MSc Computer Engineering

Cloud Computing

Letter Frequency

Lorenzo Bataloni

Kevin Boni

Edoardo Pantè

Academic Year 2023/24

1. Introduction

The purpose of this project was to exploit the functionality of the Hadoop Framework in order to obtain a MapReduce program that receives as input files of different sizes and languages and outputs how frequent each letter has occurred on each file.

The dataset was a .txt file and the languages chosen were Italian, English and French. The sizes for each language were: 50MB, 500MB, 1GB and 3GB.

2. Hadoop

To run the implementations with the different files and several reducers, a bash script was used.

```
for file in ${files[@]};
do
for ver in ${versions[@]};
   do
   for num in ${reducers[@]};
       do
       echo ---------------------------------------------------------------------
       echo starting execution version: ${ver}, file: ${file}, num reducer: ${num}
       echo ---------------------------------------------------------------------
       hadoop jar ./lettercount-${ver}/target/lettercount-${ver}-1.0-SNAPSHOT.jar
it.unipi.hadoop.LetterCount ${file} results/${ver}/${file}/${num} ${num};
       echo waiting for killing container....
       sleep 300; # Wait 5 min. to let all containers finish (60sec x 5)
       done;
   done;
done
```

2.1 Cluster Structure

Hadoop is designed to run on distributed systems. Therefore, a cluster with three different virtual machines was created. We have a master node, called **namenode** that coordinates the execution of jobs and file management on HDFS. We then have two nodes, called **datanodes** on which the jobs are executed and on which the replication of the sectors that make up the files is done. In the table below you can see some specifications of the cluster.

IP	Role	Hostname
10.1.1.82	Namenode	hadoop-namenode
10.1.1.83	Datanode	hadoop-datanode-2
10.1.1.143	Datanode	hadoop-datanode-3

3. Implementation

Three different implementations have been developed: Naive, Opt and Opt-combiner.

The Naive implementation presents a classical approach and has two MapReduce pipelines. The first pipeline was used to count the number of letters and the second one to compute the frequency of the occurrence of each letter. Both pipelines use a combiner to pre-aggregate the mapper results.

The Opt-Combiner implementation uses a single MapReduce pipeline with the use of a combiner in between for initial aggregation of results.

The Opt implementation represents the most optimized version that has been developed. This uses a single MapReduce pipeline without using combiners. In fact, an initial pre-aggregation is done directly in the mapper, which in addition to calculating the total also calculates the number of occurrences for each letter that is encountered.

3.1 Naive

In the Naive implementation we used the standard MapReduce techniques offered by Hadoop. For this we have two sequential pipelines. The first, called the Count Pipeline, deals with going to extract the total number of letters from the document. When this pipeline is finished, the value just computed is sent to the second pipeline. At this point, the second pipeline, called Frequency Pipeline, is run, in which all occurrences for each letter are counted and the report is then run to extract its frequency within the analyzed document.

3.1.1 Count Mapper

```
// Naive - Count Mapper
counter = 0
for each letter in lowerCase(document){
  if letter in ["a" ... "z"]{
    counter += 1
  }
}
write("TOTAL" , counter)
```

3.1.2 Count Combiner

```
// Naive - Count Combiner
counter = 0
for each mapper{
  counter += mapper.counter
}
write("TOTAL" , counter)
```

3.1.3 Count Reducer

```
// Naive - Count Reducer
total = 0
for each combiner{
  total += combiner.counter
}
write("TOTAL" , total)
```

3.1.4 Frequency Mapper

```
// Naive - Frequency Mapper
occurrency = {}
for each letter in lowerCase(document){
  if letter in ["a" ... "z"]{
    if letter not in occurrency{
      occurrency[letter] = 1
    }else{
      occurrency[letter] =+ 1
    }
  }
}
for each key in occurrency{
  write(key , occurrency[key])
}
```

3.1.5 Frequency Combiner

```
// Naive - Frequency Combiner
for each key in keys{
  sum = 0
  for each mapper {
    sum += mapper.occurrency[key]
  }
  write(key , sum)
}
```

3.1.6 Frequency Reducer

```
// Naive - Frequency Reducer
total_letters = read("TOTAL")
for each key in keys{
  sum = 0
  for each value in combiner[key].values{
    sum += value
  }
  freq = sum / total_letters
  write(key , freq)
}
```

3.2 Opt-Combiner

In the Opt-Combiner implementation, we decided to use a single pipeline to avoid the overhead due to all the network-level communications and cluster memory management. In this implementation, the mapper is responsible for both counting the total number of letters and emitting each character that is encountered in the format <character 1>. At this point the Combiner takes care of aggregating the data by going to sum the values for each character. Once the combiner execution is finished, the reducer has all the data to calculate the frequency for each character.

3.2.1 Mapper

```
// Opt-Combiner - Frequency Mapper
counter = 0
for each letter in lowerCase(document){
  if letter in ["a" ... "z"]{
    counter += 1
  }
  write(letter , 1)
}
write(TOTAL_LETTERS , counter)
```

3.2.2 Combiner

```
// Opt-Combiner - Frequency Combiner
for each mapper{
  sum = 0
  for each key in mapper.keys{
    sum += read(mapper[key])
  }
  write(key , sum)
}
```

3.2.3 Reducer

```
// Opt-Combiner - Frequency Reducer
total_letters = read(TOTAL_LETTERS)
for each combiner{
  sum = 0
  for each key in combiner.keys{
    sum += read(mapper[key])
  }
  freq = sum / total_letters
  write(key , freq)
}
```

3.3 Opt

In the Opt implementation, the combiner is eliminated to go to exploit the in-combiner property. This property allows you to not use the combiner and go for aggregation directly in the mapper. In cases where the logic is not too complex, this technique avoids the communication overhead due to the presence of combiners. In fact, in this case the mapper, in addition to calculating the total number of characters, will go and create a HashMap in the format <character, n> where n is the number of occurrences of the specific character in the input fragment analyzed by the mapper. At this point the reducer, once the total is obtained by summing the results obtained by the mapper can easily calculate the frequency.

3.3.1 Mapper

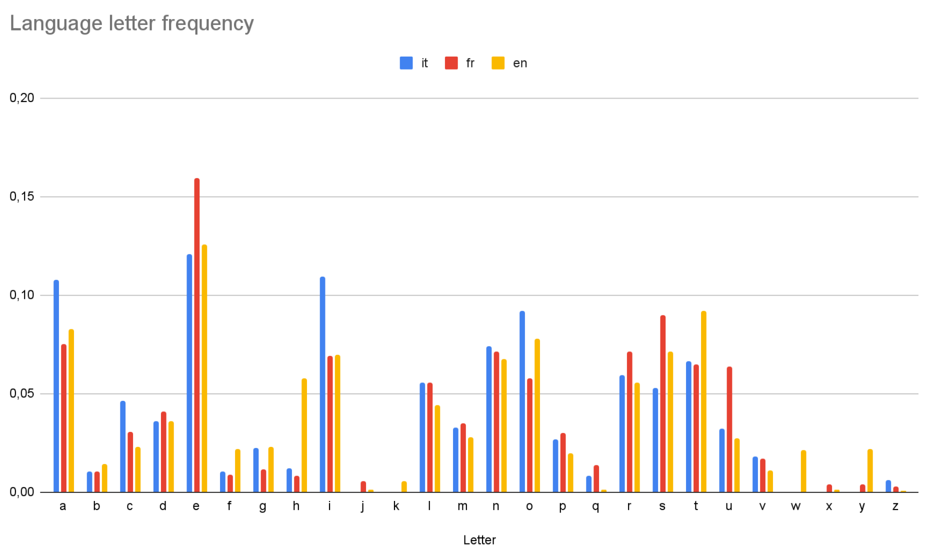
```
// Opt - Frequency Mapper
occurrency = {}
counter = 0
for each letter in lowerCase(document){
  if letter in ["a" ... "z"]{
    if letter not in occurrency{
      occurrency[letter] = 1
    }else{
      occurrency[letter] =+ 1
    }
    counter += 1
  }
}
write(TOTAL_LETTERS , counter)
for each key in occurrency{
  write(key , occurrency[key])
}
```

3.3.2 Reducer

```
// Opt - Frequency Reducer
total_letters = read(TOTAL_LETTERS)
for each mapper{
  sum = 0
  for each key in mapper.keys{
    sum += read(mapper[key])
  }
  freq = sum / total_letters
  write(key , freq)
}
```

4. Experimental Results

4.1 Frequency comparison

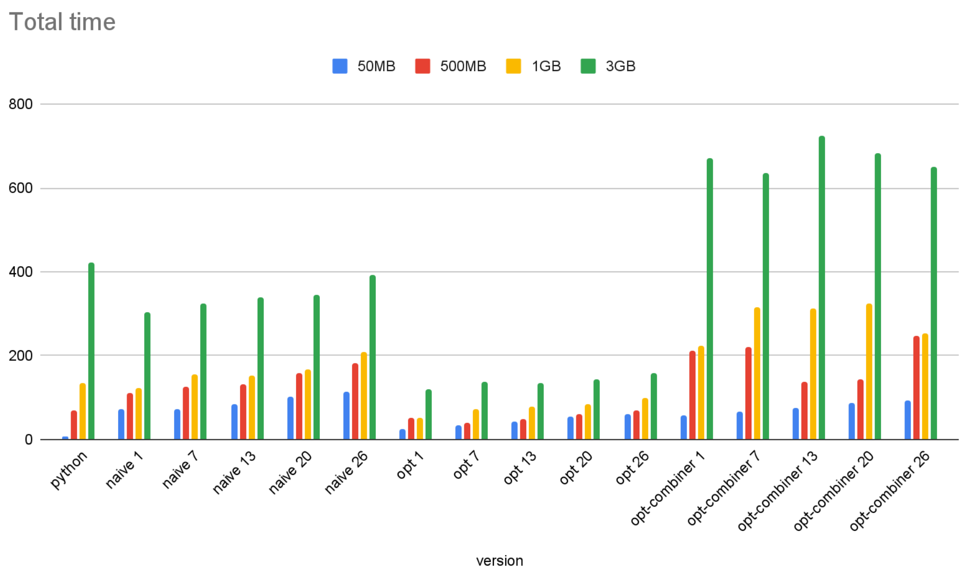


4.2 Execution time

In the following section we discuss the execution times of the various implementations

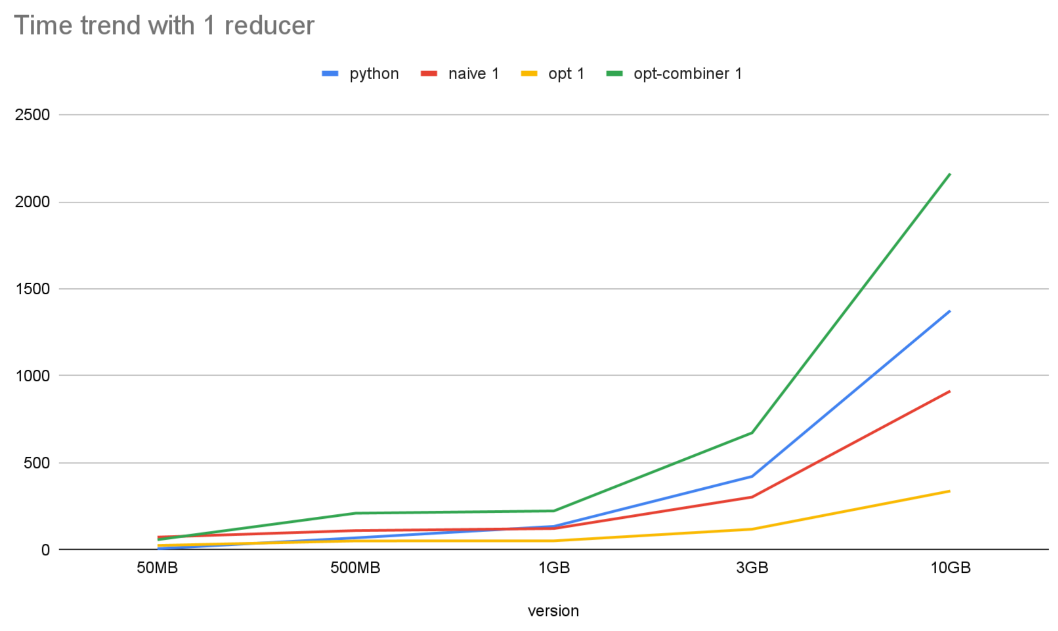
4.2.1 Total

The following chart shows the total execution times of each implementation as file sizes change. As we obviously can see, independently of the versions used, the larger the file the longer the execution time.



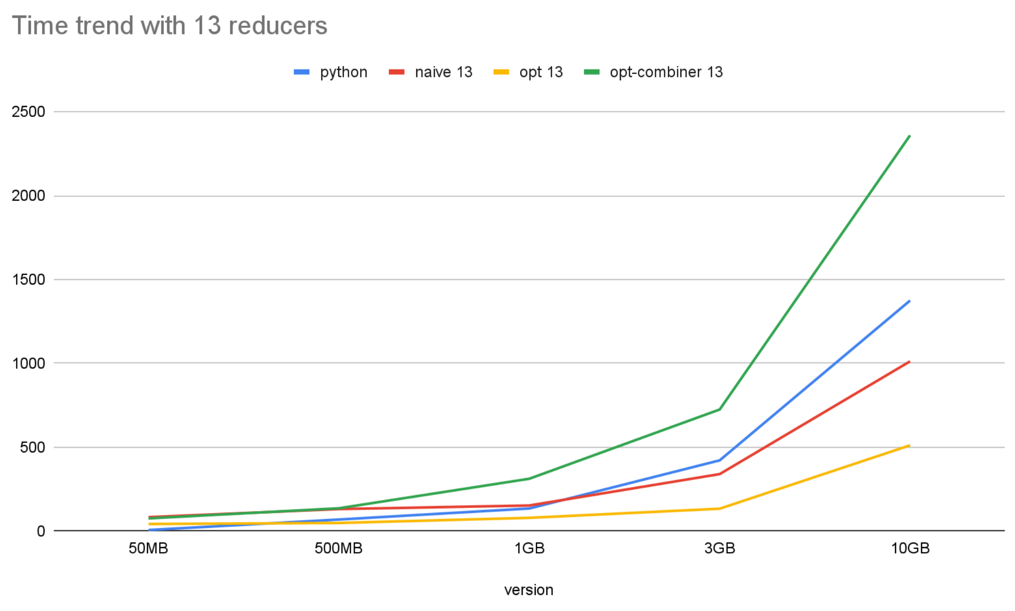
4.2.2 1 Reducer

The following chart shows the execution time trend of each version with 1 reducer. As we can see with this configuration for files smaller than 1GB we have almost the same execution times. As the file size increases the implementation that shows the best performance is the opt one.



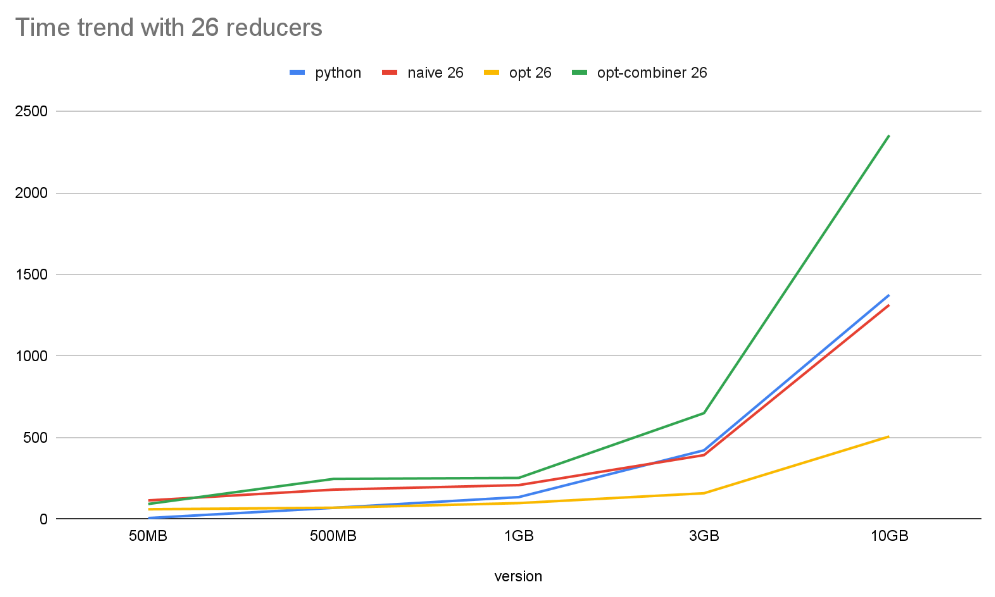
4.2.3 13 Reducers

The following chart shows the execution time trend of each version with 13 reducers. As we can see the time gap between the versions is almost the same.



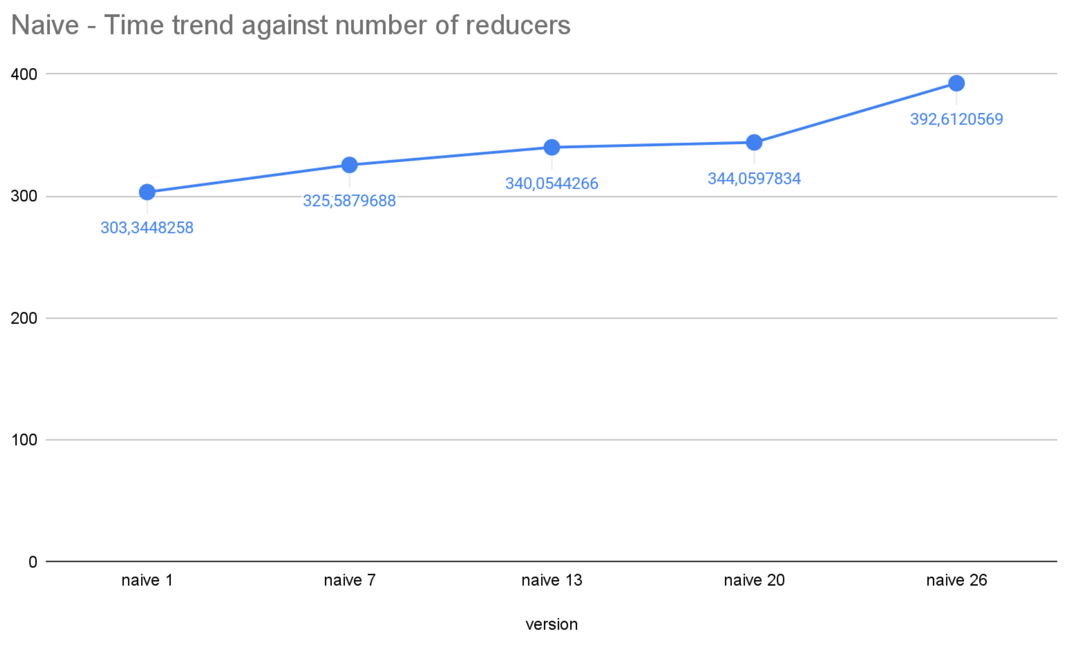
4.2.4 26 Reducers

The following chart shows the execution time trend of each version with 26 reducers. With this configuration we can see that still the most performing version is the opt one but the time gap between the naive and the Python implementation for larger files narrowed.

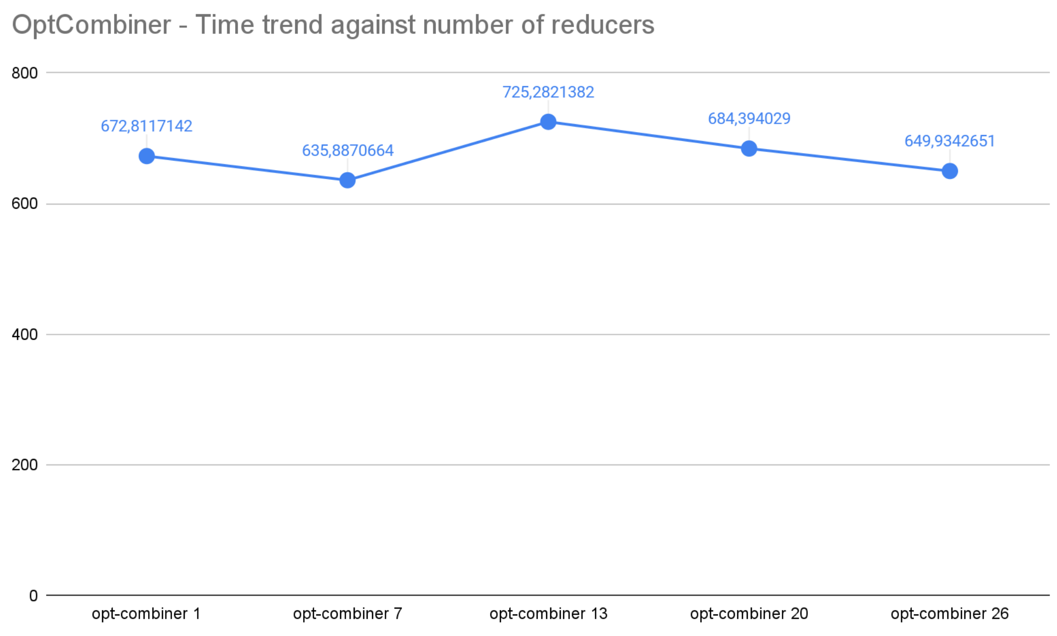


4.3 Time against reducer number

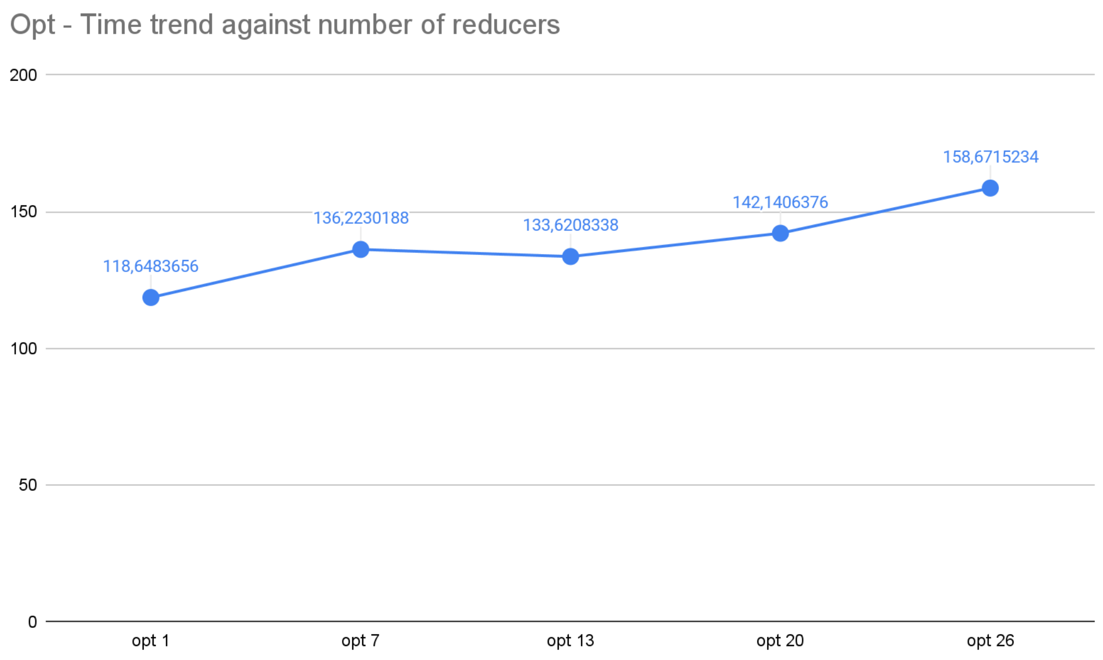
4.3.1 Naive time trend



4.3.2 OptCombiner time trend

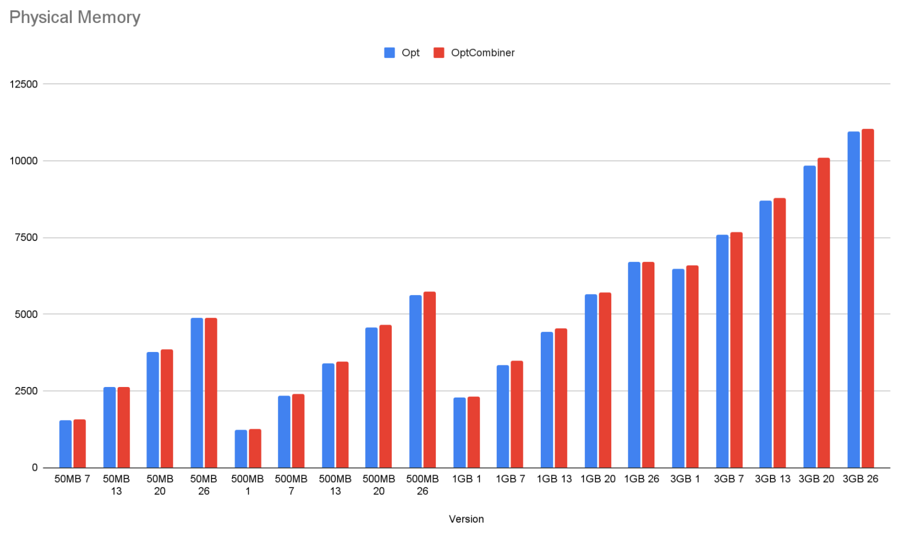


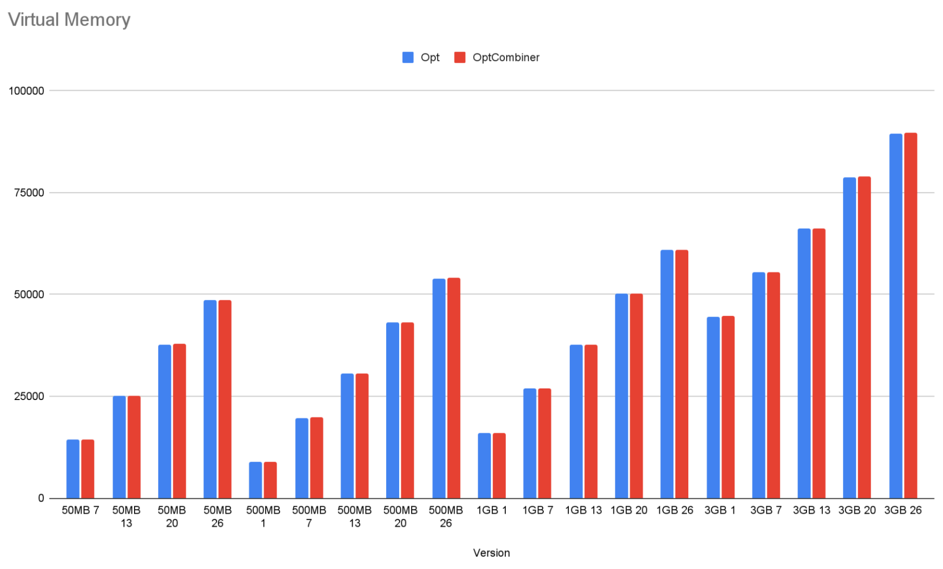
4.3.3 Opt time trend

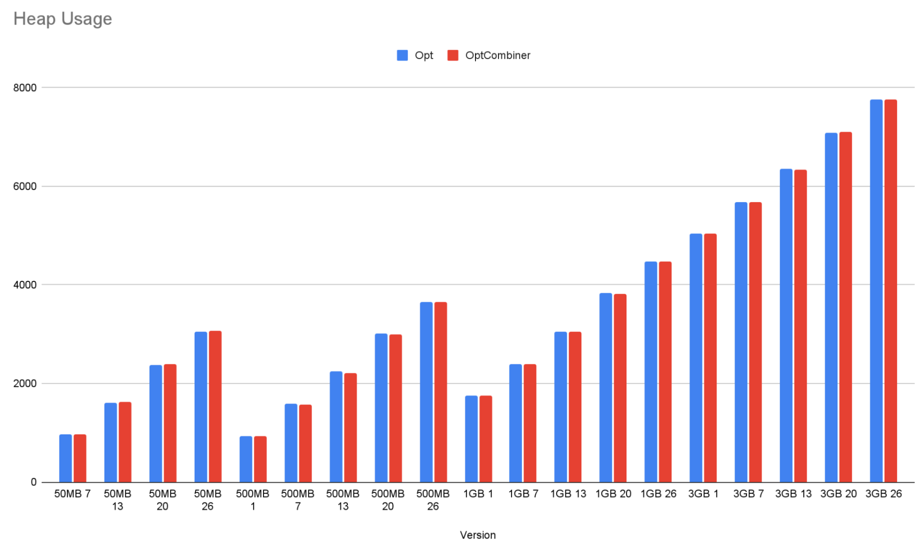


4.4 Memory usage

In addition to the time required to execute the job, another extremely important factor is memory utilization. Given the nature of Hadoop, it is important to consider both physical and virtual memory. In addition, we analyzed the memory used for the heap, Hadoop and our jobs being implemented in Java.
In the following graphs you can see the difference between the opt and optcombiner implementations for physical memory, virtual memory, and heap.







As you can see in the graphs, the memory occupied is exactly the same. Considering that the jobs have a limited memory range, it is easy to imagine that both versions used all the memory available to them. What is important to note is that this emphasizes when the opt version and the in-mapper technique is actually very important at the efficiency level. In fact, for the same amount of resources used, the opt takes an important amount of time less than the opt-combiner. This makes the opt version the ideal version in that it is possible to achieve the same results using the same resources but taking much less time.