Université Libre de Bruxelles

INFO-H417 : Database systems architecture

Project report

Professor:
Stijn Vansummeren

Students :
Miró-Manuel Matagne
Duc Minh Nguyen
Linh Tran Ngoc

Academic Year 2020 - 2021

Contents

1 Introduction

The goal of this project, as part of the course of Database systems architecture, is to compare the performances of different methods for reading from and writing to secondary memory. The main source of performance loss is due to I/O operations, and we will therefore try to determine which implementations can reduce these I/O costs to the maximum, and which are less efficient. In addition, it is required to implement an external-memory merge-sort algorithm to examine its performances under different parameters.

In order to reach this goal, this report will describe first of all the machine environment on which the experiments were ran on. Then, after an explanation of the code architecture, the concept of mapping will be explained in detail. Finally, the different experiments as well as the Multi-way merge will be discussed.

2 Hardware and general setup

The implementation of this project was done in C++, using Visual Studio C++ on Windows. The virtual environment used was CLion for Windows.

All the measures presented in this report have been done on the same computer in order to be able to easily compare the values. This machine runs on Windows 10, uses a HDD disk, and has a 8GB memory. We are conscious that the results might be slower than on a machine using an SSD as access latency might be 1 or 2 orders of magnitude faster in that case.

3 Code architecture

The code architecture is generally speaking quite straightforward. The folders `InputStreams` and `OutputStreams` respectively group all the different input stream and output stream implementations. The folder `Experiments` groups all the 3 experiments that have been made throughout the project.

3.1 Input streams

All 4 input stream implementations inherit from one common parent class called `InputStream` in order to avoid duplicating code.

`InputStream1` uses the `read` function in order to read characters one by one until an end of line is detected. `InputStream2` makes use of `fgets` in order to fetch an entire line at the time from the considered file. `InputStream3` contains an internal buffer of size `B` (called `buffer`) and fills it up everytime it is required to read something that is not contained in it. `InputStream4` uses file mapping in order to read lines from a file. As this is an important part of our project, a theoretical point about file mappings will be made at section 4.

3.2 Output streams

All 4 output stream implementations also inherit from a common class called `OutputStream`.

These 4 implementations work very similarly to the input streams, in a symmetric way. `write` is used instead of `read` and `fputs` instead of `fgets`. Once again, `OutputStream4` makes use of file mappings.

3.3 Experiments

A separate class is made for each one of the experiments. Notice however that only the header files are present for these classes since all experiments implement a template function which allows to avoid repeating code for each different input or output classes we want to apply the experiment to. The template is able to execute the experiment for any input stream given for `Experiment1` and `Experiment2`, and for any combination of input and output streams for `Experiment3`.

3.4 Multi-way merge sort

The `MultiwayMerge` class is used to execute the external multi-way merge sort. It does not require a template since it uses only one input and output stream implementation.

3.5 Testing

The experiments have to be tested, and in particularly their times of execution have to be measured in order to see the influence of each parameter on the efficiency of execution. The `Chrono` class is used in order to handle time measurements, it acts like a chronometer. All the tests on the several experiments as well as the Multi-way merge sort have been implemented in the `Measurement` class. Although this function might be quite long (there are many parameters to vary for each experiment), it is quite simple and is only used to measure execution times of experiments with varying parameters.

It is important to point out that all measurements present in this report are average values : the tests are executed multiple times in order to have more reliable values. The number of repetitions of a test depends on the expected execution time (when a test is extremely long, fewer repetitions will be made in order to avoid running tests for an excessive time).

4 Mapping

Since file mappings are present in our implementations and that it is not a very straightforward process, a theoretical explanation of this functionality is given.

Mapping a file can be compared to the action of creating a link between the disk (where the file is stored) and the physical memory (where the process' address space is). Indeed, the memory-mapped files allow us to access files on the disk as they were present in physical memory through pointers (like accessing dynamic memory). Reading a memory-mapped file (MMF) can be done by dereferencing a pointer in the designated range of addresses and similarly, writing data can be done by assigning a value to a dereferenced pointer. However, reading/writing to the disk is handled at a lower level to improve performance, so the file input/output (I/O) is not done exactly at the time the reading/writing of the memory-mapped file is performed.

MMF I/O operations have the advantage of being fast because data transfers are performed in a fix number of pages of data. Depending on the OS, the size of a page can vary. In our case, it is 4096 bytes and the MMF uses 16 pages (65536 bytes) for each I/O (it corresponds to the allocation granularity of the OS). Those pages of memory are managed by the virtual-memory manager (VMM) and because all the disk I/O operations have been standardised when it is performed by this VMM, it is a lot faster (routine memory accesses).

The main advantage of this method is that we can map a big file in memory even if this file is a lot bigger than the available RAM. Indeed, it is the principle of virtual memory that we use. We can access the whole file through a pointer as it was in RAM, but the I/O is performed only when accessing a part of the file not already in RAM. Each I/O contains 65 536 bytes (in our case) which obviously contains itself the part of the file needed. It is only if we want to read another part of the file that is not in those 65 536 bytes that another I/O will be performed. Note that if multiple I/Os are performed, older chunks of data of the file will be removed from the RAM at some point (otherwise the whole concept of mapping would be irrelevant if we kept everything in RAM). So by using MMF I/O all the I/O interactions now occur in the user space (`read()` and `write()` include a pointer to a buffer in the process' space where the data is stored, so the kernel has to copy it, whereas MMF is directly in the file process' address space, so process has a direct access to the file).

An important remark has to be made nevertheless, using MMFs for simply reading a file into RAM compared to other methods of reading will not differ a lot in terms of performance. However, for different cases it can be a lot faster, if we want to modify a record of a large database by over-writing it, each time a new part of the record is needed, another file read is required (a lot of I/Os are needed). In the MMF approach, when the record is first accessed, all the page(s) of memory containing the record are read into memory. So, there is no system call overhead when accessing memory mapped files after the initial call. It may be possible that we do not see improvement of the performance using file mapping, here is why : methods that can use the disk cache will not be significantly slower unless a random reading is performed. Indeed, memory maps allow to keep using pages without additional overhead until you are done while `read()` does not.

Depending on systems, memory-mapped file functions differ. Here is how we implemented it on Windows with C++. To perform the memory mapping file, we call `CreateFileMapping`. This will only create a memory-mapped file object (very few resources needed even if it is a large file) into the physical memory. Then we need to map a view of a memory-mapped file, to do so, we need a valid handle to the MMF object (it is returned by the `CreateFileMapping` function). The `mapViewOfFile` function allows us to map a view of the file in order to perform our writing/reading. It is only when a view of the file is mapped that resources are loaded in the process's address space. We implemented this in such a way that we map and unmap parts of the file. Note, however, that only a multiple of the allocation granularity can be used for the offset (which is why we choose to round the size of the buffer `B` to a multiple of this value). In fact, this method of mapping part of the file is not really relevant (beside the pedagogical aspect), because one of the advantages of the view of the file is that we can read a file that is bigger than the process's address space, so every I/O is performed only when needed (the mapping does not imply I/O, only accessing a mapped region of the file will induce I/O !). Thus mapping the whole file will not bring the whole file in RAM, accessing part of the file mapped, however, will provoke an I/O (if the corresponding resources have not already been brought into RAM).

For the writing part of the project, we use a copy function in order to put data into the file (data and the file are both in the process's address space). It is like copying data from a buffer to another. Note that when the view is unmapped (or when the file-mapping object is deleted), changes are automatically written into the disk.

5 Experiment 1 : Sequential reading

The goal of this first experiment is to determine which of the 4 input stream implementations is more efficient for sequential reading, as well as finding the pros and cons of each implementation.

The code relative to this experiment is located in the `Experiment1` class, which implements a template function called `length`, that can be used with any input stream class. This function calls the `readln` function of the appropriate input stream class, and computes the length of each line, which is added to a counter `sum`. Once the end of the file is reached, the total length of the file is returned.

5.1 First implementation

In the first implementation of the input stream (class `InputStream1`), we used the `read` system call function, which can be seen as a function provided by the kernel. Without going into too much detail, a function from the system call is generally slower and need to switch between the user space and the kernel. The advantage is that we have more control and the reading is performed directly when asked (which is not the case for the `fgets` function for example). The `readln` function reads characters one by one until reaching the end of the line. Therefore, for each line, the number of necessary I/O operations to return the line is equal to the number of characters in the line. By extension, when reading the file completely and computing its length in this experiment, the total number of I/O operations is inevitably equal to the number of characters contained in the file, which is `N`. Therefore, we can write the cost function of `length` when the `InputStream1` class is used, which we will call `length1` as follows :

$$C(\texttt{length1}) = N \tag{1}$$

The number of I/O therefore evolves linearly in function of the size of the file being read. This relation can be verified by measuring the execution time of the function on several files of different sizes, and plotting the execution time in function of the file size. These measurements have been computed on every file of the dataset, taking the average execution time on 10 total executions per file. These results are available in the appendix A. The most meaningful results in order to verify the cost function are shown in the table 1 below.

File name	File length (N)	Execution time in ms (t)	N/t
`aka_name`	73004383	288342,21	253,19
`char_name`	215711567	868975	248,24
`comp_cast_type`	45	1,93	23,35
`complete_cast`	2414495	9242,43	261,24
`movie_link`	656584	2611,33	251,44

Table 1: Main results of Experiment 1 using InputStream1

As per the cost function defined at equation 1, the ratio $\frac{N}{t}$ should be a constant. For most of the files tested, this value fluctuated between 245 and 265, but some values are particularly low. For instance, as shown in the table, the file `comp_cast_type` has a ratio of 23,35 which is surprisingly low compared to the rest of the results. This is justified by the very small length of that file. Indeed, in the case of very short files, the constant costs induced by the file openings and closings are not negligible for example. Therefore, the execution time will be higher when put in proportion to the file length. For larger files, the execution time is so big that the constant costs can be neglected, or at least their impact is way less noticeable. Notice that these surprisingly low ratio values occur for files that have a length up to around 2000 at least (the `info_type` file presents a ratio of 48, for a length of 1815, see appendix A).

By analyzing these results, or the complete table at appendix A.1 and neglecting the very short files, an average value for the ratio $\frac{N}{t}$ is computed, and is equal to 246,74 ms. The standard deviation is then calculated on these same values, and equals to 17,94. In order to compare these values with the other implementations where the ratios will have much different values, we compute the coefficient of variation, defined as the quotient of the standard deviation by the mean. The results are summarized here below :

$$\mu = 246,74$$
$$\sigma = 17,94$$
$$c_v = \frac{\sigma}{\mu} = 7,27\%$$

The coefficient of variation is relatively small when the very short files are neglected, we can therefore consider the cost function established at equation 1 is correct. In order to confirm this result, a graph plotting the execution time of the `length` function on all files of the dataset is computed :

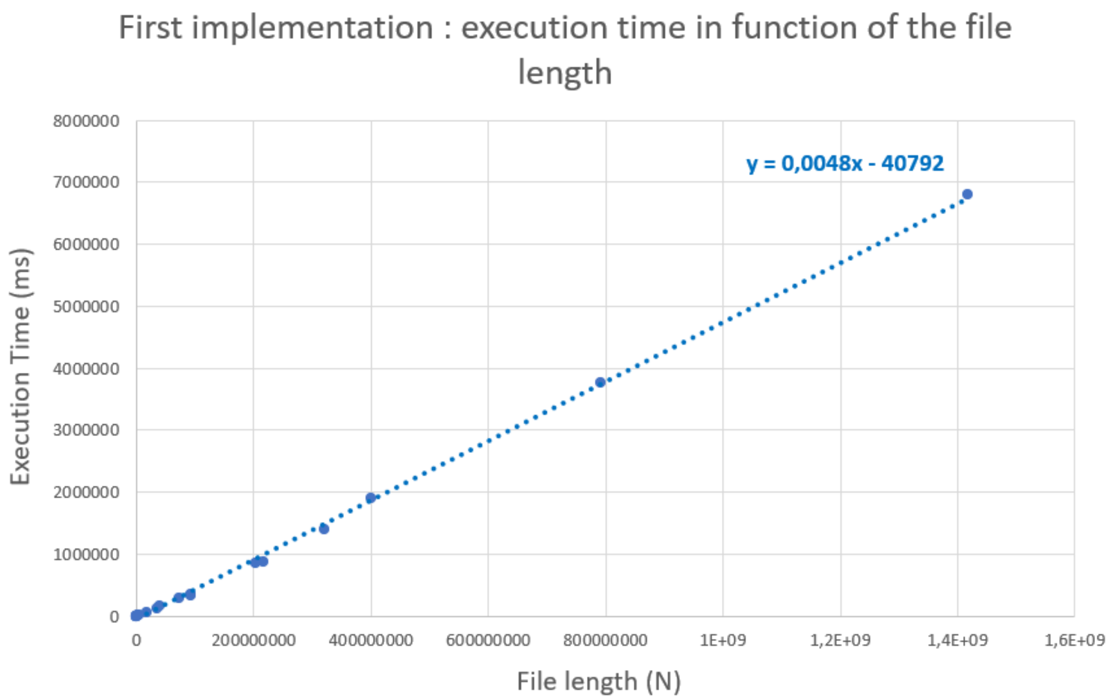


Figure 1: Execution time of the length function in Experiment 1 using InputStream1 in function of the file length

The data seems to be arranged in a linear fashion. Indeed, the trendline of this set of data is computed and we obtain an affine function equation as displayed on the graph. Notice that the intercept is at 40792ms, which is very small compared to the execution times we measured.

Furthermore, we can estimate the cost of a single I/O operation involving the fetching of a single character. This is given by $\frac{t}{N}$, which gives, using the average computed previously, $4,05.10^{-6}s$. Notice that this corresponds to the value of the slope of the trendline on the previous graph (where the values are in milliseconds). We were expecting a value quite a bit higher than this one (around $10^{-3}s$ for a HDD for the access latency), we believe the fact that we are reading the file sequentially might considerably reduce the cost of the I/O since the disk head will always be right next to the next character to be read (if the data is stored sequentially on disk).

5.2 Second implementation

In the second implementation of the input stream, the `readln` function is based on the `fgets` function defined in the C library `stdio`. It is indeed a library call (so no mode switching like in a system call), and uses buffered I/O's which allows to use the cache to have better performance. This function uses an internal buffer in order to retrieve a line from a file. This function might be a little bit misleading, as it takes as argument an integer `n` representing the maximal number of arguments to be read. One could therefore believe that this integer is the size of the buffer used to read a line from the file. However, by taking a look at the source code of `fgets`[1], it is clear that the function used to refill the buffer (`__srefill`) does not depend on `n`. The calculations should therefore not be too different when modifying `n`, in terms of I/O operations. It is worth noticing that `fgets` is an optimized function that takes advantage of several hardware optimizations and uses the cache to improve performance. We will call `S` the size of the internal buffer, which is in our case equal to 4096 (as it is defined by the `BUFSIZ` constant of the `stdio` library). This buffer is refilled only when it is empty, therefore the cost function appears to be :

$$C(\texttt{length2}) = \left\lceil \frac{N}{S} \right\rceil \tag{2}$$

Notice the ceiling function around the fraction. This is explained by the fact that an I/O operation will be necessary to store the last elements of the file in a buffer, even though there might be less than `S` characters left in the file.

In order to verify this cost function, the execution time of this implementation of the Experiment 1 is computed on all the files of the dataset, and the results can be found in appendix A.4. The main findings, on files of different sizes are presented below :

File name	File length (N)	Execution time in ms (t)	N/t
`aka_name`	73004383	3667,61	19909,17
`char_name`	215711567	11619,2	18565,10
`comp_cast_type`	45	0,151	298,01
`complete_cast`	2414495	282,008	8561,80
`movie_link`	656584	69,1148	9499,90

Table 2: Main results of Experiment 1 using InputStream2

First of all, it is clearly noticeable that this implementation is much faster than the first one. The $\frac{N}{t}$ ratio seems to have values that are much more diversified than previously. This is due to the ceiling function, which breaks the linearity of the relation, and as a consequence leads to the fact that $\frac{N}{t}$ is not a constant. Furthermore, this method is optimized in order to work rapidly with the operating system, the parameter `S` is chosen widely in order to optimize the efficiency of the I/O operations. As previously, results are quite different on the smaller files like `comp_cast_type.csv` due to the non negligible constant costs. By computing the average value of the $\frac{N}{t}$ ratio, as well as the disparity of the values through the standard deviation and the coefficient of variation, we obtain the following results :

$$\mu = 15109,41$$
$$\sigma = 4357,23$$
$$c_v = \frac{\sigma}{\mu} = 28,83\%$$

The coefficient of variation is, as expected, higher than with the first implementation, due to the ceiling function present in the cost, and due to the optimizations of the `fgets` function which also might break the linearity. In order to confirm this cost function is correct, we can plot the execution times in function of the file lengths for all the files :

[1]https://android.googlesource.com/platform/bionic/+/ics-mr0/libc/stdio/fgets.c

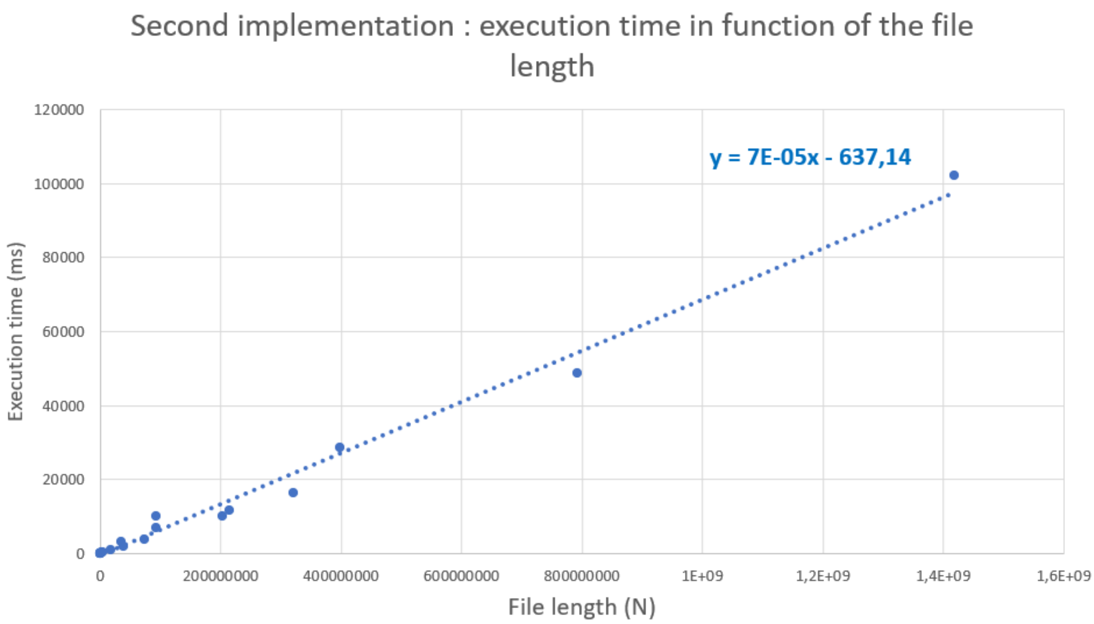


Figure 2: Execution time of the length function in Experiment 1 using InputStream2 in function of the file length

We can observe that, even though the linearity is not as *strong* as in the previous implementation, the trendline still approximates the results relatively well, and that our cost function is probably correct for this implementation.

5.3 Third implementation

In the third implementation of the input stream (class `InputStream3`), we used the same reading function as in the first implementation but equipped with a buffer. Unlike the second implementation, no buffered I/O's are performed, only direct I/O's are being performed here. The `readln` function uses a buffer of size `B` in order to read a line from a file. Every time the buffer is empty, it is reloaded with the `B` next characters of the file to be read. Therefore, it is important to notice that there could be more than 1 I/O operation per line if `B` is smaller than the length of the considered line. On the other hand, there could also be less than 1 I/O operation per line, if the combined length of 2 successive lines is smaller than `B` for instance. Therefore, in order to read a file in its entirety to find the total length, the cost is directly related to the number of bufferings of `B` characters that will be necessary throughout. This allows us to define a cost function for the `length` function using the `InputStream3` class, which we will call `length3` :

$$C(\texttt{length3}) = \left\lceil \frac{N}{B} \right\rceil \tag{3}$$

This cost function is therefore inversely proportional to `B` when the same file is being treated. When `B` is maintained constant, the cost should evolve quite linearly in regard to the number of characters in the file, although the ceiling function should introduce *steps* in this function which slightly breaks the linearity. Table 26 shows an overview of the results obtained for `B` = 100. The complete results are in appendix A.3.

File name	File length (N)	Execution time (t)	ceil(N/B)	ceil(N/B)/t
`aka_name`	73004383	8785,92	7300439	830,92
`char_name`	215711567	25043,9	21571157	861,33
`comp_cast_type`	45	14,50	5	0,34
`complete_cast`	2414495	338,03	241450	714,28
`movie_link`	656584	82,33	65659	797,49

Table 3: Results of Experiment 1 using InputStream3

This table outputs the ratio $\frac{\lceil \frac{N}{B} \rceil}{t}$, which should be close to a constant. We do however observe that for a constant `B`, not only is this value highly impacted by the small files (as before), but the fluctuations of the ratio also seem a bit more important than during the first implementation. By computing the average value of that ratio, as well as the standard deviation and the coefficient of variation without considering the very small files, we obtain :

$$\mu = 811,42$$
$$\sigma = 109,81$$
$$c_v = \frac{\sigma}{\mu} = 13,53\%$$

Although this coefficient of variation remains small enough in order to confirm the cost function established earlier for a fixed value of `B`, it is nearly twice higher than the one obtained during the first implementation. We suppose these variations are due to the fact that although the number of I/O operations is given by the formula 3, the duration of each ones of these I/O operations will not necessarily be the same. Indeed, the last I/O operation will typically be shorter : it has the same access latency as the previous ones, but less data has to be transferred on average, so the time taken for the throughput will typically be smaller. On top of that, the ceiling function breaks the linearity as described previously. Nevertheless, we can build a graph in order to prove the linear character of the cost function 3 with a constant `B` :

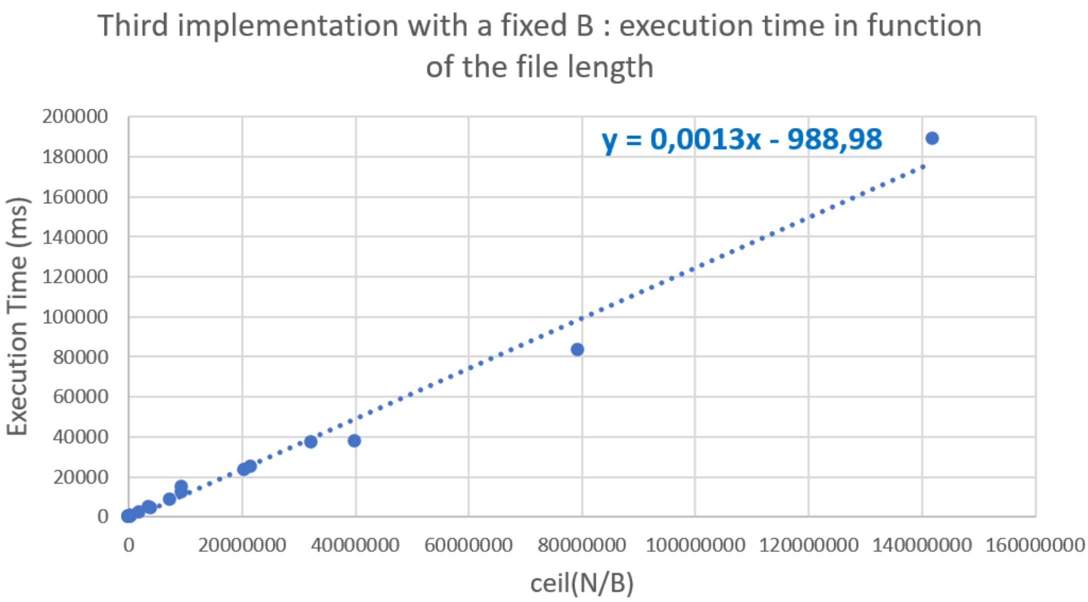


Figure 3: Execution time of the length function in Experiment 1 using InputStream3 with a constant B = 100, in function of the file length

We see the linear trendline approximates quite well the values plotted, and has an intercept of 988.98ms, which is very small compared to the measured lengths. Therefore, these results obtained with B $= 100$ are a first justification of the cost formula 3.

Next, it is necessary to show the impact of `B` on the execution time. In order to do that, a same file `aka_name` is read multiple times with several different `B` values. The main findings of this experiment are presented below :

B	Execution time is ms (t)	ceil(N/B)	ceil(N/B)/t
10	31596,1	7300439	231,06
50	10836,9	1460088	134,73
100	8151,63	730044	89,56
150	7225,41	486696	67,36
200	6692,75	365022	54,54

Table 4: Main results of Experiment 1 using InputStream3 on aka_name

We do indeed observe that when `B` gets bigger, the execution time tends to decrease. Furthermore, the hypothesis we made concerning the variability of the ratio $\frac{\lceil \frac{N}{B} \rceil}{t}$ is way more visible in this case and seems to be correct : when `B` is bigger, the cost of each I/O operation is bigger (so the ratio decreases when `B` increases) because there is more data to be fetched. This is therefore a limitation of our cost function, which represents the number of I/O operations, but is not completely and directly linked to the execution time. By plotting the results of this experiment on `B` values going from 10 to 1000 with a step of 10 on the `aka_name` file, the following graph is obtained :

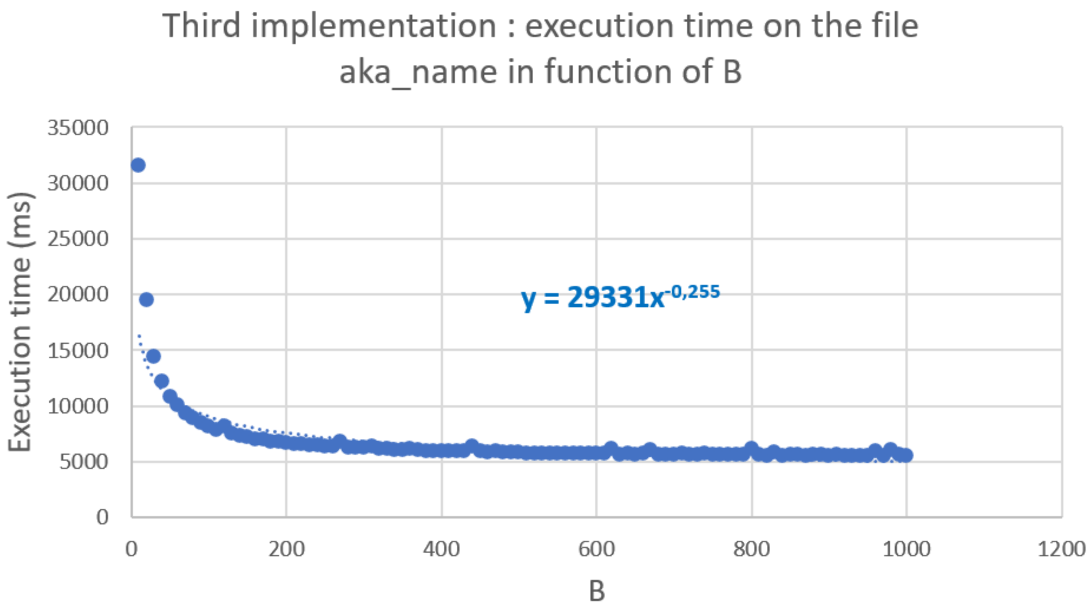


Figure 4: Execution time of the length function in Experiment 1 using InputStream3 on the aka_name file in function of B

The full results are displayed at Appendix A.3.

We do indeed see decreasing times when `B` increases, which confirms the established cost function. This method however presents some limitations, in particular from around B $= 500$, where the execution time seems to become constant no matter `B`. Concerning the execution times observed, they are greater than the ones observed on the same file with the second implementation using `fgets`.

5.4 Fourth implementation

For the fourth implementation, we used file mapping to read. For a detailed explanation of this method, please refer to section 4. As the file is read sequentially, the cost function for the fourth implementation is the following, where `G` is the allocation granularity (explained in section 4) :

$$C(\texttt{length4}) = \left\lceil \frac{N}{G} \right\rceil \tag{4}$$

`G` is equal to 65536 (2^{16}) on the machine the tests have been ran on.

Indeed, as a sequential reading is performed, `B` has no influence on the cost because whatever size of the file we map, we will never want to access information in pages that were fetched previously during the execution. The number of I/O operations performed is therefore directly related to the allocation granularity.

First, execution times of the `length` function using memory mapping on several files of different sizes are computed, in order to prove the pseudo-linear characteristic of the cost function established at equation 4 in function of `N`. These measurements are done with a constant value of `B` = 65536. The full results are displayed at Appendix A.4, and some of the main findings are reported in the table below :

File name	File length (N)	Execution time (t)	ceil(N/G)	ceil(N/G)/t
`aka_name`	73004383	5361,78	1140	0,21
`char_name`	215711567	16252,8	3292	0,20
`comp_cast_type`	45	0,25	5	3,99
`complete_cast`	2414495	317,221	37	0,12
`movie_link`	656584	85,5874	11	0,13

Table 5: Results of Experiment 1 using InputStream4, with B = 65536

From these measurements, we can expect $\frac{\lceil \frac{N}{G} \rceil}{t}$ to be a constant. As previously explained, the shorter files present dissimilarities compared to the other ones because of the important constant costs. We can therefore compute the average value of this ratio, as well as the standard deviation and the coefficient of variation in order to evaluate if it is relatively close to being constant. These measurements do not include the files that are considered too small which give non-interpretable data, the results are shown below :

$$\mu = 0,192$$
$$\sigma = 0,042$$
$$c_v = \frac{\sigma}{\mu} = 21,8\%$$

The coefficient of variation is higher than the one we got during the first implementation, but is still low enough in order to consider this data is pseudo-constant. We can also point out that this solution seems quite comparable with the third implementation in terms of execution time (when `B` is very large), but also less efficient than the second implementation using `fgets`. By plotting the results of this experiment, we obtain the following graph :

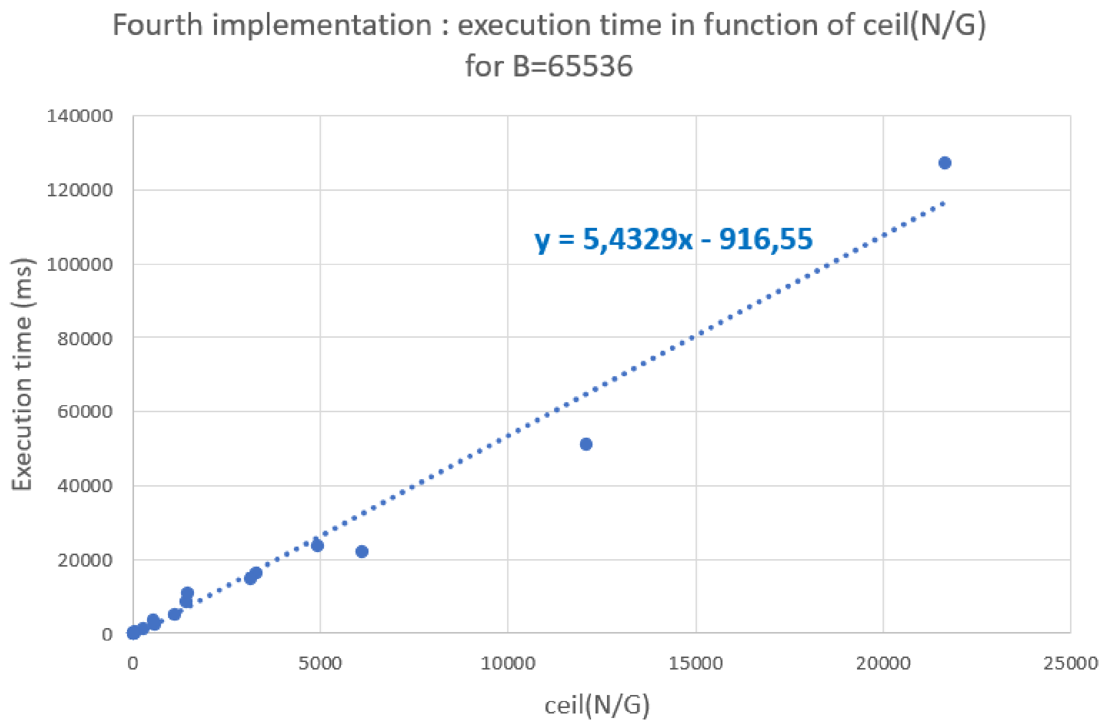


Figure 5: Execution time of the length function in Experiment 1 using InputStream4 for B=65536

We can therefore confirm the previously obtained results : the relation seems linear, but the fluctuations around the trendline seem higher than they were for the 2 first implementations.

In order to justify completely the cost function established at equation 4, we must prove that `B` does not influence the execution time. In order to do that, the function was applied to a single file `aka_title` with `B` values ranging from 65536 to 3276800 (50 times `G`). The results are briefly described in the table below, as well as in the following graph :

B	Execution time is ms (t)
131072	2369,42
524288	2318,54
1048576	2355,03
1703936	2366,81
2752512	2346,66

Table 6: Main results of Experiment 1 using InputStream4 on aka_title

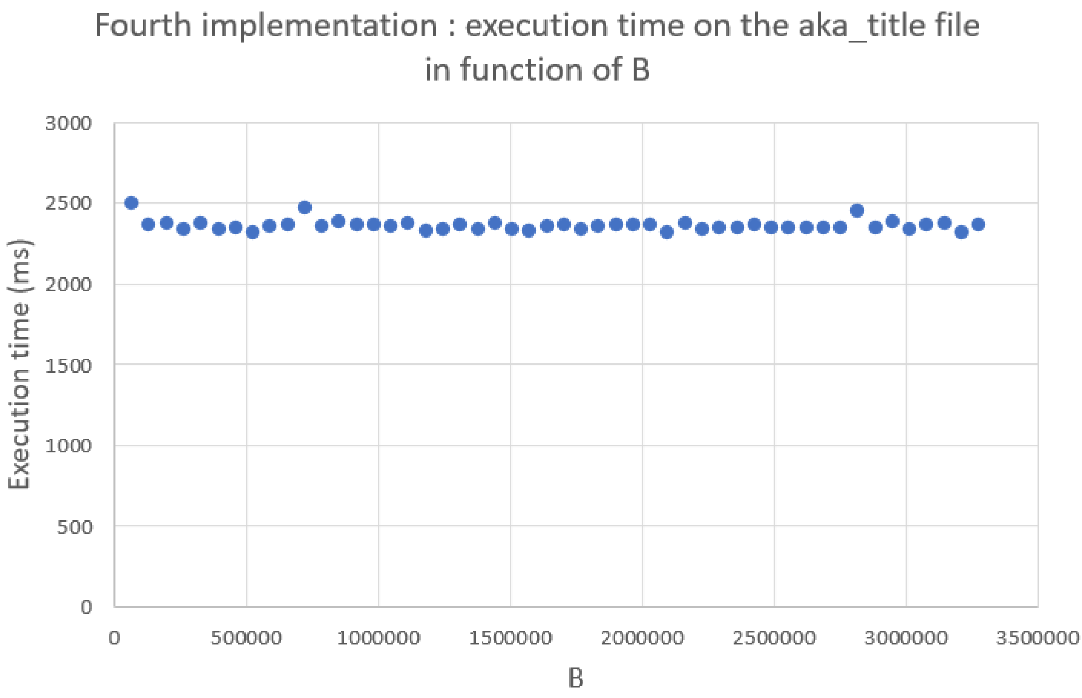


Figure 6: Execution time of the length function in Experiment 1 using InputStream4 on aka_title in function of B

We can indeed see that the execution time seems to be constant whatever `B`. This can be proven by calculating, as previously, the mean, standard deviation and coefficient of variation of this list of execution times :

$$\mu = 2362,71$$
$$\sigma = 33,22$$
$$c_v = \frac{\sigma}{\mu} = 1,4\%$$

This extremely small coefficient of variation indeed indicates that the execution time is independent of `B` as expected.

5.5 Comparison of the different implementations

By simply analyzing the data we have collected up to now, it seems obvious that the sequential reading of a file using the first implementation of the `readln` function is definitely not the best. This result is expected, as the number of I/O operations is equal to the number of characters in the file, which is technically the maximum number of operations to do in order to read a file (without any unnecessary operations).

By considering the established cost functions, one could think the third implementation would work best, when a huge value of `B` is used. This is however not the case, because the times tend to stagnate quite rapidly when `B` increases as it was shown on figure 4.

By plotting all the execution times for the 4 implementations in function of B for the `aka_name` file, we obtain the following results :

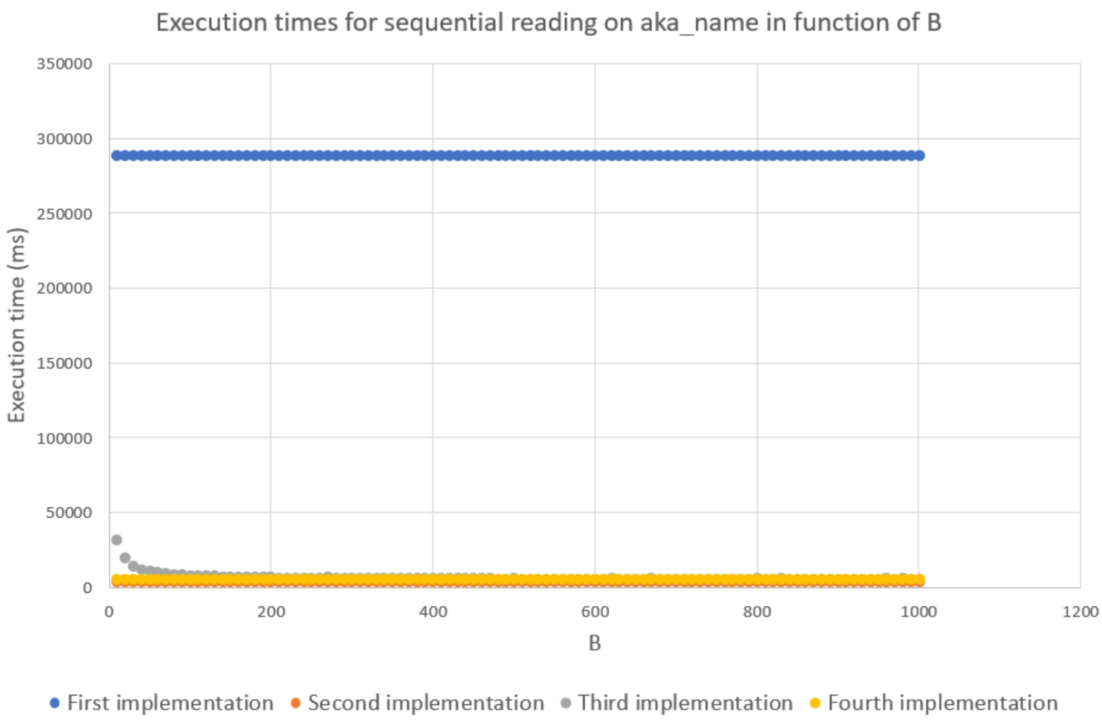


Figure 7: Execution time of the length function in Experiment 1 for all different implementations on aka_name

It is now clear that the first implementation is less efficient than the 3 others, so it is removed from the graph in order to allow clearer interpretation :

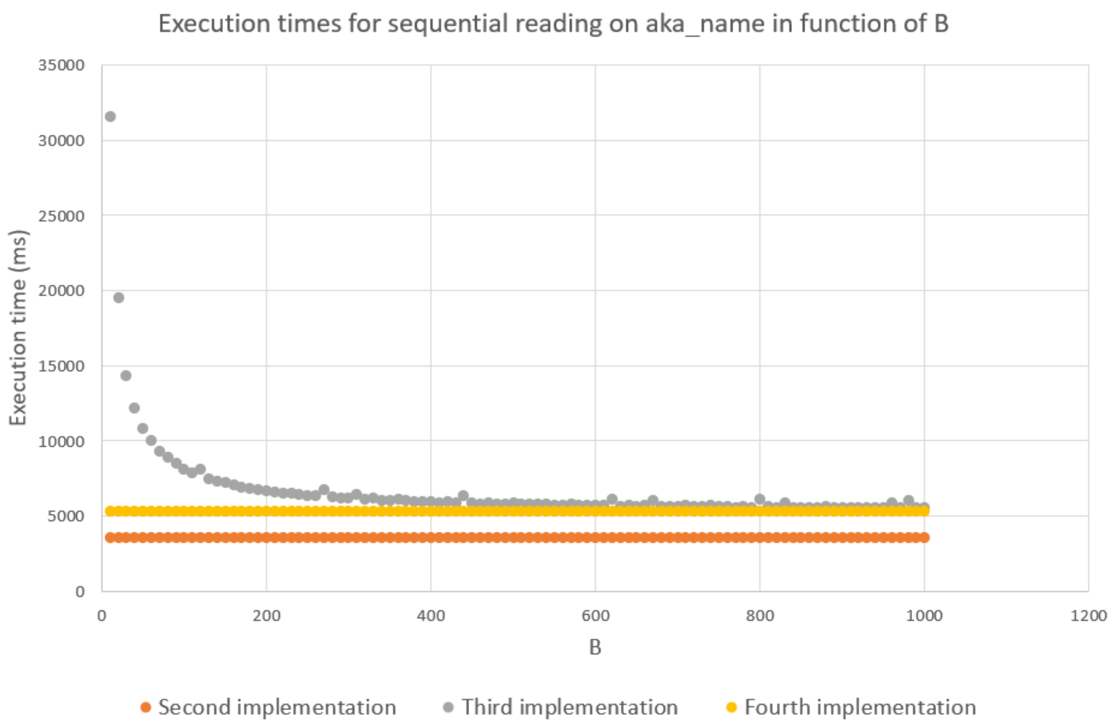


Figure 8: Execution time of the length function in Experiment 1 for the second, third and fourth implementations on aka_name

The second implementation, using the built-in `fgets` method, is therefore the most efficient to execute sequential reading on `aka_name`, which has around $7,3.10^7$ characters. A justification for this behavior would be that `fgets` uses the computer cache very efficiently. Indeed, this function is extremely optimized on order to adapt to the hardware, and the buffer sizes are chosen specially to allow the usage of cache, which is extremely rapid. The third implementation using the buffering is comparable to the memory mapping in terms of time when `B` is very large. The fourth implementation using the mapping is not particularly efficient in the case of sequential reading, because the advantage of mapping is mainly visible when we want to access data that has already been previously read.

We observe that time taken by the direct I/O's performed by the system call function is longer than the time taken by the buffered I/O's and the file mapping. Moreover, we can see that for this specific case (sequential reading) the buffered I/O's (managed by the OS) performed better than the file mapping. As explained above, we can deduce that the usage of cache is very well optimized for this kind of reading. These results must be validated on other files of various sizes. The same graph is computed for `complete_cast` (size 2.10^6) at figure 9 and for a smaller file `movie_link` (size 6.10^5) at figure 10.

Figure 9: Execution time of the length function in Experiment 1 for the second, third and fourth implementations on complete_cast

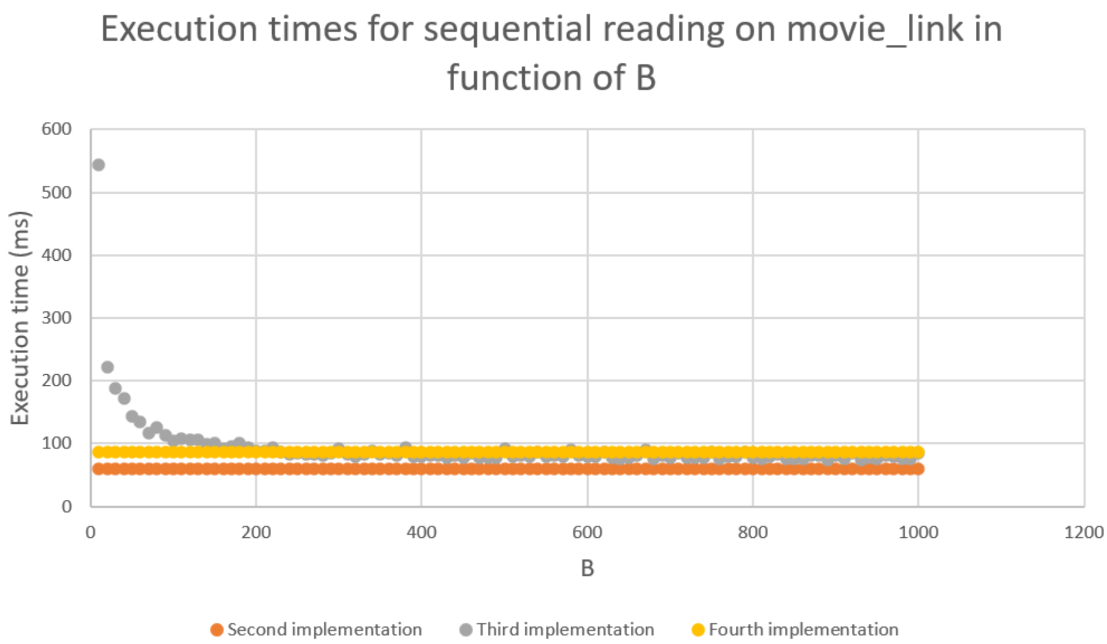


Figure 10: Execution time of the length function in Experiment 1 for the second, third and fourth implementations on movie_link

The results obtained are very similar to the previous ones, the second implementation always giving slightly better results, and the third and fourth implementation giving extremely similar results when `B` is very large. It is important to remember the extremely short files are neglected in this analysis, because the constant costs arising from the file openings and closings for example are non negligible and therefore don't really give useful data to study the I/O costs.

6 Experiment 2 : Random reading

The objective of this second experiment is to determine which of the 4 input stream implementations is more efficient for random reading, as well as finding the pros and cons of each implementation.

The code relative to this experiment is located in the `Experiment2` class, which implements a template function called `randjump`, as the `Experiment1` class, that can be used with any input stream class. This function calls the `readln` function of the appropriate input stream class, but in contrary to the first experiment, it takes as input a parameter `j` which indicates the number of times the following process will be repeated :

- compute a random number position between 0 and the size of the input file;
- seek to the computed position, i.e. set the cursor to the computed position;
- add the length of the string returned by `readln` to a counter `sum`.

Once the process above is done `j` times, the total of the counter `sum` is returned.

Before the actual measurements on the 4 different implementations, it was expected that random reading would bring some differences compared to the first experiment through the way it reads the files. Indeed, in the first experiment, the implementations are compared by observing their execution times in function of the size of the files, while the random reading might be independent to the size of the files due to the fact that it will not read entirely a file from the start to the end but by jumping to different positions randomly generated from a seed which will be a constant in order to to have the same scenarios for the 4 `randjump` implementations. Therefore, the execution times of a random reading will obviously be dependent on the number of iterations `j` given in parameter. It is expected that the second implementation of `InputStream` will be less efficient than previously since it uses an internal buffer, which will need to be refilled many times in the case of random readings. The same reasoning applies to the 3rd implementation, where the difference should be even more noticeable, because no cache optimization is used for this one. The fourth implementation should be way more efficient here, as the advantages of file mappings fully come in hand when randomly reading a file.

6.1 First implementation

As mentioned in the subsection 5.1, in the first implementation of the input stream (class `InputStream1`), the `readln` function reads characters one by one until reaching the end of the line and as a result, the number of necessary I/O operations to return a line is equal to the number of characters in each line. However, the `randjump` function makes j random jumps in the file, it means that `readln` will not be necessary called from the start of a line. As a result, the total number of I/O operations is inevitably equal to the product of `l` (length of the string returned by `readln`) and `j` (number of jumps in the file). Therefore, we can write the cost function of `randjump` when the `InputStream1` class is used, which we will call `randjump1` as follows :

$$C(\texttt{randjump1}) = l * j \tag{5}$$

The number of I/O therefore evolves in function of 2 parameters, `l` and `j`. As a result, two measurements are made, the first one is the execution time of the function on several files of different sizes by plotting the execution time in function of the file size with a fixed number of iterations `j` as a parameter. Then, the second measurement is the execution time of `randjump1` in function of the number of iterations with an arbitrary file.

The first measurement has been computed on multiple files of the dataset with a seed = 10 and a seed = 100 as shown in the tables 7 and 8 below. As the first experiment, the function `testFiles2` from the `Measurement` class is used by taking the average execution time on 10 total executions per file with a constant number of iterations `j` = 10.

File name	Execution time in ms (t)	Sum
`cast_info`	15.0057	273
`comp_cast_type`	1.99667	82
`keyword`	6.99008	141
`movie_info_idx`	3.92387	117

Table 7: The sums returned by the randjump function in Experiment 2 using InputStream1 with a constant j = 10 and a seed = 10

File name	Execution time in ms (t)	Sum
`cast_info`	12.6152	257
`comp_cast_type`	0.68175	62
`keyword`	6.21133	132
`movie_info_idx`	9.60508	136

Table 8: The sums returned by the randjump function in Experiment 2 using InputStream1 with a constant j = 10 and a seed = 100

The biggest file `cast_info`, in term of the total size, has the largest sum due to the longer size of the lines. However, it is neither proportional to the total size of a file nor the size of the lines because of the randomness, but it has an influence on the sum output. Furthermore, the randomness and the non-proportionality can be proven by noticing that with a seed equal to 10, the sum returned in the file `keyword` is bigger than the one with the file `movie_info_idx`. One would be tempted to say that it is due to the longer lines in `keyword` but with a constant seed equal to 100 it is the inverse, i.e. the sum returned with the file `movie_info_idx` is higher than the one returned with `keyword`. We can notice that the execution time evolves in regard to the sum.

In the second measurement, the `randjump1` function reads the file `cast_info` by varying the number of iterations `j`. As before, the average execution time on 10 total executions is taken. It is obvious that more the number of iterations `j` increases, and more the sum returned by `randjump1` will increase as well as the execution time. In order to confirm this statement, a graph plotting the execution time of the `randjump` function on the iterations applied on the file `cast_info` is computed :

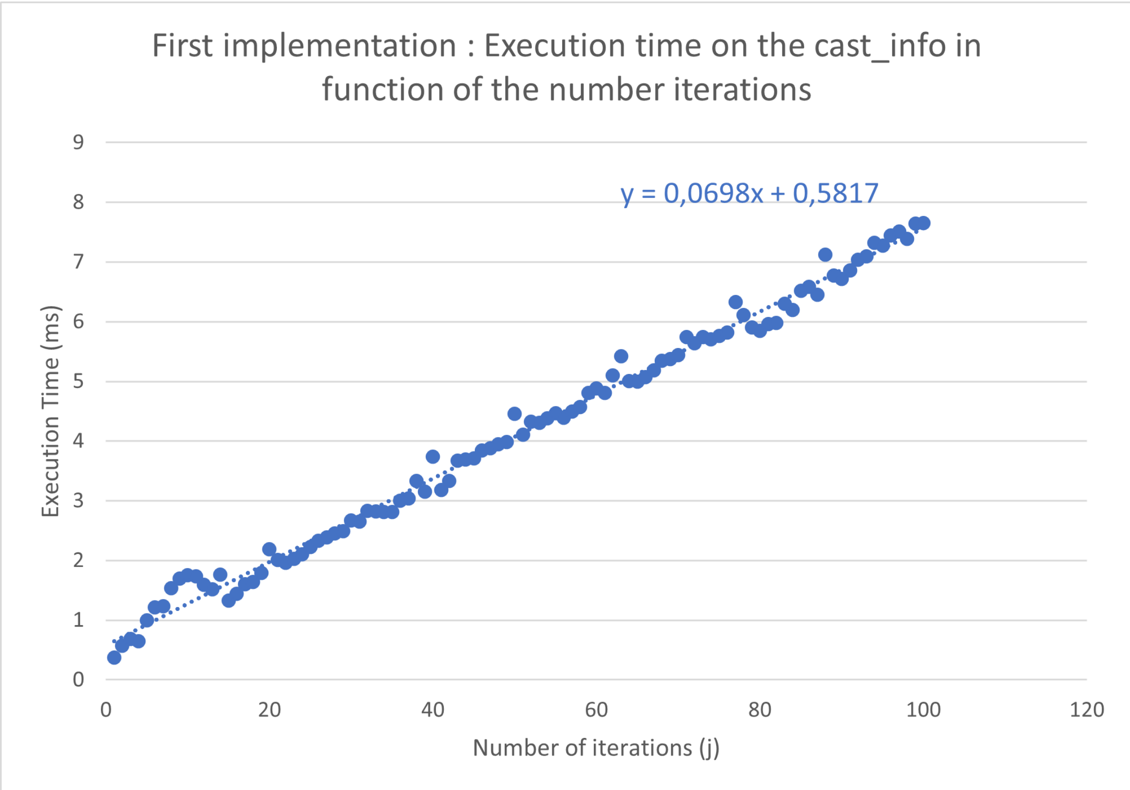


Figure 11: Execution time of the randjump function in Experiment 2 using InputStream1 on the cast_info file in function of the number of iterations

The graph seems to be arranged in a linear fashion despite some weak variations. Indeed, the trendline of this set of data is computed and an affine function equation is obtained as displayed on the graph.

The same linear behavior is noticed for larger numbers of iterations as shown in the table 30 of the appendix B.1.

6.2 Second implementation

Unlike the second implementation in the first experiment, using the second implementation in random reading is less efficient due to the fact that the size of the internal buffer (called S) of the `fgets` function does not have influence anymore. Indeed, the buffer will have to be refilled multiple times for each call of the `randjump` function.

In order to observe the `randjump` function using the `InputStream2` class which we will call `randjump2`, 2 measurements will be computed, as the previous subsection. The first measurement computes the execution time of `randjump2` in function of the total size of different files of the dataset (taking the average execution time on 10 total executions per file). Then the second one computes the execution time of `randjump2` by varying the number of iterations `j`.

As shown on the two tables 9 and 10 below, the exact same observation than the first implementation can be made by running `randjump2` on the same files as the first experiment to put in evidence the differences. It confirms that the total size of a file does not influence the sum returned by the `randjump2` function. One difference we can notice is that the execution time is faster than the `randjump1`.

File name	Execution time in ms (t)	Sum
`cast_info`	0.51673	273
`comp_cast_type`	0.37535	82
`keyword`	0.33984	141
`movie_info_idx`	0.37693	117

Table 9: The sums returned by the randjump function in Experiment 2 using InputStream2 with a constant j = 10 and a seed = 10

File name	Execution time in ms (t)	Sum
`cast_info`	9,97019	257
`comp_cast_type`	0,57382	62
`keyword`	5,39875	132
`movie_info_idx`	5,35759	136

Table 10: The sums returned by the randjump function in Experiment 2 using InputStream2 with a constant j = 10 and a seed = 100

In the second measurement, the `randjump2` function reads the file `cast_info` by varying the number of iterations `j`. As the first implementation, more the number of iterations `j` increases, and more the sum returned by `randjump2` will increase as well as the execution time. In order to confirm this statement, a graph plotting the execution time of the `randjump` function on the iterations applied on the file `cast_info` is computed :

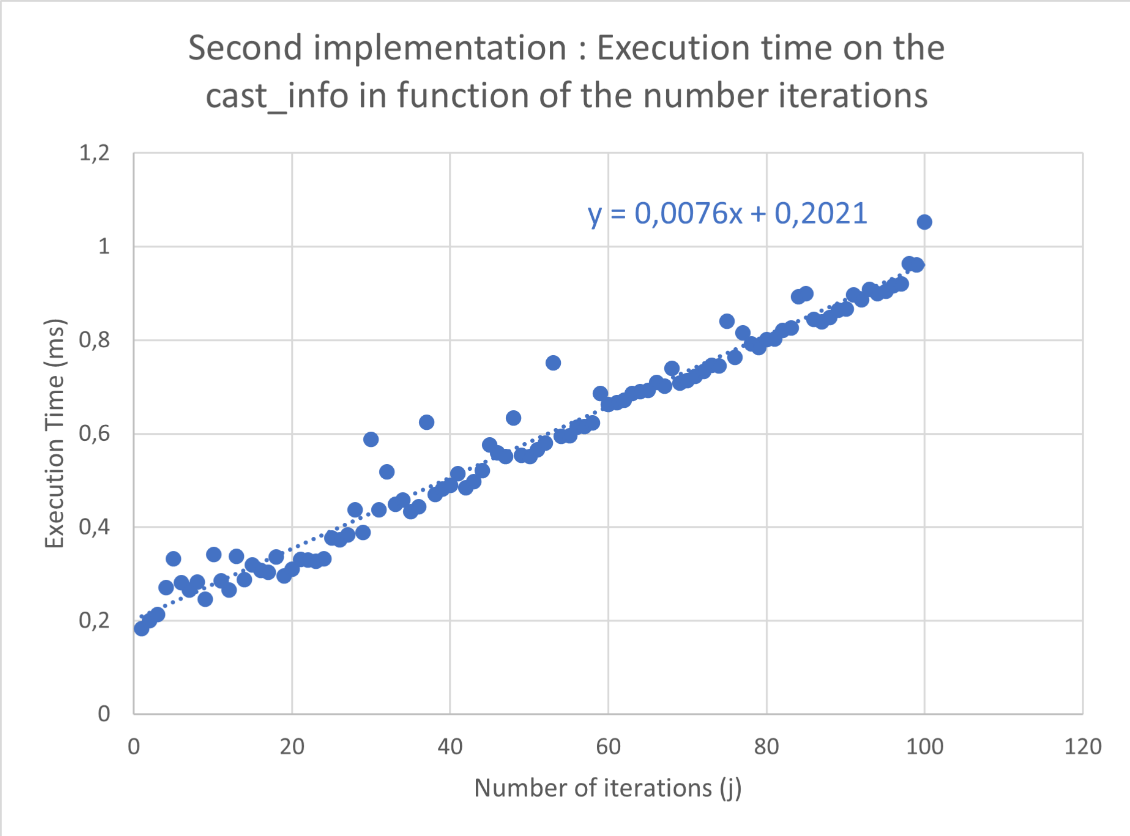


Figure 12: Execution time of the randjump function in Experiment 2 using InputStream2 on the cast_info file in function of the number of iterations

The graph seems linear with a few variations. The results for larger numbers of iterations are displayed in the appendix B.2 and feature a similar linear fashion. Notice that `randjump2` is faster than `randjump1` as shown on the figure 12.

6.3 Third implementation

In this third implementation of the `InputStream3` class, the `readln` function uses a buffer of size `B` in order to read a line from a file as already said in the first experiment. Every time the buffer is empty, it is reloaded with the `B` next characters of the file to be read. However, since it is random reading, random jumps occur within the file being read, for each jump, the buffer is refilled. Therefore, as for the second implementation, its performance will be reduced in comparison to the first experiment. Note that the `randjump` function using the `InputStream3` class will be called `randjump3` from now on to simplify.

Three measurements are made, the first one is the execution time of the function on several files of different sizes by plotting the execution time in function of the size file with a number fixed of iterations `j` as a parameter and a constant `B` equal to 100. Then the second measurement is the execution time of `randjump3` in function of the number of iterations with an arbitrary file (constant `B` = 100). And finally, the execution time is plotted in function of different values of `B`.

As the previous implementations, the first measurement has been computed on multiple files of the dataset with a seed = 10 and a seed = 100 to prove the non-influence of the total size of a file, as shown in the tables 11 and 12 as follows :

File name	Execution time in ms (t)	Sum
cast_info	0.49542	273
comp_cast_type	1.67674	82
keyword	3.16489	141
movie_info_idx	7.68705	117

Table 11: The sums returned by the randjump function in Experiment 2 using InputStream3 with a constant j = 10, B = 100 and a seed = 10

File name	Execution time in ms (t)	Sum
cast_info	0.32084	257
comp_cast_type	0.3616	62
keyword	0.55428	132
movie_info_idx	0.4215	136

Table 12: The sums returned by the randjump function in Experiment 2 using InputStream3 with a constant j = 10, B = 100 and a seed = 100

Like the two previous implementations, we can see that the sum returned from the `keyword` file is greater than the one returned from the `movie_info_idx` file for a seed equal to 10 but for a seed equal to 100, the inverse occurs. It shows therefore the sums returned are indeed not proportional to the total size of a file.

In the second measurement, the `randjump3` function reads the file `cast_info` by varying the number of iterations `j` with a fixed buffer size `B` equal to 100. As before, the average execution time on 10 total executions is taken. We will plot the execution time in function of the first 1 to 100 iterations :

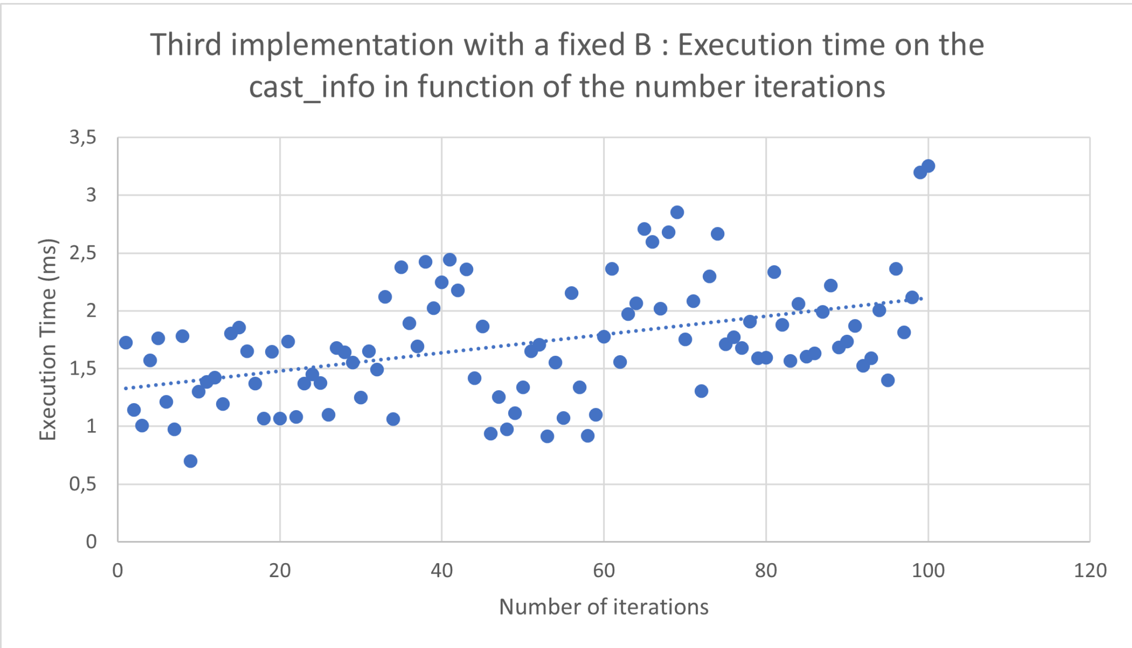


Figure 13: Execution time of the randjump function in Experiment 2 using InputStream3 on the cast_info file with a constant B = 100, in function of the number of iterations

The graph seems not linear and the execution times are very low and random, we suppose that these random variations are due to the low value of iterations of the `randjump3` which is executed too rapidly, and because of the constant costs such as the `seek`, `open` and `ftell` functions. In order to observe more accurately the `randjump3` function, we will plot on larger iterations as 1000 to 10000 as shown on the Figure 14 below :

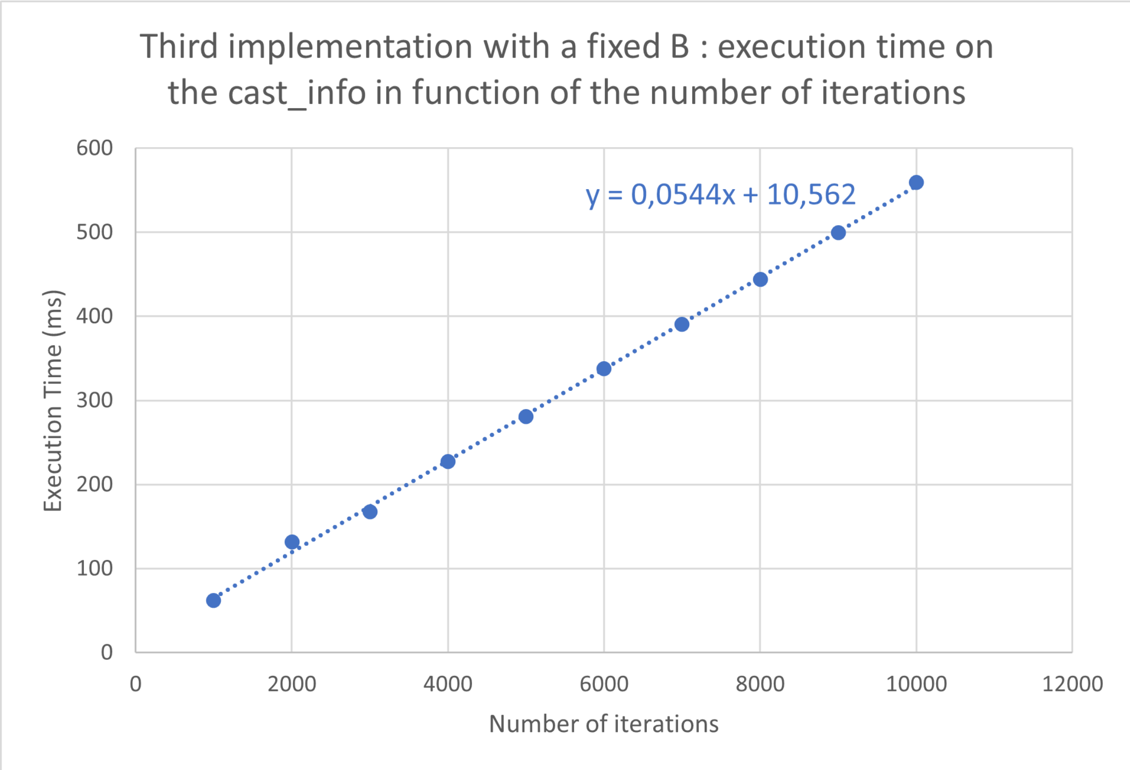


Figure 14: Execution time of the randjump function in Experiment 2 using InputStream3 on the cast_info file with a constant B = 100, in function of the number of iterations

Now we see the linear trendline approximates quite well the values plotted.

Next, it is necessary to show the impact of `B` on the execution time. In order to do so, a same file `cast_info` is read multiple times with a number of iterations `j` equal to 10000 with several different `B` values. The main findings of this experiment are presented below :

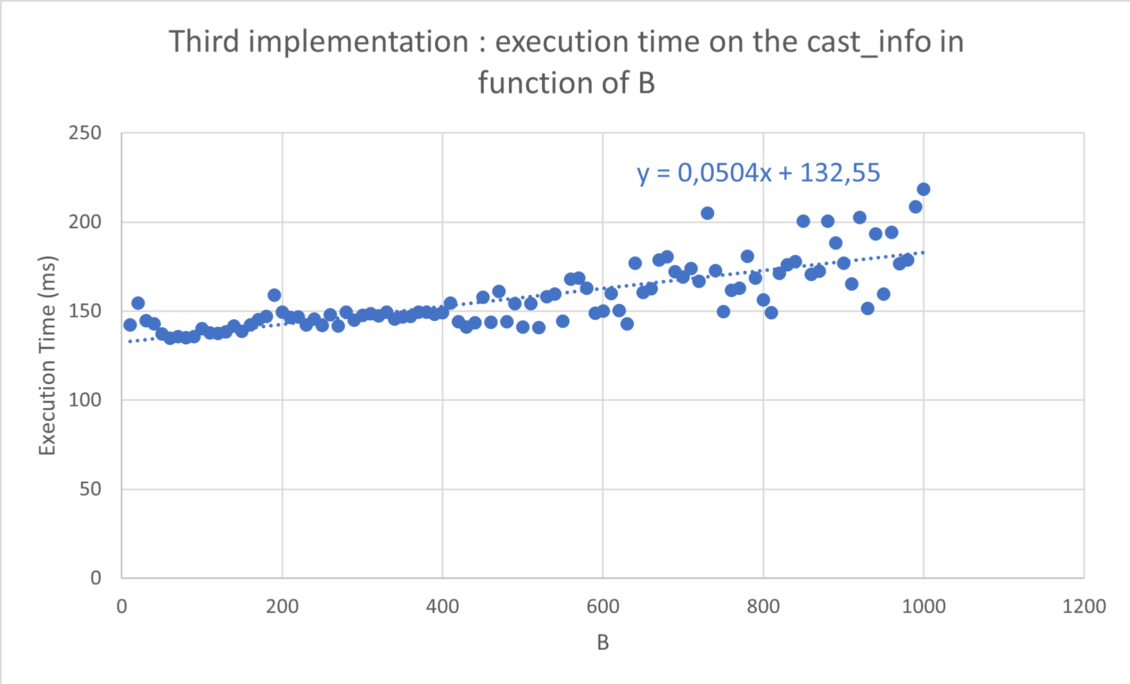


Figure 15: Execution time of the randjump function in Experiment 2 using InputStream3 on the cast_info file in function of B with j = 10000

The full results are displayed in the tables 33 and 34 of the Appendix B.3.

In contrary to the first experiment, the execution time in function of `B` grows linearly, i.e. when `B` gets bigger, the execution time tends to increase. As a result, when `B` is bigger, the cost of each I/O operation is bigger because there is more data to be fetched, thus, the time taken for the throughput will be greater. However, since in the `randjump3` function, the buffer of size `B` is reset on each call of the `seek` function, the previous buffer is never used. Therefore, it is useless to increase `B` to very large values in this implementation, and it leads to large execution times.

6.4 Fourth implementation

For this last implementation of random reading, 3 measurements will be computed as for the previous implementation. We will call the `randjump` function using the `InputStream4` class `randjump4`. In contrary to the sequential reading where `B` had no influence on the execution time, it will be important in this case because as the reading is random, we might need to access pages that were already fetched into memory previously. Therefore, the size `B` will have a direct influence on the execution time of the `randjump4`.

The first measurement is made once again to prove the non influence of the total size of a file on the sequential reading. This measurement is done on multiple files of the dataset with a constant number of iterations j and a fixed B = 65536 (allocation granularity) and we vary the seed between 10 and 100. The results are the same as the others implementations as shown on the 2 tables 13 and 14, i.e. `randjump4` does not depend on the total size of a file.

File name	Execution time in ms (t)	Sum
`cast_info`	4,06032	273
`comp_cast_type`	0,79132	82
`keyword`	6,59532	141
`movie_info_idx`	9,16791	117

Table 13: The sums returned by the randjump function in Experiment 2 using InputStream4 with a constant j = 10, B = 655336 and a seed = 10

File name	Execution time in ms (t)	Sum
`cast_info`	13,7111	257
`comp_cast_type`	0,26802	62
`keyword`	7,94977	132
`movie_info_idx`	13,1686	136

Table 14: The sums returned by the randjump function in Experiment 2 using InputStream4 with a constant j = 10, B = 655336 and a seed = 100

Then the second measurement will be computed by plotting the `randjump4` on the `cast_info` file with a constant B equal to the allocation granularity in function of the number of iterations as shown on the Figure 16 below :

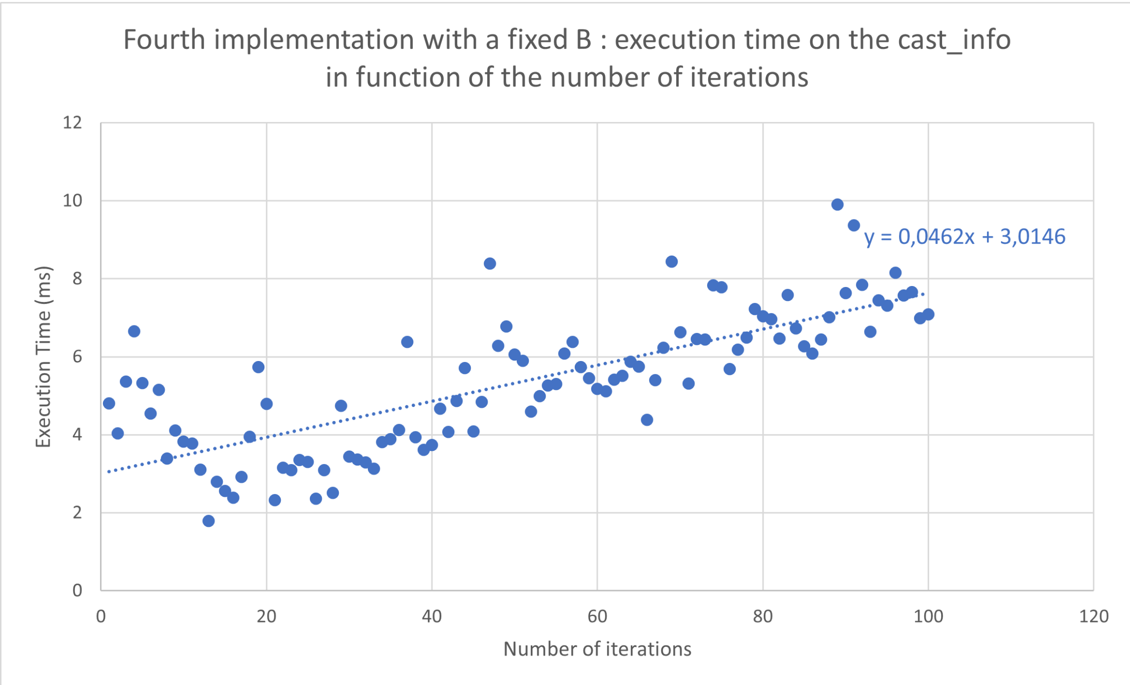


Figure 16: Execution time of the randjump function in Experiment 2 using InputStream4 on the cast_info file with a constant B = 65536, in function of the number of iterations

The graph features a linear behavior with some fluctuations as expected. Indeed, the execution time will be more random due to the small values of `j` and the constant costs of other function in the `randjump4` as for the third implementation. The execution on larger number of iterations presents a more linear behavior (see the table 35 in the Appendix B.4). Notice that the results are not so different in comparison to the third implementation in terms of execution time when `j` is very large, but less efficient than the second implementation.

Finally, in order to completely observe the `randjump` function using the `InputStream4` class, we will plot the execution time of this function on the `cast_info` file by varying the size B from 655360 (10*65536) to 65536000 (1000*65536). The results are shown in the following graph :

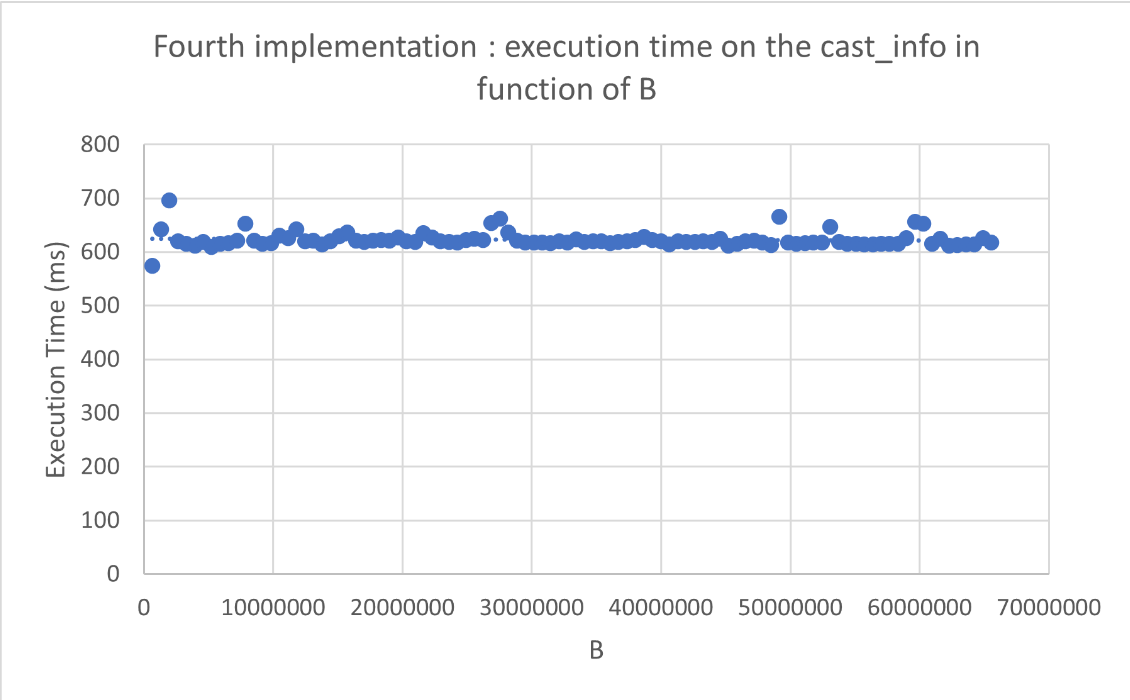


Figure 17: Execution time of the randjump function in Experiment 2 using InputStream4 on the cast_info file with a constant j = 10000, in function of B varying from 655360 to 65536000

The complete results are displayed in the table 36 in the Appendix B.4.

As `B` grows, we see that the execution time is relatively constant despite some variations. It is due to the fact that it is random. Indeed, with a different seed the graph will be a bit different dependently to the fact that whether the positions generated will be in the previously fetched pages or not. The execution time will therefore be different (when the position generated is not in the current mapped part, it will be unmapped, and we map the page in which the current position is).

However, when B is at bigger values, the execution time decreases drastically. We can explain it by the fact that when B is big, the part of the file being mapped will consequently grow. As a result, if the part being mapped gets bigger, the probability of a generated position being in the already mapped page is higher, i.e. less unmappings will occur and the execution time will be smaller. Indeed, unmapping parts of a file that have already been accessed will make the previous I/O unusable and so, due to the random reading, we could have to re do the same I/O later leading to a longer execution time. As explained previously, mapping the whole file will not bring it in RAM, only the parts of the file we want to access will be brought. Moreover, we have the possibility to "re use" previous I/O's when we need to access to an already accessed (and mapped) region.

One could imagine the best case is when `B` is equal to the total number of characters in the file. In order to prove this statement, a graph is built by plotting the execution time of the `randjump4` function with a constant number of iterations `j` equal to 10000, in function of B values going from 1310720000 (20000*65536) to 2129920000 (32500*65536) with a step of 655360 (10 times 65536) on the `cast_info` file, the following graph is obtained :

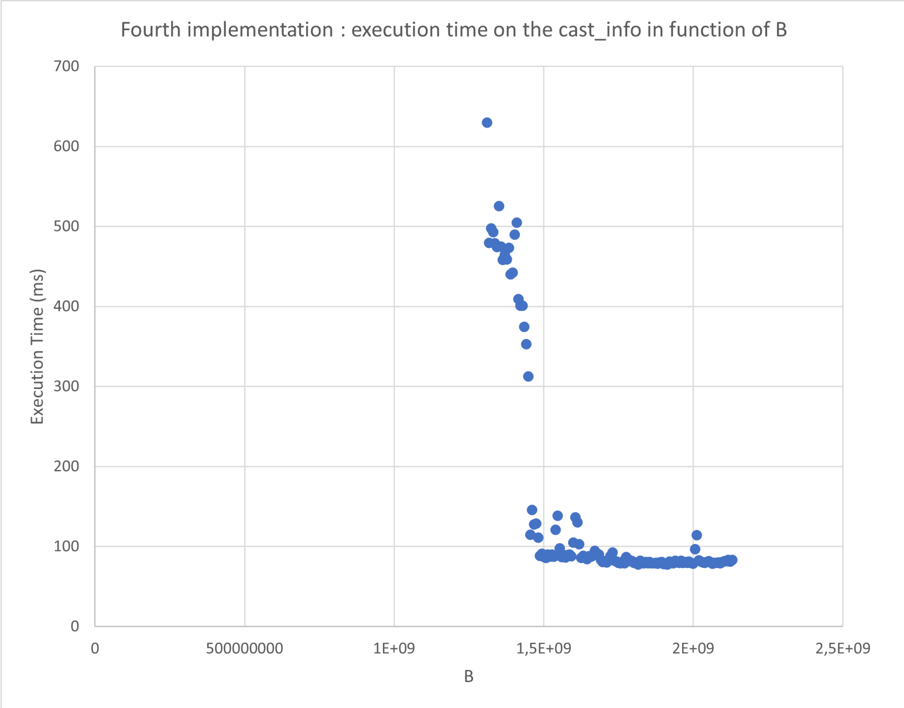


Figure 18: Execution time of the randjump function in Experiment 2 using InputStream4 on the cast_info file with a constant j = 10000, in function of B varying from 1310720000 to 2129920000

The results are displayed in the table 37 in the Appendix B.4.

The graph seems constant after a sharp drop around `B` = 1448345600 (22100 times 65536) where the execution time is equal to 312,412 ms. It is due to the fact that the total size of the `cast_info` file is equal to 1418137141, therefore, when `B` reaches approximately this value, the execution time drops. Thus, the execution time is drastically low in comparison with the previous results in the Figure 17. Concerning the execution times observed, they are smaller than the ones observed on the same file with the third implementation.

6.5 Comparison of the different implementations

After observing and collecting the data of the different implementations, it is obvious that the random reading of a file using the first implementation of `InputStream` is definitely the least efficient as for the first experiment.

By considering only the execution time of the 4 implementations with small `B` values, the third implementation is the most efficient. Indeed, the third implementation with a small size `B` will be the fastest among the 4 implementations. A comparative table is built below to justify this statement. The results are taken from the measurements on the `cast_info` file with a number of iterations `j` equal to 10000 and a seed equal 10, note that the size of the buffer `B` is equal to 10 for the third implementation and equal to 65536 for the fourth implementation.

Implementation	Execution time in ms (t)
`implementation 1`	797.049
`implementation 2`	248.819
`implementation 3`	142.298
`implementation 4`	544.931

Table 15: Execution time measured from randjump function using the 4 InputStream on the cast_info file, with j = 10000, B = 10 (third implementation) and B = 65536 (fourth implementation) and a seed = 10

The fastest implementation is indeed the third one as shown on the table 15.

However, by taking into account bigger size `B`, the result is different. Indeed, the fourth implementation is the most efficient when `B` is very big, as shown on the table 18 we can see when `j` is equal to 10000, the execution time is extremely faster in comparison to the others implementations especially the third one. To illustrate the comparison, the table 16 is built. The execution times are taken from the measurements on the `cast_info` file of the 4 implementations with the same parameters than the previous table except for the size of the buffer `B` of the fourth implementation which will be equal to 1500774400 (22900 times 65536).

Implementation	Execution time in ms (t)
`implementation 1`	797.049
`implementation 2`	248.819
`implementation 3`	142.298
`implementation 4`	86.8736

Table 16: Execution time measured from randjump function using the 4 InputStream on the cast_info file, with j = 10000, B = 10 (third implementation) and B = 1500774400 (fourth implementation) and a seed = 10

The fourth implementation is nearly 2 times faster than the third implementation. This result was expected, as the mapping is particular efficient for random reading.

7 Experiment 3 : Combined reading and writing

In this experiment, we are combining `k` different files into one unique file. To do so, we read one line of each input file and write them in an output file in a round robin fashion. The goal is to determine which pair of input stream and output stream works the best. For the input streams, we only considered the second and fourth implementation (respectively the best implementation identified in experiment 1 and 2).

All the analysis in the experiment 1 of the different implementation of `InpuStream` can be transposed for the four implementations of `OutputStream`.

Before starting this experiment, we thought that `InputStream2` was going to be the fastest way to read the files because of the sequential reading. Indeed, even if we read the files in a round robin fashion, each file is still read sequentially. Therefore, the mapping would not bring any added value.

For the fastest way to write in the files, we supposed that it was going to be `OutputStream4` or `OutputStream2` based on the study made in the first experiment. Theoretically, the third implementation of `OutputStream` with a enormous `B` would have been the fastest one, but as we saw, the increase of speed with `B` will normally stop at a certain point. The first implementation of `OutputStream` will for sure be the slowest (based on what we studied in the first experiment). The advantage of the fourth implementation is that an I/O occurs only when we access a part of the file we want to write, as explained in section 4. Then, we modify in the RAM the buffer by copying our different lines without doing any I/O's. All the I/O's occurs only when unmapping or after a long time to update the content of the disk. It allows to have a faster way to write but, because we do less I/O's, if a crash occurs, it is more likely that we lose what we have modified.

Based on what we saw, the second implementation of `OutputStream` can be the fastest one, but the cache can only store a limited amount of bytes before flushing everything to the disk. The advantage of the cache when writing is less important than when reading as the OS cannot bring data while working on other process as in experiment 1 when reading.

However, if the cache can store and wait long enough to limit the number of I/O's, it can be nearly as fast as the fourth `OutputStream` implementation. To wrap up everything, we thought that because the cache does not add as much value as when reading, the fourth implementation of `OutputStream` was going to be the fastest way to write in the file.

Here, implementations that have a buffer are even more memory consuming (RAM memory) when using a big `k`. Indeed, we do not read a whole file and then read another one, instead we keep the buffer of every file in RAM and proceed to the reading in a round robin fashion. Because the mapping works with multiples of 16 pages, it can also become quickly memory consuming.

The first implementation of output stream is obviously less memory (RAM) consuming but also a lot slower because we do an I/O for each line read.

Different sets of files were created in order to measure the times of execution of this experiment, each letter corresponding to a set as displayed below. The further in the alphabetical order the letter is, the bigger sum of the file sizes is.

A = link_type,kind_type

B = link_type,kind_type,info_type,company_type,comp_cast_type,role_type,movie_link

C = complete_cast,keyword

D = keyword,company_name,movie_info_idx,movie_keyword,movie_companies,aka_title, aka_name

E = complete_cast,keyword,movie_link,company_name,movie_info_idx,link_type,kind_type, info_type,company_type,comp_cast_type,role_type,movie_keyword,movie_companies, aka_title,aka_name

F = movie_info,person_info

G = movie_info,person_info,name,cast_info,title,movie_keyword,char_name

H = title,link_type,kind_type,info_type,company_type,comp_cast_type,role_type,complete_cast, keyword,movie_link,company_name,movie_info_idx,movie_keyword,movie_companies,

aka_title,aka_name,movie_info,person_info,
name,cast_info,char_name

The parameter `k` correspond the number of files.

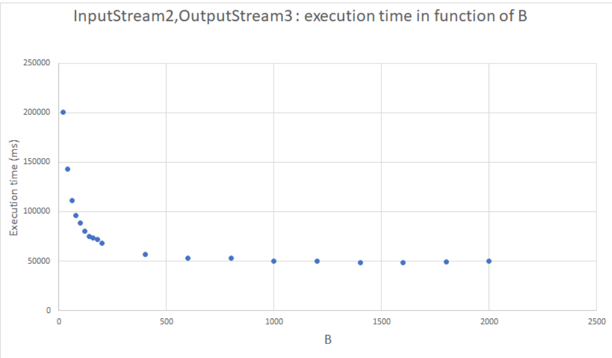
- k=2 for A,C and F.
- k=7 for B,D and G.
- k=15 for E.
- k=21 for H.

7.1 Variation of B

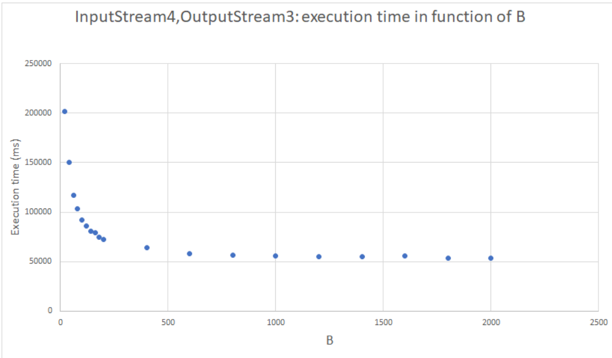
As we can observe in the graphs below, we have result similar to the ones we got in experiment 1 for `InputStream3` and `InputStream4`. The analysis done in experiment 1 for `InputStream3` and `InputStream4` can be transposed correspondingly for `OutputStream3` and `OutputStream4`.
The execution time drops as we increase `B` for `OutputStream3` for both pairs `Inputstream2/OutputStream3` and `InputStream4/OuputStream3` until a stagnation at around `B`=1000. However, the execution time stays constant as we increase `B` for `OutputStream4` for both pairs `InputStream2/OutputStream4` and `InputStream4/OutputStream4`.

Note that the use of `InputStream2` or `InputStream4` does not influence the shape of the graph when changing `B` of the `OutputStream` peer (as expected).

In the next analysis, we will consider the execution time with `B`=2000 for every pair containing `OutputStream3` and the mean execution time of all the `B` values for every pair containing `OutputStream4`.

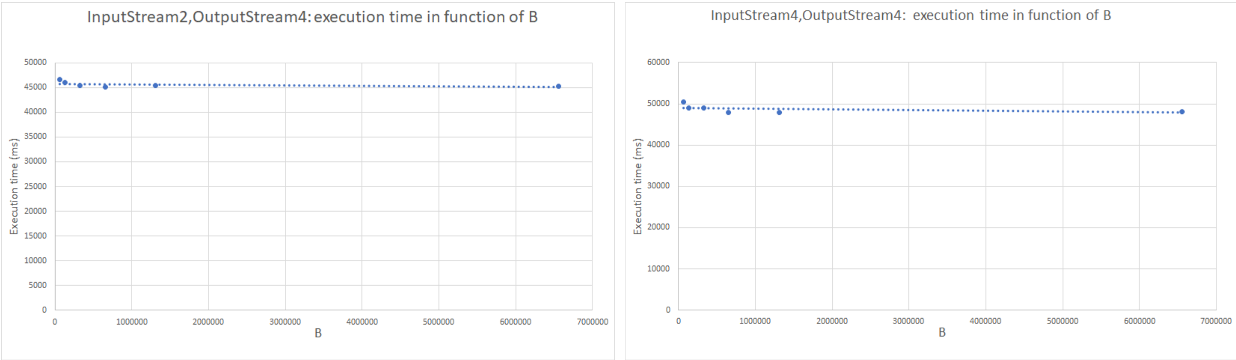


(a) InputStream2,OutputStream3



(b) InputStream4,OutputStream3

Figure 19: Execution time in function of B on the set of files E



(a) InputStream2,OutputStream4 (b) InputStream4,OutputStream4

Figure 20: Execution time in function of B on the set of files E

7.2 Comparison of the pairs of InputStream/OutputStream

The execution times of this experiment on the different pairs of input and output streams, and applied to the different batches of files presented above are displayed :

Implementation\Files	A	B	C	D	E	F	G	H
`InputStream2,OutputStream1`	13.458	6114.02						
`InputStream4,OutputStream1`	13.909	6314.68						
`InputStream2,OutputStream2`	2.5805	116.485	1105.13	68432	50550.9	140699	444650	469675
`InputStream4,OutputStream2`	2.7017	137.185	1215.01	89475.1	57618.2	144932	697026	706880
`InputStream2,OutputStream3`	1.8721	113.727	1046.87	47096.9	49823.3	127123	427604	436578
`InputStream4,OutputStream3`	2.0466	131.698	1198.47	54977	53551.9	136303	614177	623149
`InputStream2,OutputStream4`	2.6061	113.4295	1049.21	42646.2	45608.6	110564.8	390415.2	4064633
`InputStream4,OutputStream4`	2.9045	117.9432	1144.02	48628.8	48786.55	119011	572653.7	579699.2

Table 17: Time (in ms) in function of the files and the implementation

Remainder : for the third and fourth implementation of output streams, we respectively used in the table `B` = 2000 and the average time on all the different `B` values that were tested (see appendix C).

Every empty cell corresponds to measurements that were not done because of the excessive amount of time it would have taken.

By analysing this table, we can see that the second implementation of input streams is always the fastest way to read. We can also see that for every set of files, the pair `InputStream2/OutputStream4` is always the fastest except for the column "A". Indeed, for very small files, the third implementation of `OutputStream` is the fastest way to write in the destination file. In fact this third implementation of `OutpuStream` has almost the same execution time as the `OutputStream4` for columns B and C. We can deduce that by using a buffer size chosen by ourselves for `OutputStream3`, we can have similar or better execution time than `OutputStream4` which is not very optimal for small file sizes. However, when we work on very large files, `OutputStream4` becomes much more interesting.
We can also see that `OutputStream2` does not perform as well as we expected, probably due to the fact that the cache can't optimize the writing as well as it does for the reading. As expected, `OutputStream1` is the slowest.
Also as expected, the execution time grows for every pairs of `InputStream` and `OutputStream` as the total size of the set grows (either by using a bigger `k` or by using bigger files).

8 Multi-way merge

In order to sort very large relations, it is possible to use the Two-Phase Multiway Merge-Sort algorithm. Considering the memory contains M buffers, the algorithm is separated into 2 phases :

- The file is divided into portions of M bytes, each portion of M bytes is written to a separate file. Notice that the lines in the file can't be separated, therefore lines are added to the file until its size exceeds M. These $\left\lceil \frac{N}{M} \right\rceil$ files are sorted on the column k.
- d files out of the $\left\lceil \frac{N}{M} \right\rceil$ files are merged together in sorted order (following the merge-sort algorithm) with the help of a priority queue. This process is repeated until there is only one big sorted file left in the queue.

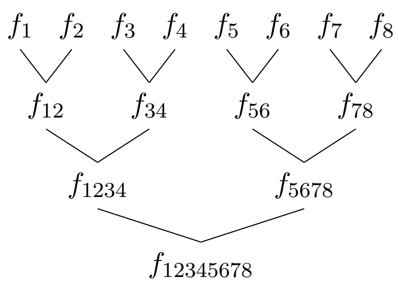
In order to evaluate the cost of this algorithm, we need to take into consideration the file size N, as well as M and d. The reading and writing functions used will be the optimal ones found in Experiment 3, which means the input streams used will be from the class `InputStream2` and the output streams will be from `OutputStream4`.

First of all, during the first phase, the whole file is read and written once. The cost of this phase is therefore the cost of reading N characters and writing N characters. This is therefore :

$$\left\lceil \frac{N}{S} \right\rceil + \left\lceil \frac{N}{G} \right\rceil$$

Indeed, the cost of reading a file sequentially using `InputStream2` was detailed in the Experiment 1, where S is the size of the internal buffer used by `fgets`, and the cost of writing N elements sequentially to a file using `OutputStream4` is the same as reading sequentially this file.

Next, these $\left\lceil \frac{N}{M} \right\rceil$ files are merged together. In order to make the cost function simpler, we will provide an upper bound to the cost function, using the *big O* notation. Indeed, the problem encountered when figuring out the cost function is that if $\left\lceil \frac{N}{M} \right\rceil \; mod \; d \neq 0$, then the last files are merged with files that have already been merged previously (and are therefore longer). The cost function becomes in that case quite complicated to evaluate, especially that even when $\left\lceil \frac{N}{M} \right\rceil \; mod \; d = 0$ it is very plausible that after a round or a few rounds we may end up with a number of files that is not a multiple of d. The *worst case scenario* is therefore when $\left\lceil \frac{N}{M} \right\rceil \; mod \; d^r = 0$, where r is the number of recursive *rounds* of merging that will be necessary. This is the case we will use in order to establish the cost function. Notice that, by focusing on this scenario, we can visualize the merging procedure as a tree. For clarity, let us represent the case where $\left\lceil \frac{N}{M} \right\rceil = 8$ and d $= 2$:



In this tree, $f_{i...j}$ corresponds to the file obtained by merging all files ranging from i to j. We can see that in total, the whole file has to be read and written 3 times in this example case. This helps us understand how to derive a function corresponding to the number of *rounds* r that are necessary to run the algorithm. Since the final result is always a single file, we can write :

$$\frac{\left\lceil \frac{N}{M} \right\rceil}{d^r} = 1 \tag{6}$$

from which we can establish :

$$r = \frac{\log \left\lceil \frac{N}{M} \right\rceil}{\log d} \tag{7}$$

which gives us the number of iterations of merging. Therefore, we can deduce the cost function of the multi-way merge sort algorithm we have implemented :

$$C(merge) = O\left(\frac{\log \left\lceil \frac{N}{M} \right\rceil}{\log d}\left(\left\lceil \frac{N}{S} \right\rceil + \left\lceil \frac{N}{G} \right\rceil\right)\right) \tag{8}$$

In order to verify the cost function, we will study independently the effects of variations of the parameters `M`, `N` and `d`.

First, the execution time of the `extsort` function is called on the `company_name` file (`N` = 17802021), with a constant `d` = 20 and `M` varying between 10000 and 300000. Looking at the cost function, we would expect the execution time to decrease when `M` increases, following an inverse logarithmic trend, because of the factor $\log \left\lceil \frac{N}{M} \right\rceil$ which is the only factor containing `M` in the cost formula. The main findings are showed in the table below (full results at appendix D) :

M	Execution time (t)	t/log(ceil(N/M))
40000	145973	55098
70000	134600	55931
110000	129970	58822
190000	113753	57651

Table 18: Execution time of the multi-way merge sort algorithm on company_name in function of M

The product $\frac{t}{\log \left\lceil \frac{N}{M} \right\rceil}$ is computed at it should be a constant. By computing the average value as well as the standard deviation and coefficient of variation of this ratio for all computed values, we obtain :

$$\mu = 57538,96$$
$$\sigma = 1880,17$$
$$c_v = \frac{\sigma}{\mu} = 3,27\%$$

The coefficient of variation being extremely small, we can consider that our assumptions were correct. In order to further justify the dependency on `M` of the cost function, the results are plotted, with a logarithmic scale being used on the horizontal axis :

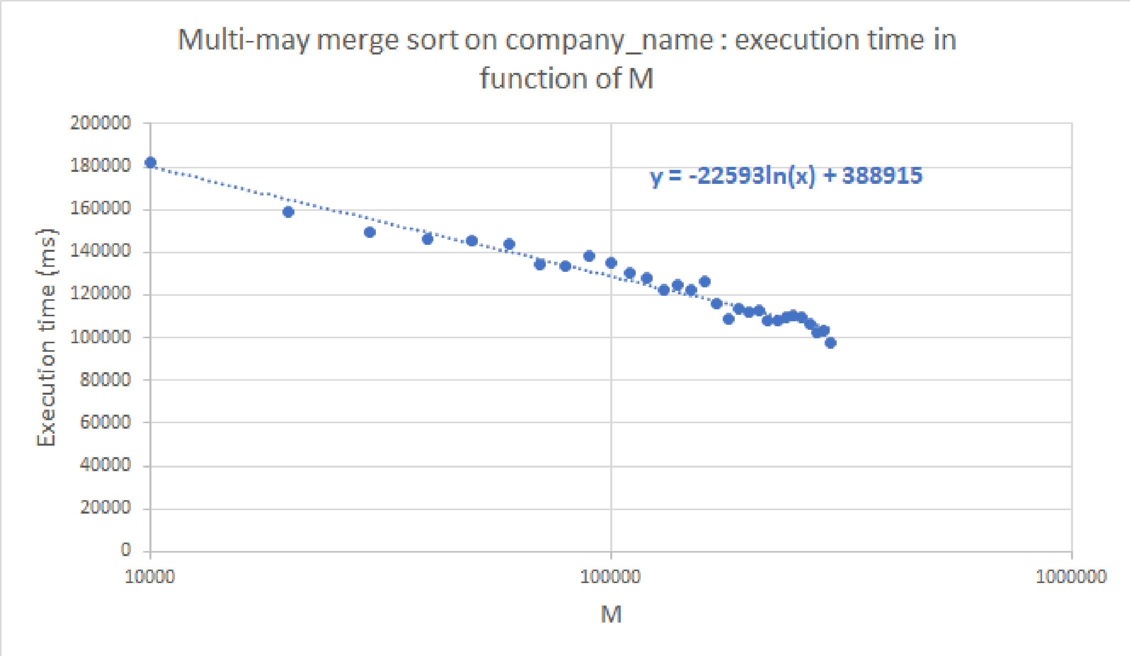


Figure 21: Execution time of the extsort function on the company_name file in function of M, with constant values of d and N

We observe that the logarithmic trendline approximates the results quite accurately, and that its equation actually corresponds to an inverse logarithmic function. Therefore, we can consider that the execution time depends on `M` in an inverse logarithmic fashion as expected.

Next, the influence of `N` on the execution time has to be shown. Since this parameter appears in both factors of the cost function established at equation 8, we expect a $N.log(N)$ evolution regarding the execution time when `M` and `d` are constant. Therefore, the multi-way merge sort algorithm was applied to most the files on the dataset of different lengths (the excessively long files were ignored due to the long execution times), while maintaining the other parameters constant. The very short files were also ignored as the `M` and `d` constant values (respectively 200000 and 200) did not really make sense for extremely short files. The main findings are shown below, and full results are available at appendix D :

File name	File length (N)	Execution time (t)	N.log(N)	N.log(ceil(N/M))/t
`aka_name`	73004383	694323	574059004	269,54
`keyword`	3791536	17931,8	24943815	270,4
`movie_keyword`	93799307	1010650	747786800	247,91
`title`	203877939	2046890	1694097278	299,67

Table 19: Execution time of the multi-way merge sort algorithm on several files in function of N, with constant values of M and d

The ratio $\frac{N.\log\left\lceil\frac{N}{M}\right\rceil}{t}$ should behave more or less as a constant if we follow the cost function, although this is not entirely true as the `N` can not exactly be factored out of the second factor of equation 8. It is nevertheless a decent indicator to confirm that the established cost function is correct. The same statistical features as previously are computed :

$$\mu = 237,58$$
$$\sigma = 63,88$$
$$c_v = \frac{\sigma}{\mu} = 26,8\%$$

As expected, the coefficient of variation is higher than previously as this ratio does not exactly reflect the cost function, but it is still sufficiently low to confirm our hypothesis on the influence of `N` on the execution time. A graph plotting these execution times in function of $N.log(N)$ is presented :

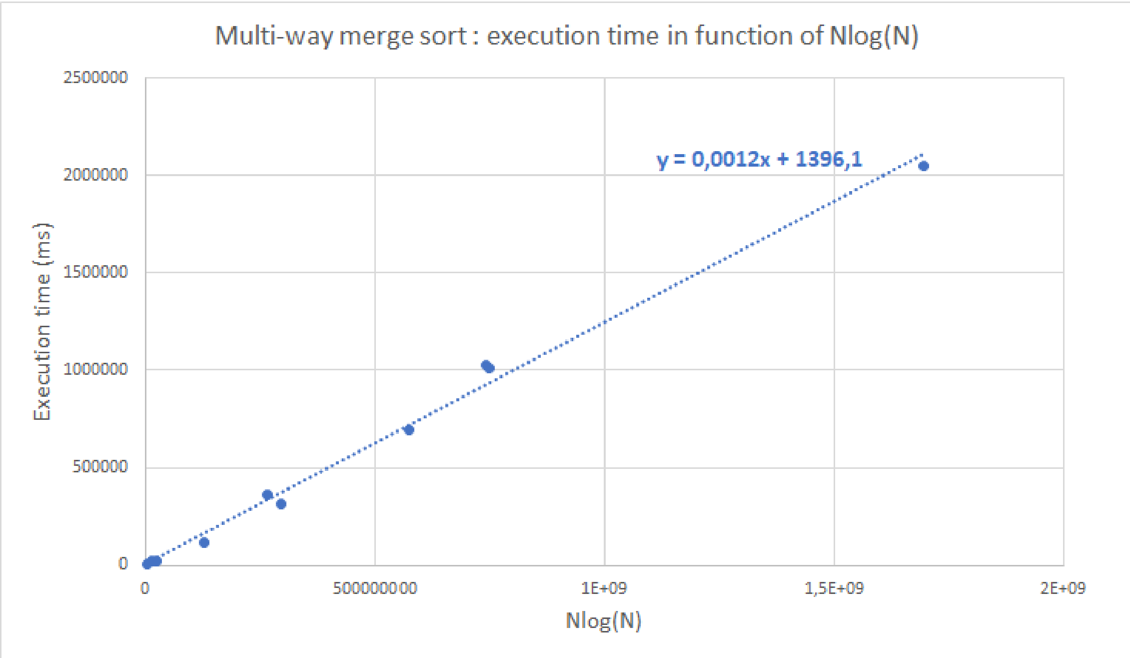


Figure 22: Execution time of the extsort function on several files in function of N.log(N), with constant values of d and M

The linear trendline seems to approximate the results well, and the intercept of this trendline is minimal (1396,1), therefore our assumption seems correct : the execution time of the multi-way merge sort evolves in function of $N.log(N)$.

Finally, it is necessary to prove that the execution time of the multi-way merge sort algorithm depends on `d`, in an inverse logarithm fashion (see cost function at equation 8). Therefore, the execution times have been measured for `d` varying between 2 and 20 for the `company_name` file with `M` = 100000. The main results are shown below (full results at appendix D) :

d	Execution time (t)
2	167956
5	139705
10	116257
14	111429

Table 20: Execution time of the multi-way merge sort algorithm on company_name in function of M

By plotting the results of this execution in function of `d` on a graph with a logarithmic horizontal scale, we obtain :

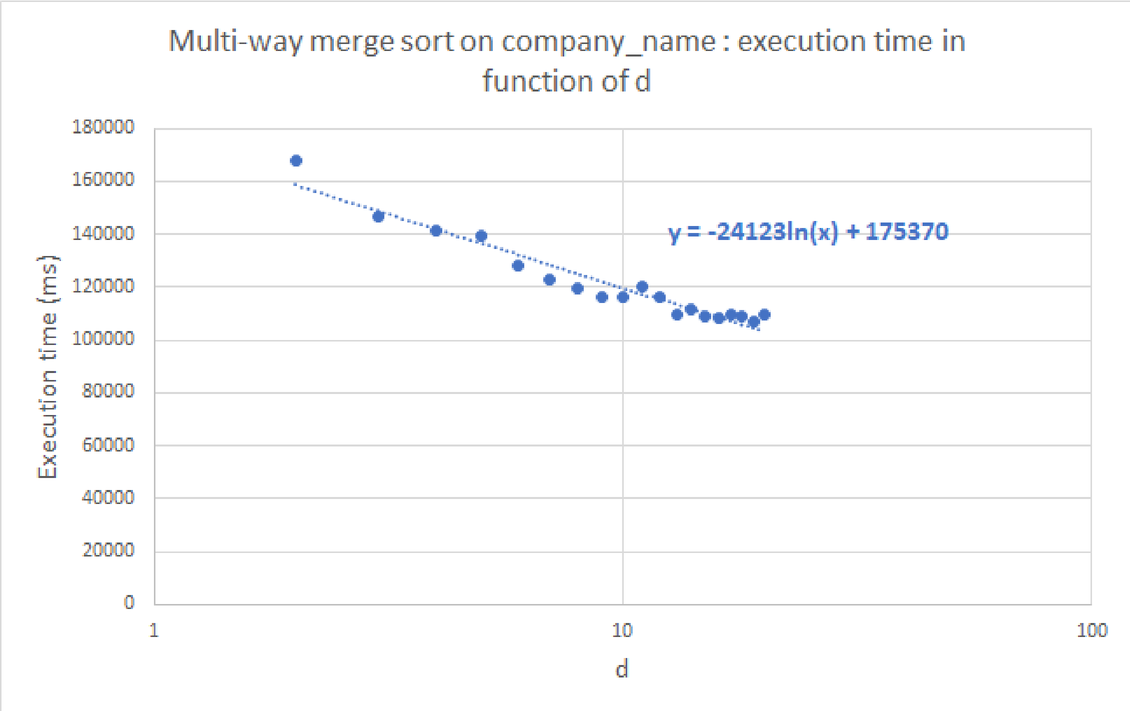


Figure 23: Execution time of the extsort function on several files in function of d, with constant values of N and M

We observe that the trendline seems to approximate the results pretty well, and that it indeed corresponds to an inverse logarithmic function. Therefore, we can assume that the multi-way merge sort execution time actually depends on `d` in an inverse logarithmic fashion as expected.

Therefore, we can conclude that the cost function established at equation 8 seems verified by the experimental results.

9 Conclusion

In this project, 4 different implementations for reading from and writing to a secondary memory have been done. By comparing their performance through experiments. After observing the results obtained from the first and second experiments, the two most efficient implementations for the reading of files were determined to be the second and the fourth implementations. This determination allowed to proceed with the next experiment which combined the reading implementations with the writing ones to establish the best pair of input stream implementation and output stream implementation. As a result, the second implementation of the input stream is the fastest way to read while for the output stream, the fourth implementation is more efficient with very large files and the third implementation is more interesting with small files. Finally, in order to fulfill the objectives of this project, an external-memory Two-Phase Multiway merge-sort algorithm has been implemented. Through different comparisons with plots, this merge-sort algorithm is confirmed to be dependent to the 3 parameters n, d and M. Through this project the group has put into practice different concepts seen in the theoretical courses which has contributed a lot to our understanding.

A Appendix 1 : Experiment 1

A.1 First implementation

File	Average time (ms)	Length	(Length/Time)
aka_name	288342,2	73004383	253,1866061
aka_title	155003	38858446	250,6947995
cast_info	6804791	1418137141	208,4027476
char_name	868975	215711567	248,2367928
comp_cast_type	1,92637	45	23,35999834
company_name	69193,9	17802021	257,2773178
company_type	2,02237	92	45,49118114
complete_cast	9242,43	2414495	261,2402799
info_type	39,7328	1928	48,52414126
keyword	13847,7	3791536	273,8025809
kind_type	7,62033	85	11,15437258
link_type	23,898	261	10,92141602
movie_companies	370852	93112104	251,0761813
movie_info	3752617,4	792010495	211,0554876
movie_info_idx	137217	35335875	257,5182011
movie_keyword	362712	93799307	258,6054694
movie_link	2611,33	656584	251,4366242
name	1393964	321191609	230,416
person_info	1595901,2	399133124	250,0988933
role_type	4,12577	160	38,78063974
title	856427	203877939	238,0564123
		AVG	**246,7402929**
		STD	**17,9428158**
		CV	**0,072719439**

Figure 24: Execution time of the length function in Experiment 1 using the first implementation on all the files of the dataset

A.2 Second implementation

File	Average time (ms)	Length	(Length/Time)
aka_name	3667,61	73004383	19905,16522
aka_title	1795,06	38858446	21647,43574
cast_info	102248	1,418E+09	13869,58318
char_name	11619,2	215711567	18565,09631
comp_cast_type	0,151	45	298,013245
company_name	921,501	17802021	19318,50427
company_type	0,15554	92	591,4877202
complete_cast	282,008	2414495	8561,796119
info_type	0,35546	1928	5423,957689
keyword	330,201	3791536	11482,50914
kind_type	9,6749	85	8,785620523
link_type	0,17404	261	1499,655252
movie_companies	6938,79	93112104	13419,06932
movie_info	48641,7	792010495	16282,54142
movie_info_idx	3240,73	35335875	10903,67757
movie_keyword	9915,18	93799307	9460,171878
movie_link	69,1148	656584	9499,904507
name	16333,1	321191609	19665,07332
person_info	28511,8	399133124	13998,87499
role_type	0,17812	160	898,2708287
title	10162,5	203877939	20061,78982
		AVG	**15109,41285**
		STD	**4357,228925**
		CV	**0,288378441**

Figure 25: Execution time of the length function in Experiment 1 using the second implementation on all the files of the dataset

A.3 Third implementation

File	Average time (ms)	Length	ceil(N/B)	ceil(N/B)/t
aka_name	8785,92	73004383	7300439	830,9248206
aka_title	4530,44	38858446	3885845	857,7191178
cast_info	189036	1,418E+09	141813715	750,1942223
char_name	25043,9	215711567	21571157	861,3337779
comp_cast_type	14,5047	45	5	0,344715851
company_name	2034,13	17802021	1780203	875,1667789
company_type	11,025	92	10	0,907029478
complete_cast	338,033	2414495	241450	714,2793751
info_type	18,7994	1928	193	10,26628509
keyword	550,009	3791536	379154	689,3596287
kind_type	0,95146	85	9	9,459146995
link_type	4,581	261	27	5,893909627
movie_companies	12442,1	93112104	9311211	748,3632988
movie_info	83438,9	792010495	79201050	949,2101406
movie_info_idx	5144,85	35335875	3533588	686,8204126
movie_keyword	15221,9	93799307	9379931	616,2128906
movie_link	82,332	656584	65659	797,4906476
name	37347,6	321191609	32119161	860,0060245
person_info	37769,1	399133124	39913313	1056,77162
role_type	10,5923	160	16	1,510531235
title	23236,6	203877939	20387794	877,4000499
			AVG	**811,4168537**
			STD	**109,8062751**
			CV	**0,135326589**

Figure 26: Execution time of the length function in Experiment 1 using the third implementation with B=100 on all the files of the dataset

B	t	ceil(N/B)	ceil(N/B)/t
10	31596,1	7300439	231,0550669
20	19525,9	3650220	186,9424713
30	14372,8	2433480	169,3114772
40	12207,9	1825110	149,5023714
50	10836,9	1460088	134,7329956
60	10039,9	1216740	121,1904501
70	9355,44	1042920	111,4773864
80	8931,16	912555	102,176537
90	8484,24	811160	95,60785645
100	8151,63	730044	89,55803931
110	7906,95	663677	83,93590449
120	8130,3	608370	74,82749714
130	7507,11	561573	74,8054844
140	7320,34	521460	71,23439622
150	7225,41	486696	67,35894572
160	7054,26	456278	64,68119973
170	6962,26	429438	61,68083352
180	6834,09	405580	59,34659918
190	6760,15	384234	56,83808791
200	6692,75	365022	54,53991259
210	6612,3	347640	52,57474706
220	6566,44	331839	50,53560224
230	6518,69	317411	48,69245201
240	6469,82	304185	47,01599117
250	6405,8	292018	45,58649973
260	6335,56	280787	44,31920777
270	6773,49	270387	39,91841724
280	6271,93	260730	41,5709359
290	6220,33	251740	40,47052166
300	6221,11	243348	39,11649207
310	6416,13	235499	36,70421266
320	6133,76	228139	37,19398868
330	6168,23	221226	35,86539412
340	6087,27	214719	35,2734477
350	6047,58	208584	34,49049041
360	6126	202790	33,10316683
370	6072,04	197310	32,49484522
380	5980,69	192117	32,12288214
390	5975,13	187191	31,32835604
400	5938,87	182511	30,73160382
410	5915,56	178060	30,10027791
420	5932,06	173820	29,30179398
430	5917,52	169778	28,69073531
440	6363,75	165920	26,07267727
450	5912,75	162232	27,43765591
460	5841,5	158706	27,16870667
470	5904,99	155329	26,30470162
480	5823,7	152093	26,11621478
490	5814,98	148989	25,62158425
500	5877,3	146009	24,84287003
510	5778,5	143146	24,77217271
520	5790,4	140394	24,24599337
530	5788,85	137745	23,79488154
540	5780,01	135194	23,38992493
550	5761,76	132736	23,03740524
560	5748,25	130365	22,67907624
570	5769,25	128078	22,20011267
580	5715,02	125870	22,02441986
590	5706,23	123737	21,68454479
600	5720,09	121674	21,27134363
610	5716,85	119680	20,9346056
620	6147,33	117750	19,15465739
630	5685,1	115880	20,38310672
640	5739,34	114070	19,87510759
650	5678,24	112315	19,77989659
660	5691,18	110613	19,43586392
670	6067,21	108962	17,9591608
680	5651,24	107360	18,99760053
690	5635,88	105804	18,77328829
700	5619,73	104292	18,55818696
710	5708,56	102824	18,01224827
720	5615,39	101395	18,05662652
730	5636,69	100007	17,74215009
740	5691,82	98655	17,33276878
750	5648,98	97340	17,23142939
760	5614,12	96059	17,11025058
770	5605,87	94811	16,91280747
780	5610,79	93596	16,68142989
790	5605,55	92411	16,48562585
800	6115,44	91256	14,92222964
810	5648,15	90129	15,95726034
820	5570,03	89030	15,98375592
830	5887,21	87958	14,94052361
840	5569,51	86910	15,60460435
850	5595,2	85888	15,35030026
860	5590,05	84889	15,18573179
870	5557,53	83914	15,09915376
880	5645,39	82960	14,69517606
890	5603,96	82028	14,63750633
900	5558,77	81116	14,59243682
910	5608,13	80225	14,30512488
920	5550,57	79353	14,29636956
930	5534,9	78500	14,1827314
940	5557,77	77665	13,97412991
950	5546,97	76847	13,85386977
960	5924,26	76047	12,83653992
970	5551,78	75263	13,55655303
980	6045,61	74495	12,32216435
990	5587,31	73742	13,19812217
1000	5566,24	73005	13,11567593

Figure 27: Execution time of the length function in Experiment 1 using the third implementation on aka_name with B varying from 10 to 1000

A.4 Fourth implementation

File	Average time (ms)	Length	ceil(N/65536)	ceil(N/65536)/t
aka_name	5361,78	73004383	1114	0,207766824
aka_title	2521,57	38858446	593	0,235170945
cast_info	127241	1418137141	21640	0,170070968
char_name	16252,8	215711567	3292	0,202549715
comp_cast_type	0,25072	45	1	3,988513082
company_name	1290,66	17802021	272	0,21074489
company_type	0,25496	92	1	3,922183872
complete_cast	317,221	2414495	37	0,116637928
info_type	10,9712	1928	1	0,091147732
keyword	369,457	3791536	58	0,156987146
kind_type	0,25826	85	1	3,872066909
link_type	0,27686	261	1	3,611933829
movie_companies	8466,79	93112104	1421	0,167832201
movie_info	51280,3	792010495	12086	0,235685049
movie_info_idx	3511,95	35335875	540	0,153760731
movie_keyword	10786,7	93799307	1432	0,132756079
movie_link	85,5874	656584	11	0,128523591
name	23826,5	321191609	4901	0,205695339
person_info	22285,3	399133124	6091	0,273319183
role_type	0,2556	160	1	3,912363067
title	14670,1	203877939	3111	0,212063994
			AVG	**0,191502928**
			STD	**0,041939788**
			CV	**0,219003377**

Figure 28: Execution time of the length function in Experiment 1 using the fourth implementation on all the files of the dataset

A.5 Comparisons

B	LENGTH1	LENGTH2	LENGTH3	LENGTH4
10	288342,2	3587,4	31596,1	5361,78
20	288342,2	3587,4	19525,9	5361,78
30	288342,2	3587,4	14372,8	5361,78
40	288342,2	3587,4	12207,9	5361,78
50	288342,2	3587,4	10836,9	5361,78
60	288342,2	3587,4	10039,9	5361,78
70	288342,2	3587,4	9355,44	5361,78
80	288342,2	3587,4	8931,16	5361,78
90	288342,2	3587,4	8484,24	5361,78
100	288342,2	3587,4	8151,63	5361,78
110	288342,2	3587,4	7906,95	5361,78
120	288342,2	3587,4	8130,3	5361,78
130	288342,2	3587,4	7507,11	5361,78
140	288342,2	3587,4	7320,34	5361,78
150	288342,2	3587,4	7225,41	5361,78
160	288342,2	3587,4	7054,26	5361,78
170	288342,2	3587,4	6962,26	5361,78
180	288342,2	3587,4	6834,09	5361,78
190	288342,2	3587,4	6760,15	5361,78
200	288342,2	3587,4	6692,75	5361,78
210	288342,2	3587,4	6612,3	5361,78
220	288342,2	3587,4	6566,44	5361,78
230	288342,2	3587,4	6518,69	5361,78
240	288342,2	3587,4	6469,82	5361,78
250	288342,2	3587,4	6405,8	5361,78
260	288342,2	3587,4	6335,56	5361,78
270	288342,2	3587,4	6773,49	5361,78
280	288342,2	3587,4	6271,93	5361,78
290	288342,2	3587,4	6220,33	5361,78
300	288342,2	3587,4	6221,11	5361,78
310	288342,2	3587,4	6416,13	5361,78
320	288342,2	3587,4	6133,76	5361,78
330	288342,2	3587,4	6168,23	5361,78
340	288342,2	3587,4	6087,27	5361,78
350	288342,2	3587,4	6047,58	5361,78
360	288342,2	3587,4	6126	5361,78
370	288342,2	3587,4	6072,04	5361,78
380	288342,2	3587,4	5980,69	5361,78
390	288342,2	3587,4	5975,13	5361,78
400	288342,2	3587,4	5938,87	5361,78
410	288342,2	3587,4	5915,56	5361,78
420	288342,2	3587,4	5932,06	5361,78
430	288342,2	3587,4	5917,52	5361,78
440	288342,2	3587,4	6363,75	5361,78
450	288342,2	3587,4	5912,75	5361,78
460	288342,2	3587,4	5841,5	5361,78
470	288342,2	3587,4	5904,99	5361,78
480	288342,2	3587,4	5823,7	5361,78
490	288342,2	3587,4	5814,98	5361,78
500	288342,2	3587,4	5877,3	5361,78
510	288342,2	3587,4	5778,5	5361,78
520	288342,2	3587,4	5790,4	5361,78
530	288342,2	3587,4	5788,85	5361,78
540	288342,2	3587,4	5780,01	5361,78
550	288342,2	3587,4	5761,76	5361,78
560	288342,2	3587,4	5748,25	5361,78
570	288342,2	3587,4	5769,25	5361,78
580	288342,2	3587,4	5715,02	5361,78
590	288342,2	3587,4	5706,23	5361,78
600	288342,2	3587,4	5720,09	5361,78
610	288342,2	3587,4	5716,85	5361,78
620	288342,2	3587,4	6147,33	5361,78
630	288342,2	3587,4	5685,1	5361,78
640	288342,2	3587,4	5739,34	5361,78
650	288342,2	3587,4	5678,24	5361,78
660	288342,2	3587,4	5691,18	5361,78
670	288342,2	3587,4	6067,21	5361,78
680	288342,2	3587,4	5651,24	5361,78
690	288342,2	3587,4	5635,88	5361,78
700	288342,2	3587,4	5619,73	5361,78
710	288342,2	3587,4	5708,56	5361,78
720	288342,2	3587,4	5615,39	5361,78
730	288342,2	3587,4	5636,69	5361,78
740	288342,2	3587,4	5691,82	5361,78
750	288342,2	3587,4	5648,98	5361,78
760	288342,2	3587,4	5614,12	5361,78
770	288342,2	3587,4	5605,87	5361,78
780	288342,2	3587,4	5610,79	5361,78
790	288342,2	3587,4	5605,55	5361,78
800	288342,2	3587,4	6115,44	5361,78
810	288342,2	3587,4	5648,15	5361,78
820	288342,2	3587,4	5570,03	5361,78
830	288342,2	3587,4	5887,21	5361,78
840	288342,2	3587,4	5569,51	5361,78
850	288342,2	3587,4	5595,2	5361,78
860	288342,2	3587,4	5590,05	5361,78
870	288342,2	3587,4	5557,53	5361,78
880	288342,2	3587,4	5645,39	5361,78
890	288342,2	3587,4	5603,96	5361,78
900	288342,2	3587,4	5558,77	5361,78
910	288342,2	3587,4	5608,13	5361,78
920	288342,2	3587,4	5550,57	5361,78
930	288342,2	3587,4	5534,9	5361,78
940	288342,2	3587,4	5557,77	5361,78
950	288342,2	3587,4	5546,97	5361,78
960	288342,2	3587,4	5924,26	5361,78
970	288342,2	3587,4	5551,78	5361,78
980	288342,2	3587,4	6045,61	5361,78
990	288342,2	3587,4	5587,31	5361,78
1000	288342,2	3587,4	5566,24	5361,78

Figure 29: Execution time of the length function in Experiment 1 using all the implementations on aka_name with B varying from 10 to 1000

B Appendix 2 : Experiment 2

B.1 First implementation

Iterations	Average time (ms)	Sum
100	19,8876	2052
200	25,2662	4207
300	25,9037	6214
400	32,0613	8082
500	44,8529	10261
600	48,7866	12366
700	57,9411	14298
800	72,3914	16403
900	76,7728	18564
1000	81,9804	20585
2000	159,575	41328
3000	242,634	62234
4000	320,111	83300
5000	404,636	104084
6000	485,65	124624
7000	557,904	144875
8000	638,603	165627
9000	717,297	186091
10000	797,049	206576

Figure 30: Execution time of the randjump function in Experiment 2 using the first implementation on the cast_info file in function of large number of iterations

B.2 Second implementation

Iterations	Average time (ms)	Sum
100	4,77173	2052
200	18,1898	4207
300	16,0279	6214
400	13,8294	8082
500	18,8686	10261
600	21,3854	12366
700	21,3214	14298
800	25,7332	16403
900	22,7223	18564
1000	29,1971	20585
2000	143,523	41328
3000	153,204	62234
4000	173,845	83300
5000	197,101	104084
6000	187,592	124624
7000	207,32	144875
8000	264,937	165627
9000	192,719	186091
10000	248,819	206576

Figure 31: Execution time of the randjump function in Experiment 2 using the second implementation on the cast_info file in function of large number of iterations

B.3 Third implementation

Iterations	Average time (ms)	Sum
100	7,65031	2165
200	14,2564	4242
300	16,7692	6425
400	24,7671	8551
500	33,981	10636
600	44,3552	12978
700	41,499	15141
800	44,2708	17323
900	51,1397	19537
1000	56,4301	21679
2000	131,35	43639
3000	167,318	65077
4000	227,703	87102
5000	280,503	109122
6000	338,069	130937
7000	390,28	152509
8000	444,053	173438
9000	499,736	195409
10000	559,267	217366

Figure 32: Execution time of the randjump function in Experiment 2 using the third implementation on the cast_info file with a constant B = 100, in function of large number of iterations

B	Average time (ms)	Sum
10	142,298	206576
20	154,372	206576
30	144,53	206576
40	142,891	206576
50	137,066	206576
60	134,901	206576
70	135,756	206576
80	135,21	206576
90	135,738	206576
100	140,212	206576
110	137,7	206576
120	137,46	206576
130	138,374	206576
140	141,517	206576
150	138,597	206576
160	142,386	206576
170	145,293	206576
180	146,928	206576
190	159,02	206576
200	149,41	206576
210	146,569	206576
220	146,819	206576
230	142,25	206576
240	145,444	206576
250	141,994	206576
260	147,991	206576
270	141,655	206576
280	149,536	206576
290	144,841	206576
300	147,541	206576
310	148,383	206576
320	147,341	206576
330	149,505	206576
340	145,579	206576
350	146,857	206576
360	147,044	206576
370	149,463	206576
380	149,564	206576
390	148,083	206576
400	148,997	206576
410	154,4038	206576
420	144,0886	206576
430	140,9236	206576
440	143,5764	206576
450	157,73	206576
460	143,853	206576
470	161,0208	206576
480	144,0728	206576
490	154,1616	206576
500	141,1706	206576
510	154,1706	206576
520	140,6316	206576
530	158,1628	206576
540	159,6746	206576
550	144,3604	206576
560	167,9912	206576
570	168,5996	206576
580	162,7522	206576
590	148,689	206576
600	149,8984	206576
610	159,826	206576
620	150,4144	206576
630	142,7794	206576
640	176,9776	206576
650	160,4942	206576
660	162,5026	206576
670	178,8442	206576
680	180,4184	206576
690	172,2814	206576
700	169,2222	206576
710	174,0662	206576
720	166,6324	206576
730	205,104	206576
740	172,6408	206576
750	149,7412	206576
760	161,6194	206576
770	162,873	206576
780	180,6742	206576
790	168,6382	206576
800	156,1466	206576
810	149,1708	206576
820	171,1766	206576
830	176,0656	206576
840	177,805	206576
850	200,614	206576
860	170,585	206576
870	172,3044	206576
880	200,384	206576
890	188,321	206576
900	176,7914	206576
910	165,2132	206576
920	202,726	206576
930	151,6324	206576
940	193,3314	206576
950	159,7178	206576
960	194,1226	206576
970	176,5678	206576
980	178,7506	206576
990	208,634	206576
1000	218,512	206576

Figure 33: Execution time of the randjump function in Experiment 2 using the third implementation on the cast_info file with B varying from 10 to 1000 and j = 10000

B	Average time (ms)	Sum
10000	486,919	206576
11000	501,472	206576
12000	539,704	206576
13000	564,197	206576
14000	599,334	206576
15000	632	206576
16000	662,792	206576
17000	693,385	206576
18000	722,628	206576
19000	761,774	206576
20000	787,248	206576
21000	853,991	206576
22000	860,888	206576
23000	883,573	206576
24000	920,877	206576
25000	948,203	206576
26000	980,073	206576
27000	1012,04	206576
28000	1047,47	206576
29000	1083,08	206576
30000	1125,06	206576
31000	1229,48	206576
32000	1175,42	206576
33000	1208	206576
34000	1241,01	206576
35000	1268,45	206576
36000	1304,78	206576
37000	1332,82	206576
38000	1381,18	206576
39000	1410,8	206576
40000	1436,65	206576
41000	1471,24	206576
42000	1501,86	206576
43000	1697,47	206576
44000	1759,56	206576
45000	1612,43	206576
46000	1651,62	206576
47000	1686,55	206576
48000	1680,54	206576
49000	1673,08	206576
50000	1995,72	206576

Figure 34: Execution time of the randjump function in Experiment 2 using the third implementation on the cast_info file with B varying from 10000 to 50000 and j = 10000

B.4 Fourth implementation

Iterations	Average ti	Sum
100	8,35968	2150
200	14,0492	4401
300	18,5894	6504
400	28,2143	8469
500	30,2253	10743
600	36,9993	12942
700	49,182	14972
800	49,8882	17175
900	56,3039	19432
1000	58,7572	21551
2000	135,484	43271
3000	181,076	65154
4000	240,315	87199
5000	278,724	108963
6000	327,128	130471
7000	386,82	151698
8000	441,374	173433
9000	500,439	194871
10000	544,931	216325

Figure 35: Execution time of the randjump function in Experiment 2 using the fourth implementation on the cast_info file with a constant B = 65536, in function of large number of iterations

B	Average time (ms)	Sum						
655360	574,446	207601	27525120	662,548	207601	54394880	615,715	207601
1310720	642,694	207601	28180480	636,47	207601	55050240	615,529	207601
1966080	696,489	207601	28835840	621,618	207601	55705600	614,66	207601
2621440	619,632	207601	29491200	617,671	207601	56360960	614,66	207601
3276800	615,8	207601	30146560	617,21	207601	57016320	615,116	207601
3932160	611,985	207601	30801920	617,922	207601	57671680	615,836	207601
4587520	618,978	207601	31457280	616,272	207601	58327040	615,45	207601
5242880	609,566	207601	32112640	619,544	207601	58982400	625,746	207601
5898240	614,761	207601	32768000	618,077	207601	59637760	655,996	207601
6553600	616,963	207601	33423360	623,435	207601	60293120	652,287	207601
7208960	621,637	207601	34078720	618,636	207601	60948480	615,475	207601
7864320	693,348	207601	34734080	619,896	207601	61603840	624,616	207601
8519680	620,666	207601	35389440	619,974	207601	62259200	611,618	207601
9175040	615,21	207601	36044800	616,235	207601	62914560	612,904	207601
9830400	617,025	207601	36700160	618,435	207601	63569920	613,765	207601
10485760	631,104	207601	37355520	619,777	207601	64225280	614,611	207601
11141120	625,269	207601	38010880	622,355	207601	64880640	626,191	207601
11796480	642,488	207601	38666240	627,719	207601	65536000	617,454	207601
12451840	620,376	207601	39321600	622,91	207601			
13107200	621,717	207601	39976960	619,842	207601			
13762560	614,705	207601	40632320	613,966	207601			
14417920	620,185	207601	41287680	620,551	207601			
15073280	629,077	207601	41943040	618,346	207601			
15728640	636,728	207601	42598400	618,348	207601			
16384000	621,122	207601	43253760	619,835	207601			
17039360	619,136	207601	43909120	618,507	207601			
17694720	621,099	207601	44564480	624,307	207601			
18350080	622,029	207601	45219840	612,281	207601			
19005440	620,783	207601	45875200	615,453	207601			
19660800	627,175	207601	46530560	619,584	207601			
20316160	619,848	207601	47185920	620,695	207601			
20971520	619,362	207601	47841280	617,415	207601			
21626880	635,624	207601	48496640	613,059	207601			
22282240	626,874	207601	49152000	665,952	207601			
22937600	620,564	207601	49807360	617,371	207601			
23592960	618,962	207601	50462720	615,477	207601			
24248320	617,738	207601	51118080	616,365	207601			
24903680	622,434	207601	51773440	617,455	207601			
25559040	624,642	207601	52428800	617,158	207601			
26214400	622,59	207601	53084160	646,461	207601			
26869760	653,44	207601	53739520	618,287	207601			

Figure 36: Execution time of the randjump function in Experiment 2 using the fourth implementation on the cast_info file with B varying from 65536 to 65536000 and j = 10000

B	Average time (ms)	Sum						
1310720000	629,55	207601						
1317273600	479,256	207601	1599078400	104,9646	207601	1880883200	78,4138	207601
1323827200	497,638	207601	1605632000	136,5192	207601	1887436800	79,9592	207601
1330380800	493,008	207601	1612185600	130,1274	207601	1893990400	80,5882	207601
1336934400	478,86	207601	1618739200	103,0018	207601	1900544000	77,9722	207601
1343488000	474,302	207601	1625292800	85,7642	207601	1907097600	78,1846	207601
1350041600	525,352	207601	1631846400	88,3928	207601	1913651200	77,4034	207601
1356595200	474,83	207601	1638400000	86,024	207601	1920204800	81,1044	207601
1363148800	458,32	207601	1644953600	84,0608	207601	1926758400	80,4268	207601
1369702400	464,63	207601	1651507200	87,9208	207601	1933312000	78,8774	207601
1376256000	458,57	207601	1658060800	87,0684	207601	1939865600	82,3238	207601
1382809600	473,222	207601	1664614400	89,0218	207601	1946419200	79,8718	207601
1389363200	440,366	207601	1671168000	94,5058	207601	1952972800	79,5724	207601
1395916800	442,318	207601	1677721600	91,663	207601	1959526400	82,0276	207601
1402470400	489,738	207601	1684275200	90,0528	207601	1966080000	79,5492	207601
1409024000	504,948	207601	1690828800	83,054	207601	1972633600	80,665	207601
1415577600	409,188	207601	1697382400	80,3668	207601	1979187200	79,435	207601
1422131200	401,078	207601	1703936000	81,8678	207601	1985740800	81,1448	207601
1428684800	400,606	207601	1710489600	79,8408	207601	1992294400	79,6778	207601
1435238400	374,382	207601	1717043200	83,5156	207601	1998848000	78,765	207601
1441792000	352,866	207601	1723596800	88,2014	207601	2005401600	96,5054	207601
1448345600	312,412	207601	1730150400	92,6426	207601	2011955200	114,4146	207601
1454899200	114,6226	207601	1736704000	82,0496	207601	2018508800	82,7848	207601
1461452800	145,7226	207601	1743257600	81,0444	207601	2025062400	80,943	207601
1468006400	127,5082	207601	1749811200	79,3804	207601	2031616000	79,969	207601
1474560000	128,4316	207601	1756364800	79,276	207601	2038169600	79,3842	207601
1481113600	111,2634	207601	1762918400	80,9716	207601	2044723200	80,5236	207601
1487667200	88,2618	207601	1769472000	79,2068	207601	2051276800	81,6886	207601
1494220800	90,8296	207601	1776025600	86,544	207601	2057830400	80,1464	207601
1500774400	86,8736	207601	1782579200	81,6104	207601	2064384000	78,5156	207601
1507328000	85,815	207601	1789132800	82,5856	207601	2070937600	79,6138	207601
1513881600	89,8234	207601	1795686400	81,5602	207601	2077491200	79,3006	207601
1520435200	87,4112	207601	1802240000	79,7214	207601	2084044800	80,2886	207601
1526988800	89,728	207601	1808793600	79,217	207601	2090598400	78,8944	207601
1533542400	87,0782	207601	1815347200	77,728	207601	2097152000	80,4384	207601
1540096000	121,1008	207601	1821900800	82,3038	207601	2103705600	81,7884	207601
1546649600	138,617	207601	1828454400	79,2554	207601	2110259200	81,8084	207601
1553203200	97,6336	207601	1835008000	79,2592	207601	2116812800	83,2022	207601
1559756800	86,7128	207601	1841561600	80,4288	207601	2123366400	80,851	207601
1566310400	89,2328	207601	1848115200	79,0746	207601	2129920000	83,1116	207601
1572864000	86,0966	207601	1854668800	80,5002	207601			
1579417600	89,5436	207601	1861222400	79,036	207601			
1585971200	89,7538	207601	1867776000	78,8498	207601			
1592524800	87,9948	207601	1874329600	79,534	207601			

Figure 37: Execution time of the randjump function in Experiment 2 using the fourth implementation on the cast_info file with B varying from 1310720000 to 2129920000 and j = 10000

C Appendix 3 : Experiment 3

A = link_type,kind_type
B = link_type,kind_type,info_type,company_type,comp_cast_type,role_type,movie_link
C = complete_cast,keyword
D = keyword,company_name,movie_info_idx,movie_keyword,movie_companies,aka_title,
aka_name
E = complete_cast,keyword,movie_link,company_name,movie_info_idx,link_type,kind_type,
info_type,company_type,comp_cast_type,role_type,movie_keyword,movie_companies,
aka_title,aka_name
F = movie_info,person_info
G = movie_info,person_info,name,cast_info,title,movie_keyword,char_name
H = title,link_type,kind_type,info_type,company_type,comp_cast_type,role_type,complete_cast,
keyword,movie_link,company_name,movie_info_idx,movie_keyword,movie_companies,
aka_title,aka_name,movie_info,person_info,
name,cast_info,char_name

The parameter k corresponds to the number of files.

- k=2 for A,C and F.
- k=7 for B,D and G.
- k=15 for E.
- k=21 for H.

B \ files	A	B	C	D	E	F	G	H
20	2.7984	348.364	3146.775	256671	200145.1	522209	1.85E+06	1.92E+06
40	2.4657	294.813	2755.22	216324	143275	419120	1.26E+06	1.37E+06
60	2.3554	227.318	2162.74	174769	111621	320816	979280	1.06E+06
80	2.1458	226.186	1908.02	147152	96014.5	275810	843360	908807
100	2.1457	194.339	1718.57	137235	88202.4	247398	751176	824213
120	2.1112	167.961	1645.79	121560	80430.5	225686	698587	757953
140	2.11	159.64	1559.54	118677	75121.7	209789	654416	707384
160	2.0894	151.396	1496.64	111405	73207.8	203786	626532	683284
180	2.1415	141.537	1488.5	105502	71589.9	191232	609867	646340
200	2.1201	136.335	1435.05	104887	67896.7	186192	586400	629764
400	2.1024	134.218	1235.71	86268.1	56791.7	152744	502703	532783
600	2.0962	133.534	1130.47	82904.8	53229.9	142879	472759	527115
800	2.0647	126.851	1105.62	78822.7	53019.2	141336	454349	503109
1000	1.9914	129.375	1084.42	53626.1	50309.6	143226	450683	483385
1200	1.881	114.843	1091.83	49635	50020.6	130313	465640	477155
1400	1.8721	115.6	1103.05	42868.8	48749.7	127930	445603	434547
1600	1.8724	116.34	1049.51	42499.2	48771	129024	488503	445476
1800	1.8765	115.037	1085.53	48802.7	49117.3	125876	446415	429899
2000	1.8721	113.727	1046.87	47096.9	49823.3	127123	427604	436578

Figure 38: Execution time (in ms) of the pair InputStream2,OutputStream3 in function of B and the files

B\files	A	B	C	D	E	F	G	H
20	3.6504	395.331	33457.2	195421	201564	614423	1.91E+06	2.25E+06
40	3.3145	331.18	2966.46	146884	150095	478749	1.53E+06	1.60E+06
60	3.145	234.966	2821.5	118753	117191	361826	1.27E+06	1.27E+06
80	3.0676	210.051	2398.48	101332	103799	307751	1.11E+06	1.16E+06
100	2.7142	190.101	2242.12	91799.2	92148.4	271947	1.01E+06	1.07E+06
120	2.4546	185.61	2120.7	86847.2	85845.9	250892	929928	962718
140	2.3598	168.606	1749.87	80346.1	81042.4	231387	912935	927552
160	2.2156	170.554	1572.15	76699.3	79075.4	222253	835499	879393
180	2.1564	166.69	1510.28	73786.1	74972.3	211533	840833	835650
200	2.1486	159.365	1513.2	71114.3	72475.6	204641	804007	809620
400	2.1121	146.324	1328.82	61130.5	64467.2	177213	748194	794142
600	2.0456	134.616	1245.69	57477.2	58402.4	157371	689429	680293
800	2.0693	128.207	1230.22	57990.9	56542.5	149161	734013	668924
1000	2.0679	120.422	1215.6	55049.6	56222.5	143359	674732	651864
1200	2.0456	116.497	1219.62	53986.2	55254.9	141394	709783	677435
1400	2.0497	120.817	1220.39	53547.1	54769.8	140502	641006	707816
1600	2.0465	131.865	1173.98	54075.5	55811.4	137836	663139	676364
1800	2.0467	136.47	1179.86	52821.6	53948.5	138079	632629	638047
2000	2.0466	131.698	1198.47	54977	53551.9	136303	644177	623149

Figure 39: Execution time (in ms) of the pair InputStream4,OutputStream3 in function of B and the files

B\files	A	B	C	D	E	F	G	H
65536	2.6611	115.788	1079.83	42713	46597.7	111515	394951	407167
131072	2.5541	112.414	1035.78	42412	45931.5	110656	389953	406182
327680	2.361	112.388	1051.25	42723	45327.1	110312	386902	404163
655360	2.9615	112.598	1059.83	42609	45136.5	110315	396235	404235
1310720	2.4299	113.908	1029.41	42601	45428.5	110303	387999	403523
6553600	2.6691	113.481	1039.18	42819	45230.1	110288	386451	402530
Mean value	2.60611667	113.4295	1049.21333	42646.1667	45608.5667	110564.833	390415.167	404633.333

Figure 40: Execution time (in ms) of the pair InputStream2,OutputStream4 in function of B and the files

B\files	A	B	C	D	E	F	G	H
65536	2.9893	122.544	1172.24	48742.9	50545.3	120042	579235	590157
131072	2.8206	111.798	1162.99	48672.2	49069.5	119695	577355	584375
327680	2.8292	117.244	1161.67	48610.1	49051.5	118595	571764	579654
655360	2.9293	114.978	1155.21	48574.8	47890.4	118493	568900	575155
1310720	2.7293	119.43	1099.46	48582.9	47999.5	119007	571024	574111
6553600	3.1293	121.665	1112.54	48589.6	48163.1	118234	567644	574743
Mean value	2.9045	117.943167	1144.01833	48628.75	48786.55	119011	572653.667	579699.167

Figure 41: Execution time (in ms) of the pair InputStream4,OutputStream4 in function of B and the files

D Appendix 4 : Multi-way Merge sort

M	Average time (t)	t/log(ceil(N/M))
10000	182307	56083,00474
20000	158778	53825,28224
30000	149701	53969,90827
40000	145973	55097,98036
50000	145741	57093,59292
60000	143695	58111,26285
70000	134600	55930,91738
80000	133389	56802,25004
90000	138139	60147,64389
100000	134898	59878,7396
110000	129970	58822,86345
120000	127892	58849,99453
130000	122551	57354,71539
140000	124929	59286,45071
150000	122407	58975,77953
160000	126142	61556,16367
170000	115869	57327,13905
180000	108678	54457,84896
190000	113753	57651,10446
200000	112284	57456,53339
210000	112833	58480,30124
220000	108280	56736,10164
230000	108324	57250,83717
240000	109450	58371,42612
250000	110546	59518,69157
260000	109880	59754,76756
270000	106751	58669,09719
280000	102242	56606,76205
290000	102864	57389,24176
300000	97286,9	54712,38736
	AVG	**57538,95964**
	STD	**1880,165646**
	CV	**0,032676393**

Figure 42: Execution time of the multi-way merge sort algorithm on company_name in function of M

File name	File length (N)	Average time (t)	N.log(N)	N.log(ceil(N/M))/t
aka_name	73004391	694323	574059003,7	269,5364772
aka_title	38858446	313140	294915609,7	284,1770016
company_name	17802021	114098,2	129073006,9	304,9081072
complete_cast	2414495	21334,9	15411302,21	126,0662414
keyword	3791536	17931,8	24943814,61	270,3822435
movie_companies	93112112	1,03E+06	742010995,7	241,6399684
movie_info_idx	35335875	365914	266722811,7	217,0840753
movie_keyword	93799307	1,01E+06	747786800,5	247,9138787
movie_link	656584	3454,79	3819539,731	114,4217036
title	203877939	2,05E+06	1694097278	299,6678875
			AVG	**237,5797584**
			STD	**63,87730709**
			CV	**0,26886679**

Figure 43: Execution time of the multi-way merge sort algorithm on several files in function of N

d	Average time (t)
2	167956
3	146952
4	141550
5	139705
6	128514
7	123042
8	119560
9	116540
10	116257
11	120090
12	116000
13	109457
14	111429
15	109108
16	108690
17	109983
18	109119
19	107175
20	109627

Figure 44: Execution time of the multi-way merge sort algorithm on company_name in function of d