int count = countWritable.get();
            stripe.put(neighbor, new IntWritable(count + 1));              // update occurrence
            }
        }

        // now emit all the word and associated stripes -> format emit: (word,stripes) -> all word have a stripes (h
        for (Map.Entry<String, MapWritable> entry : wordSeenMap.entrySet()) {
            Text word = new Text(entry.getKey());              // get key (word)
            MapWritable stripe = entry.getValue();             // get stripe
            context.write(word, new MapWritable(stripe));      // emit (word, stripe)
        }
```

Figure 16: Code of the map in case n-gram stripes count.

3.4.3 Stripes base

Basic implementation in which for each distinct word in the line, an associative array is created, which will contain the other neighbors that make up the window (from word_{i+1} to word_{i+n-1}) as the key and the number of occurrences for that window, in that record, as the value.
These images show the map function 17. In this case, the combiner must be different from

the reducer because if the reducer logic were used as a combiner, the output types would not match the input types required by the reducer. The reducer requests a pair of (Text, Iterable<MapWritable>) as input and returns pairs (Text, Text).

```
protected void setup(Context context) throws IOException, InterruptedException {
    Configuration conf = context.getConfiguration();    // retrieve configuration object
    window = conf.getInt("window",2);                   // get the configuration to contain the N of N-Gram (def
    wordSeenMap = new HashMap<>();                      // set the hash map
}
// map function

protected void map(final LongWritable key,final Text value,final Context context)
        throws IOException, InterruptedException {
    String[] words = value.toString().toLowerCase()
            .replaceAll("[^\\p{L}0-9\\s'-]", " ")           // removes unwanted characters, keeps only letters,
            .replaceAll("(?<=\\s)['-]+|['-]+(?=\\s)", " ") // removes isolated '-' or ''' between spaces
            .replaceAll("(^|\\s)['-]+", " ")                // removes '-' or ''' at the beginning of the word
            .replaceAll("['-]+(\\s|$)", " ")                // removes '-' or ''' at the end of the word
            .trim()                                         // removes leading and trailing whitespace
            .split("\\s+");                                 // splits the string into an array of words, using o

    boolean errorWord = false;      // indicates if empty words were found in the current window, you should mov
    if (words.length < window)      // check for record withs size less than the window of N-gram
        return;

    for (int i = 0; i < words.length - (window - 1); i++)      // iterate over each word obtained from the line
    {
        String w1 = words[i];                           // take the current word
        if (w1.isEmpty())                               // control check for the key
            continue;
        StringBuilder wordsKey = new StringBuilder();   // the builder for the key formed by the union of the va
        errorWord = false;                              // reset var

        for (int j = 1; j < window; j++)        // iteration to retrieve all the word in the current window
        {
            if (words[i + j].isEmpty())         // check if at least one word is empty
            {
                errorWord = true;               // set var
                break;                          // exit to the creation of the key string
            }
            wordsKey.append(words[i + j]);      // add word
            if (j != (window-1))                // check if is not the last iteration
                wordsKey.append(",");           // add ','
        }
        if (errorWord)   // control check for the key
            continue;                   // go to next iteration

        if (!wordSeenMap.containsKey(w1))   // check if the current word has it already been seen or not
            wordSeenMap.put(w1, new MapWritable());     // set the stripes for the new word

        // update the value for the n-gram occurrence
        MapWritable stripe = wordSeenMap.get(w1);       // get stripe for the current word
        Text neighbor = new Text(wordsKey.toString());  // neighbours of the current word (word[i+1],...,word[i+

        if (stripe.containsKey(neighbor))       // neighbor already seen, value must be updated
        {
            IntWritable countWritable = (IntWritable) stripe.get(neighbor); // get old occurrence
            int count = countWritable.get();
            stripe.put(neighbor, new IntWritable(count + 1));              // update occurrence
        }
        else                                    // new neighbor
            stripe.put(neighbor, new IntWritable(1));
    }
    if (isMemoryThresholdExceeded())    // check the used memory
        flush(context);     // emit and flush memory
}

// to close the data structures used fo in mapping

protected void cleanup(Context context) throws IOException, InterruptedException {
    flush(context);         // emit and flush memory
}
```

Figure 17: Code of the map in case n-gram stripes with in-map combining.

3.4.4 Stripes InMap Combining

In the version with in-map combining, the mapper (shown in image 13) keeps an associative array that stores the occurrence of each word window found, updating it each time it is encountered. This associative array isn't for a single call of the map function, but for all. It is initialised at the beginning with the setup function and then cleaned up at the end with the cleanup function. Memory usage is monitored as in the previous inmap case.

4 Result

4.1 General

Tests were carried out to assess the performance of each methodology and configuration discussed in this documentation. More specifically, for each configuration, ten jobs were run with a single reducer and ten jobs with two reducers. All command line output data for each job was collected and is available in the result folder on GitHub. The jobs were run ten times to obtain statistically better values that were not subject to the uncertainty of a single sampling. The code provided can run any number of identical jobs. For these tests, I chose ten as a good compromise between test execution time and a test number high enough to provide valid data. I observed that the data between one job and another did not change much, and since a job in the configuration used for the project lasted more than an hour, I chose 10 (which, although low, could provide good results by keeping the execution time for each test below 12 hours).

For the comparison, not all the various statistics provided will be considered, but only a selection consisting of the following fields:

- Bytes Read: Total number of bytes read by the job from input files via the configured input format;
- Map output bytes: Size in bytes of all mapper output;
- Spilled Records: Number of records temporarily written to disk due to memory constraints (in map phase);
- Combine input records: Records read by a combiner (if used);
- Combine output records: Records output by a combiner;
- Reduce input records: Total records sent to the reducer (across all keys);
- Bytes Written: Total number of bytes written by Reduce to the file system (e.g. HDFS) as the final output of the job
- Peak Map Physical memory: Maximum physical memory used by any single map task;
- Peak Reduce Physical memory: Maximum physical memory used by any reduce task.

For these fields, the averages calculated are based on all 10 jobs for each test and will be displayed.
It is essential to consider these results in light of the testing environment. As a solo project, I was unable to utilize Hadoop to its full potential in the distributed cluster version, but I worked on a single VM in pseudo-distributed mode.
The VM used was also single-core, so I was unable to take advantage of parallelism, which is key

to performance in these architectures.

Below, graphs relating to the various data obtained from the tests will be presented and discussed, with each graph highlighting a distinct aspect.
In the following graphs, if cases of 1 reducer and 2 reducers are not specified, it is because the data do not change between the two cases.

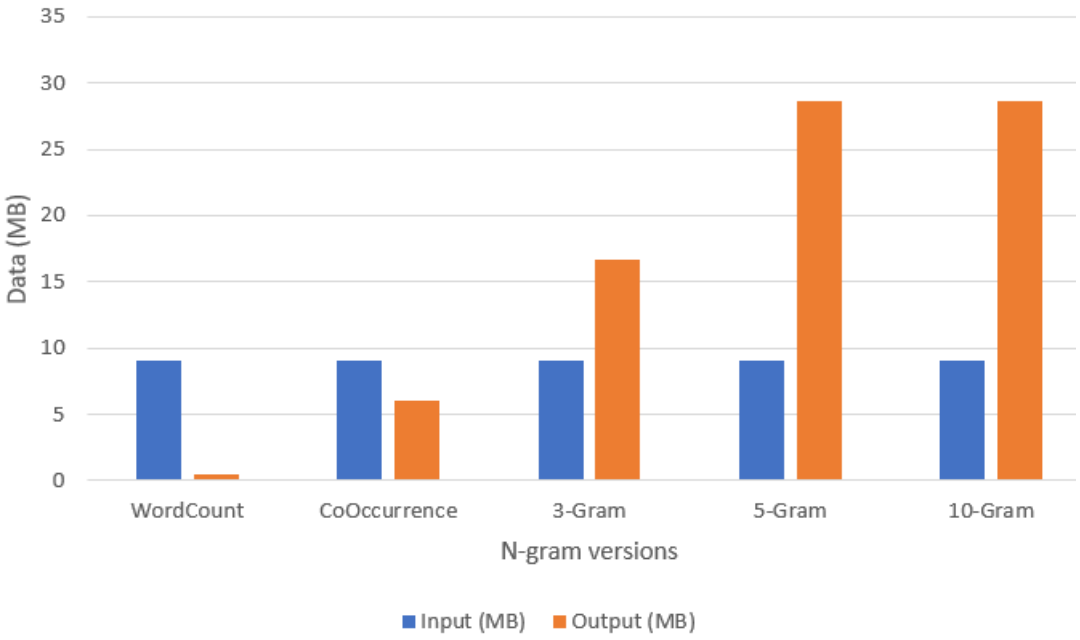

Figure 18: Image showing input and output data based on the size of the window being examined.

Image 18 shows the input and output data (in MB) generated by the jobs for each problem (window size) tested. It can be seen that the output data grows rapidly, exceeding the input data (the text corpus) by 3-grams. It can also be seen that as the window size increases, the output continues to increase but at a slower rate. This is an interesting fact that may seem counterintuitive.
Theoretically, given a vocabulary "v" (representing the number of distinct terms in the corpus) and given a window "n", the number of all possible different combinations is given by v^n. Therefore, as n increases in the n-gram, the theoretical combinations grow exponentially. For this reason, one might expect the output to grow much more than what is seen as the window increases.
In practice, output growth tends to flatten out as the window increases for the following reasons:

- Sentence length: given an n-gram and a sentence of "L" words, you get (L − n + 1) sequences. Therefore, the larger n is, the fewer sequences will be obtained from the sentence. If the sentences are short, this leads to a drastic reduction in the number of large n-grams that can be generated (in our dataset, sentences are generally not very long). So, despite the combinatorial growth of possible different n-grams, the output per document shrinks.
- Frequency distribution: 2-grams and 3-grams are very frequent and diverse, so the output file has many different sequences and occurrences. With 5-grams or 10-grams, there are

many more theoretical combinations, but those found in practice are few (perhaps even recurring), and this leads to a stabilisation in the growth of the generated output.

So, to recap :

- for small windows (2-grams/3-grams), there are fewer possible combinations, but the texts can explore them much more.
- large windows (5-gram / 10-gram), the theoretical space is enormous, but the actual texts are too short and too variable to cover it. This, together with linguistic redundancy, leads to fewer observable real combinations, which limits the growth of the output file. So the "explosive" growth remains only theoretical; in practice, it flattens out.

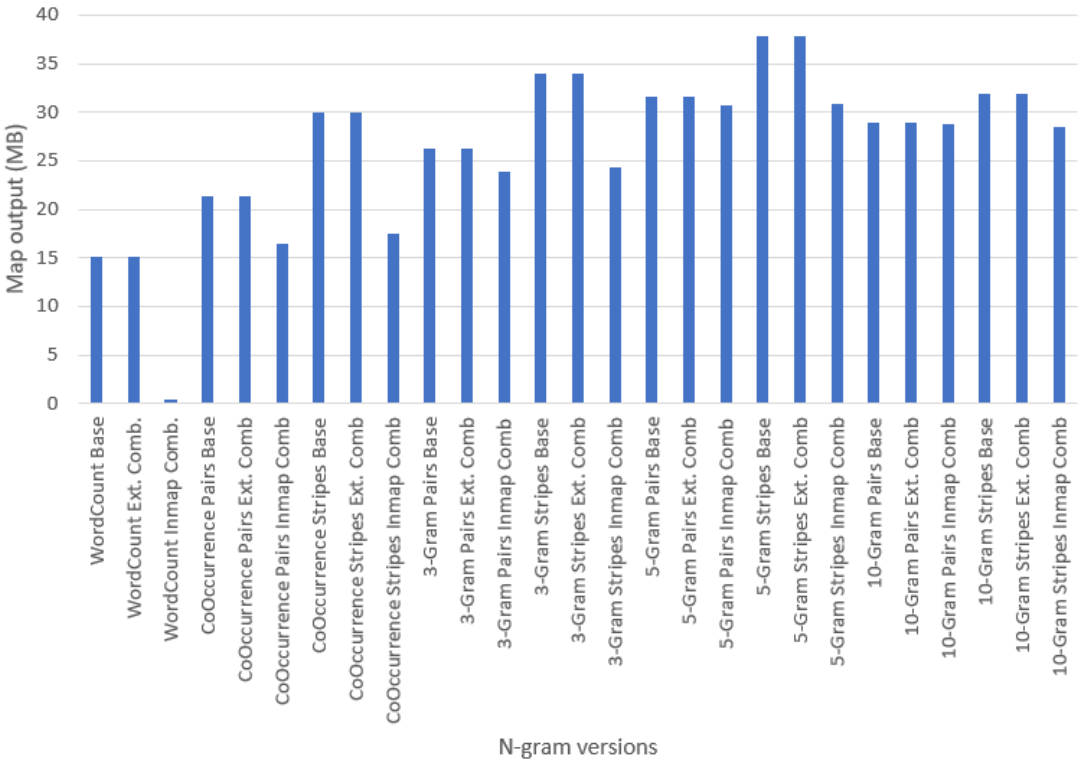

Figure 19: Image showing the map output bytes based on the version.

This image 19 shows the Map Output Bytes metric (in MB) generated by the jobs for each problem (window size) tested. Several key observations can be drawn:

- Growth with window size: the overall size of the intermediate data increases as the window size grows. This is expected, since larger windows generate a greater number of co-occurrences, and consequently, more intermediate records are emitted by the mappers.
- Effect of in-mapper combining: the in-mapper combining technique consistently produces smaller intermediate outputs because it aggregates values locally within the map() function before emitting them. The reduction is particularly evident in simple tasks such as WordCount, while in higher-order n-grams the benefit is still present but less pronounced due to the sheer number of unique combinations that need to be retained.

- External combiner impact: the external combiner does not influence the reported map output bytes. This is because the combiner is applied only after the mapper has already produced its output, during the spill/sort and shuffle phases, just before the data is sent to reducers. Consequently, it can reduce the shuffle size and the volume of data delivered to the reducers, but it does not affect the raw size of the intermediate output emitted by the mappers.
- Stripes vs. Pairs approaches: the results clearly indicate that Stripes approaches emit more data compared to Pairs. This is because while Pairs emit a simple (word, co-occurring word) key–value pair, Stripes emit a structure of the form (word, associative array of co-occurrences). The associative array introduces additional serialization overhead and therefore results in larger map output sizes. Even with in-mapper combining, Stripes remain heavier than Pairs.
- Saturation effect with higher-order n-grams: the growth in intermediate data size tends to plateau when moving from 5-grams to 10-grams. While there is still an increase, it is marginal compared to the jump observed between lower-order n-grams (e.g., WordCount to 3-grams). This saturation effect is likely due to the characteristics of the input dataset: the available text does not provide sufficient long-range contexts to generate substantially more unique co-occurrences when the window size increases beyond a certain point.

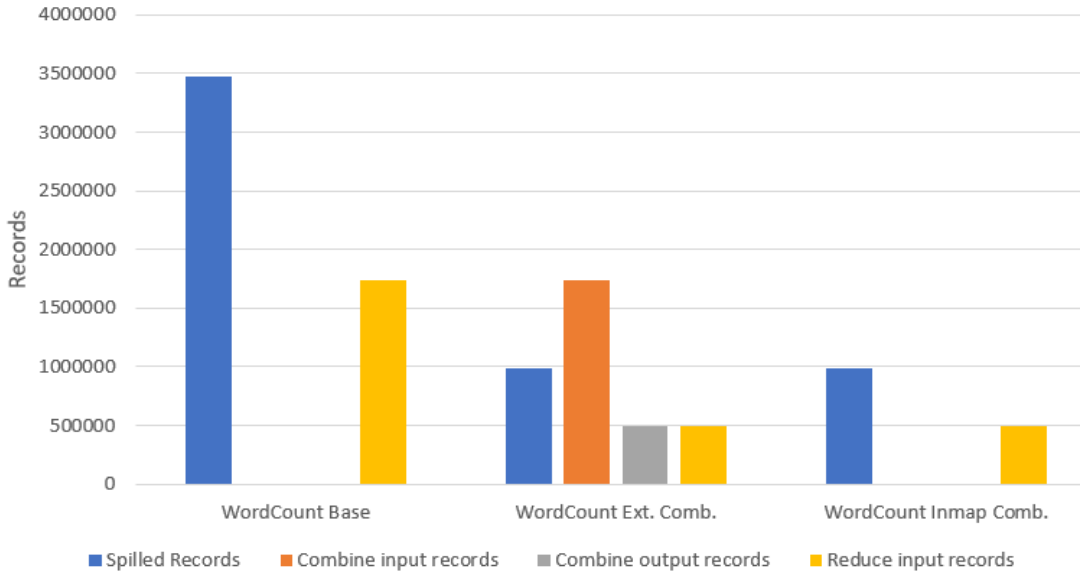

Figure 20: Image showing the spilled records, combine input and output records, and reduce input records for word count.

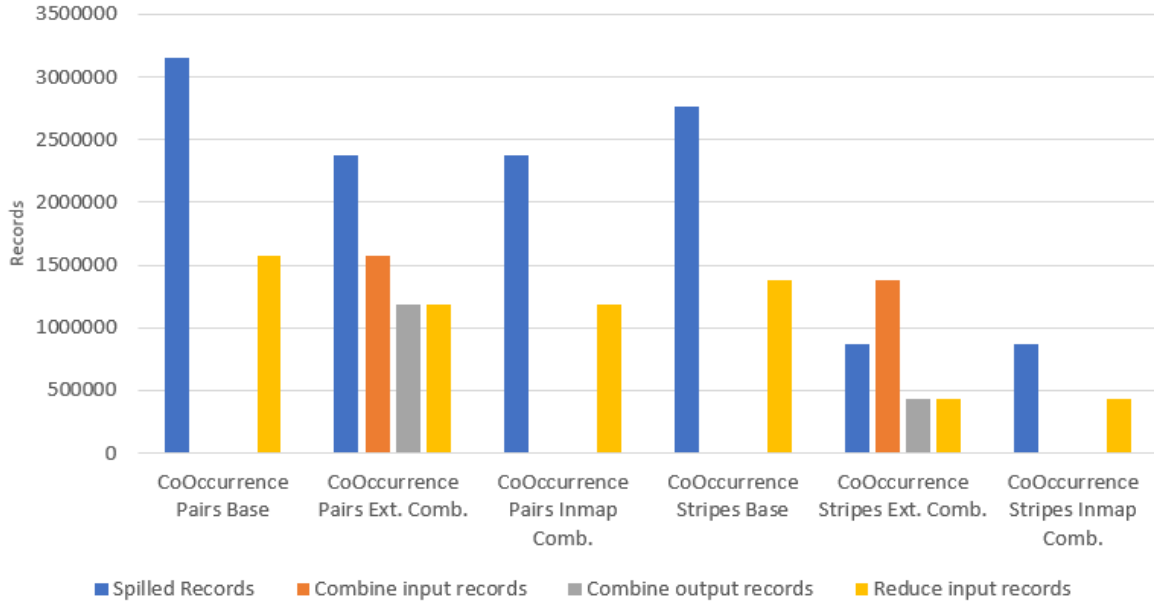

Figure 21: Image showing the spilled records, combine input and output records, and reduce input records for co-occurrence count.

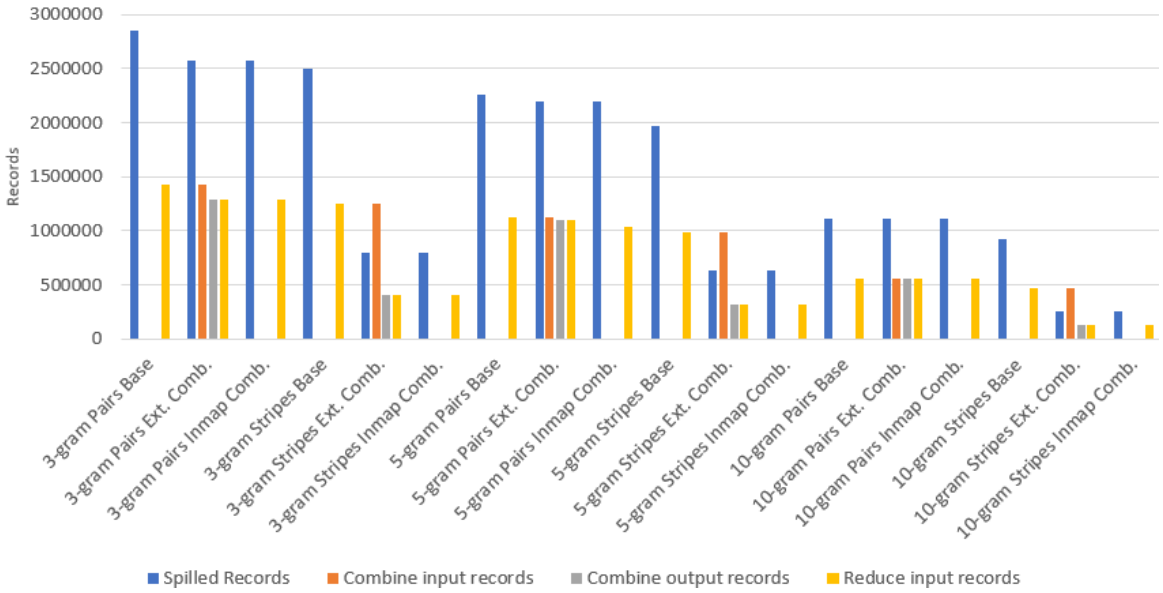

Figure 22: Image showing the spilled records, combine input and output records, and reduce input records for n-gram count.

The images 20 21 22 show that in the baseline and in-mapper combining cases, the combiner records are set to zero, in the former because no combiner is applied, and in the latter because the aggregation is directly handled inside the mapper code.

Across all examined cases, a consistent trend emerges: the baseline implementations emit the largest number of records from the map tasks, resulting in a higher number of spilled records (i.e., intermediate outputs written to disk) as well as more input records for the reducers. Both the external combiner and the in-mapper combining strategies achieve local aggregation of mapper outputs, thereby reducing the number of spilled records and the volume of data passed to the reducers. This effect is particularly visible in the Stripes versions, which consistently emit fewer records than the Pairs versions, as each stripe aggregates multiple co-occurring words under a single key, leading to a reduction of roughly one third in both spilled and reducer input records. However, this comes at the cost of heavier records, as already observed in the map output bytes, where Stripes approaches generate larger outputs despite producing fewer records. Another important observation is the general decrease in the number of spilled and reduced input records as the window size increases from 3-grams to 10-grams. This reflects the saturation effect already discussed in the map output analysis: although larger windows generate more co-occurrences in principle, the dataset does not provide enough long-range context to sustain linear growth, and therefore the number of unique intermediate records declines. Taken together, these results show that combining strategies are effective in reducing the intermediate data both written to disk and transferred over the network, with in-mapper combining achieving comparable efficiency to the external combiner while avoiding the overhead of the Hadoop combiner mechanism, and Stripes proving more efficient than Pairs in terms of record reduction, though at the expense of higher serialization costs per record.

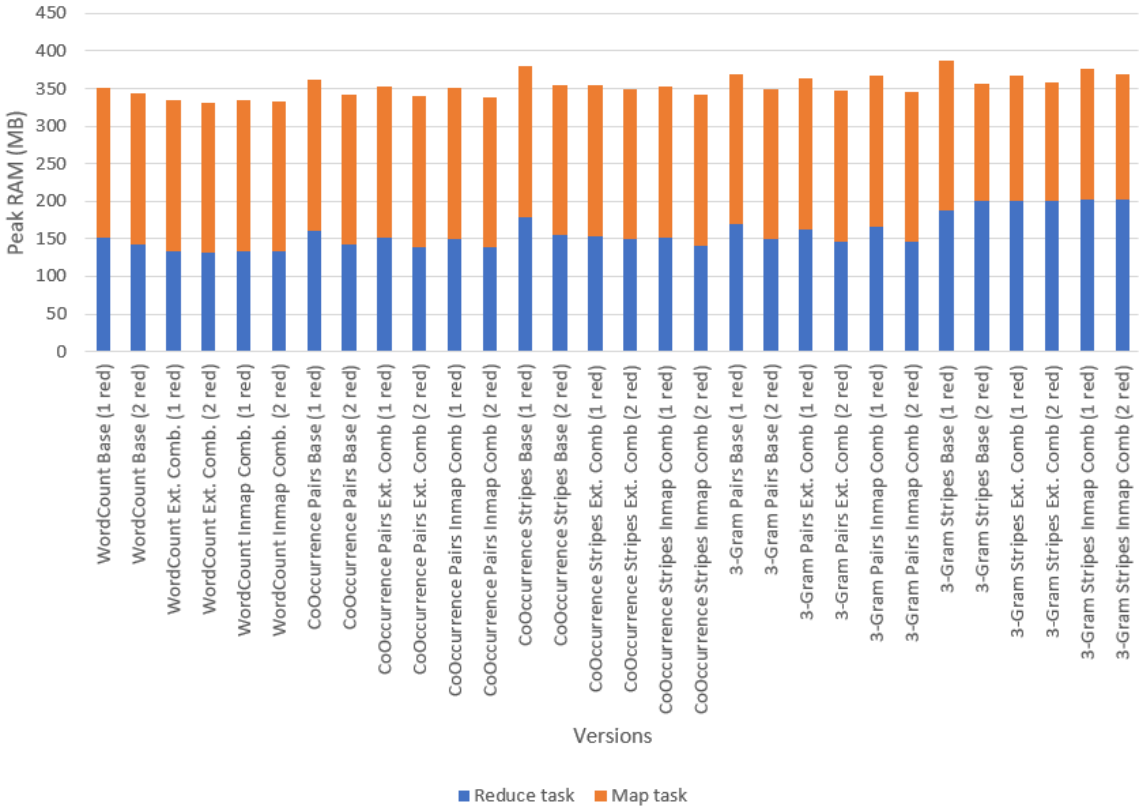

Figure 23: Image showing the peak RAM usage by the reduce and map tasks in wordcount, occurrence, and 3-gram versions.

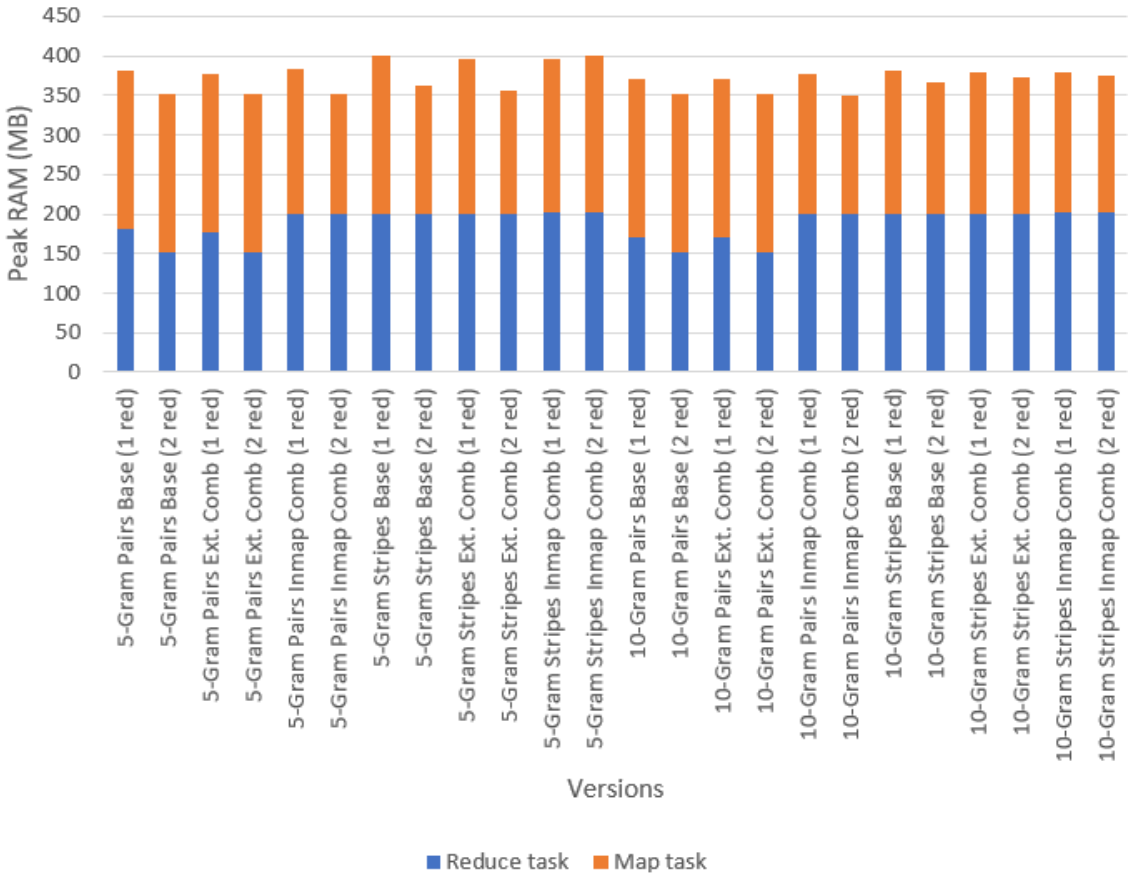

Figure 24: Image showing the peak RAM usage by the reduce and map tasks in 5-gram and 10-gram versions.

The images 23 24 show the averages of the values obtained from the command line. Be aware that the memory values shown by Hadoop in this case do not necessarily indicate the memory actually used by the tasks, but indicate the reserved memory, i.e., the memory that the container's JVM has allocated.
When a Map or Reduce task starts, a JVM process is launched with a heap configuration that depends on:

- Xms: memory initially allocated to the JVM heap.
- Xmx: maximum memory that the JVM can allocate to the heap.

These values also depend on the maximum available memory that has been configured for each task. The initially allocated value will usually be a certain percentage of the maximum value. You need to be careful at this point if you want to see the memory actually consumed because there may be cases where the memory initially allocated is more than what is needed for the task. This leads to a value that is always the same, which does not reflect the actual usage of the tasks but only the amount initially allocated.
To avoid this problem, I configured the initially allocated memory to be very small (5MB). This way, it is almost certain that the task will need more RAM, causing the request and allocation of

additional RAM, which leads to a value that is more accurate to the RAM actually used. This should provide a much more accurate estimate at the cost of a slight increase in time (due to memory reallocation).
These parameters can be set in the 'mapred-site.xml' configuration file by modifying the values associated with these fields:

- mapreduce.map.memory.mb: RAM allocated to each Map task.
- mapreduce.reduce.memory.mb: RAM allocated to each Reduce task.
- mapreduce.map.java.opts: JVM options for Map tasks (e.g., -Xms, -Xmx).
- mapreduce.reduce.java.opts: JVM options for Reduce tasks.

The image 23 24 shows the memory usage of various MapReduce implementations for different tasks (WordCount, CoOccurrence, 3-Gram, 5-Gram, and 10-Gram) using combinations of reducers and combining strategies.
Observable trends:

- Impact of version on memory usage: Across all tasks, the memory consumption for the "base" implementation is generally higher compared to implementations that use in-mapper combining or external combining. This is because in-mapper combining reduces the amount of intermediate data that needs to be held in memory before being sent to the reducers, thereby lowering peak memory usage.
- Effect of the number of reducers: Configurations with 2 reducers show lower peak memory usage compared to configurations with only 1 reducer. This occurs because the workload is partitioned across multiple reducers, reducing the amount of data that each reducer needs to process at any one time (the memory pressure is distributed).
- Impact of window size on memory usage: there is a slight upward trend in memory usage as the n-gram window increases. This makes sense because larger windows generate more intermediate data, which in turn requires more memory to store and process before the shuffle phase.

The lowest memory usage is observed for the in-mapper combining strategy with 2 reducers. These configurations are effective because they minimize the intermediate data held in memory (via in-mapper combining) and distribute the workload across multiple reducers, reducing the peak memory requirements per reducer.

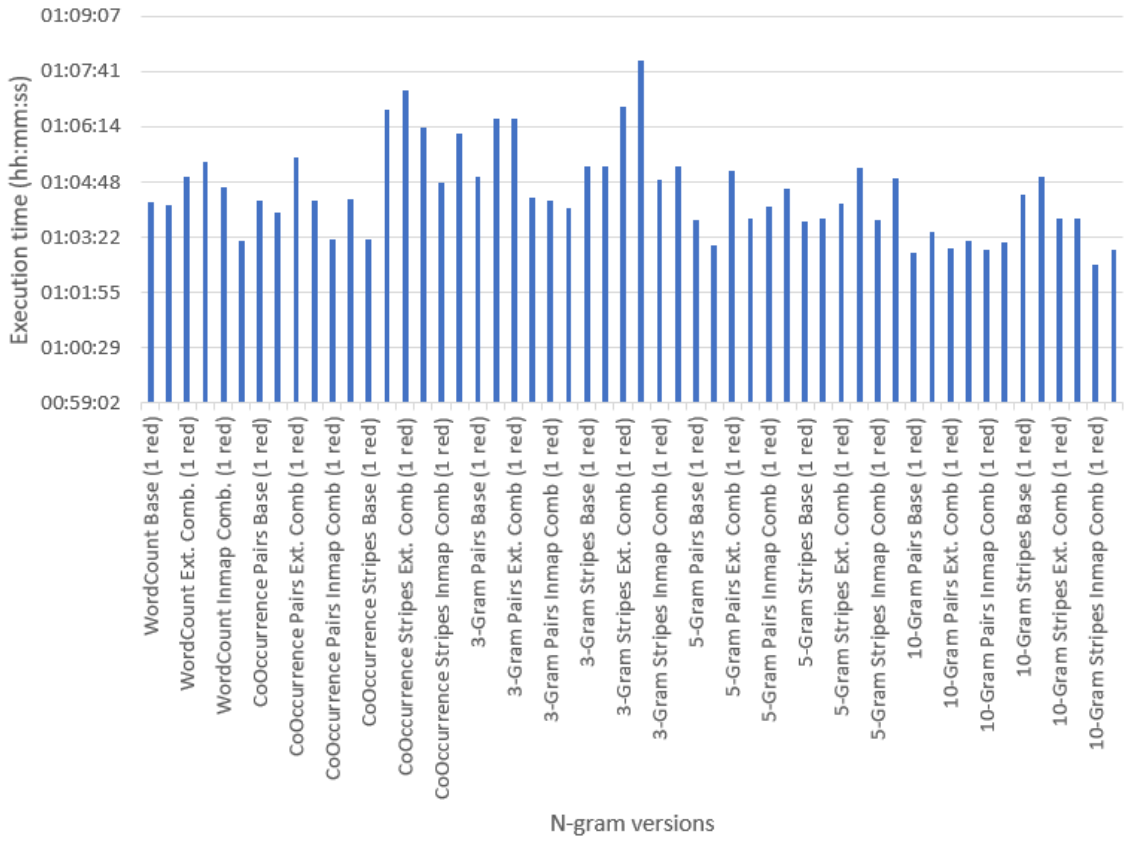

Figure 25: Image showing the average execution time for each version.

The image 25 shows the average execution times for each version seen in this project. It is necessary to remember that these times were obtained with Hadoop in pseudo-distributed mode running on a VM with a single CPU. Therefore, these times refer to jobs that could not take advantage of the parallelization offered by the map and reduce paradigm. In general, all times are very similar to each other, and there are no marked differences between the different code versions.

4.2 Summary

Overall, the results confirm the theoretical expectations of MapReduce behavior. In-mapper combining emerges as the most effective technique for reducing the volume of intermediate data, while the external combiner only impacts shuffle traffic without influencing the raw mapper output. The Stripes approach consistently incurs higher serialization overhead than Pairs, even though it reduces the number of records. Furthermore, the experiments highlight a saturation effect in higher-order n-grams, where the number of occurrences and the corresponding intermediate and output data cease to grow significantly. Finally, execution times remain largely stable across all configurations, which can be attributed to the use of pseudodistributed mode, where the benefits of reduced shuffle traffic do not translate into measurable runtime improvements.