ECOLE
POLYTECHNIQUE
DE BRUXELLES

Database Systems Architecture report

Algorithms in Secondary Memory

Yasmina El Oudghiri / Boussant-roux Luc

Contents

1 Introduction

Nowadays, the data-sets stored in computers are getting bigger and bigger. It cannot be stored in the RAM. The use of external memory is often associated with a large amount of time for a memory search. As a consequence, external memory algorithms are really important to limit this time. We will first present the several ways to write or read data from memory. Then, the report will focus on a multi-way merge sort algorithm. The goal is to analyze the performance of external-memory algorithms.

2 Environment

The experiments were done on a macbook air with a macOS distribution. The laptop has 1.3GHz Intel Core i5 for a processor and has a 4Go RAM. We programmed the algorithms using Java where we documented all of our functions with a Javadoc. The Javadoc is generated using our Makefile and is in the "doc/javadoc" folder. A usage of our application is also in this documentation. We use a benchmark class for all of the tests. We then write all of our results in a csv file that we will study afterwards. For the launch of our tests we use our Makefile. The test data generated is of size 3Go.

3 Streams

Every input/output stream is an implementation of an only input/output interface. This permits to easily use each stream with the same functions. All of the streams have qualities regarding certain types of files. We added a stream that needed to be interpreted in the internal buffer section. It seems that there could have been two possibilities of internal buffers.

3.1 Expected Behavior

3.1.1 No Buffer

The first implementation that needed to be done is a direct stream with no buffers. The data is read/written one element at a time. We can assume that this implementation will be the one with the longest times for reading and writing large files. For each of the request to read or write, there is an exchange between memory and disk. We can estimate the number of I/Os needed using this cost formula :

$$N^2 * k$$

with N being the number of 32-bit integers in the file and k being the number of streams opened.

3.1.2 Buffered Stream

The second implementation is a buffered stream. This means that we will no longer read an element at a time but a certain number of elements, the buffer length B. We can then say that the implementation will be a bit faster than the first one for large files. However, when the files are small it can happen that the first implementation is faster than one with a buffer. The cost formula associated with this implementation is :

$$\lceil N^2/B \rceil * k$$

with B being the size of the Buffer used.

3.1.3 Internal Java Buffer

At first, we had come up with a stream using the Java Buffer from the second implementation but also an internal memory buffer. The internal buffer is then used for the reading and writing of the data. This implementation can be faster because of the fact that internal memory is more easily reachable by a process. However, the cost formula is the same as the one in the second implementation.

3.1.4 Internal Buffer

We then created another implementation for the internal memory buffer. This time we didn't use at all the Java buffer which meant that we read data one at a time to put into an internal memory buffer. As a consequence, the cost formula for I/0s becomes :

$$N * k$$

3.1.5 Memory Mapping

The fourth implementation is the one that is predicted as the fastest stream for large files. Whenever a process is created, a virtual memory page is also created. This method maps the data in the file inside the memory of the computer. Most often the memory is mapped in the Kernel space which means that it is really easily accessible. It is mapped by creating a byte-by-byte correlation between a portion of the file and the memory. As a consequence, you can limit the number of system calls to a minimum. The I/0s cost formula becomes :

$$N * k$$

3.2 Experimental observations

To illustrate our algorithms we chose to use files of size 26.2MB. To figure out the size of the files we wanted to use we had to find out what size files have in average. We used a study by Berkeley from 2013 about the size of files as shown in figure 1.

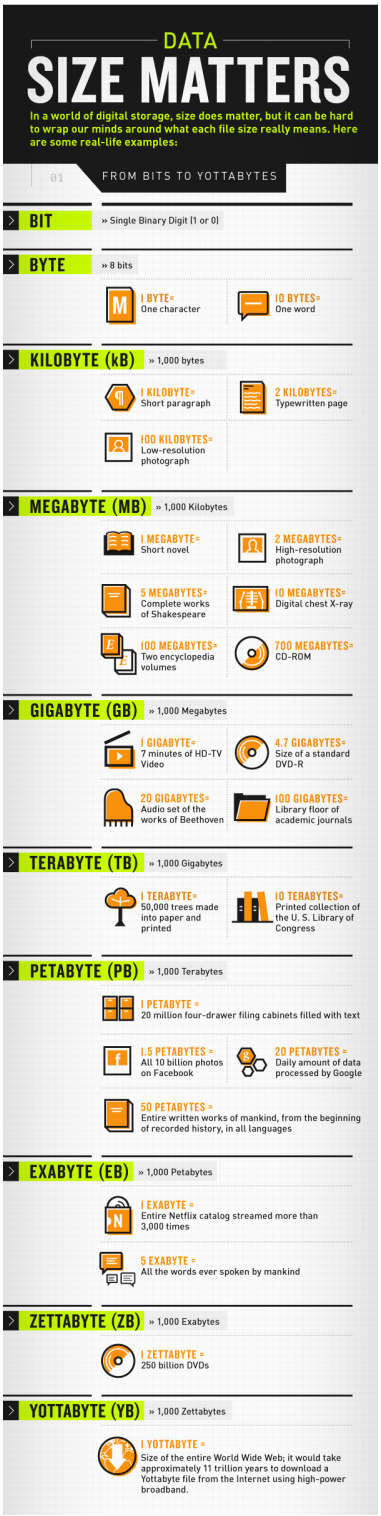

Figure 1: Image of a study by Berkeley (University of California)

We thought that choosing a file size in megabytes was a good starting point for testing real-life application to our algorithms. However, considering the computer we test the algorithms on, we cannot go over a certain file size. Moreover, we knew that the size of a mail attachment couldn't go over 25MB. Thus, we decided to choose files of size a little over 25MB to not be have it be easily divisible.

3.2.1 First test

We wanted to experiment two different factors of the streams. The first one is how the number of streams affect the time it takes to read/write files. Therefore, we tested for each algorithm the time it took to copy files with a certain number of streams. We only changed the number of streams to do this experiment and took a buffer length of 1024 bytes for all of the buffer-related methods.

We then experiment both the input and output streams. After looking at the results, we thought that the results would appear clearer with a logarithmic scale regarding the time. To have a correct approximation for the results we did the experiments multiple times and took an average of the results. However, the time it took to test everything was too long to have more than 4 tests per method per number of stream. Thus, we used at least 2 tests to average the results and more when we could. This explains why there can be some slight errors in the results but only punctual errors.

The number of streams is the number of input and the number of output streams opened. After that, for each input/output stream couple opened we copy a file.

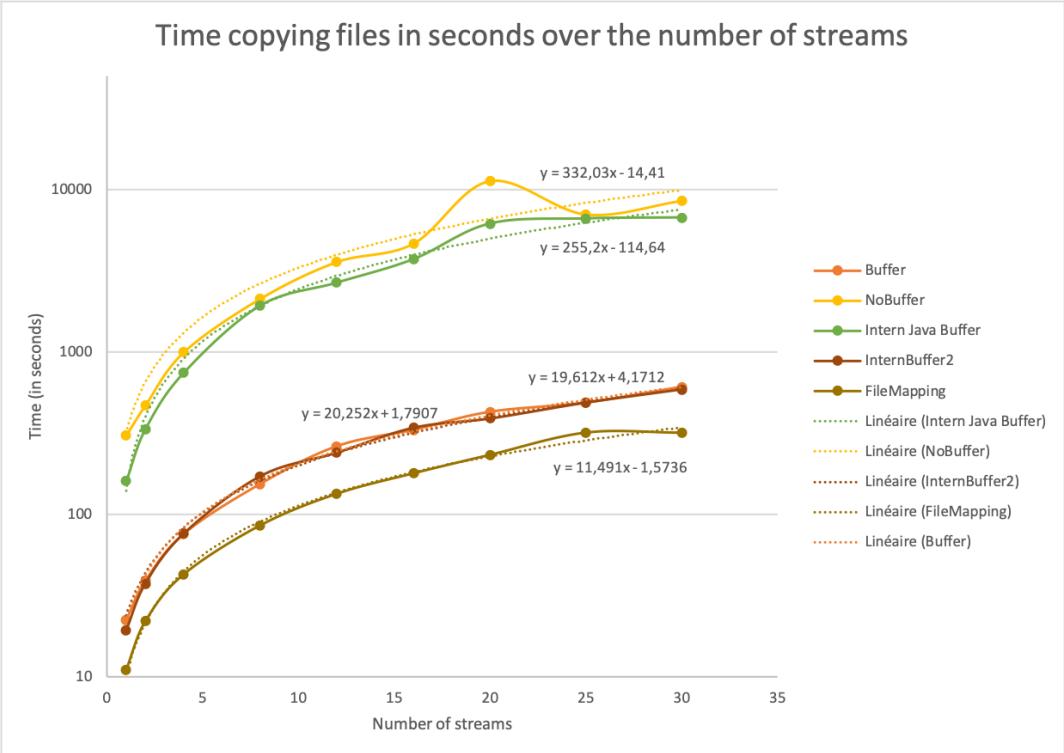

Figure 2: Graph showing the effects of the number of streams for each method

In figure 2, we can see that there are some methods that are faster for any number of stream. We can group the "NoBuffer" and "Inter Java Buffer" method together as the slowest methods. Then, the "FileMapping" method is the fastest method. Lastly, we have the "InternBuffer2" and "Buffer" method as the methods in the middle.

We can see a slight error for 20 streams in the "NoBuffer" method but we can still interpret the results correctly by extrapolating the other results.

Lastly, we can see the different equations for the approximations of the results. All of them are different but what is common to all is that they follow a linear tendency. As a consequence, we can think that we can predict the time it takes for any number of streams to copy files.

3.2.2 Second test

The second factor that we wanted to try and analyze is the size of the buffer. We only then test the implementations that use a buffer. The idea is to test over the same files as before with only one stream and then change the buffer length each time. We will again make multiple tests to have a good average of the results. We tested 2 times for each implementation and each buffer length.

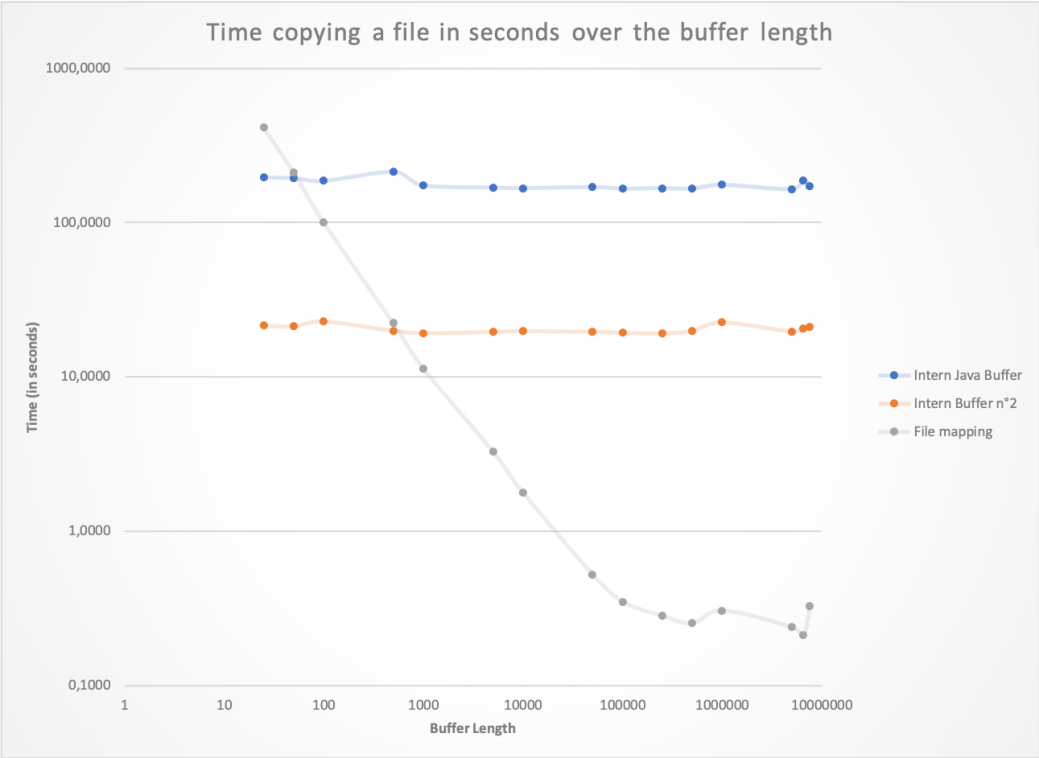

Figure 3: Graph showing the effects of buffer length for each method involving a buffer

We can see, in figure 3, 2 different type of behavior. Both intern buffer implementations have almost no effect from the different buffer length experimented. It seems that when we change the buffer length the "Inter Java Buffer" and "InternBuffer2" doesn't seem to be affected by the change. Whereas, the "FileMapping" is very affected by the buffer length.

With a small buffer, we can see that the "FileMapping" method is the slowest method. Ten, the larger the buffer is the less time it takes to copy the files. We then have files that are copied very fast. Lastly, we tested what would happen if the buffer length was higher than the file size and we got as a result a higher number than when the buffer size is the size of the files.

The optimal value of the buffer for the "FileMapping" method is the value of the file size. For the other methods, the optimal value is also the value of the file size but the difference isn't as much important as for the "FileMapping" method.

3.3 Overview

We can see that the expected behavior was pretty much observed in the different experiments that we had. However, some of the results weren't predicted correctly. We can see that the fact that the "Intern Java Buffer" method is similar to the "NoBuffer" method

wasn't predicted. This can be explained by the way it was implemented and the use of a Java Buffer and an Internal Memory Buffer.

Moreover, we can see that both "InternBuffer 2" and "Buffer" method have almost the same behavior. We thought that the fact that the buffer was stored in internal memory would have gotten us a better algorithm. Nonetheless, it seems that both methods are faster than the "NoBuffer" and "Intern Java Buffer" method.

Lastly, the fastest method "FileMapping" is very dependent of the buffer length. This could have been expected because of the time it takes to map part of the file each time. Therefore, when the buffer length reaches the size of the file we then get a unique map for the entire file and then it is only an access to the kernel memory which is very fast.

4 Multi-way merge sort

4.1 Expected Behavior

The multi-way merge sort algorithm has two distinct phases to its algorithm. The first phase consists of producing the initial runs. The point is to first divide the elements in the file into buffers where we will sort each individually using a main memory algorithm of our choice. The second phase is more complex and needs to be repeated until all the buffers have been merged together. During every step, we will merge a certain number of buffers into one output buffer. This will be done until we have only one final sorted file.

The analysis of the cost estimation of this algorithm resides in the division of each of the different steps. The first phase has a cost of $2 * \lceil N/B \rceil$ which is the number of streams that will be opened and then sorted.

The second phase is a bit more complicated. We first need to calculate the cost of a group of steps in the loop and then count the number of times we will loop over this group of steps. We will study group of steps because we want to try to count how many times the algorithm goes through the whole file. Thus, the cost of this group in the loop is the number of buffers that are read and sorted on each run. We know that this is $2 * \lceil N/B \rceil$ I/Os operation. Moreover, we know that the number of runs is reduced by a d factor every time. Therefore, with the master theorem for cost, we have a final cost for phase 2 of :

$$\theta(\lceil N/B \rceil * log_d(\lceil N/B \rceil))$$

Since the cost estimation of phase 2 dominates the one of phase 1, we can see that the final cost estimation of the multi-way merge sort algorithm is:

$$\theta(\lceil N/B \rceil * log_d(\lceil N/B \rceil))$$

4.2 Experimental observations

We will be experimenting 3 different factors of our implementation. We will be doing 5 iterations for each of the test that we will make to establish an average for each result. When testing a factor, the others will be fixed therefore we will only change the factor that we are studying at the moment.

The files that we will experiment on will all be files that have been created randomly. Thus, we can expect considering the large size of the files that the files aren't sorted.

4.2.1 File size

The first factor that we are going to study is the file size. We will be studying how the file size affects the time it takes to sort the file. For these tests, we fixed the number of streams, d, and the size of the main memory. The parameters will be :

$$\begin{cases} d = 29 & streams \\ M = 300000 & 32 - bitintegers \end{cases}$$

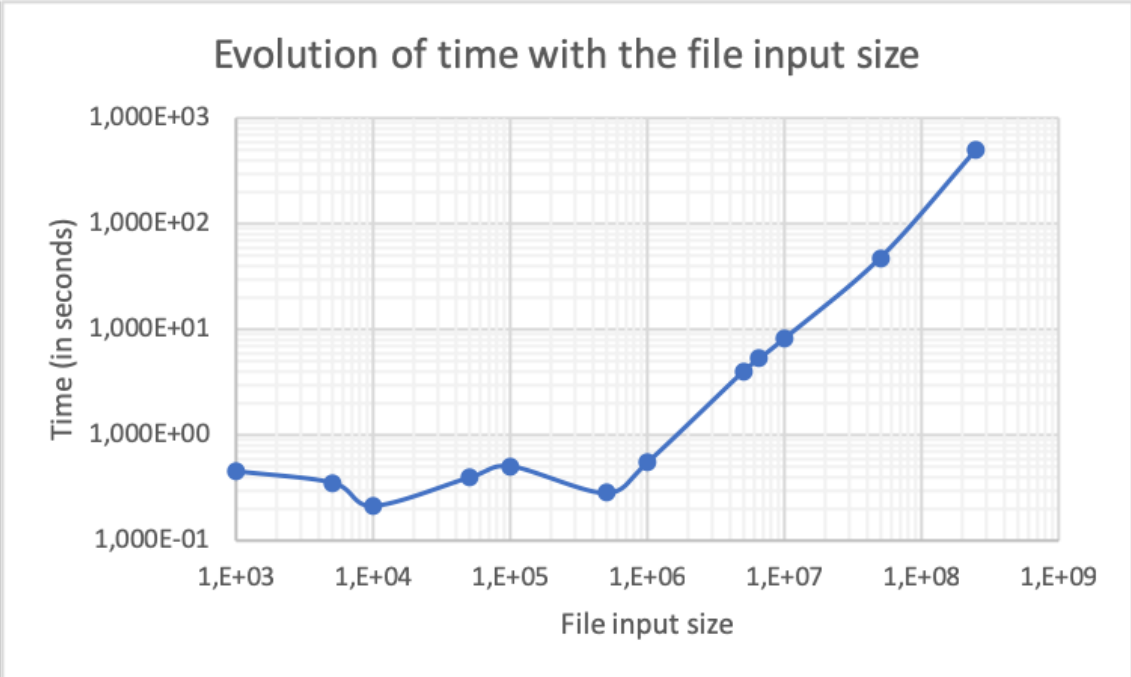

Figure 4: Graph showing the effects of the file input size on our multi-way merge sort implementation

We can see in figure 4 that there are two parts to the graph that we are studying. The first part shows that even though the input size increases the time it takes to sort the file doesn't increase. However, after the size reaches 1 million, the time escalates as quickly as

the size builds up.

4.2.2 Size of the main memory

A second factor that we can study is the size of the main memory. We then fix the other parameters as:

$$\begin{cases} d = 29 & streams \\ N = 6553600 & 32 - bitintegers \end{cases}$$

In our implementation of the multi-way merge sort, we can modify the memory size when we modify the "buffer length". This is how we will modify the memory size in this experiment.

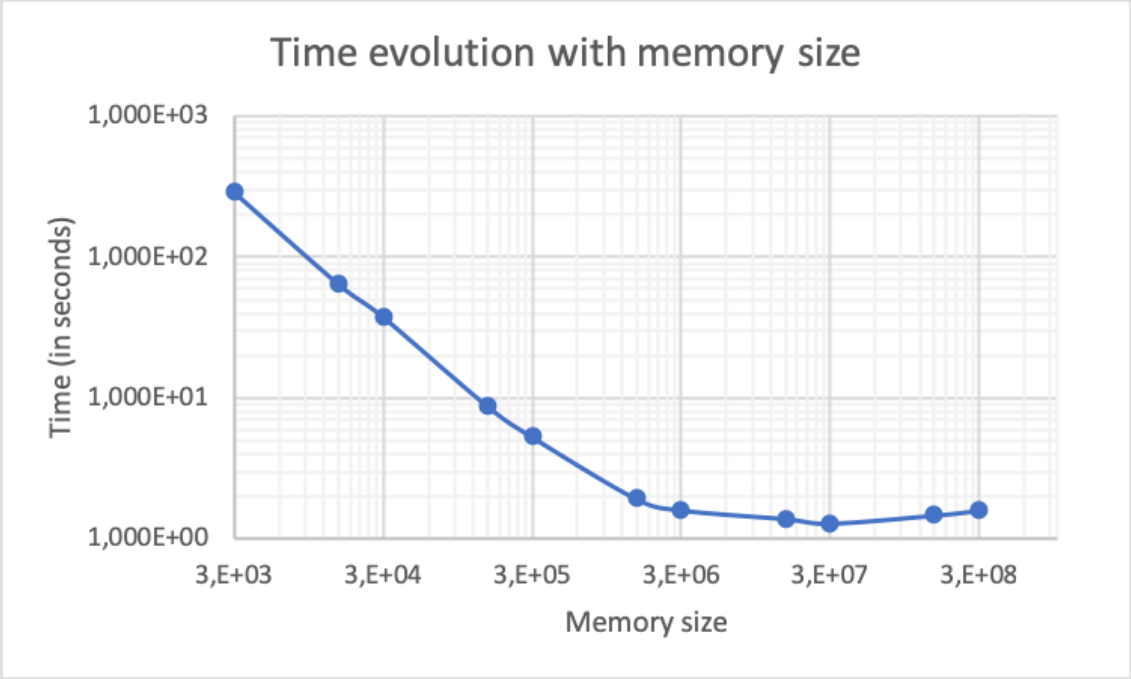

Figure 5: Graph showing the effects of memory size on our multi-way merge sort implementation

In figure 5, the evolution of the time is completely related to the memory size. We then can see the two same parts to the graph as before. However, this time the part that stagnates is when the memory size is bigger than $3 * 10^6$ 32-bit integers. Moreover, this time the time it takes to sort the file shortens a bit more than the increase in memory size until we reach $3 * 10^6$.

4.2.3 Number of streams

The last factor that needs to be studied is the number of streams to merge in one pass. The parameters that will be fixed for this study will have for value :

$$\begin{cases} M = 31200 & 32 - bitintegers \\ N = 6553600 & 32 - bitintegers \end{cases}$$

In our implementation, we cannot modify directly the number of streams without changing the memory. Therefore, we will have the number of streams modified but also what we call the "buffer length". This will permit to fix the memory size to the number specified above.

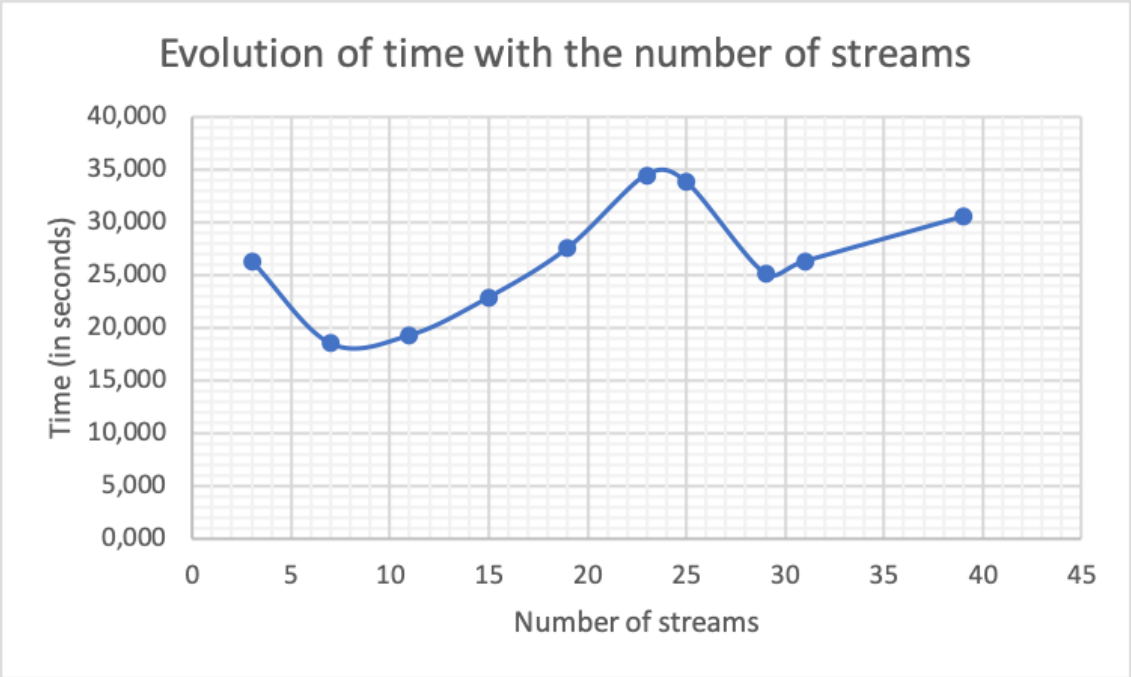

Figure 6: Graph showing the effects of the number of streams on our multi-way merge sort implementation

In figure 6, the number of streams has an impact over the time it takes to sort the files. Here we took rather small files to be able to lower the stream as much as possible. We can still see that the most of the time when the stream increases the time it takes to sort the file escalates also. However, when the number of streams is too low then the time increases. We can see that for this file size we have an optimal number of stream of 9.

4.2.4 Comparison with a main-memory sort

If we look at all of the different tests that we made we then can figure out what seems to be the optimal factors that we need to use for the next tests. We have implemented a

merge sort algorithm for the internal memory sort. We can call this method if the input file fits in memory. Here we will choose :

$$\begin{cases} M = 2000000 & 32 - bitintegers \\ d = 30 & streams \end{cases}$$

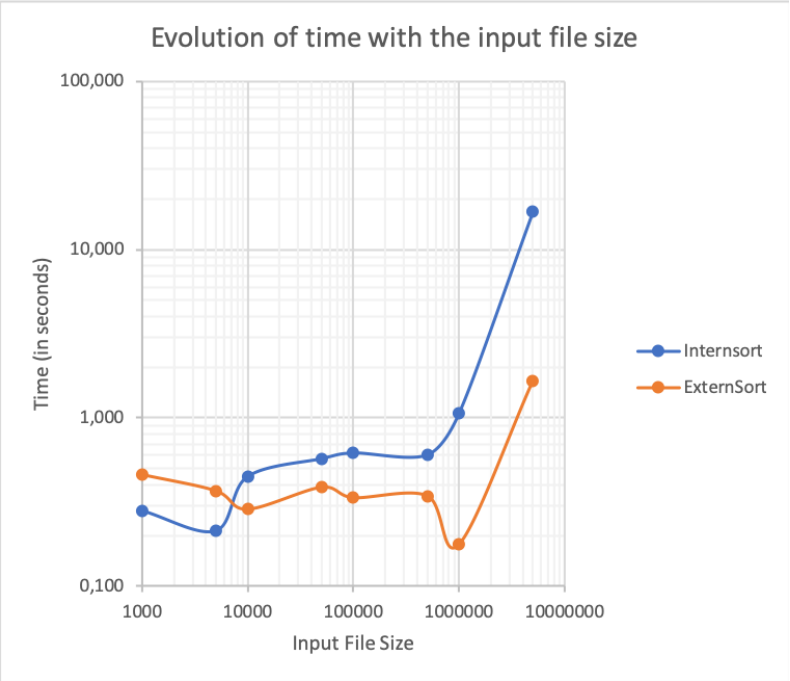

Figure 7: Graph showing the difference between an external memory algorithm and an internal memory algorithm

In figure 7, we can see that the internal memory algorithm is a bit faster than the external memory one for most of the values. This is very important because we can see the algorithm is faster. However, the evolution of both algorithms seems to follow the same line which means the difference between the two will not increase.

4.3 Overview

We can see that the evolution of the time it takes to sort a file is as expected. We have three different phenomenons that can be explained.

Firstly, we can see that when the file input size is bigger than the memory size, the time it takes to sort the file increases. This is as we could have expected it to be because it is normal that the time it takes to sort a file increases with its size.

However, we can see that when we compare this algorithm with the internal memory sort algorithms our implementation then isn't very efficient. This is because we do additional work splitting the file in buffer even though we don't need to.

Moreover, we have the proof that when the memory size is bigger than the size of the file the time it takes to sort the file is at his lowest. This regroups to what has already been said before.

To conclude, our implementation is very important because we can then sort files bigger than our memory in a rather correct time. We can see that the use of this implantation can be very important nowadays considering the increasing size of our files.

5 Conclusion

Our conclusion to this report will reflect on what we learned while doing this project. This project showed us a real-life experience of what an external memory algorithm could do and how it is implemented.

It taught us that the conception of this kind of algorithm resides in the division of the project in multiple tasks. This is a phenomenon that is very important in this project. We tried to divide as much as possible the different tasks in this project. Therefore, we also wanted to re-use as much code as possible which we did by using our interfaces and our Main class.

Another thing this project had us learn is that a project is as good as its documentation. This is the reason why we tried to document as much as possible the code and also why we created the Javadoc. It was very important while trying to interact between ourselves. It was much easier to share our code with the documentation.

Regarding the streams, we learned a lot with the file mapping. The research that we did on this was very important to understand really how the internal memory of a computer works. But, also how better to interact between an external memory.

Moreover, we proved that the access time towards disk memory is bigger than the access time is internal memory. Regarding databases, we can now better apprehend the way we make requests and the way we conceive the indexes of our databases. Indeed, if we make a request (update for example), performance-wise, it is better to have 4KB blocks than the whole table as one block (time isn't linearly proportional to the size).