Université Libre de Bruxelles

INFO-H417 : Database System Architecture

Project

Algorithms in Secondary Memory

Authors:
Soufiane AJOUAOU 000459811
Yahya BAKKALI 000445166
Maxime HAUWAERT 000461714

October 26, 2023

Contents

1 Introduction and Environment

1.1 Introduction

The goal of this project is to compare the performance of different methods for reading and writing to secondary memory by using them in different algorithms.

Four methods will be implemented, one that uses one character at a time, one using a buffering mechanism, one that uses a buffer of a predefined size and one using memory mapping.

Four algorithms will be implemented, one that reads the file and computes the sum of the lengths of each line of a file, one that reads randomly the file and computes the lengths of the next lines, one that merges different files and one that sorts a file.

In this report, the implementations of the different methods will be seen then all the results will be shown and discussed.

1.2 Environment

Here are the specifications of the computer used for the benchmark :

- CPU : Intel Core i7-8750H
- RAM : 8GB DDR4, 2400Mhz
- Memory : 480 GB SSD with 540 MB/s read speed and 555 MB/s write speed
- Operating system : Ubuntu 20

The programming language used for this project was Java 13.

The following external libraries have been used :

- JMH : It has been used to easily make benchmarks.
- Junit : It has been used to test the different implementations.

1.3 Definitions

For the different implementations, a notation will be used and it is as follows :

- One refers to the first implementation.
- Buffered refers to the second implementation.
- OneBuffer refers to the third implementation.
- MMap refers to the fourth implementation.

To estimate the costs the following parameters will be used :

- B(R) : The number of blocks that R occupies on disk.
- B(l_i) : The number of blocks that a line i occupies on disk.
- S : The number of main memory buffers available.
- D : The default buffer size of the Buffered implementation and it is equal to 8 kB based on its source code.

2 Observations on reading and writing

2.1 Implementation

The figure 1 shows an UML diagram describing the structure of the implementation:

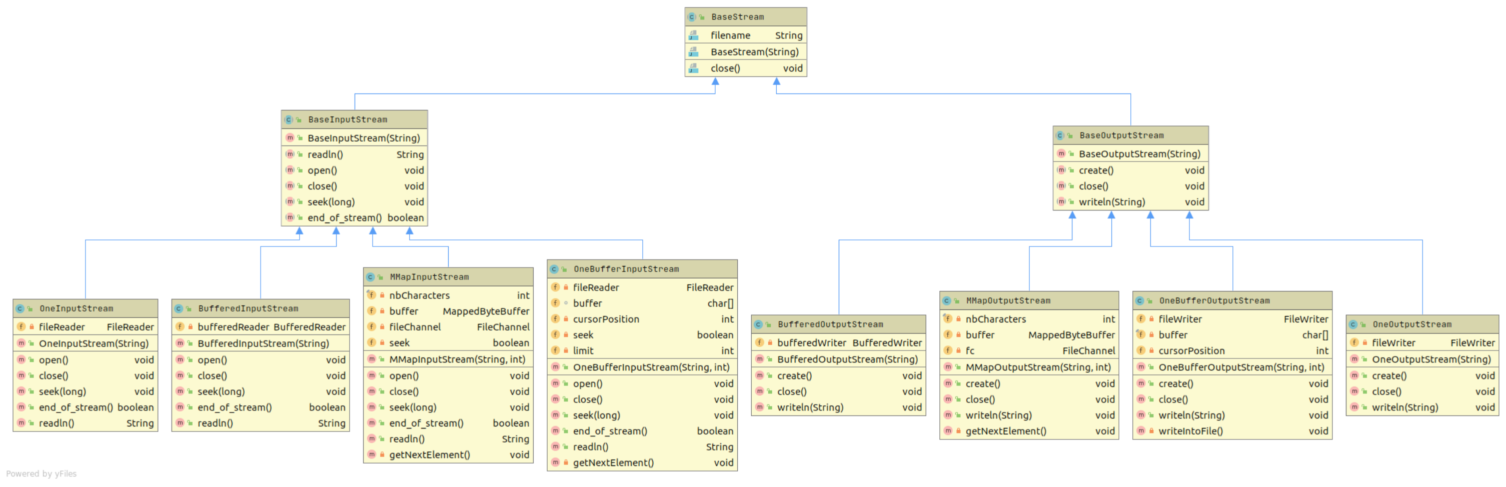


Figure 1: Stream structure

In order to facilitate the use of different implementations, it is useful to use abstract classes. In addition, a factory model has been introduced via different classes such as BaseStream, Generator and BaseAlgo. The purpose of using this type of methodology is to avoid code duplication and to obtain a generic code. This type of class allows us to instance abstract classes without knowing the type of the abstract class.

For the different implementations, the open/create and close methods are basic : they simply create a new object instance, close the stream and free all the system resources associated with it. For all implementations other than MMap, there is no predefined method to seek in the file.

To do this, the procedure chosen was as follows :

- Close the current stream
- Re-open the stream
- Skip n position

The `readln` and the `writeln` of each of these implementations will be explained in the following sections.

2.1.1 One

This method reads and writes the files byte by byte. It has a `FileReader` attribute.

- readln : It works by calling multiple times the `read` method of its `fileReader`. Each time the method is called, the type of the character read is verified, if it is an `end-of-line` or an `end-of-stream` the method returns the string else the character is added to the string.

- writeln : It works by calling the `write` method of its `fileReader` multiple times. Each character of the specified string is written in the file with the method. Then an `end-of-line` character is added.

Algorithm 1 readln

```
1:  if end of stream then
2:      return null
3:  else
4:      Create an empty line to store the next line
5:      Read the first character
6:      while the character is not an end of line or end of stream do
7:          Add the character to the line
8:          Read the next character
9:      end while
10:     return the line
11: end if
```

Algorithm 2 writeln

```
1: for each character in the line do
2:     Write the character and flush it into the file
3: end for
4: Write the end of line character and flush it into the file
```

2.1.2 Buffered

This method reads and writes the files with a predefined buffering mechanism. It has a `BufferedReader` that is wrapped around a `FileReader`.

- readln : This method just return the result of the call of the method `readLine()` of its `bufferedReader`.

- writeln : The `writeln` method just calls the method `write()` of its `bufferedReader` with the specified string with an `end-of-line` at the end.

Algorithm 3 readln

```
1: return the line returned by the readLine() method
```

Algorithm 4 writeln

```
1: Write the line with end of line character and flush them into the file
```

2.1.3 OneBuffer

This method reads and writes the files with a buffer of a predefined size B.

- readln : For each character of the buffer its type is verified, if it is an `end-of-line` or an `end-of-stream` the method returns the string, else the character is added to the string. When the buffer is empty, it is filled with a call of the `read` method of its `fileReader` that reads at the maximum B characters.

- writeln : While there are still characters to write, the buffer is filled with the next characters and they are written to the file with a call of the `write` method of its `fileReader` then adds an `end-of-line` at the end.

Algorithm 5 readln

```
 1: if end of stream then
 2:    return null
 3: else
 4:    Create an empty line to store the next line
 5:    while the character is not an end of line or end of stream do
 6:       if Cursor position exceeds buffer size then
 7:          Read the next B character(s) from the file
 8:          Set the cursor position to zero
 9:       end if
10:       Read the character from the buffer at the cursor position
11:       Add the character to the line
12:       Increment cursor position
13:    end while
14:    return the line
15: end if
```

Algorithm 6 writeln

```
 1: for each character in the line do
 2:    if cursor position exceeds buffer size then
 3:       Write the next B character(s) and flush it into the file
 4:       Set the cursor position to zero
 5:    end if
 6:    Add the character to the buffer at the cursor position
 7:    Increment cursor position
 8: end for
 9: Add the end of line character to the buffer at the cursor position
10: Increment cursor position
```

2.1.4 MMap

2.1.4.1 Concept

Memory mapping allows you to assign directly byte by byte a part of the file. The file must be present on the disk or on a device accessible from a file descriptor. This allows us to directly manipulate mapped portions of the file through a pointer. Writing or reading the file becomes as much simple as manipulating data from a variable. The advantage of this method is that it is not linked to RAM unlike the most common I/O methods. It is also advantageous for large files as the memory mapping allows to access specific areas of the file quite fast. A file can also be mapped by several applications simultaneously.

2.1.4.2 Methods

This method reads and writes the files by memory mapping. It maps a part of a file into a memory. It allows the program to modify this part of a file like a basic object in the dynamic memory.

- readln : For each element of the buffer its type is verified, if it is an `end-of-line` or an `end-of-stream` the function returns the string else it is added to the string. When the buffer is empty, it is filled thanks to a call of the `map` method of its `fileChannel`, it maps the next B characters. The use of a `ByteArrayOutputStream` object was needed in order to support special characters, such as ó.

- writeln : While there are still characters to write, the buffer is filled with the next characters and they are written to the file with a call of the `put` method. When the buffer is full it is replaced by the new buffer of the next mapped part.

Algorithm 7 readln

```
if end of stream then
    return null
else
    Create an empty line to store the next line
    while the character is not an end of line or end of stream do
        if the buffer has no remaining characters then
            Map the next B character(s) from the file
        end if
        Get the character from the mapped buffer
        Add the character to the line
    end while
    return the line
end if
```

Algorithm 8 writeln

```
for each character in the line do
    if the mapped buffer size has been exceeded then
        Map the next B character(s)
    end if
    Put the character into the mapped buffer
end for
Put the end of line character into the mapped buffer
```

2.2 Experiment 1.1

2.2.1 Discussion

The sequential reading is an algorithm that reads sequentially the specified file and computes the sum of the length of all the lines.

It has a cost that depends on the cost of `readln` on the entire file of the method used.

- One : The cost expected is equal to $B(R)$ I/O operations, as it reads the file byte by byte.
- Buffered : The cost expected is equal to $\lceil \frac{B(R)}{D} \rceil$ I/O operations [1].

- OneBuffer : The cost expected is equal to $\lceil \frac{B(R)}{S} \rceil$ I/O operations. As each time the reading is done from the buffer and not from the disk [1], until the buffer becomes empty then an I/O operation is performed to refill it.

- MMap : The cost expected is the same as the OneBuffer but as described in its JavaDoc "mapping a file into memory is more expensive than reading or writing a few tens of kilobytes of data via the usual read and write methods" so the cost can be more than expected.

After estimating the cost, an implementation ranking can be made for the expected results. First it is expected that the OneBuffer is the best, then the Buffered followed by the MMap and lastly the One.

2.2.2 Experimental observations

Before comparing different implementations, it is necessary to determine the optimal buffer size for the implementations that use it. In order to use it at its full power. For all the benchmarks of this experiment, the parameters were set to 1 iteration of warmup and 3 iterations of measurement. The average time will be calculated in seconds

2.2.2.1 Optimal buffer size values

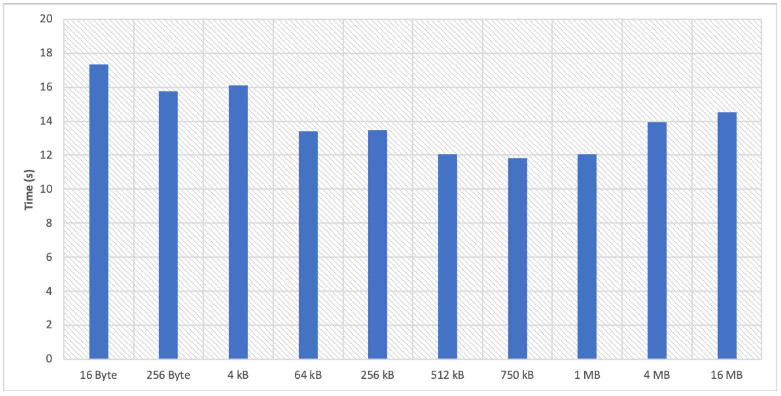
The benchmark was run on values belonging to three different scales.

Scale	**Value**
Bytes	16, 256
kB	4, 64, 256, 512, 750
MB	1, 4, 16

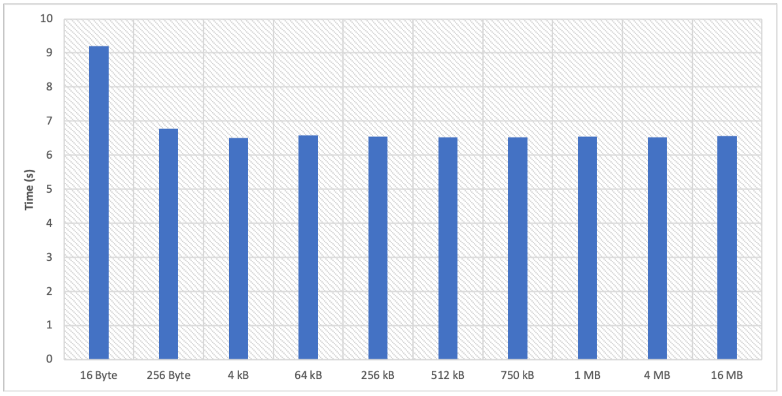
For OneBuffer implementation, the file chosen for the benchmark was "cast_info.csv"; its size is almost 1,4 GB. Depending on the results of the benchmark analysis in the figure 2a, increasing the buffer size at a certain point can speed up the execution time but exceeding it afterwards can make it worse again. For the upcoming length programs benchmarking, the buffer size is set to 750 kB.

[1] $\mathcal{O}(n)$, n refers the buffer size, will be ignored

For MMap implementation, the file chosen for the benchmark was "keyword.csv"; its size is almost 4 MB. Depending on the results of the benchmark analysis in the figure 2b, all values may be optimal except 16 Byte which is slower. To verify this, further benchmarking tests were carried on other files and for each of them, the different buffer sizes neither increase nor decrease the speed because they have almost identical execution time. To keep the same size as the OneBuffer implementation the buffer size is also set to 750 kB.



(a) OneBuffer



(b) MMap

Figure 2: Buffer size parameter tuning

2.2.2.2 Files benchmark

The benchmark was performed on all files, except for the MMap implementation where the execution time started to increase and become important at some point. Also, the buffer size has been set to 750 kB for OneBuffer and MMap. The table below shows the obtained values.

	One	Buffered	OneBuffer	Mmap
Files	Units (s/op)			
comp_cast_type.csv	0,001	0,0001	0,001	0,001
kind_type.csv	0,001	0,0001	0,001	0,001
company_type.csv	0,001	0,0001	0,001	0,001
role_type.csv	0,001	0,0001	0,001	0,001
link_type.csv	0,001	0,0001	0,001	0,001
info_type.csv	0,001	0,001	0,001	0,003
movie_link.csv	0,042	0,007	0,016	0,671
complete_cast.csv	0,16	0,019	0,03	2,36
keyword.csv	0,247	0,031	0,056	3,579
company_name.csv	1,059	0,129	0,221	15,898
movie_info_idx.csv	2,094	0,196	0,371	32,994
aka_title.csv	1,627	0,297	0,55	34,448
aka_name.csv	3,148	0,363	0,882	64,66
movie_companies.csv	3,148	0,5	0,981	85,379
movie_keyword.csv	4,058	0,583	1,012	88,95
title.csv	8,486	1,338	2,489	
char_name.csv	7,71	1,569	2,752	
name.csv	11,168	2,011	4,147	
person_info.csv	13,424	1,908	3,863	
movie_info.csv	33,935	3,469	8,641	
cast_info.csv	51,614	5,256	11,751	

To facilitate the reading of the result obtained, the figure 3 groups together these four implementations and their execution times.

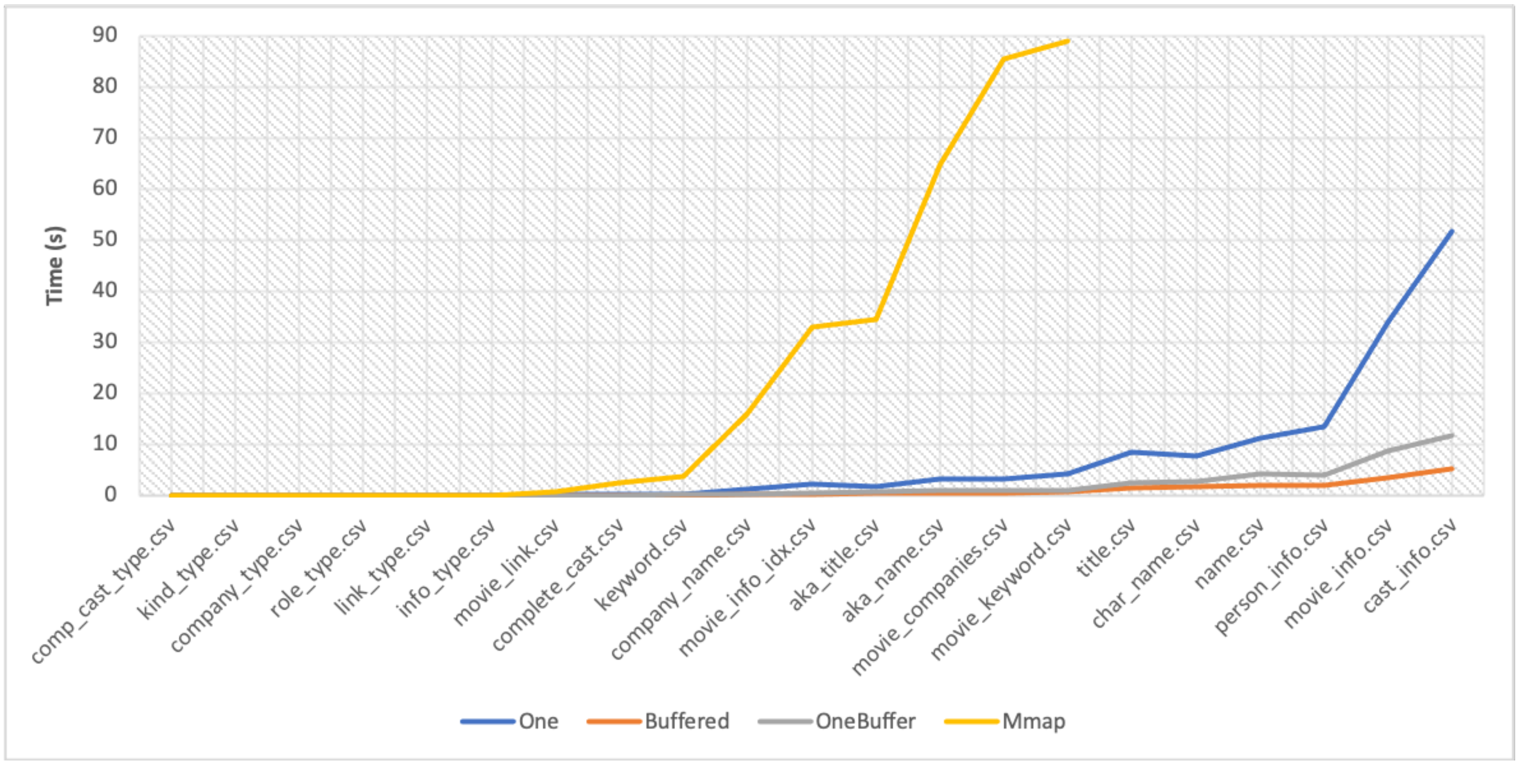


Figure 3: Length program execution time

2.2.3 Explanation

The ranking obtained does not follow the expected one. Because the Buffered is the best and not the OneBuffer. The choice of the OneBuffer was made because both use a memory buffer but the size of the OneBuffer is too big. The performances are quite close, so this difference is explained by the fact that the Buffered is well developed by java and optimized to the maximum. The other difference is that the One was better than the MMap, the reason why the MMap expected to be more performant is that it has a buffer in memory but as said in JavaDoc the simple reading can be better for small files and also the manual management of the UTF8 encoding for the MMap can also increase the execution time.

2.3 Experiment 1.2

2.3.1 Discussion

The random reading is an algorithm that randomly jumps, j times, to a position in the specified file and reads the next line from that position and calculates the sum of the length of all lines read. Its cost depends on the value of j and the cost of `seek` and `readln` of the method used.

As explained earlier, there is no predefined seek method for the One, Buffered and OneBuffer implementations. Therefore, an approach was followed to create it. The chosen method was too expensive because it is not executed as an atomic operation like the MMap. As three steps are necessary, first the stream must be closed, then it must be reopened so that the cursor can be placed at the beginning of the file stream and finally a certain number of bytes must be skipped to position the cursor at the desired location. The estimated cost for this block of operation is as follows :

- Close : release the resources + free the buffer [2].
- Reopen : allocate the resources + an-in memory buffer [2].
- Skip n bytes : 1 I/O operation + fill the buffer [2].

On the other hand the MMap uses a `FileChannel` method that allows to move the cursor and read from any position directly.

- Seek[3] :
  - One : The cost expected is equal to j I/O operations.
  - OneBuffer : The cost expected is equal to j I/O operations.
  - Buffered : The cost expected is equal to j I/O operations.
  - MMap : The cost expected is equal to 0 I/O operations.
- readln :
  - One : The cost expected is equal to $\sum_{i=0}^{j} B(l_i)$ I/O operations.
  - OneBuffer : The cost expected is equal to $\sum_{i=0}^{j} \lceil \frac{B(l_i)}{D} \rceil$ I/O operations.

[2]Does not apply for the One implementation

[3]Consider only the I/O cost

- Buffered : The cost expected is equal to $\sum_{i=0}^{j} \lceil \frac{B(l_i)}{S} \rceil$ I/O operations.
- MMap : The cost expected is equal to $\sum_{i=0}^{j} \lceil \frac{B(l_i)}{S} \rceil$ I/O operations.

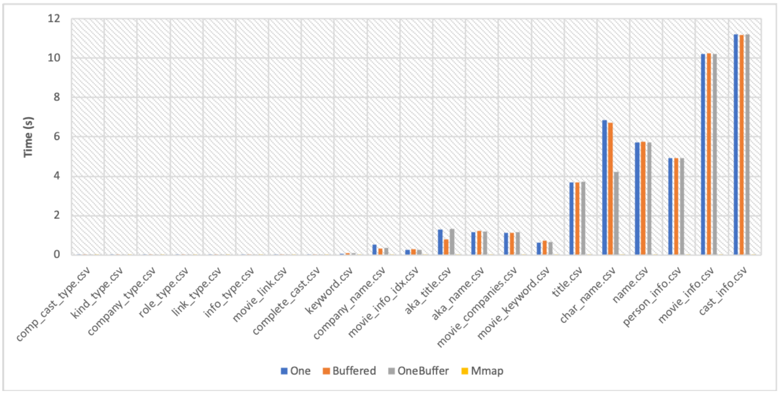
The estimated overall cost is as follows :

Implementation	**Cost (I/O operation)**
One	$j + \sum_{i=0}^{j} B(l_i)$
Buffered	$j + \sum_{i=0}^{j} \lceil \frac{B(l_i)}{D} \rceil$
OneBuffer	$j + \sum_{i=0}^{j} \lceil \frac{B(l_i)}{S} \rceil$
MMap	$\sum_{i=0}^{j} \lceil \frac{B(l_i)}{S} \rceil$

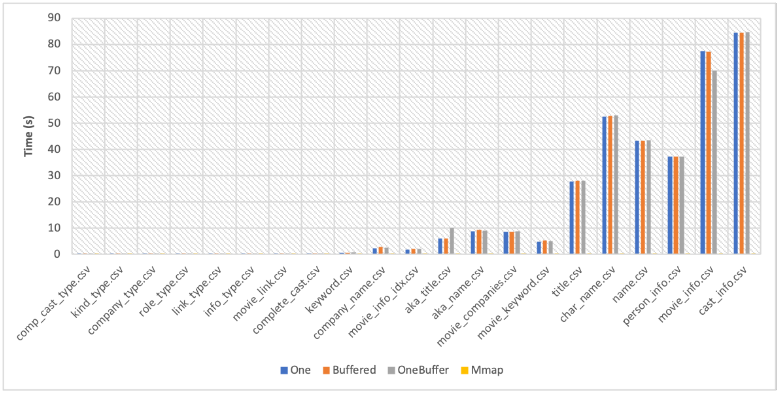
After estimating the cost, an implementation ranking can be made for the expected results. First it is expected that the MMap is the best, then the One followed by the Buffered and lastly the OneBuffer.

2.3.2 Experimental observations

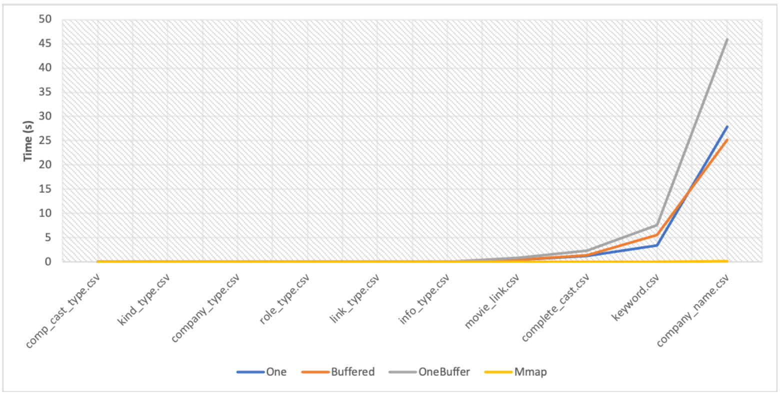
The benchmark was realized using four j-values which are 10, 100, 1000 and 10000. For the sub-benchmarks 10 and 100 all files were tested, except for the sub-benchmarks 1000 and 10000 where the execution time started to increase and become important after a certain file size. In addition, the buffer size was set to 750 kB for OneBuffer and MMap as for the first experiment. Figure 4 groups these four sub-benchmarks and their execution times.



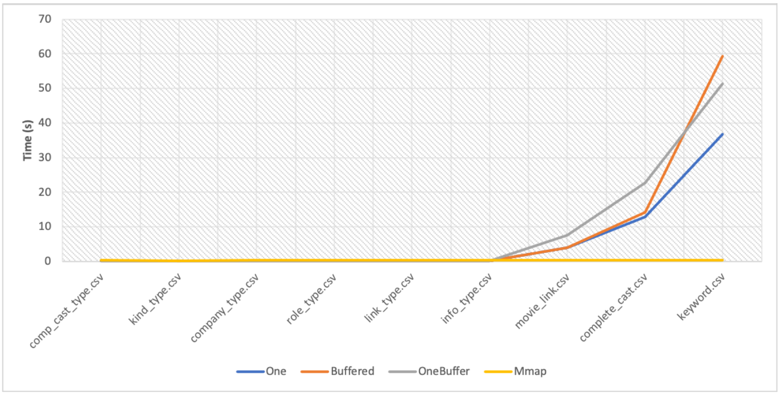
(a) j = 10



(b) j = 100



(c) j = 1000



(d) j = 10000

Figure 4: Randjump program execution on different values j

2.3.3 Explanation

The ranking obtained follows the expected one. MMap has the best performance and far exceeds that of other implementations, even for the largest j and file sizes. For other implementations and a smaller j equals 10 or 100. The difference goes almost unnoticed, on the other hand when the j becomes big the difference starts to dig in as the One remains faster followed by the Buffered and finally the OneBuffer which is well in line with the predictions made, because the ignored costs for the seek start to make the difference.

2.4 Experiment 1.3

2.4.1 Discussion

The RRMerge is an algorithm that merges the content of all the specified files. The first lines of every files are written in the output file, then the second lines etc., until there are no lines left.

It has a cost that depends on the number and the size of the files to merge, the cost of `readln` of the input stream used and the cost of `writeln` of the output stream used.

As the readln has already been seen in the first experiment, only the writeln will be.

- One : The cost expected is equal to $B(R)$ I/O operations, as it writes into the file byte by byte.
- Buffered : The cost expected is equal to $\lceil \frac{B(R)}{D} \rceil$ I/O operations [4].
- OneBuffer : The cost expected is equal to $\lceil \frac{B(R)}{S} \rceil$ I/O operations. As each time the write is done into the buffer and not the disk [4], until the buffer becomes full then an I/O operation is performed to flush it.
- MMap : The cost expected is the same as the OneBuffer but as described in its JavaDoc "mapping a file into memory is more expensive than reading or writing a few tens of kilobytes of data via the usual read and write methods" so the cost can be more than expected.

The estimated overall I/O operation cost is as follows :

Implementation	**Read**	**Write**
One	$B(R)$	$B(R)$
Buffered	$\lceil \frac{B(R)}{D} \rceil$	$\lceil \frac{B(R)}{D} \rceil$
OneBuffer	$\lceil \frac{B(R)}{S} \rceil$	$\lceil \frac{B(R)}{S} \rceil$
MMap	$\lceil \frac{B(R)}{S} \rceil$	$\lceil \frac{B(R)}{S} \rceil$

[4] $\mathcal{O}(n)$, n refers the buffer size, will be ignored

After estimating the cost and on the basis of the results of experience 1.1 and results of experience 1.2. A top 3 (x,y) pair ranking can be done for the expected results. As the results of experiment 1.1 confirmed that Buffered was too fast compared to MMap and knowing that the same output stream will have the same performance as any input stream, the choice of MMap as input stream can already be eliminated as it will never perform better than Buffered using the same output stream. As for reading, the expected ranking of the writing implementation is as follows. First, the OneBuffer is expected to be the best, then the Buffered, followed by the MMap and finally the One. So the top 3 ranking can be as follows :

1 (Buffered, OneBuffer)

2 (Buffered, Buffered)

3 (Buffered, MMap)

2.4.2 Experimental observations

For this benchmark, different versions of this algorithm will be created. Each version represents a unique pair (x,y) of implementation of the input stream x and implementation of the output stream y. For the x pair, the best implementation as identified in experiment 1.1 and the best implementation as identified in experiment 1.2 have been used, which are Buffered and MMap. For y, all possible implementations have been tested. The files were first sorted in ascending order, then sub-sets were created each time according to the number n chosen. For example, for n = 3 the first 3 small files were taken and merged. The maximum test value was 17, so the files "name.csv", "person_info.csv", "movie_info.csv" and "cast_info.csv" were not used. The table below shows the obtained values for the different (x,y) pairs merging different file sizes. In addition, the buffer size was set to 512 kB for OneBuffer and MMap. To facilitate the reading of the result obtained, the table has been divided into two figures. Figure 5 groups the buffered input stream with the four output streams and their execution times. On the other hand, the figure 6 groups the MMap input stream with the four output streams and their execution times.

	Byte					MB										
Pair	130	222	382	643	2691	0,7 MB	3,1	6,9	24,7	60	98,9	171,9	265	358,8	562,7	778,4
(Buffered, One)	0,001	0,002	0,003	0,003	0,009	1,412	6,617	14,559	52,022	128,359	209,653					
(Buffered, Buffered)	0,001	0,001	0,001	0,001	0,003	0,079	0,417	0,747	1,445	4,71	5,946	9,867	17,173	29,219	37,8	47,005
(Buffered, OneBuffer)	0,001	0,001	0,001	0,001	0,002	0,014	0,044	0,096	0,363	0,819	1,413	2,617	3,666	4,988	7,983	10,174
(Buffered, Mmap)	0,001	0,001	0,001	0,001	0,001	0,015	0,055	0,114	0,348	0,745	1,243	2,072	2,828	3,918	6,175	8,857
(Mmap, One)	0,003	0,003	0,004	0,005	0,015	2,606	12,176	26,428	93,02	231,51	375,48					
(Mmap, Buffered)	0,002	0,002	0,002	0,004	0,009	1,285	5,874	12,673	42,662	107,013	170,846					
(Mmap, OneBuffer)	0,002	0,002	0,003	0,004	0,01	1,247	5,451	11,928	41,538	102,849	166,711					
(Mmap, Mmap)	0,002	0,001	0,003	0,003	0,01	1,23	5,49	12,096	41,249	103,288	165,711					

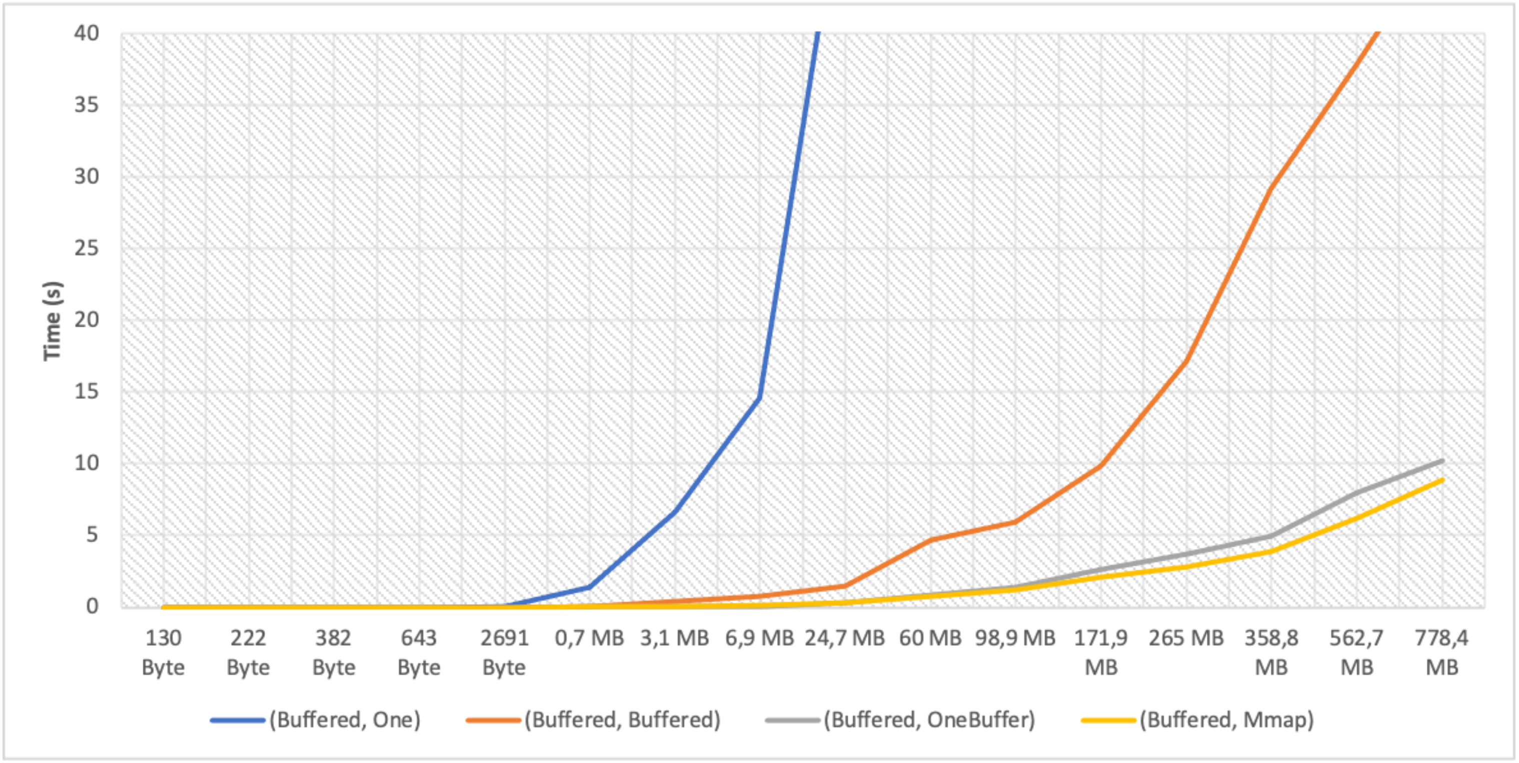


Figure 5: RRmerge program using the Buffered input stream

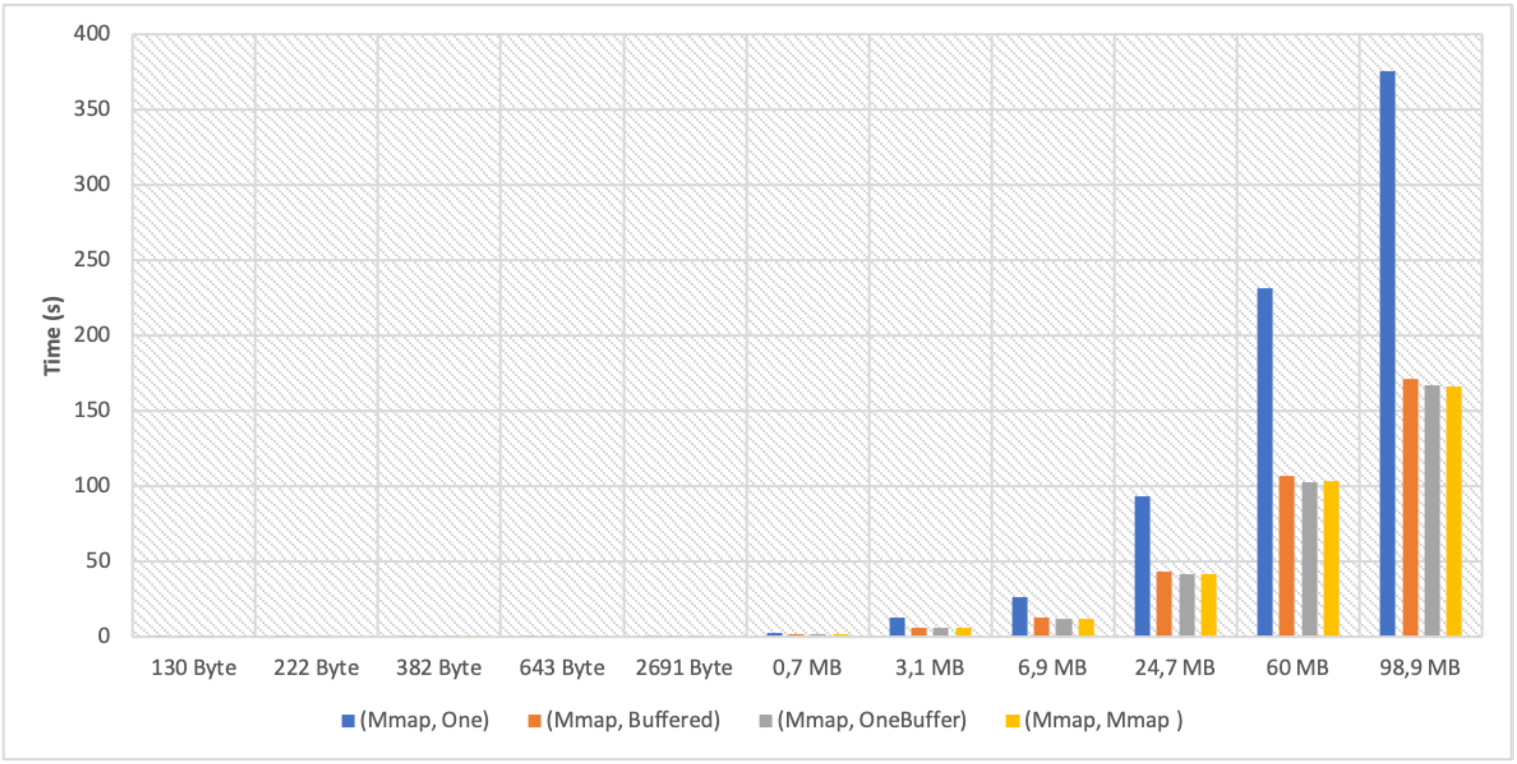


Figure 6: RRmerge program using the MMap input stream

2.4.3 Explanation

Globally the results are in agreement with the predictions made the only difference is the top 3 ranking order. The performance of MMap is too bad because it takes a long time for the same case where the Buffered made it in an instant. The output stream One has the worst execution time with the two input streams, and this has been excepted because the writing is done byte by byte, which is more expensive than the other output streams costs. However, this time the MMap outperformed the other output streams and was the best which is not expected because for reading it has the worst execution time. This could be explained by the fact that writing to the buffer immediately effects the changes to the mapped regions of the file. On the other hand, OneBuffer and Buffered have to wait until the buffer is full to commit the changes to the file. Also, the difference between the OneBuffer and the Buffered is noticeable which was not obvious for reading.

It is understandable that this difference is due to the size of the buffer which is much bigger than the Buffered one.

3 Observations on multi-way merge sort

3.1 Discussion

3.1.1 Expected behavior

Multi-way merge sort is an algorithm that sorts the lines of a file. The lines are sorted by the value of a specific column.

First the algorithm split the main file into multiple sub-files that are sorted by the value of the specific column. These files have approximately the same size M and they are put in a queue. Then the first predefined number d of files are removed from the queue and merged together. The merging is done by reading the current line of each of these files and writing the line in the new output file the one that has the lowest value in the specific column, then the next line is read from the same file. This process stops when the end of every files is reached, then the output file is added to the queue. This operation is done until the queue has d elements or fewer, in which case the operation is done one last time with the remaining files and the result is written in the predefined output file.

At the beginning the input file is split into $B(R)/M$ sorted sublists of M blocks. Then these sublists are merged into larger sublists, still sorted. After this second pass, there are $B(R)/M^2$ sorted sublists of M'^2 blocks. After the third pass, $B(R)/M^3$ of M^3 blocks. This continues until there are only one sorted sublist left. So, there are $\lceil \log_M B(R) \rceil$ passes. At each pass the entire input is read and written once, $2B(R)$. Hence the total cost of the multi-way merge is equal to $2B(R)\lceil \log_M B(R) \rceil$ I/O operations.

The estimated overall I/O operation cost is as follows :

Implementation	**Cost**
One	$2B(R)\lceil \log_M B(R) \rceil$
Buffered	$2\lceil \frac{B(R)}{D} \rceil \lceil \log_M \lceil \frac{B(R)}{D} \rceil \rceil$
OneBuffer	$2\lceil \frac{B(R)}{S} \rceil \lceil \log_M \lceil \frac{B(R)}{S} \rceil \rceil$
MMap	$2\lceil \frac{B(R)}{S} \rceil \lceil \log_M \lceil \frac{B(R)}{S} \rceil \rceil$

For this experiment, only the pair (Buffered, MMap) will be used. Its total cost is as follows : $\lceil \frac{B(R)}{D} \rceil \lceil \log_M \lceil \frac{B(R)}{D} \rceil + \lceil \frac{B(R)}{S} \rceil \lceil \log_M \lceil \frac{B(R)}{S} \rceil$

After estimating the cost, some predictions can be made about the expected parameters tuning results. First the M parameter will be predicted. The increasing of the M parameter should speed up the execution time because it represents the base of the log in the cost formula. On the other hand, the d and k parameters should not have a considerable influence on the performance, as the number of I/O operations remains the same for any of their values, the only difference would be on the complexity of the sorting algorithm and the resources allocated. The execution time of the algorithm launched on a file and on the sorted version of this file should not be different as the number of I/O operations will still be the same, the only thing that could change is the complexity of the sorting algorithm.

3.1.2 Implementation

In order to store the files to merge, it has been decided to use a queue containing the input stream linked to each of these files. To find the lowest value of the specific column in the merging process the use of a priority queue was needed. This priority queue is composed of tuples, one for each of the files to merge. In these tuples the first element is the input stream linked to a file and the second is a list of the value of each column of the current line. As the files of this project use the `csv` format, the lines are split and joined with a comma. The functions `String.split` and `String.join` have been used to split the lines in column and to join them again together.

3.2 Experimental observations

For this benchmark the best combination for reading and writing identified in experiment 1.3 will be used. As seen in experiment 1.3 result the best pair of input/output streams was (Buffered, MMap). Also, the MMap buffer size was set to 512 kB as in the experiment 1.3.

First, the value of M will be investigated. Its value will vary while keeping the value of the other variables fixed and random values have been assigned to them. It will vary between three different values : 512 kB, 1 MB and 4 MB. These values will be tested on four different files sorted by their size : “aka_title.csv”, “movie_keyword.csv”, “char_name.csv” and “person_info.csv”. The figure 7 shows the execution time of the algorithm with the different values of M on the four different files.

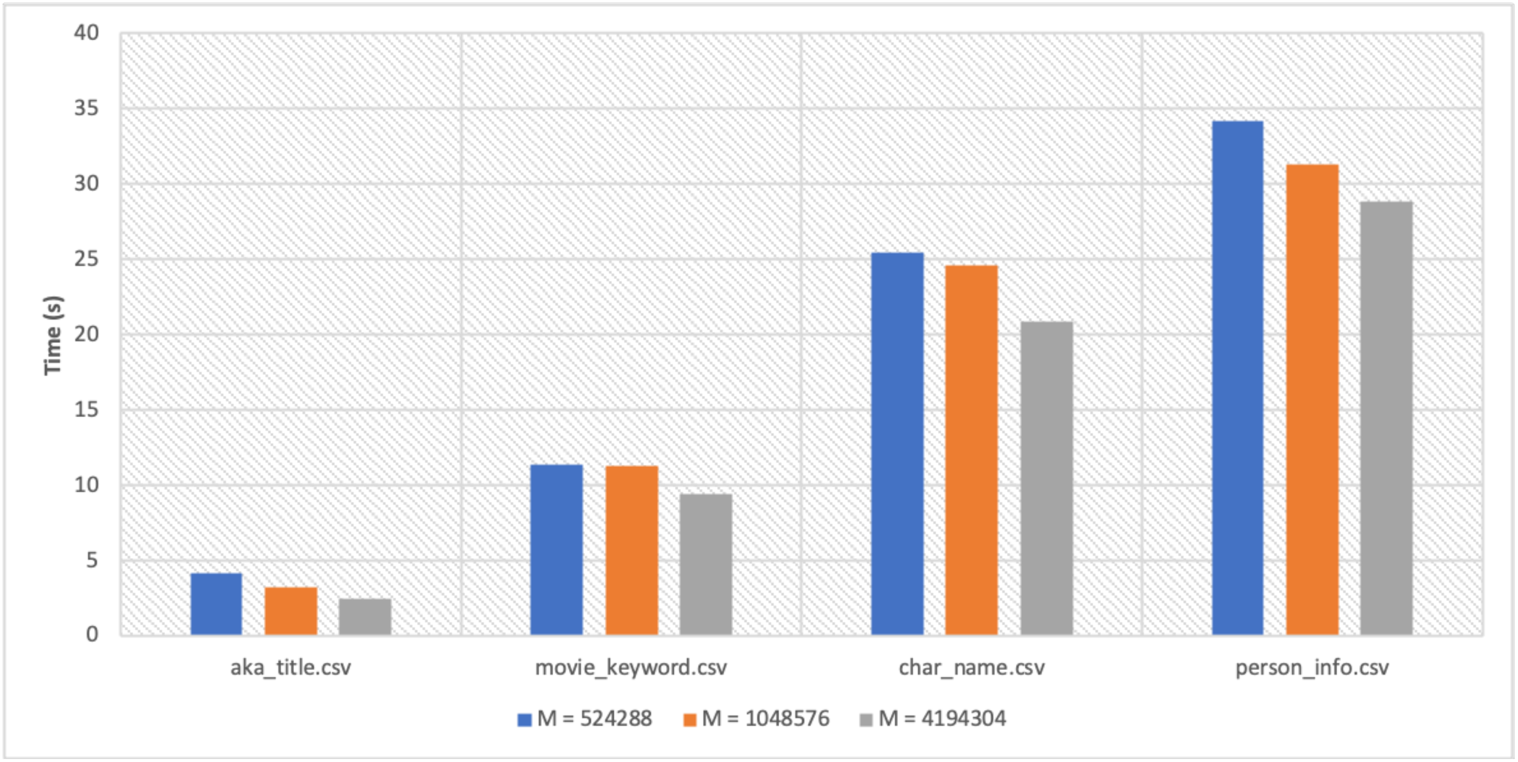


Figure 7: M parameter tuning

Next, the value of d will be investigated. Its value will vary while keeping the value of the other variables fixed, the M is set to the best value found in the M parameter tuning experiment. It will vary between eight different values : 5, 10, 15, 20, 25, 30, 35 and 40. These values will be tested on four different files sorted by their size : “company_name.csv”, “aka_name.csv”, “title.csv” and “name.csv”. The figure 8 shows the execution time of the algorithm with the different values of d on the four different files.

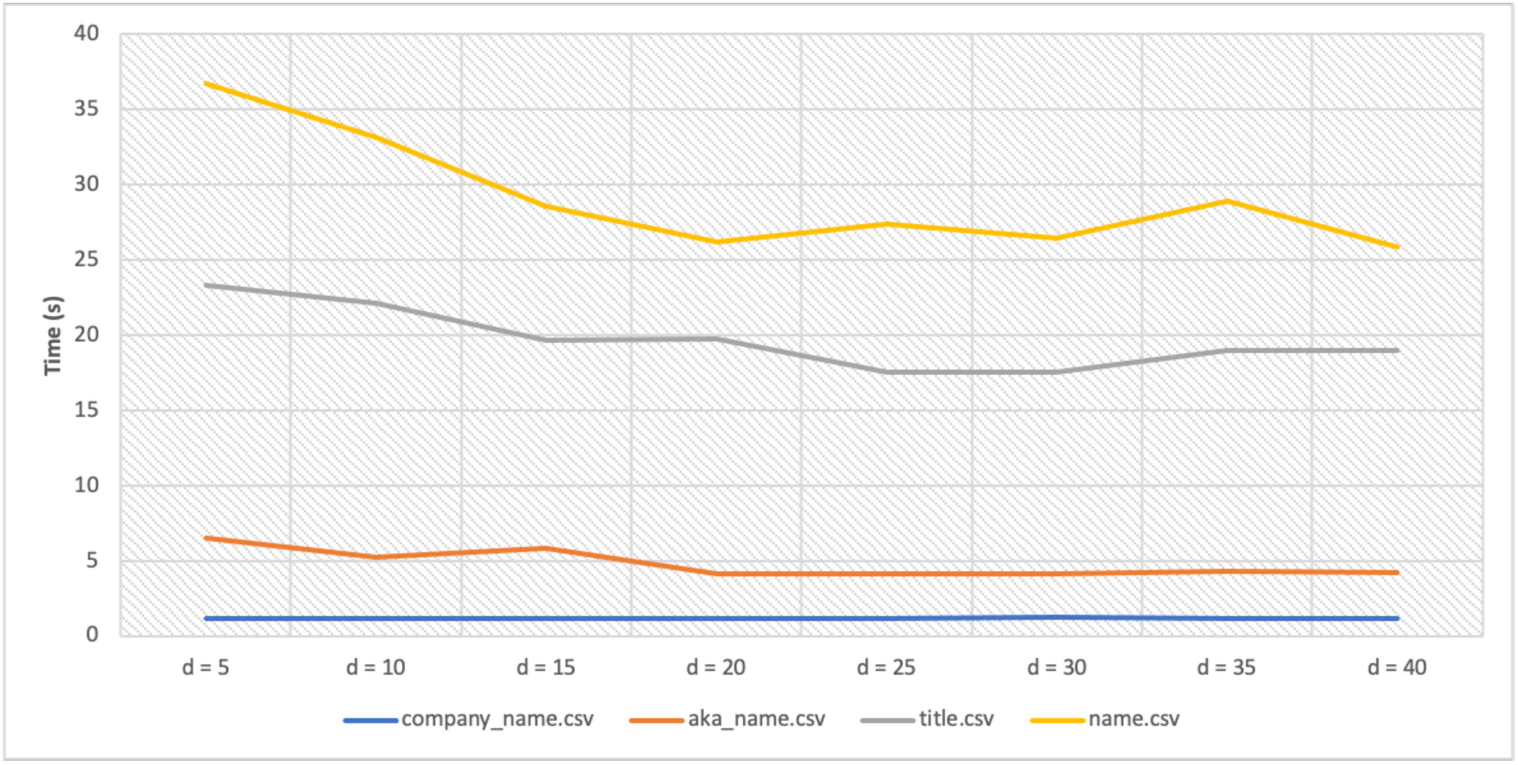
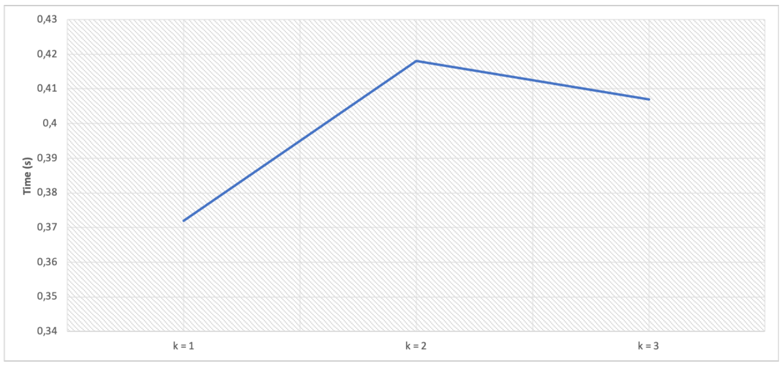
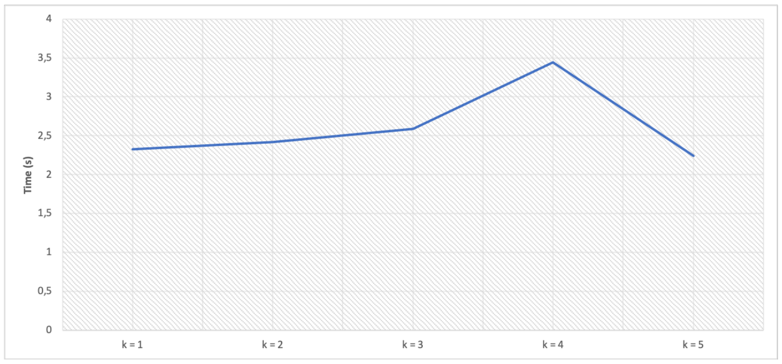


Figure 8: d parameter tuning

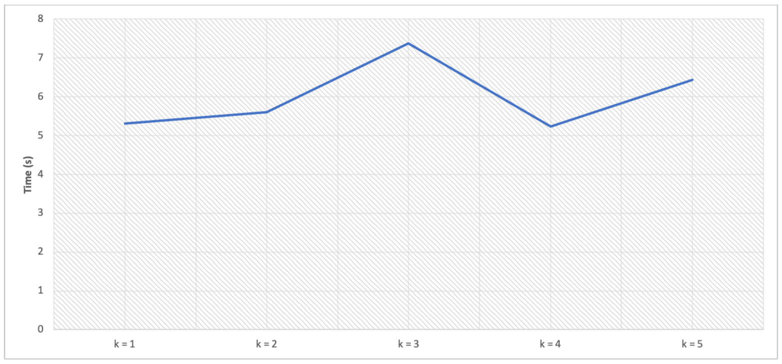
Then, it will be investigated if there is a difference in execution time when the algorithm sort different columns of the same file. The M and d parameters are set to the best optimal values found during their tuning experiment. The algorithm will be launched on all the columns of four different files : “keyword.csv”, “movie_info_idx.csv”, “movie_companies.csv” and “title.csv”. The figure 9 shows the execution time of the algorithm on the four different files on all of their columns.



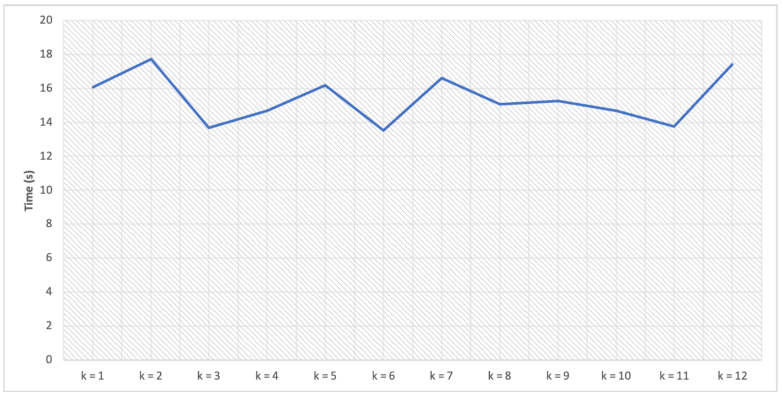
(a) keyword



(b) movie_info_idx



(c) movie_companies



(d) title

Figure 9: Files sorts according to different columns

Finally, it will be investigated if there is a difference in execution time when the algorithm is launched on an unsorted file and on a sorted one. The M is set to 4 MB and the d is set to 25 as they are the best values. The algorithm will be launched on the "cast_info.csv" file and it will be sorted by the seventh column. The figure 10 shows the execution time of the algorithm when it is launched on the file and on a presorted version of it.

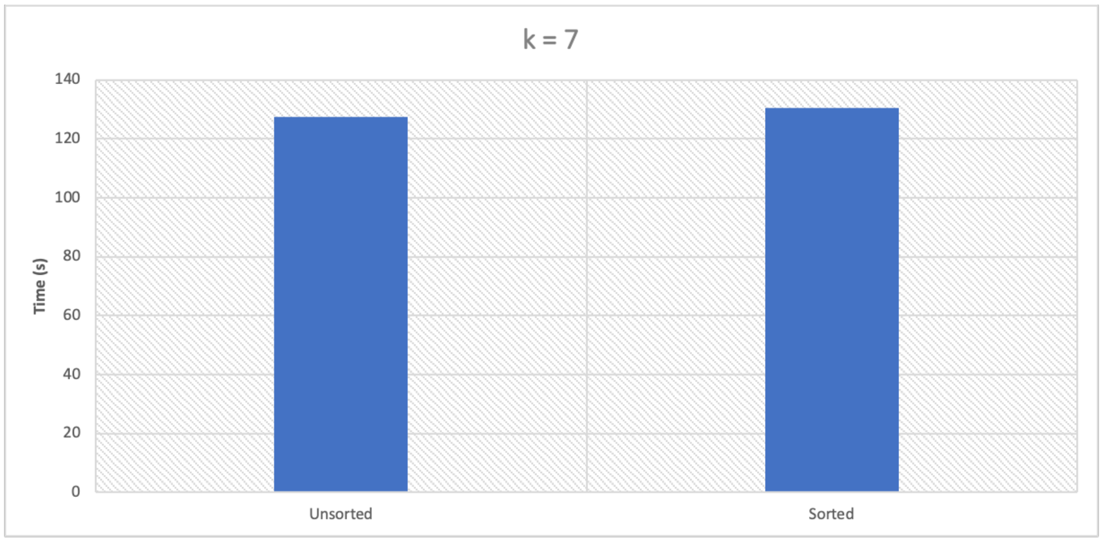


Figure 10: cast_info

3.3 Explanation

The prediction is in agreement with the result of the sub-benchmark. The results show that by increasing the value of M the execution time decreases. And that can be explained by the fact that M represent the base of the log in the cost formula, when the base increases the result decreases. In other words, the number of passes needed decreases as the number of available blocks in the memory increases.

Concerning the d parameter, the result demonstrates that the value of d for small files does not affect their execution time, but it does for bigger files. Globally there is not an optimal value for d as the execution time fluctuate in the same interval. And it comes down to the fact that the number of I/O operations needed does not depends on d as the file will be read and written entirely but it will change the number of temporary files to create. When d increases the number of temporary files decreases but the size of the memory that the program needs also increases as there are at most $\mathcal{O}(d)$ lines in memory.

About the k parameter, the result confirms what was expected as there are no big differences in the execution times. Files sorts according to different columns have no impact on the execution time of the algorithm. As the main memory sorting algorithm has almost the same performance because $O(d)$ in the test circumstances does not create a big difference for the complexity time of the algorithm.

As expected, the execution time of the algorithm on a file and on a sorted version of this file stays the same. This is due to the fact that the number of I/O operations the algorithm does not change because of the order of the lines as the length of the lines stays the same. The main memory algorithm does not know that the elements are already sorted so it will try to sort them, that means that the time complexity stays the same.

4 User manual

In the root folder of this project a `pom.xml` file has been added. This file follows the Maven Java structure. It allows to build the project and generate the benchmarks executable. All the result that were used in this report will be available in the `benchmark-logs` folder. In order to be able to reproduce them the following procedure should be followed.

```
$ mvn clean
$ mvn package
```

Once theses commands have been executed, a new directory named `target` will appear. This directory contains the executable.

When executing the benchmark jar, all the benchmarks are launched (which can take a while). To specify one benchmark only the name of the class is needed.

The command line can be as follow :

- To launch all the benchmarks of the OneBuffer (i.e. length, randjump):

  ```
  $ java -jar target/benchmarks.jar OneBuffer
  ```

- To launch the benchmarks of the length program with the OneBuffer implementation:

  ```
  $ java -jar target/benchmarks.jar OneBuffer.length
  ```

- To launch the benchmarks of the length program with the OneBuffer with a fixed BufferSize parameter :

```
$ java -jar target/benchmarks.jar OneBuffer.length -p
    BufferSize=1024
```

- To launch the benchmarks of the extsort program with a fixed input stream, a fixed output stream and k that takes the value 2 then 5 (there should only be commas in all the values the parameter should take, no space) :

```
$ java -jar target/benchmarks.jar extsort -p
    xPair=Buffered -p yPair=Mmap -p k=2,5
```

5 Conclusion

In this project four implementations of input/output have been done. By implementing them, it shows how files can be manipulated. Their performances have been observed over several experiments. It has been shown that each of these implementations have their strengths and their weaknesses, there is no perfect input or output streams for all cases. For example, when reading sequentially a file, the Buffered and the One buffer have demonstrated their superiority but in random reading the MMap showed its dominance. As it has been seen, the implementation using an in-memory buffer have always the best execution times. These two experiments have shown that the chosen input stream should be selected according to the use case to have the best performance.

Another concept learned in the project was the memory mapping. As opposed to the traditional input/output implementations, it does not use the RAM. It allows the manipulation of files directly through the dynamic memory.

It has been shown that the combining of two different types of streams for the input and output one is beneficial to obtain the best performance. As seen the pair (Buffered, MMap) and the pair (Buffered, OneBuffer) offer great performance compared to any other pairs.

The last experiment gave a real-world experience with the performance of an important external-memory algorithm, multiway merge, which sorts the lines of the files. The tuning of the different parameters allowed the finding of the optimal values for each of them.

The time complexity of the used algorithm is important, but number of I/O operations is also a value to take into account. The performances obtained on one machine might differ from the one obtained on another one, especially if the type of the secondary memory is not the same in the case of this project.