Université Libre de Bruxelles

Database systems architecture

INFO-H417

Algorithms in Secondary Memory

Group 26:

Ghita AIT OUHMANE
Hà My DUONG
Maxime LANGLET

Year 2020 - 2021

Contents

1 Introduction and environment

1.1 Aim of the project

The objective of this project is to evaluate the performance of external-memory algorithms. Two main steps are needed in order to accomplish that :

- First; a comparison of different methods that can be used to read from and write to secondary memory
- Second; the implementation of an external-memory merge-sort algorithm and the analysis of its performance under different parameters

The performance must be evaluated on the IMBD movie database, a 3.7GB dataset that is composed of CSV text files. This dataset contains information related to movies, television programs, etc.

1.2 Environment

To implement the code, our group used the C++ programming language on MacOS and on a Windows Subsystem for Linux (WSL2); this subsystem allowed us to run a GNU/linux environment directly on our Windows machines without the overhead of a dual boot or a VM.

All the test were conducted on a MacBook pro 2019 with an Intel Core i5 four cores 1,4 GHz with 8 GB of LPDDR3 2133 MHz and 256 KB and 6 MB of L2 and L3 cache respectively. The internal storage is a Solid State Drive. The exact specifications of this SSD are not listed but a BlackMagic Disk Speed Test gives an average of more or less $340MB/s$ write speed and $1240MB/s$ read speed.

2 Reading and writing

The first thing to do was to implement input and output streams, with four different I/O mechanisms to read from and write to a file.
In order to do that, we defined 8 different classes, 4 of them being for the input streams and the 4 other for the output streams.

2.1 Input stream classes

In the input stream classes, the following methods were required :

- `open`: opens an existing file for reading
- `readln`: reads the next line in the stream
- `seek(pos)`: moves the file cursor to pos so that the subsequent readln reads from the position pos to the next end of line
- `end_of_stream` : a boolean operation that returns true if the end of file has been reached.

The implementations for `open,seek(pos),end_of_stream` are quite similar. The major difference concerns the `readln` method. The different input stream classes were implemented as follows :

2.1.1 InputStream1 :

Reads one character at a time using the `read` system call that stores the number of bytes read (1 byte in our case) in a 1 byte pointer (called c) which is appended to a string containing the line until that point.Then it checks if it is equal to the new line character or the carriage return. Once the condition is satisfied the function ends and returns a string of the line read, or equivalently the characters read.

2.1.2 InputStream2 :

Reads a buffer of **fixed size B** using the function `fgets`. It stops reading when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. We chose to use a fixed buffer size of 1000 as the IMDB datasets does not contain a file with a line containing more characters than 1000 (according to an answer to a question asked by e-mail),this way we are sure to read the whole line with this size of buffer.

2.1.3 InputStream3 :

Allocates memory for a buffer of size B, where we store the B characters to be read using the system call `read` , B being a **parameter** that the user can choose. Whenever the buffer is full, we first look into it to see if the newline or carriage return is present. If the answer is yes, the offset is set as the next index using the `seek(pos)` method, else the next B characters are read and so on until the end of line is found. After that, a string containing the read string is returned.

2.1.4 InputStream4 :

Mapping a file consists of creating a new projection in the virtual address space of the calling process. This way there is a direct byte-to-byte correlation between the virtual memory and the region that has been mapped.

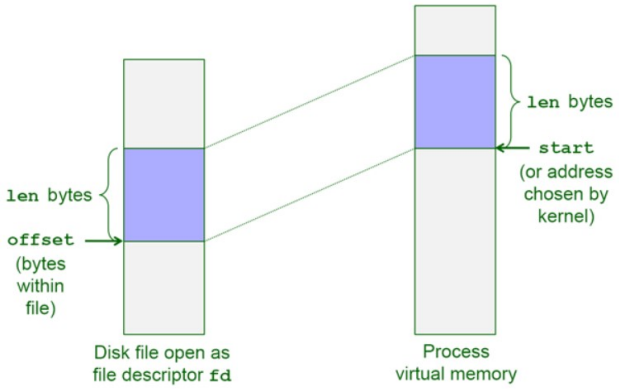

Figure 1: Memory mapping [1]

For the implementation the `mmap` function from `<sys/mman.h>` has been used.
Since this system call requires an offset proportional to the size of a page (which is 4096 for our computer), we first need to determine at which position we currently are in the file. We first get the position of the cursor, that way we can compute the page in which it is currently at. Then we can map the file into our buffer of a certain size B. The size of this buffer should be proportional to a page size since the offset is as well, otherwise some characters might not be read.
The function `readln` doesn't unmapthe mapped region right away after the line has been read; indeed in the case of sequential reading, the next line to read will probably be on the same page, so we keep that same map. If not, the previous region is first unmapped and then we map again the page of interest. Another point is that, if a line starts on one page but ends on another, then the function maps the following page (pages if necessary) until finding the end of the line. The function returns a string containing the string of the line read.

2.2 Output stream classes

In the output stream classes, the following methods were required :

- `create`: creates a new file, if it already exists we chose to overwrite it.
- `writeIn`: writes a string to the stream and terminates with the newline character
- `close`: closes the stream

The different implementations of the `writeln` method in each class were implemented as follows :

2.2.1 OutputStream1 :

Writes a string in the file character per character using the `write` system call which writes a number of bytes (`size_of(char)`in our case) from the string str given as an parameter into the output file.

2.2.2 OutputStream2 :

Writes the string in the output file using the function fputs.

2.2.3 OutputStream3 :

This function first checks if the string given as an argument is not empty, if it is the case, the function returns, else we allocate memory for a buffer of size B that we initialize (B is chosen by the user) and fill it with the B first characters of the input string, we then check each entry of the buffer looking for the carriage return or the newline characters, if we find it, we write that specific size in the file using the `write` system call and set the attribute `over` to one, this way we will exit the while loop, otherwise we will write the characters in the buffer to the file, adjust the offset and repeat, until the end of the input string is reached.

2.2.4 OutputStream4 :

Writes in a file using the mmap/munmap functions, maps a size of B that is rounded up to the next multiple of a page size. To do that, we increase the size of the file using lseek to get the file's size and ftruncate so that we can map the file and write the string in it. We loop on the number of multiple of pages B needed to write the whole string.
Two cases are taken into account, the first one is when we are at the first pass of the loop which means that we can possibly map a page that already contains some text (if the file is not empty) or a page that was already mapped with this instance of outputstream4. When we write for the first time with the instance of outputstream4, we initiate the attribute `init` to 1 (default value of that attribute being 0). That way, we know that we are forced to map the file. When it is not the first time calling write with that instance of outputstream4, we check if the current page that we have last mapped is the same as the one we are trying to map. If not, we synch and unmap the last map and map a new multiple of page size.
To write the string in the map, we have two situations, the first one being that the page we are writing to already contains some text and there is enough space to write the entire string. The second situation is the one where there is not enough space on the map to write the whole string; in this case we write to the end of the map and the rest will be written on another map.
Which leads us to the second case, where we are not at the first pass of the loop which means that we needed more multiple of pages B to write the entire string (in the case of the experiments, we never encounter this case as a line in the data set has a maximum size of 1000 characters). In that case, we synch and unmap the last map, then remap to write the rest of the string or, if it still does not fit entirely on the map, writes until the end and continues to loop.

3 Experimental setup

The following section will present the different experiments conducted for this project and the results obtained.
In order to facilitate the testing phase, a python script `DBproject.py` has been made to "automate" the process. This file contained the path to the files to test and the different buffer sizes and jump counts, and for each experiment, each file has been tested in a single execution of the process.

Each method was ran 4 times and the values retrieved are the average of those four tests.
The average times were measured using a separate class called Timer. This class uses the chrono library.The point of this class is pretty straight forward, it takes the current time at the constructor of the class and again either at the destructor or when the Stop function is called. Then the two measurements are subtracted which gives us the total time taken. We decided that a good order of magnitude was to measure in micro seconds. Another advantage of using micro-seconds is that, when dividing the number of characters read by our method by the time taken, we directly have a MB/s transfer rate.

4 Experiment 1.1 : Sequential reading

The fist experiment consists of writing a program called `length`,that takes as input a file from the IMDB dataset, and reads it sequentially, one line after the other and adds the length of the line to a counter called sum. When the entire file has been read, the sum value is returned.

The implementation of the method for the different input stream classes follows the same idea; first since each input stream is associated to a file through the `open` method, the first thing to do is to start by changing the position of the next element to be read and set it at the start of the file (position 0) with a call to `seek(0)`,in case the `readln` method has been called beforehand.
We then initialize an int value called `sum` to 0 and enter a while loop that is repeated as long as the size of the return value from the `readln` call is positive or we reached the end of the file; indeed if the file to be read is empty or if we reached the eof, the `readln` method returns an empty string. Otherwise we retrieve the returned string's size and add the value to the sum variable. Once we exit the while loop we return `sum`.
The only implementation difference is in the case of the InputStream4 class : since we in general read the lines sequentially one after the other, as said earlier we don't unmap the mapped region right away in the `readln` call, but rather at the end of the length method, this way, at the next call to readln, we first check if the region of interest is still mapped, and avoid unnecessary mappings and unmappings.

Before conducting the actual experiments, we can already try to predict the expected performance of each method .
Let N be the size of the file and B the size of the buffer:

- Method 1 : Each call to readln corresponds to one call to the read method, in addition to that we have to see when the eof is reached ; indeed in this case readln returns an empty string (of size 0) and that is the condition for breaking the while loop. So in the end we make N+1 I/O's.
- Method 2 : In this implementation ,readln reads a maximum of 1000 characters at a time (stops earlier if a newline character has been met) so the worst case scenario is the one where each character is written in a line, making the I/O cost N, and in the best case each line contains 1000 characters and the cost here is $\lceil \frac{N}{1000} \rceil$.
- Method 3 :In this method each call to readln reads B characters, so the cost would be the total number of calls to readln, which is $\lceil \frac{N}{B} \rceil$. The value is rounded up to take into account the case where the last call contains less than B characters.
- Method 4 : When mapping a file using `mmap`, the pages are not loaded directly into physical memory, but implements demand paging, where the actual readings from disks are performed in a lazy manner, on demand, only when a specific location is accessed, one page at a time.[2]
  When the operating system tries to access that data, but that it is still not present in the memory space, a page fault occurs; depending on whether the page was already loaded into memory or not, the page fault is either minor or major . Either way, this faulting leads to additional I/O costs[3][4][5] , so each time a page is accessed for the first time, a new I/O is made for each page being read (for sequential reading, the probability for the page to already be in memory is quite small) so the total cost would be around $\left\lceil \frac{N}{pagesize} \right\rceil$.

We can then expect the fourth implementation to work best, and the first one to be the worst.
The four implementations have been tested on different file sizes : `role_type.csv` (160 Bytes) , `movie_link.csv` (659 KB), `movie_info_idx.csv`(35.3 MB) and `aka_names.csv` (73 MB).
The detailed performance result for each file and buffer size can be found in a table in the appendix, we will present here some graphs to compare the performance of the different methods on each file.
These graphs represent the variation of speed (obtained by dividing the file size by the average time taken

by each method) as a function of the buffer sizes for each implementation.

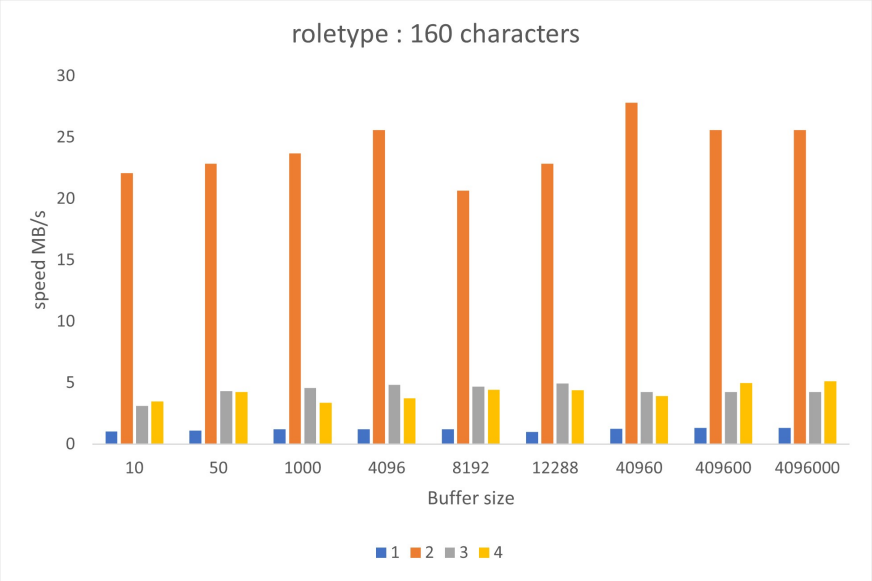

Figure 2: Sequential reading on role_type.csv

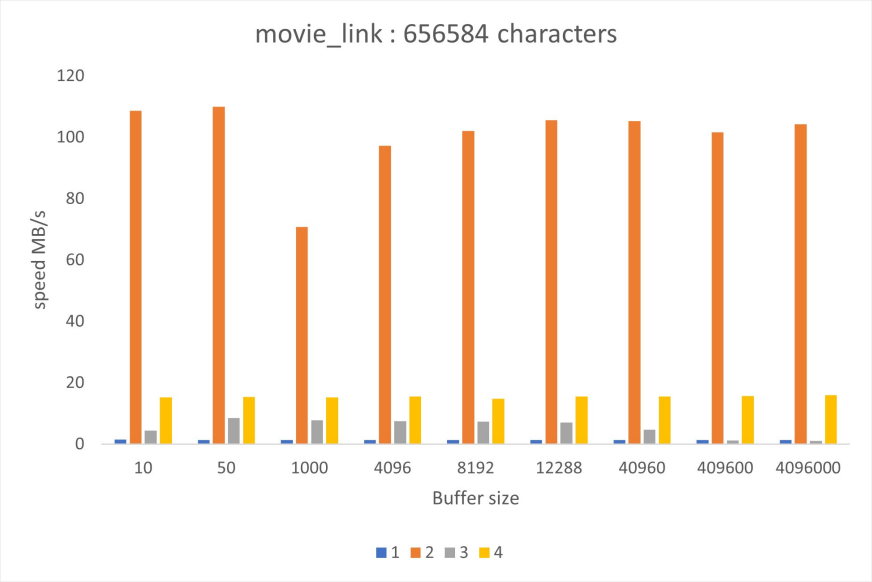

Figure 3: Sequential reading on movie_link.csv

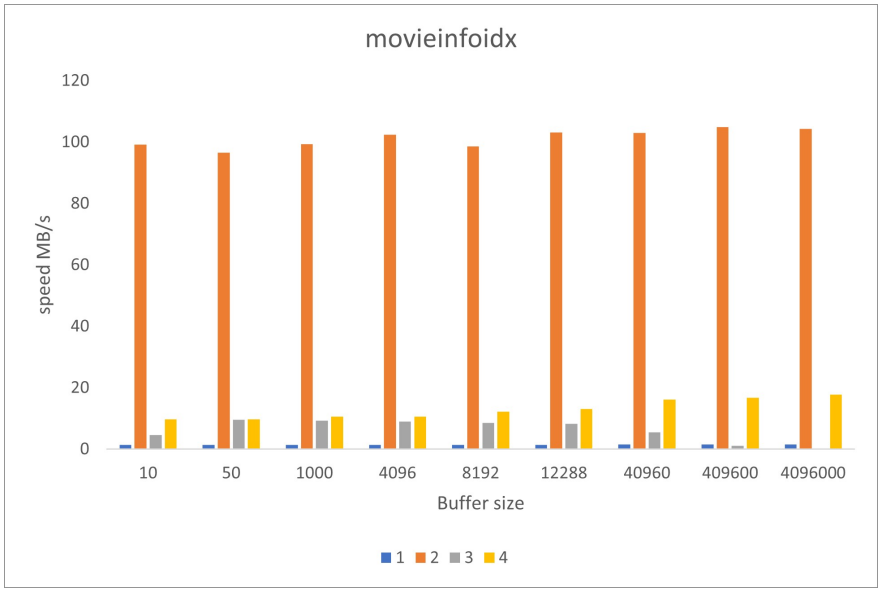

Figure 4: Sequential reading on movie_info_idx.csv

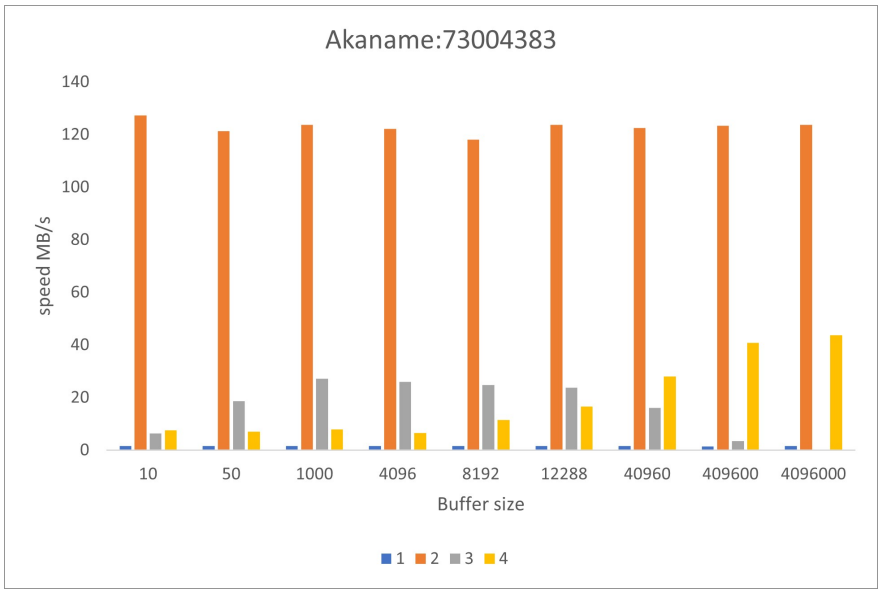

Figure 5: Sequential reading on aka_name.csv

For clarification, concerning the 4th implementation, when buffer sizes are inferior to a page size (i.e. 4096 on our machine),the size of the buffer is rounded up to that page size since the readln method we implemented can only take an integer representing the number of pages. Again, for comparison sake, we will keep method 4 in the graph even when the size of the buffer is inferior to a page size.

From a first look at the results, we can see that the worst implementation is indeed the first one, because a system call to read is made for each character of the file. However, the performance of the mapping wasn't the one expected; it is only the second best. Many factors could explain why it is the case ; as

explained earlier, the mapping is a lazy process, where the page content isn't loaded until needed, generating a page fault at each time; these page fault take some time to be processed; the process goes to a blocked state, the content is loaded ...etc. And since we read the file sequentially, we don't re-access that value, which is quite pointless in this case. The performance could also be influenced by the characteristics of the CPU and the OS on which the process is being executed.
The second implementation is the best compared to the third one because, as said before, a line's size is as max 1000, so there is one I/O made per line, where as for the buffers of size smaller than 1000, the performance is slow since more than one I/O are needed for each line and additional `lseek` calls are made to start reading from the previous offset. The third implementation presents a maximum read speed between a buffer size of 50 to 1000. Then it slowly decreases in speed until being just as slow as the first method. Our assumption is that, this method loses performances when allocating and freeing large buffer sizes at each call to readln. Moreover, it might be possible that the buffering mechanism used in method two with `fgets`is more efficient than the one used in method three with `read`, which would depend on how those functions were implemented and how low-level programmed they are.
On a side note, the performance of implementations one and two obviously don't vary for a given file since there are no buffers associated with them.

Note that, when increasing the file size, every method increased their overall speeds, apart from implementation 1 . It seems that the file size impacts the throughput in general. This could be due to a hardware optimization. Another explanation could be that the operating system caches parts of the file into the main memory to speed up the reading process. Since the first file `role_type.csv` is quite small, the process doesn't take as much time as for big files and no optimization could be required, especially since the file is read sequentially.
For the first implementation, optimization might happen, but given the big number of I/O operation, the final performance doesn't benefit from it and the optimization process isn't that noticeable .

5 Experiment 1.2 : Random reading

The second experiment is the random reading. It consists in reading until the end of a line, starting from a random position in the text. Of course, for consistency in our testing, the random numbers will be generated from a seed, that way we will always read in total the exact same number of characters between each call of the different methods and buffer sizes for a given file.

The general idea of this method is the following: after opening the file through the open function, we also initialize an integer `sum` that will correspond to the total characters read. We then initialize the seed and the first random position in the text, we need to stay in the file, to that end we need to find the size file. We then set the position of the cursor to the right position in the file and read the rest of the line. The numbers of characters read are then added to the total sum. We then repeat this process until we have reached the desired number of random jumps. The total numbers read are then returned. Depending on the method of reading used, some additional code is added to ensure the correct and optimal implementation of the method is used (e.g. unmapping).

Before conducting the experiment, we can already try to predict the performance of each implementation : Let M be the **average** size of a line in the IMDB dataset

- Method 1 : Before reading, a jump has to be made using the `lseek` system call, then the `readln` function is called on each character. This operation is repeated k times so the total cost would be $k * (M + 1)$
- Method 2 : Following the same idea, `lseek` is called k times and the total cost would be $k * (1 + \lceil \frac{M}{1000} \rceil)$ Since the maximum size of a line in this dataset is a thousand, we expect M to be less than that especially since the jump doesn't always land in the beginning of the line, so $\lceil \frac{M}{1000} \rceil$ will generally give 1 and we can expect the total cost to be around $2 * k$.

- Method 3 : For each jump we make `lseek` and then read the line from that position, which would have a cost of $\lceil \frac{M}{B} \rceil$, repeated k times, so the total would be $k * (1 + \lceil \frac{M}{B} \rceil)$
- Method 4 : For each jump, we seek and perform a mmap operation. Here since the jump are randomized, the probability that the cursor ends up in a previously accessed page is rather small, so we can consider that faulting occurs for every jump, and since we end up reading only one line, only one page will be accessed, so the total cost would be (k times repetition) $2 * k$. That means that the program will have to unmap and map the right page at each step. We thus predict some loss in effectiveness compared to a sequential reading.
  In the case where all the jumps end up on the same page (very low probability, but not impossible) the total cost would be $k + 1$!
  As seen before the performance of the mapping could be slower due to the processing of all the page faults and overheads, so the final performance could end up being much slower than this.

So we can already expect the first implementation to perform worst. All these tests were conducted on the same files as sequential reading.

First, since the first two methods don't depend on varying size of buffers, we will compare them before testing the third and fourth method.
(Again the detailed results are presented in the appendix B).

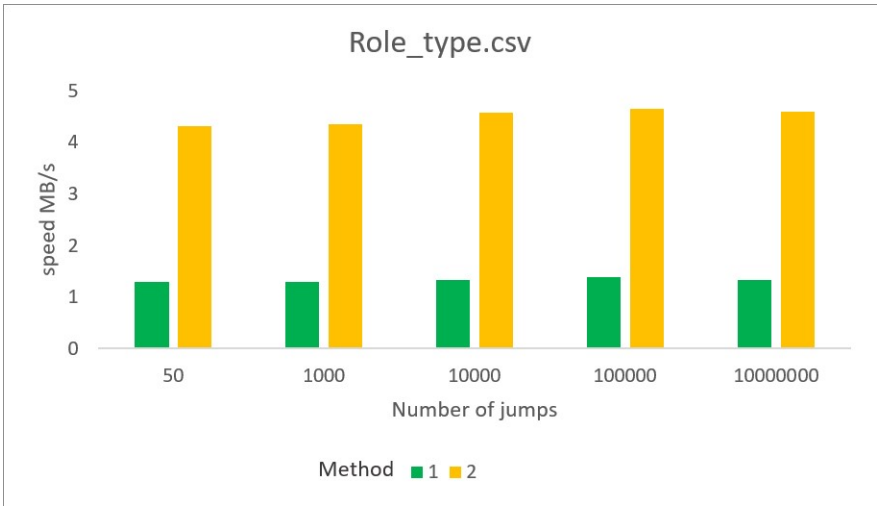

Figure 6: Random reading on role_type.csv

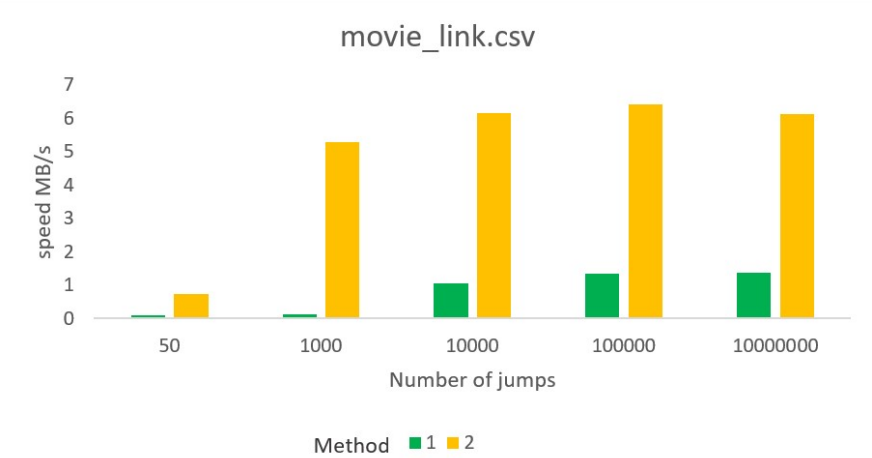

Figure 7: Random reading on movie_link.csv

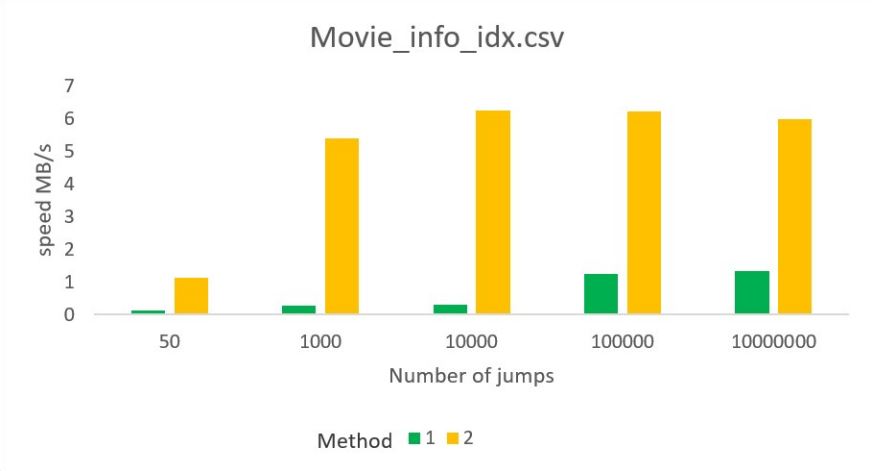

Figure 8: Random reading on movie_info_idx.csv

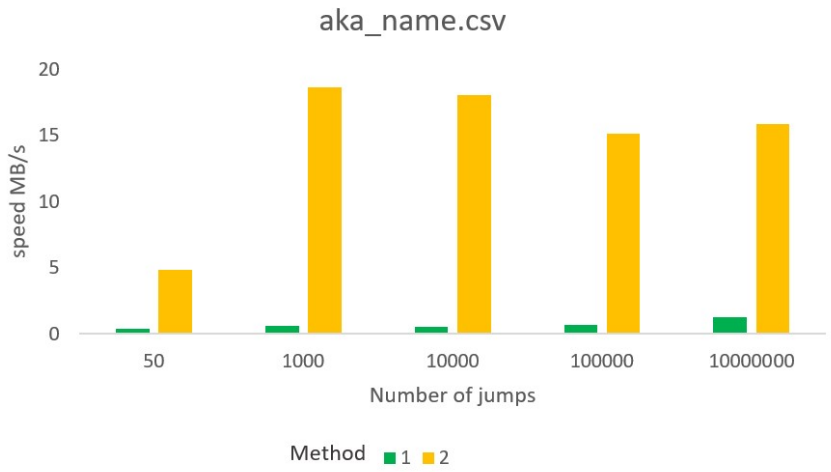

Figure 9: Random reading on aka_name.csv

We can see that these two methods presents the same number of read characters (see appendix B) which confirms our methods works as intended.

For the smallest file, we can see that the performance of both implementations is constant, no matter the number of jumps, this could be explained by the fact that the file is so small (160 characters in total) that when the jump is performed, the probability to end up on a page which was accessed before and that is in the main memory is much higher than for other files, even for 50 jumps.
We know that many operating systems actually put into memory more than what the method actually calls for, which means that if we are in the same region of the file, it is possibly already in memory thanks to the operating system. However, for bigger files, the behavior is a little different. The more jumps we make, the less time it takes; this again could be explained by the higher probability to end up in a previously accessed region ,which would be stored in the main memory, making the execution faster than a disk access.
Here again the performance grows with the file size, like for sequential reading, this can be explained by a caching mechanism , which is much more efficient for the method2 compared to method1.

Looking at the overall performances from both methods, we see reduced speeds compared to sequential reading. This is due to the cost of seeking at each iterations. This will also have an effect on the third and fourth method as we will see next. The following two methods will be treated separately since a broader spectrum of variables were tested, that way, hopefully, an optimal value can be returned.

Method 3:

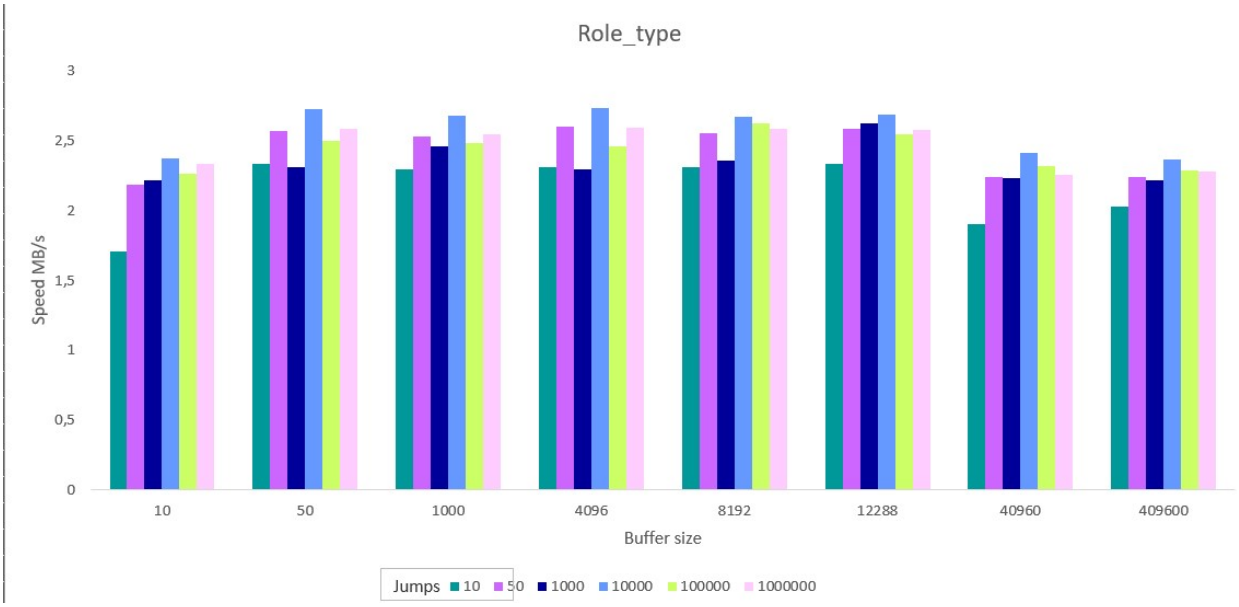

Figure 10: Random reading method3 on role_type.csv

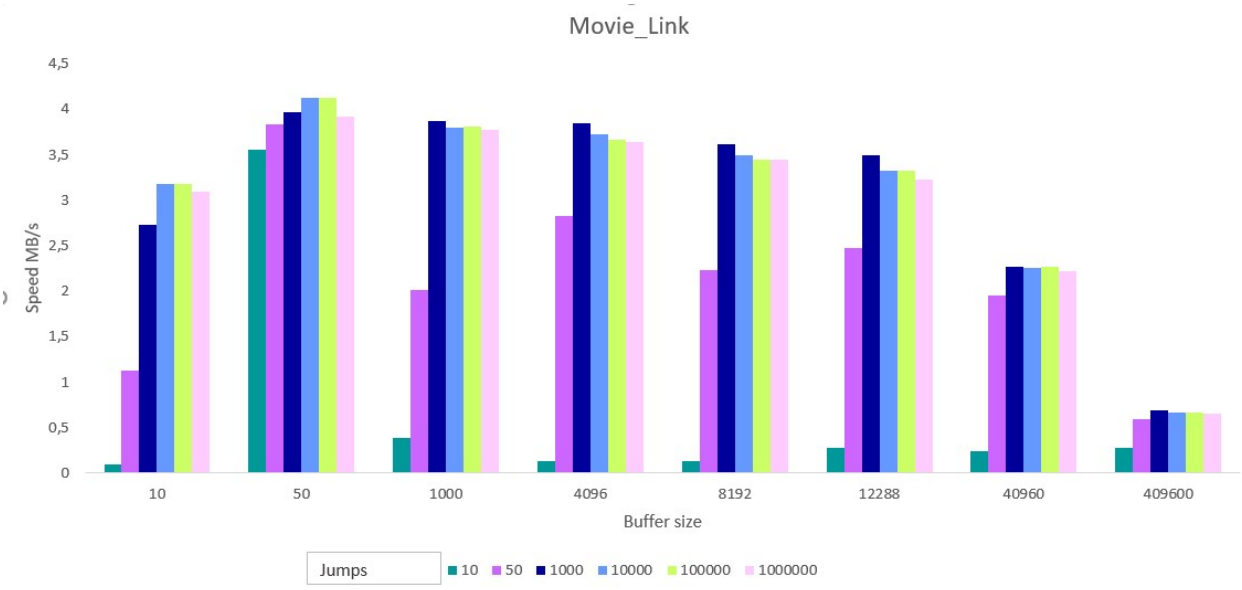

Figure 11: Random reading method3 on movelink.csv

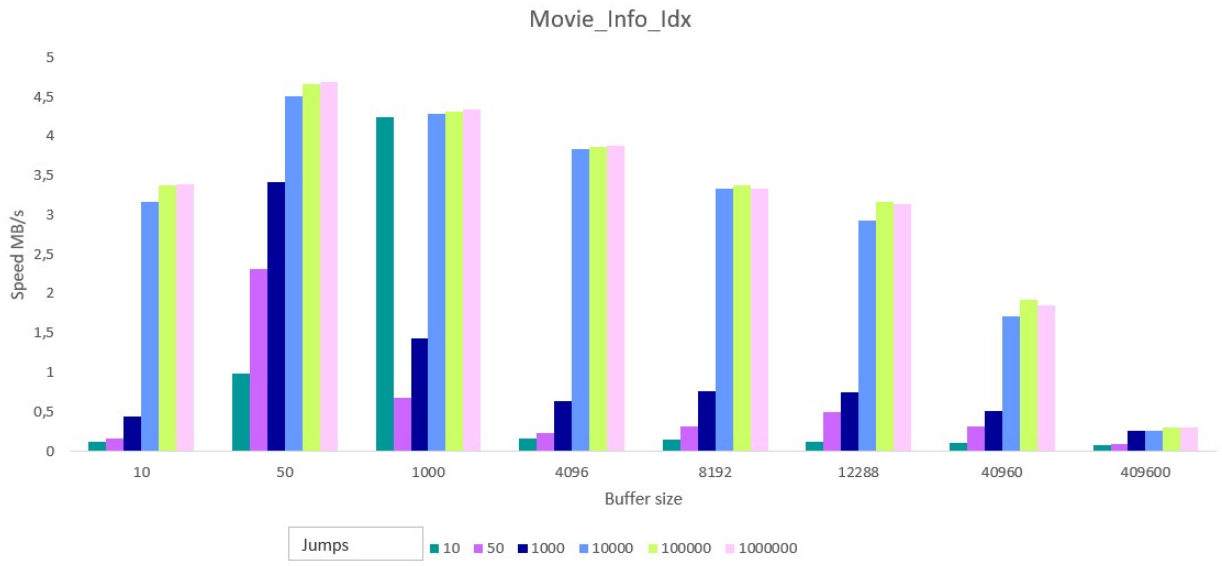

Figure 12: Random reading method3 on movie_info_idx.csv

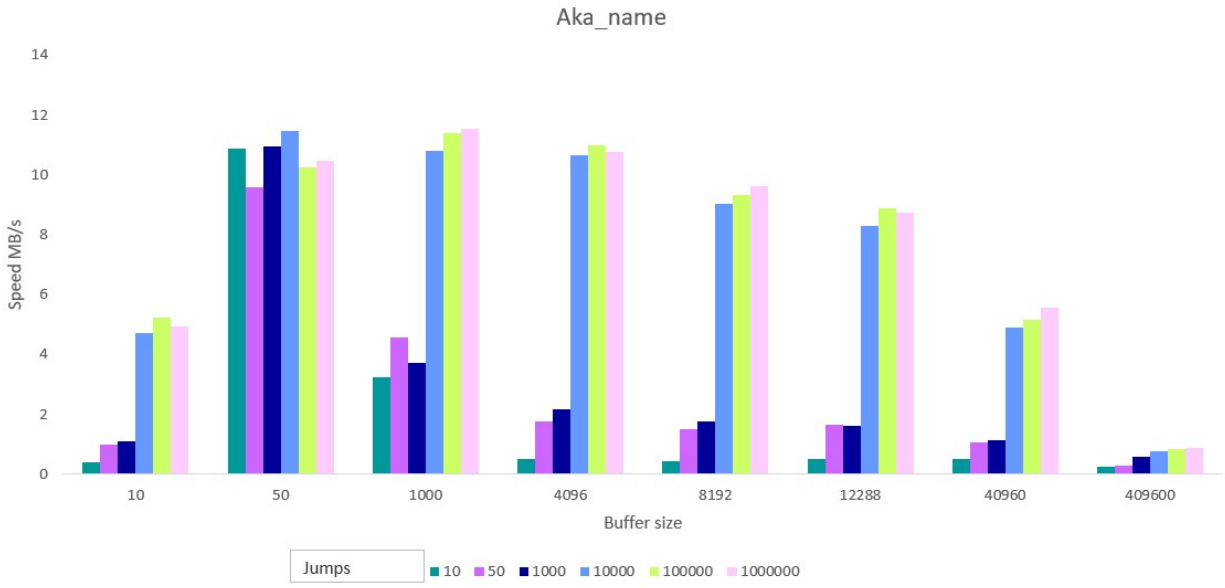

Figure 13: Random reading method3 on aka_name.csv

For the first file (`role_type.csv`), the results are similar to the first two methods: no matter the buffer size, the the overall performance doesn't change much despite the number of jumps varying. This is expected since the file is so small, and as explained before, the probability of ending up in a region read before is higher.
The other graphs have a different behaviors but keep some similarities. For the buffer size, the pattern is similar to the one from sequential reading : the performance is quite poor at the start and when increasing the buffer size, it gets better until reaching an optimal value, after that it decreases again (which we explained by the excessive amount of heap memory allocation done) . The peak in performance is reached around

50-1000 size of buffer which is expected as we consider 50-1000 to be the average size of the read content when random reading. From these graphs, we can conclude that the optimal for method 3 for random reading is a buffer size of 50.
For the influence of the number of jumps, we notice the same behavior as in the two previous methods; increasing the number of jumps increases the performance, due to caching, especially in the bigger files.
Regarding the file size, the same discussion made for the first and second implementation stays valid (caching).
Note that the bar peaking from the `movie_info_idx.csv` graph for 10 random jumps and the size of the buffer 1000 could be an experimental error since the behavior is quite irregular and far from what was expected.

Method 4:

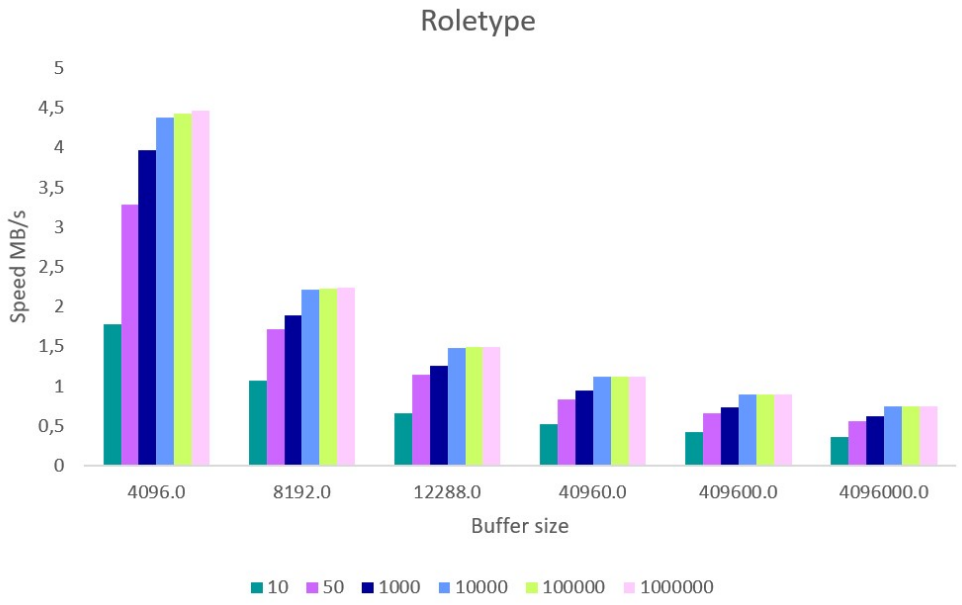

Figure 14: Random reading method4 on role_type.csv

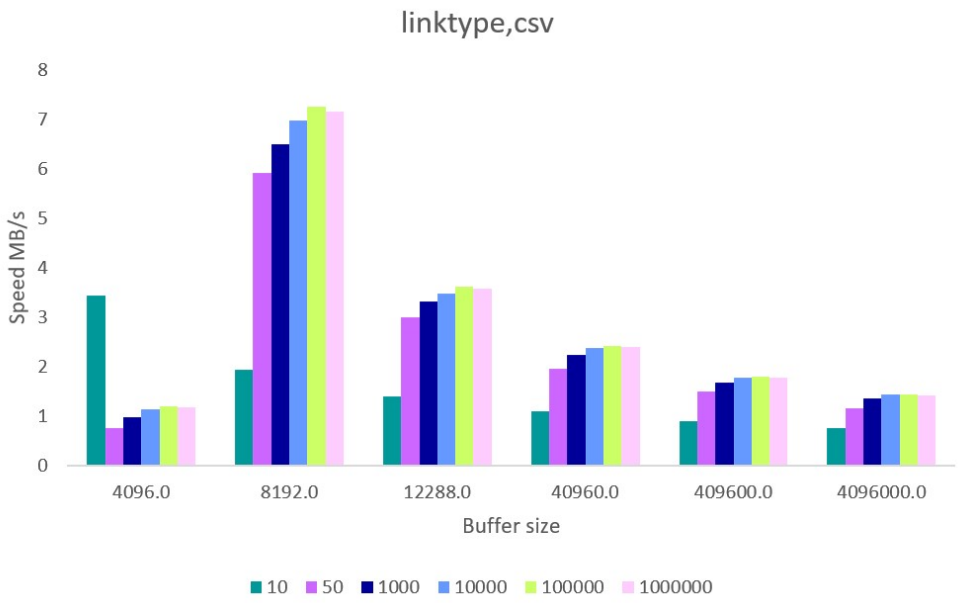

Figure 15: Random reading method4 on linktype.csv

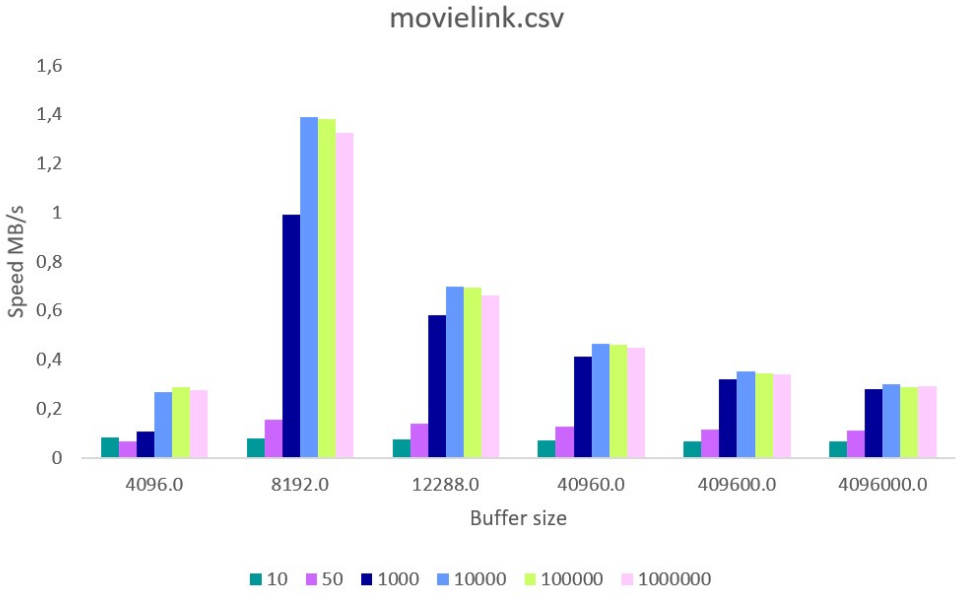

Figure 16: Random reading method4 on movie_link.csv

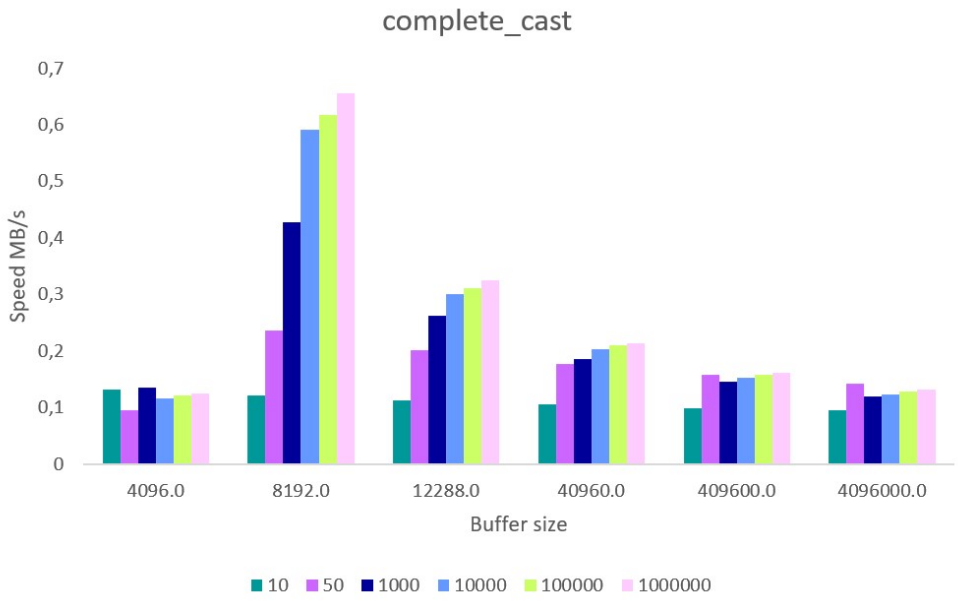

Figure 17: Random reading method4 on complete_cast.csv

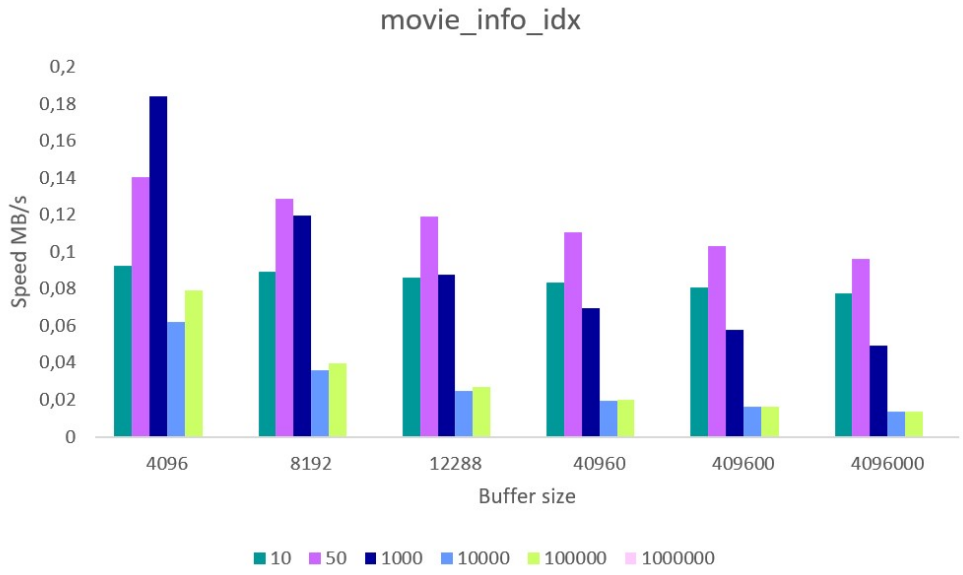

Figure 18: Random reading method4 on movie_info_idx.csv

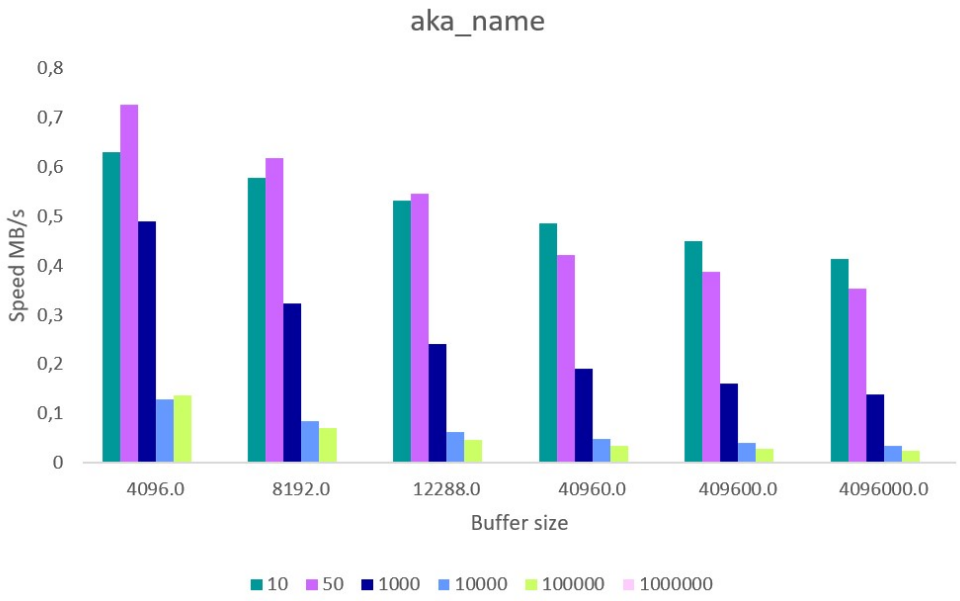

Figure 19: Random reading method4 on aka_name.csv

Note that, for our two biggest files (`aka_name.csv` and `movie_link.csv`), we did not perform tests on k=1000000 since it took an excessive amount of time and resources, and that the behavior can already be seen at that point.

We can already see the impact of giving really big values to B when reading; the performance decreases. Indeed, as explained for sequential reading, this could be due to the cost of mapping a region this big, with all the overhead it could cause, making it much slower. For `role_type.csv`, we notice that the buffer size that performs best is the one corresponding to one page; indeed it is the minimum possible value, and since the file is only 160 bytes long, loading the page once into memory is sufficient to read all the possible lines. However, the next file `linktype.csv` behaves in an unexpected way, the optimal buffer size is of 2 pages, but the file is only 261 bytes long, not that much bigger than the previous one. We couldn't really explain the origin of such behavior.
Now talking about the files in general we can see two kinds of behaviors depending on whether the files are small (role_type-movie_link-complete_cast) or very big(movie_info_idx-aka_name).If the files are rather small, the performance increases when increasing the number of jumps k, which could be justified as before by the higher probability of ending on the same page (or even region if caching plays its part and also loads the surrounding pages). However, when the files are of bigger size, the probability gets smaller, and the cost could grow significantly due to the faulting, which is costly, that will be repeated each time the `readln` function is called.

Now, to determine which implementation is the most performant, we can take a single buffer size (4096 to be relevant with the mapping) and a set number of jumps and see how each method performs.

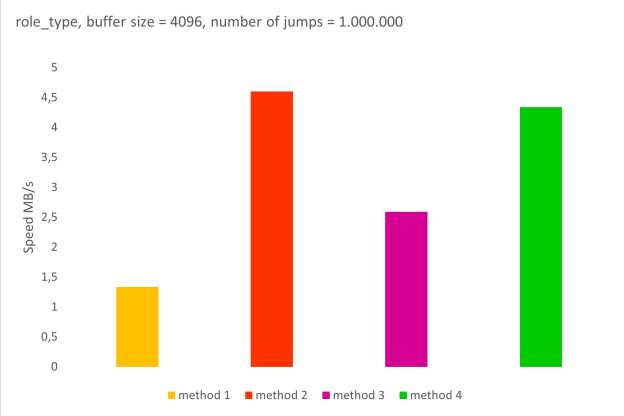

Figure 20: Comparison on role_type.csv

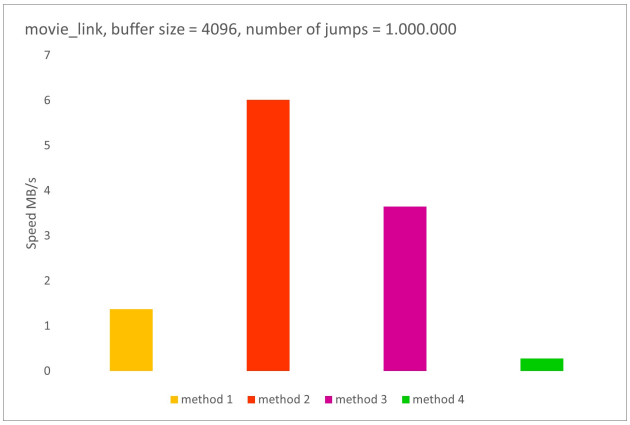

Figure 21: Comparison on movie_link.csv

We can see that the mapping does pretty well on the smallest file, however, for bigger files, the performance drops drastically, but the second implementation stays as performant.

Section Conclusion : Since so many variables were tested in this section, we can safely say that method 2 takes the crown for random reading. We realised method 3 and 4 operates differently when increasing the size of the files. Method 3 increases its read speed while method 4 decreases its read speeds. Visibly, allocating memory for a buffer size costs less than mapping the same size. Despite our research, we could not explain with a high enough level of confidence this behaviour.

6 Experiment 1.3 : Combined reading and writing

For this third experiment we now have to combine both input and output streams to create a function called `rrmerge` that will take as input k files and read one line of each file in a round robin way until all files have been read, and for each read file the content is written in a file using an output stream. There are 8 implementations of the method : 4 combining the InputStream3 (being the second best result for the random reading) with the different output streams, and 4 other combining the InputStream2 (the best implementation of the sequential reading) with the four output streams. Since method 2 was the best for sequential and random reading, we decided to take the second best method from the random reading for comparison sake.
In order to do this, we created two classes : `RrmergeMethod3` and `RrmergeMethod2`, that each have the methods called `rrmerge1`,`rrmerge2`,`rrmerge3` and `rrmerge4`.
Each method takes as parameter the name of the output file, the number of files passed, and the name of each file (in addition to the buffer sizes for methods 3 and 4). It starts by creating an instance of the corresponding output stream and creating the output file, then in order to read the files in a round-robin way, a list of InputStreams is created and allocated space corresponding to the number of files*the size of the InputStream. Then for each file an InputStream is created, the file is opened and that InputStream is stored in the list. Once all the streams are stored in the list we can start reading and writing ; we enter a while loop that continues as long as the number of files is different from 0, and for each file we call the `readln` method and verify its size, if it is different from zero we write the content to the output file using the output stream, in the other case it means we have reached the end of file ; in this case we decremente the int value of the number of files by 1, and update the list of InputStreams by closing and deleting the InputStream associated to the file that we finished reading. We continue the same way until all files have been, which means that num will be equal to 0 and we will exit the while loop.The function ends by freeing the list of InputStreams.
Before testing these two classes, we can already see what kind of results to expect:

Let k be the number of files to merge and N_k the size of each file

- RrmergeMethod2 :
  - rrmerge 1 : Since reading is done with method 2 and the writing with method 1 on each of the k files, the total cost would be $\sum_k \lceil \frac{N_k}{1000} \rceil + \sum_k N_k$
  - rrmerge 2 : Same but with the method 2 for writing : $2*\sum_k \lceil \frac{N_k}{1000} \rceil$
  - rrmerge 3 : B is the size of the writting buffer used in method3 : $\sum_k \lceil \frac{N_k}{1000} \rceil + \sum_k \lceil \frac{N_k}{B} \rceil$
  - rrmerge 4 : Here the cost for writing to the file sequentially would be depending on the page size , similary as for the sequential reading. The cost is the : $\sum_k \lceil \frac{N_k}{1000} \rceil + \sum_k \left\lceil \frac{N_k}{pagesize} \right\rceil$
- RrmergeMethod3 : Follows the same idea, but insted of $\lceil \frac{N_k}{1000} \rceil$ for reading it is $\lceil \frac{N_k}{B} \rceil$
  - rrmerge 1 :$\sum_k \lceil \frac{N_k}{B} \rceil + \sum_k N_k$
  - rrmerge 2 : $\sum_k \lceil \frac{N_k}{B} \rceil + \sum_k \lceil \frac{N_k}{1000} \rceil$
  - rrmerge 3 :$2*\sum_k \lceil \frac{N_k}{B} \rceil$
  - rrmerge 4 :$\sum_k \lceil \frac{N_k}{B} \rceil + \sum_k \left\lceil \frac{N_k}{pagesize} \right\rceil$

From the cost estimations, and knowing that the method2 performs better for sequential reading, we can already expect the RrmergeMethod2 to perform better than RrmergeMethod3. As for the writing, we could expect the mapping to do better according to the formulas but as we saw for before, the faulting and overheads slow the process a lot. As always, the 1st implementation should perform worst, so the candidates for best writing method should be 2 or 3, if the buffer size of method3 is not very big.
All the tests done for RrmergeMethod3 were done using the same size for the reading buffer (1000, which was the optimal value for reading with the third implementation in sequential reading).

The first test consisted in combining a file with itself. The output files must then be a line followed by its copy.(verifying that the methods gave the proper results are much more easier since the number of characters are double in the output file). Then we tried merging three times the same file, and also merging different files (2 and 3).
Seeing how slow the first method (rrmerge1) performed for both classes during the first tests on small and intermediate file sizes, and how long it took to run a test on larger files using it, we decided to not test it on big files. But we can affirm without any loss of generality that the 1st reading implementation is by far the worst.

The testing wasn't conducted on the biggest files of the dataset; since each time we run a test with a certain configuration, an average over four iterations is conducted, and running the experiment once on a big file is really slow, doing it 4 times in a row takes a lot of time. So we tested all of our methods on these four files : `role_type.csv` (160 bytes), `movie_link.csv` (656,584 bytes), `complete_cast.csv` (2,414,495 bytes) and `movie_info_idx.csv`(35,335,875 bytes)

The following graphs show the average time as a function of the buffer sizes, not the transfer rate like previously!

RrmergeMethod2 :

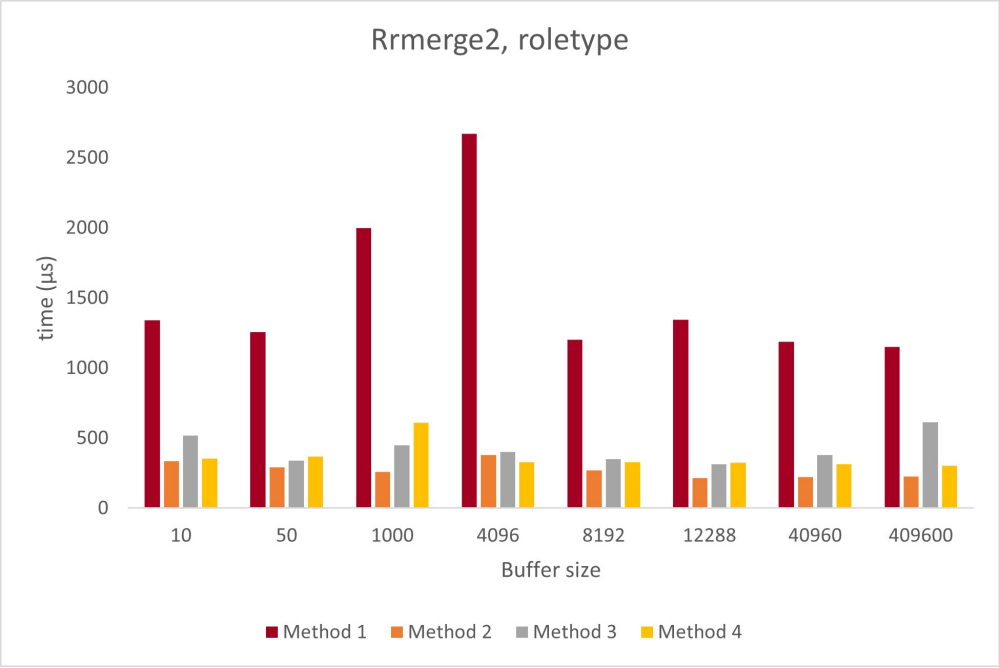

Figure 22: rrmerge 2 times on role_type.csv

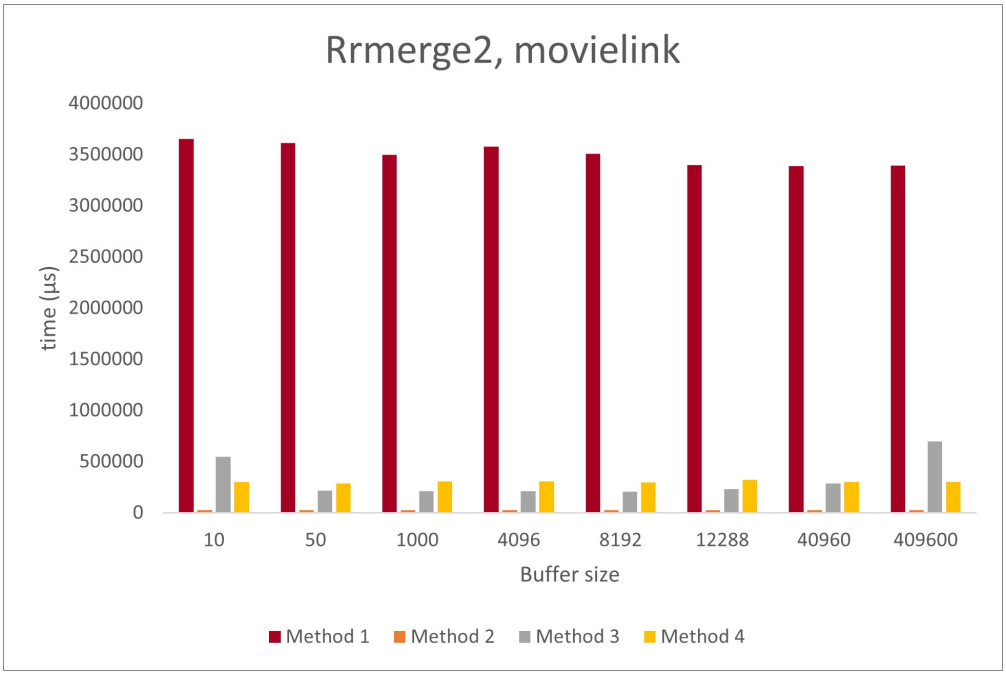

Figure 23: rrmerge 2 times on movie_link.csv

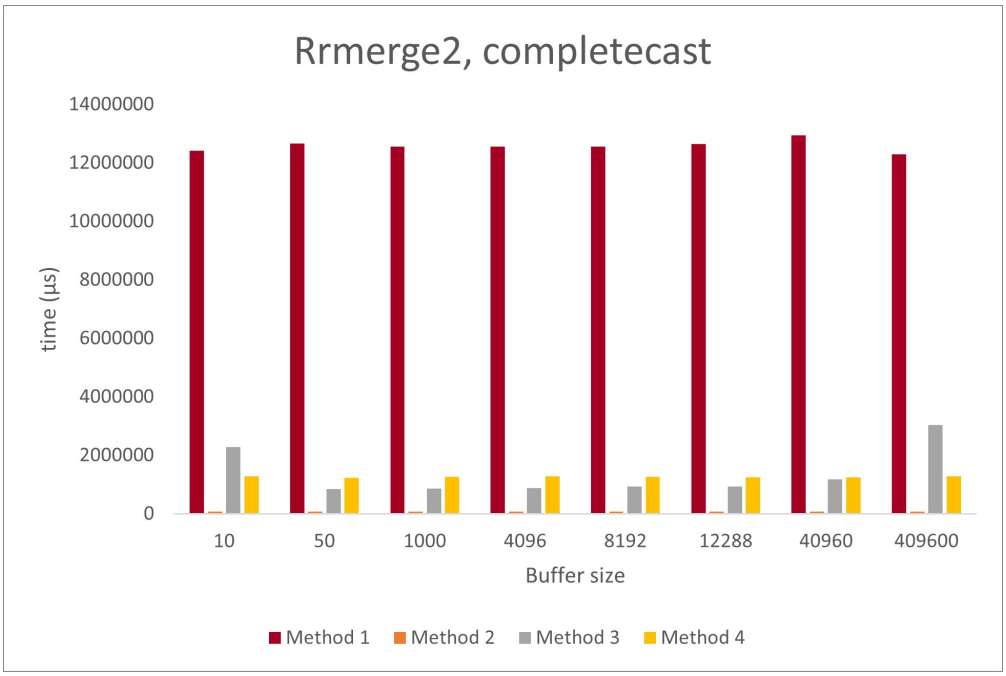

Figure 24: rrmerge 2 times on complete_cast.csv

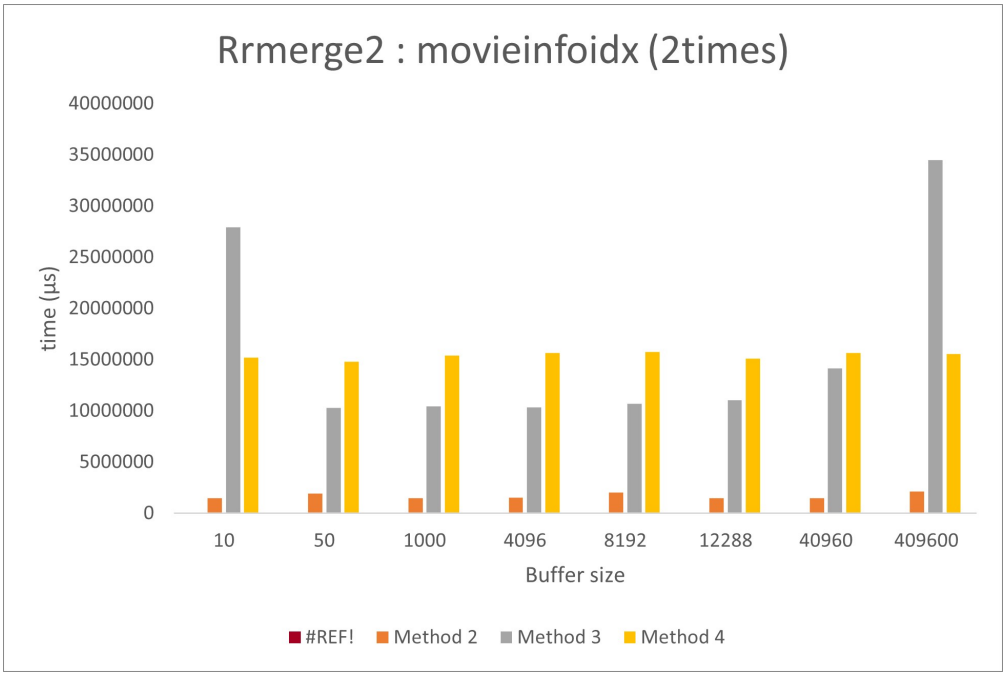

Figure 25: rrmerge 2 times on movie_info_idx.csv

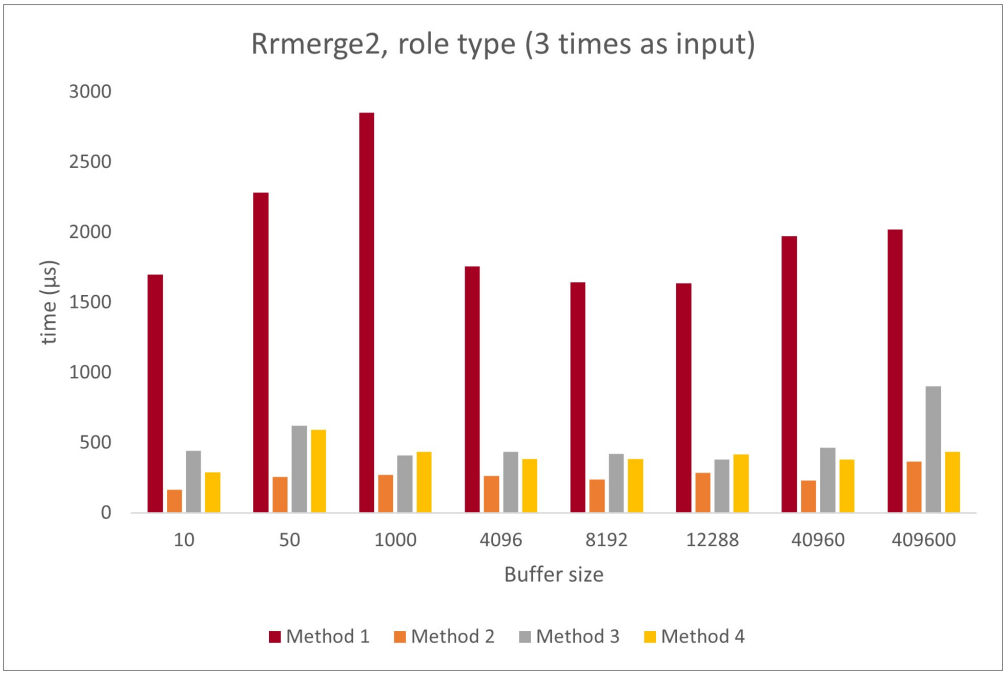

Figure 26: rrmerge 3 times on role_type.csv

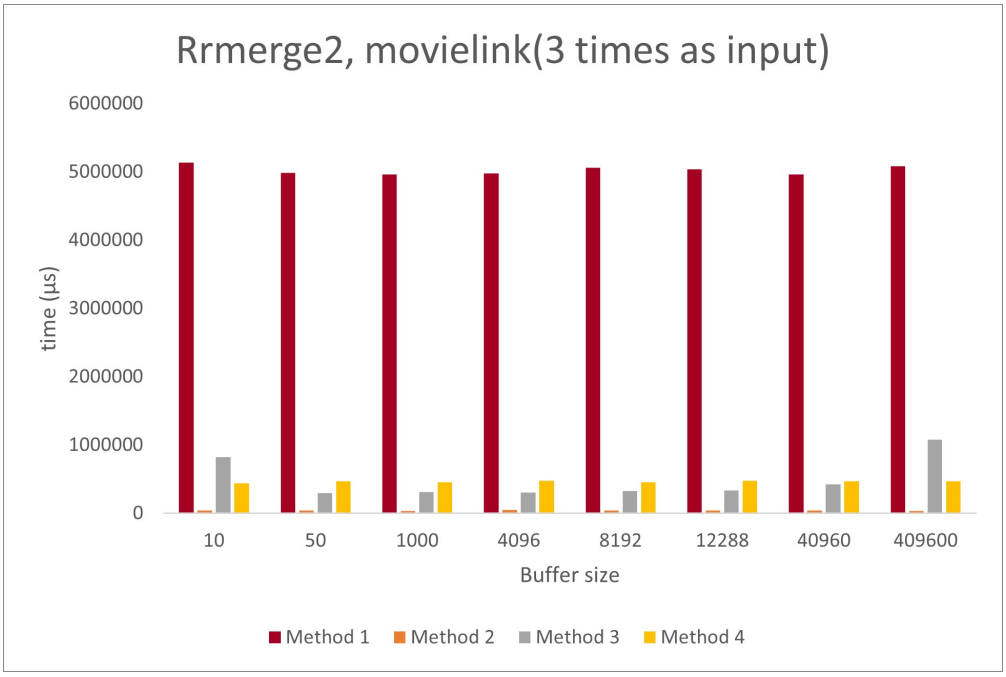

Figure 27: rrmerge 3 times on movie_link.csv

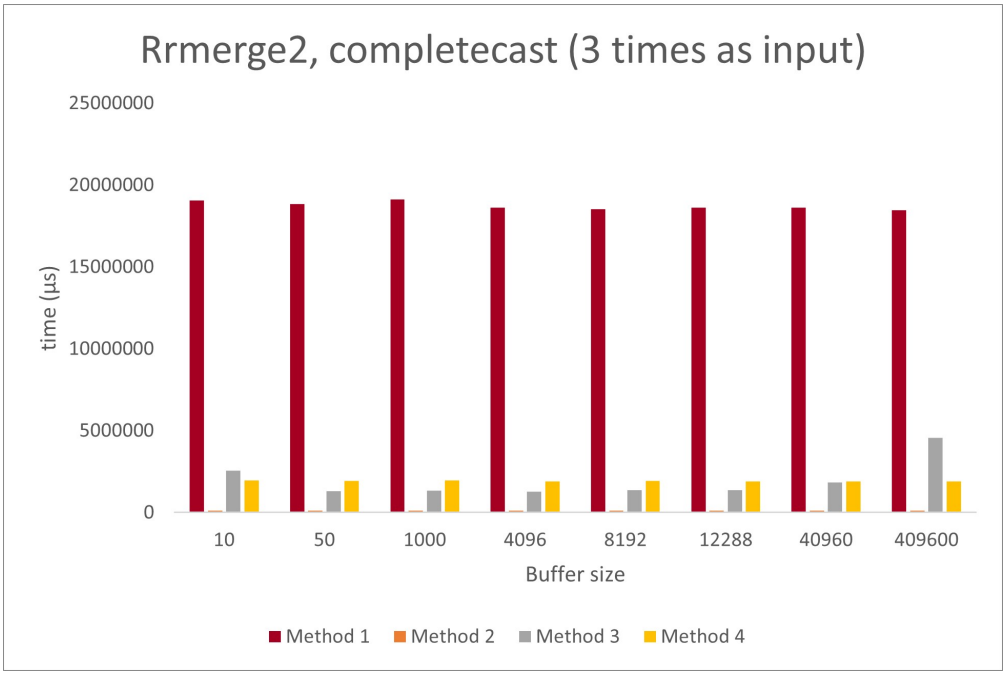

Figure 28: rrmerge 3 times on complete_cast.csv

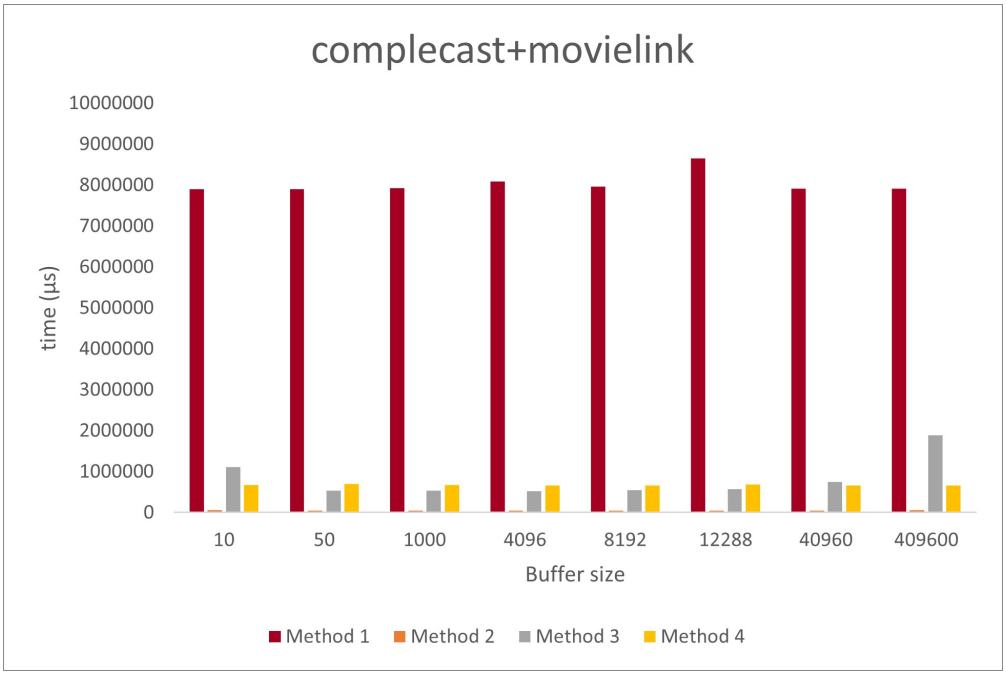

Figure 29: rrmerge2 with movie_link and complete_cast

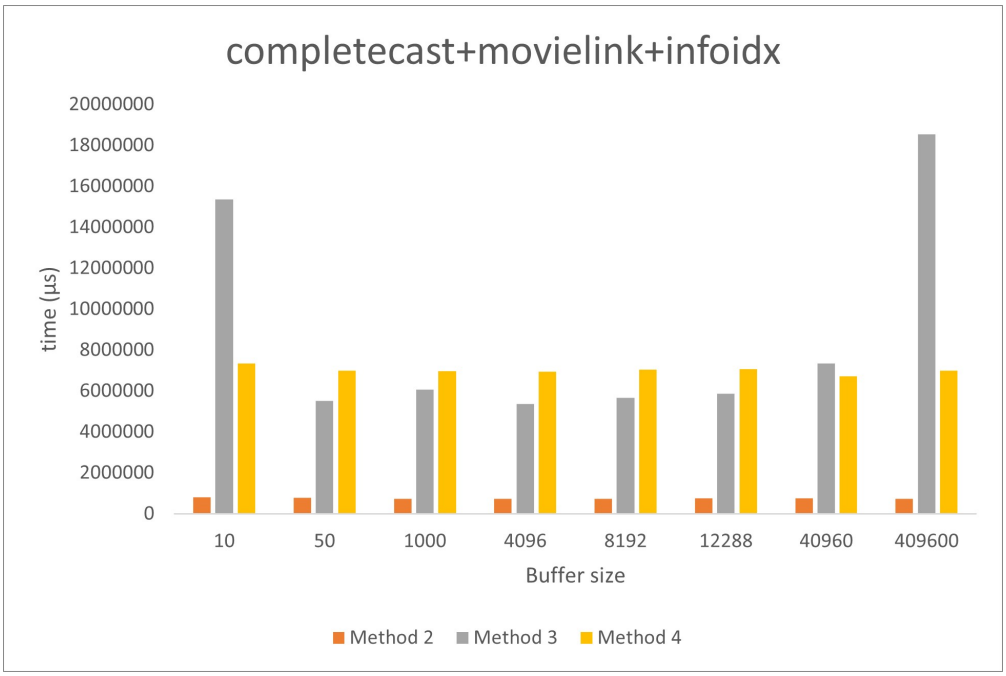

Figure 30: rrmerge2 with movie_link infoidx and complete_cast

RrmergeMethod3 :

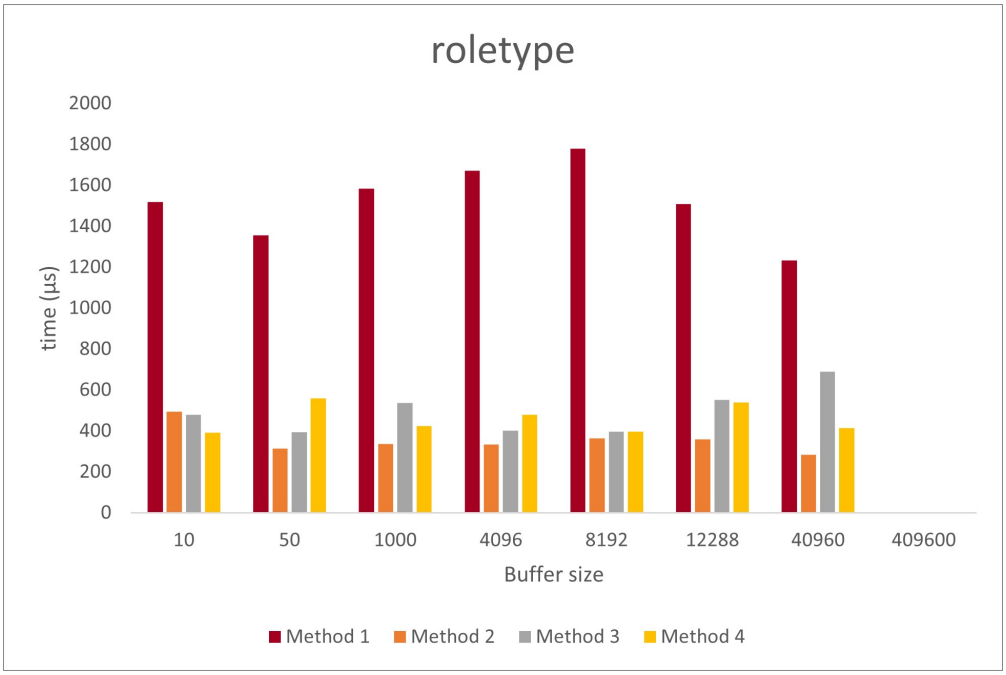

Figure 31: rrmerge 2 times on role_type.csv

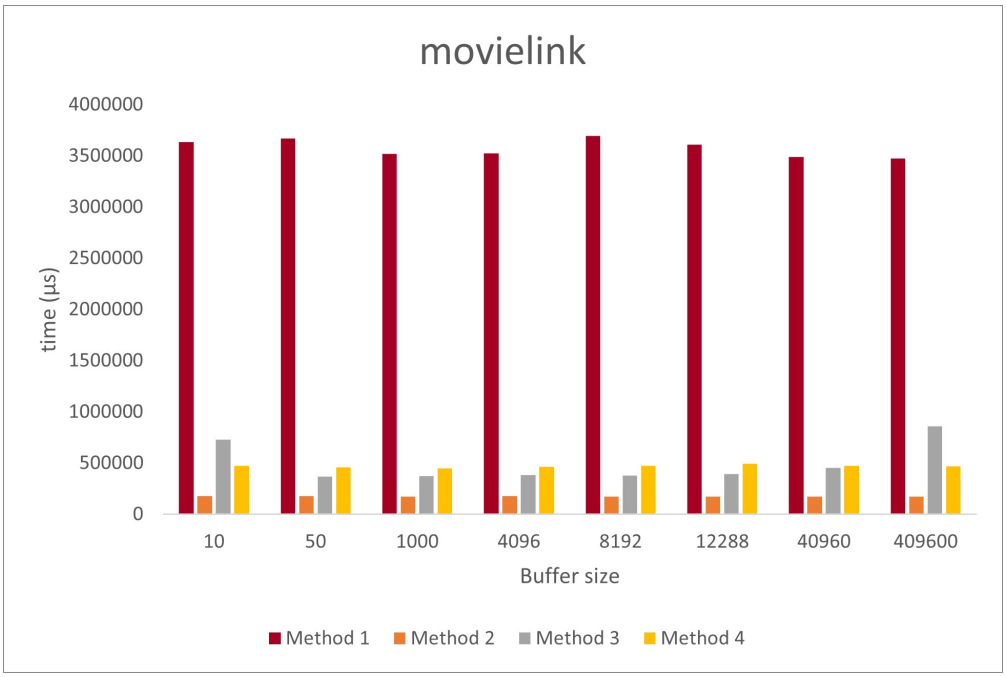

Figure 32: rrmerge 2 times on movie_link.csv

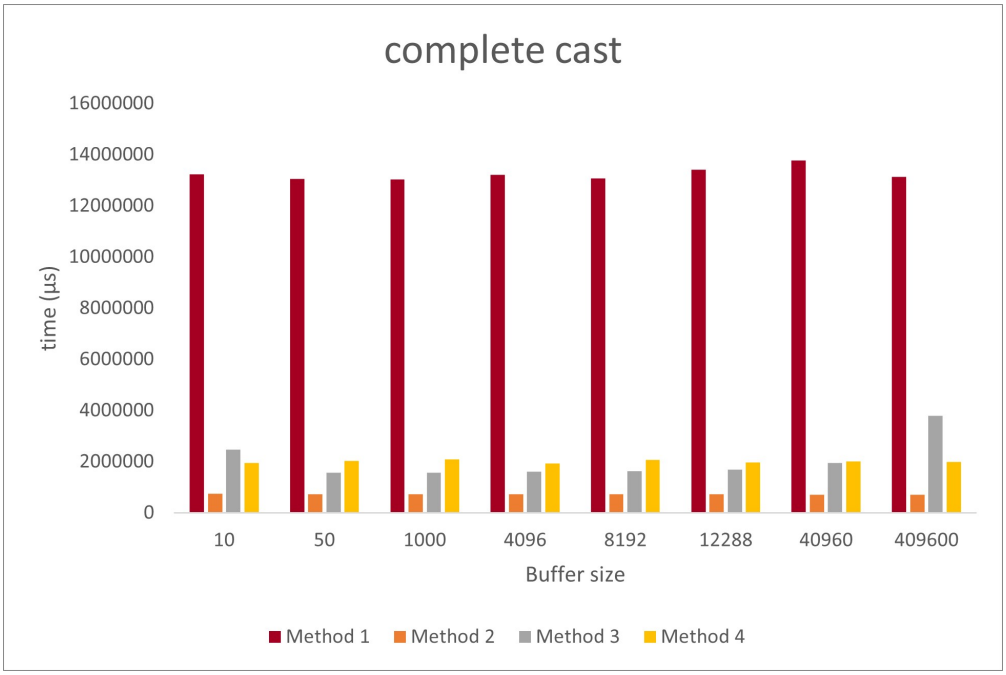

Figure 33: rrmerge 2 times on complete_cast.csv

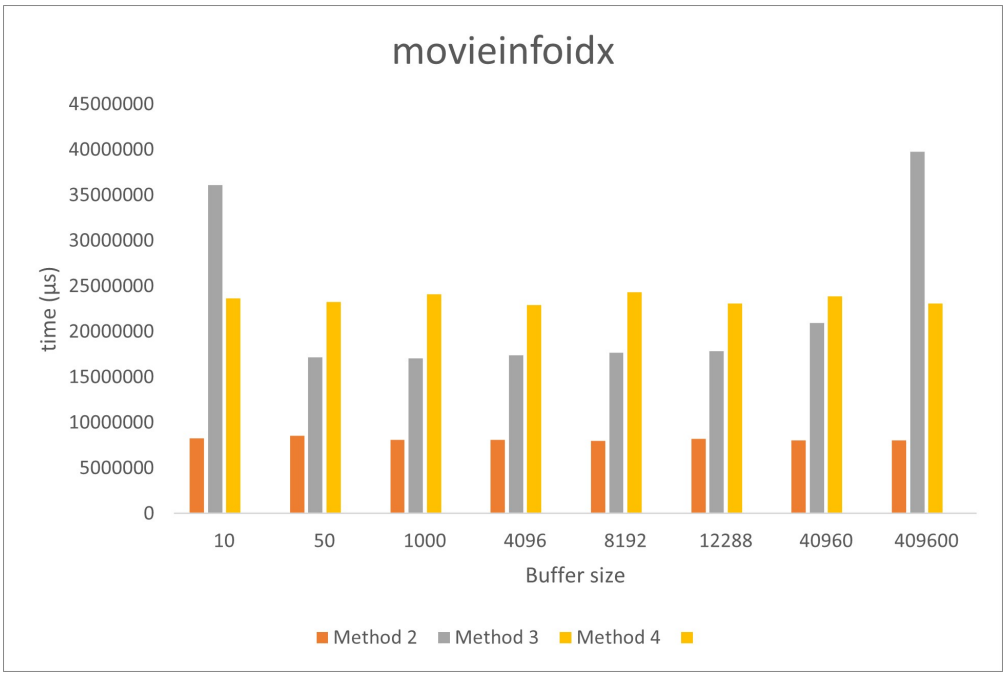

Figure 34: rrmerge 2 times on movie_info_idx.csv

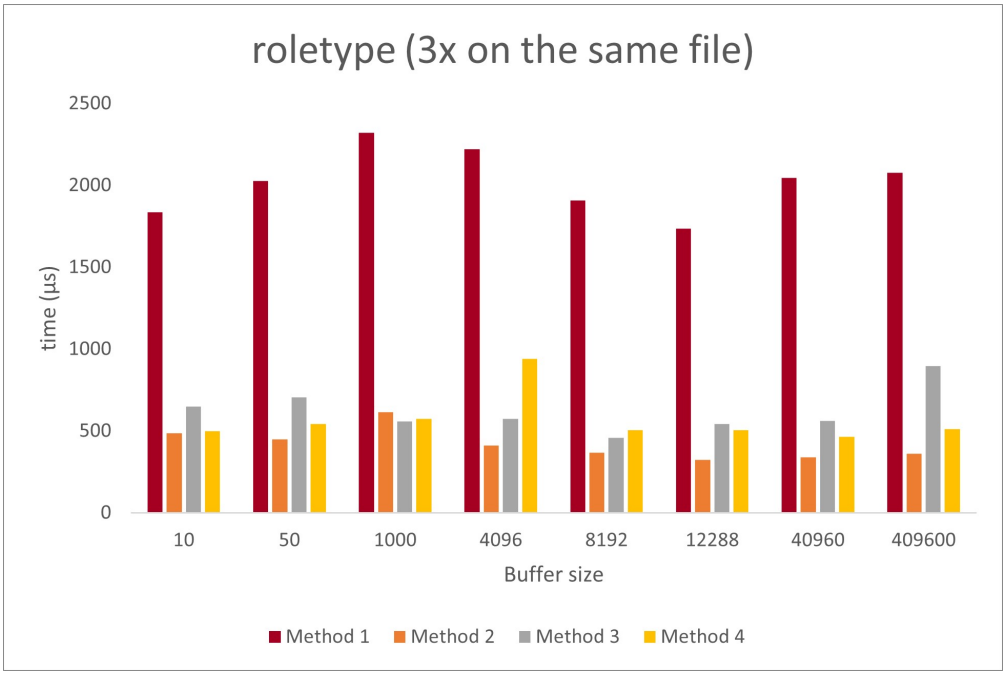

Figure 35: rrmerge 3 times on role_type.csv

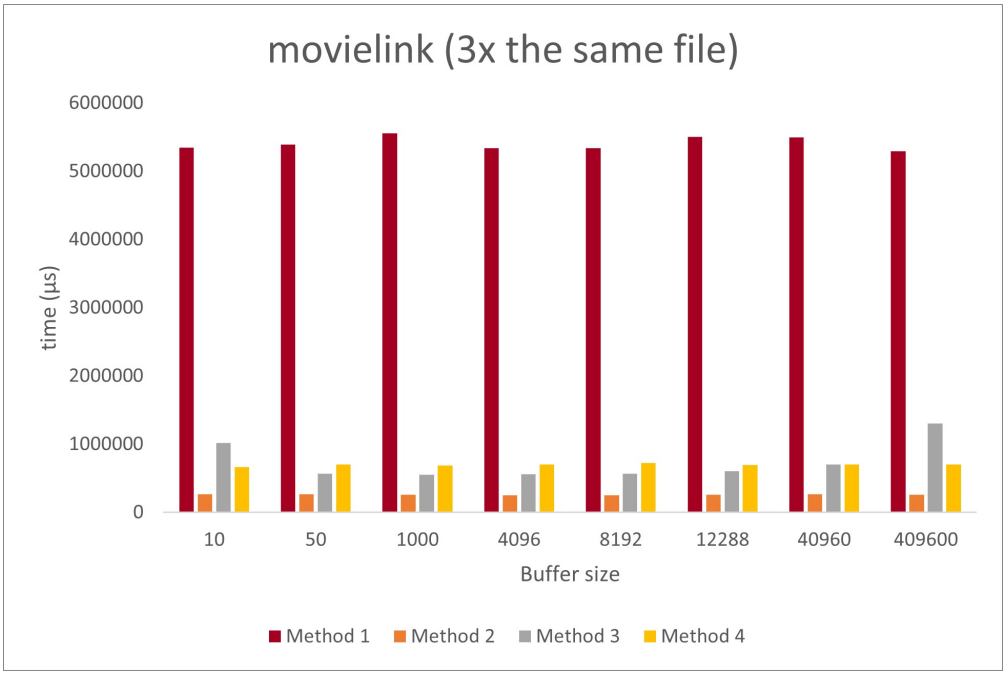

Figure 36: rrmerge 3 times on movie_link.csv

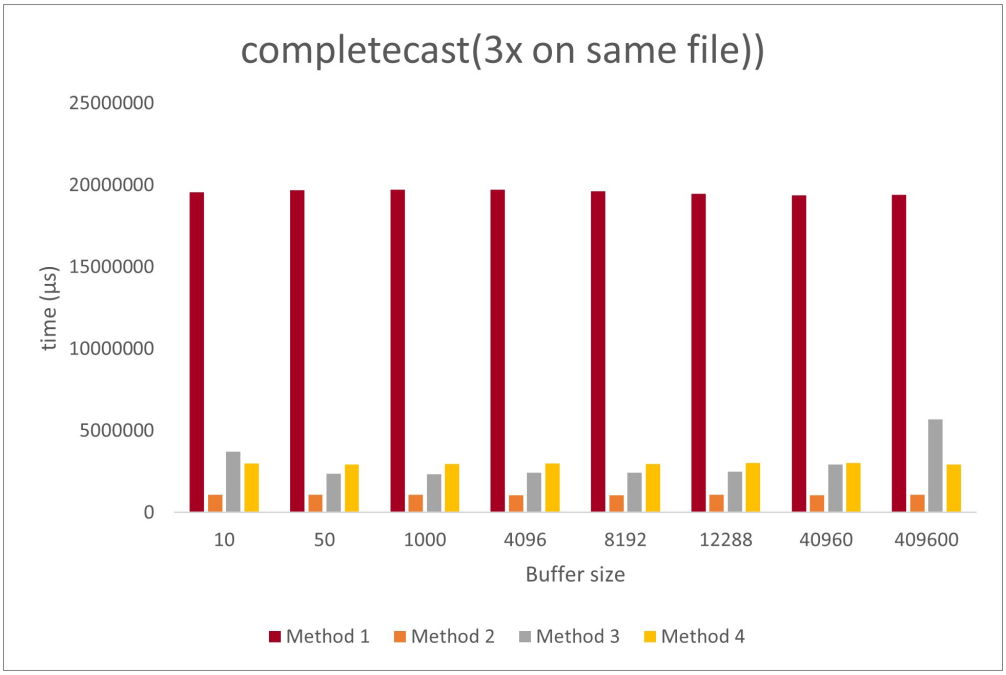

Figure 37: rrmerge 3 times on complete_cast.csv

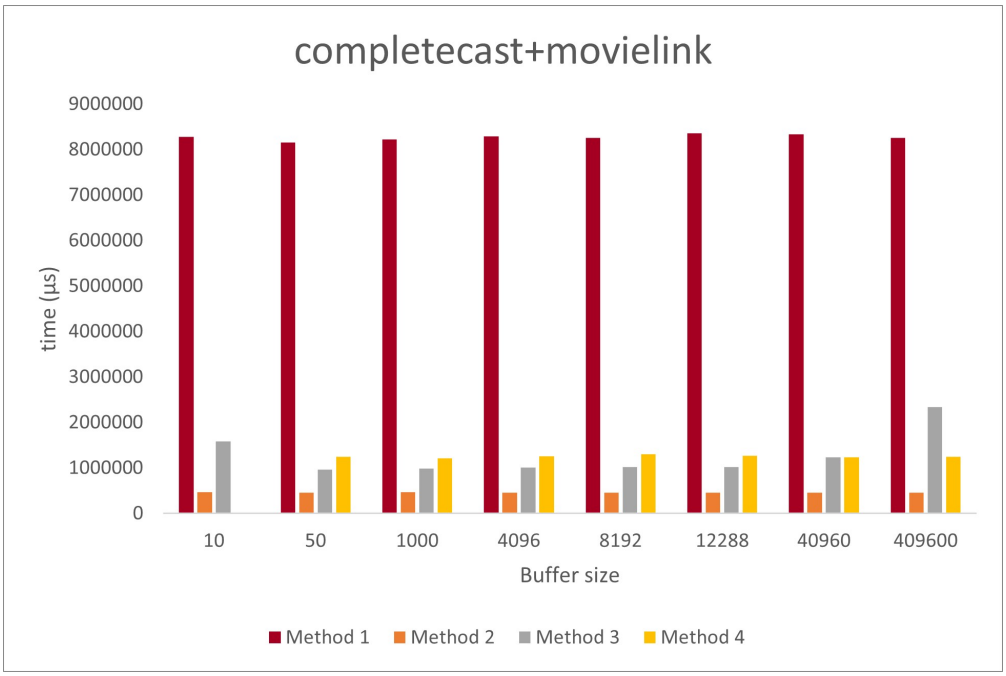

Figure 38: rrmerge on movie_link and complete_cast.csv

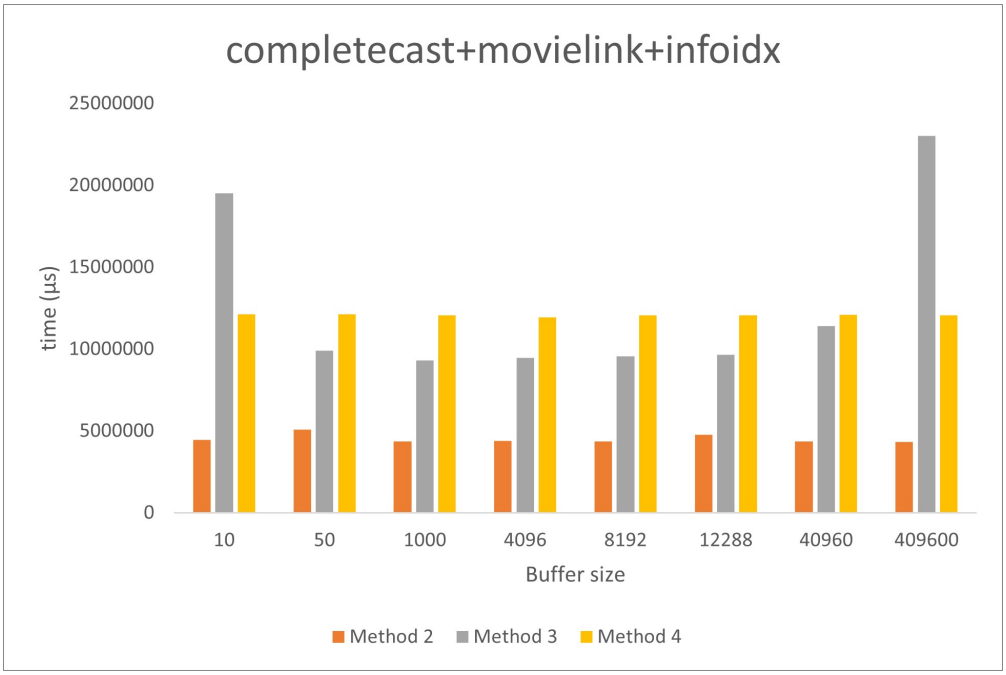

Figure 39: rrmerge on movie_link movieindex and complete_cast.csv

From a first look at the graphs we can, as said before, already conclude that writing with method 1 is the most costly of all : overall, the performance is pretty much the same.
The cost for writing with the 2nd implementation is also constant, which was expected since the buffer size is constant.
Now looking at the writing methods 3 and 4 : the method 3 always has the same behavior : it starts with a high cost, then for buffer values around 50-1000 it reaches its optimal value and after that the cost rises again; the explanation here is the same as the one for reading, for very small buffer sizes, there is more than one I/O to perform for each call to readln, however when the buffer size augments too much, the cost rises due to the excessive time taken to allocate such a big buffer size on the heap and free it at each call. If we compare the performance of rrmerge3 between the two classes, the RrmergeMethod2 class performs better, that is due to the fact that the read with method2 is far more performant, even with the optimal buffer value.
For the method 4, the values below 4096 are rounded up to match 4096 (minimum pageable). Memory mapped writing here performs better than the method3 for very small and very big buffer sizes, but overall it takes longer than the method2, this could be due many factors ; applying the `ftruncate` function every time we read a line instead of using it only when we map a new multiple of page size , or the call to `lseek` to get the file size, which could increase the total time as it is an additional I/O, in addition to the faulting, overhead and possible CPU time cost that was discussed for memory mapped sequential reading .
Here we can notice that for the big page sizes that the performance is constant, when we could expect a very high time due to the important cost for mapping such a large region; this is due to an optimization choice made from our part when coding the writeln method for mapping. Whenever the number of pages to map given as a parameter is much greater than the *needed* pages to write the string, the mapped region's size is equal to the pagesize (4096) which is why the cost almost does not vary. We could have chosen to set the size to map to the multiple just below, but if it was still too big we would have needed to adjust it once again in a recursive way which would have ended up wasting even more time.
Here again the first class performs better than the second, for the same reasons.
Now, what happens when we merge more than 2 files ? The cost seems to increase linearly. As seen during

the predictions, it is a sum of the costs, and since we are merging the same files now three times, the cost is approximately 1.5 times the previous one .
When the files being merged are different, nothing much changes, the behavior of the different functions stays the same, and here also, the rrmerge from Method2 outperforms the rrmerge from Method3.

7 Multi-way merge sort

The fourth and last experimentation is the merge sort algorithm. To implement it, we created a class `mergesort.cpp`.
In this class, we can find the method extsort which can be described in two steps : the first step is reading the whole file, storing the lines accordingly and putting them in files of more or less M bytes, the second step is the merging phase where we merge d files into one recursively. The files resulting from this first step are a little bit more than M bytes long because the condition to stop is when the file size exceed M bytes, so a line will be added to the file, resulting to the file being slightly bigger than M bytes.

In the first part, we create the vector `addresses` containing the addresses of the files created and the priority queue `mylist` containing the lines objects. Indeed, we created a structure called `Line` which has the following attributes :

1. word : a vector of string, which represents an entire line, each item in the vector being the word of a column. To split the line into a vector of strings, we used the method `split` that splits based on the separator `","`.
2. line : a string that is the line itself, used to write the whole line.
3. kchar : the k-th word, which is thus the word contained in the k-th column.
4. index : a string being the address of the file from which the line is from, used to make the link between the priority queue and the vector of addresses.
5. toseek : an integer defining where the line is in the file.

Thus, until the end of the file, we read M bytes, push them in the priority queue, create a new file (outputx.txt where x is the number of the file) and add it to the vector of addresses and finally we write the lines sorted into the new file by taking the top of the priority queue until it is empty. As a reminder, a priority queue is a data type which contains objects that are sorted with a priority. This priority is defined in our implementation by the means of `comparator`, which is a Compare type that compares strings in the lexicographical order. Our implementation thus sorts the columns based on the attribute `kchar` of the `Line` structure.For example, if we have the words "1", "9" and "10", our sorting method will return this order "1", "10", "9" as it looks at the first character and not the value of the integer.

At this stage, we have $\lceil \frac{N}{M} \rceil$ files. The second part of the algorithm is to merge those files by buckets of d files.
We have to merge until only one file remains in the vector `addresses`. For each pass, we compute the number of files there is and the number of buckets of d or less files we must merge. We then have a for loop that loops over the number of buckets we have to do for one pass.
For each bucket of d files, there are two cases : either the number of files is exactly d or it is less than d, which means we only merge the remaining files. Either way, we read the first line of each files of the buckets and push them in the priority queue `mylist`. Then, while the priority queue is not empty, we take the top of the priority queue. This line is thus the one we must write first in the merging file, we then read the next line in the file from which the line is from. If the line is not empty (its size is not 0), we add the next line to the priority queue and remove from the priority queue the old top line that we just processed. In the case where we are at the end of the file, we delete the file (as it is an intermediary file), remove it from the vector `addresses` and then pop the line just processed in the priority queue. When the priority queue is empty, which means that all the files of the bucket have been are merged, we add the created file in the vector of addresses.
We process in a similar fashion for the cases of d or less files.
It must be noted that the way we implemented the merge sort algorithm might require more I/Os than

needed. Indeed, we chose to open and close the files each time we need to read or write a line instead of having too many streams open at the same time.

7.1 Expected behavior

Before conducting the experiment, we can already try to predict the cost of the algorithm in terms of I/Os:

Firstly, the cost of the first step which is reading the file ($\lceil \frac{N}{M} \rceil$) and creating files containing M bytes of it ($\lceil \frac{N}{M} \rceil$). The total cost of this step is thus : $2 * \lceil \frac{N}{M} \rceil$

For the merging part, we based our equation on the cost seen in the course. Indeed, we have seen the sort-based set union which works as follow : on the first pass we read M blocks in the input relation, sort them and write the result to disk. In the following passes, we merge M sub-lists into one. The size is thus always multiplied by M and the number of files divided by M. The cost of the algorithm is $(2 * B(R)) * \log_M B(R))$, with `B(R)`the size of the file in terms of blocks. The $2 * B(R)$ in the equation stands for the reading and writing at each pass, the $\log_M B(R))$ is the number of passes that we do. The difference in our case is that the first step and the merging phase use different parameters while the course only uses the parameter M. We thus start with $\frac{N}{M}$ files. We then know that there will be $\log_d \lceil \frac{N}{M} \rceil$ passes as we divided the number of files by d each passes. We then multiply the number of passes of the merging phase to the cost of reading and writing.
In the end, we obtain this cost :

$$(2 * \lceil \frac{N}{M} \rceil) * (1 + \log_d \lceil \frac{N}{M} \rceil)$$

7.2 Experimental observations

We tested on the following files, sorted on the fourth column or the last column if it does not contain at least 4 columns :

- role_type.csv (160 B)
- link_type.csv (261 B)
- info_type.csv (1928 B)
- movie_link.csv (657 KB)
- complete_cast.csv (2.4 MB)
- movie_info_idx.csv (35,3 MB)

We also tested the algorithm on the file `movie_link.csv` already sorted to verify if it makes any difference. The graphs here present on the x-axis the parameter M, on the y-axis the average time in micro-seconds (taken on an average of four runs) and the different colors represent different values of d.
For the smallest files (role_type.csv,link_type.csv and info_type.csv), we had to choose other values of M and d as having values that are too big for them would not have been relevant. The tables with the actual results can be found in the appendix.

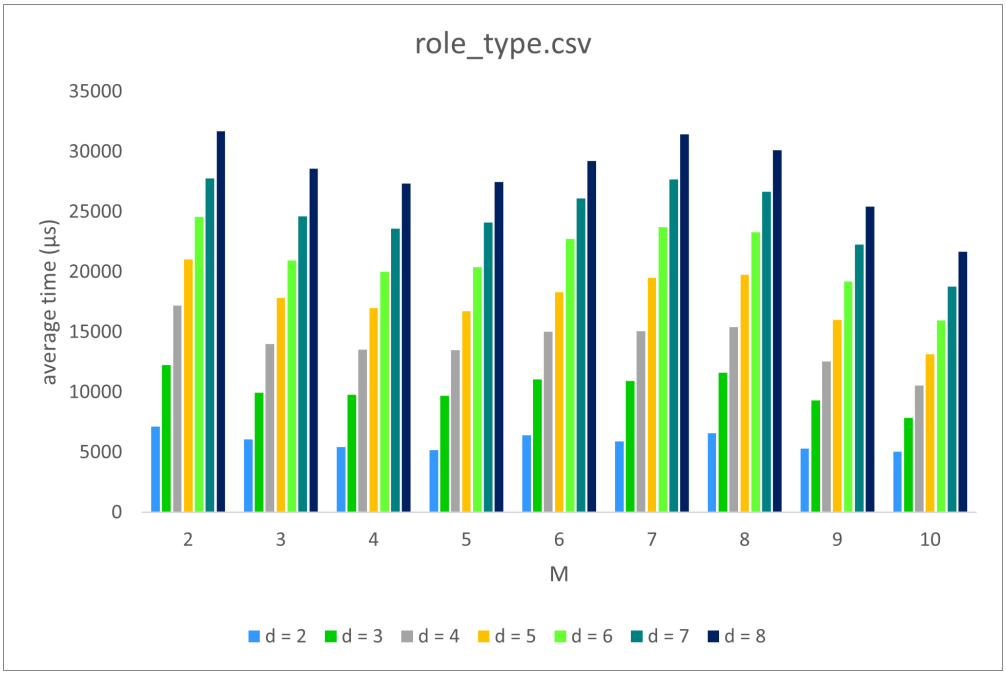

Figure 40: Merge sort on role_type.csv

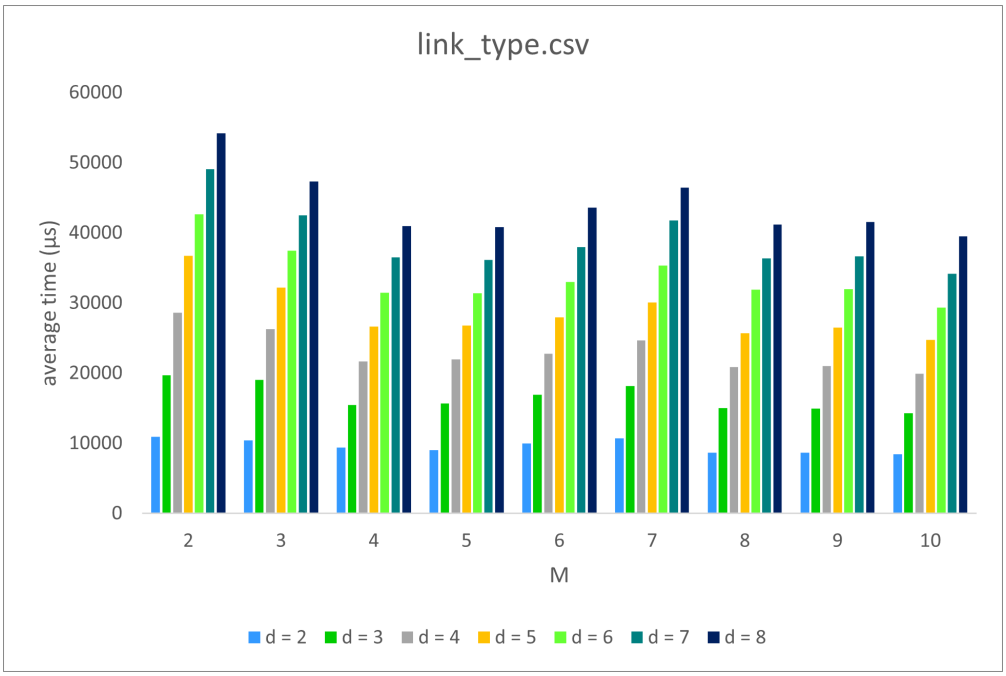

Figure 41: Merge sort on link_type.csv

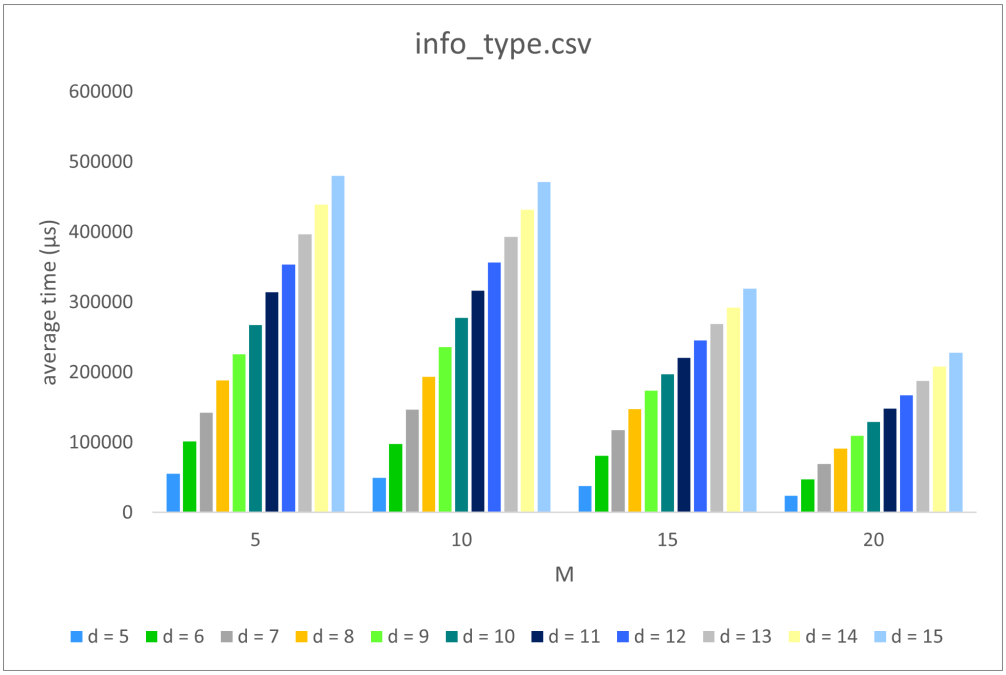

Figure 42: Merge sort on info_type.csv

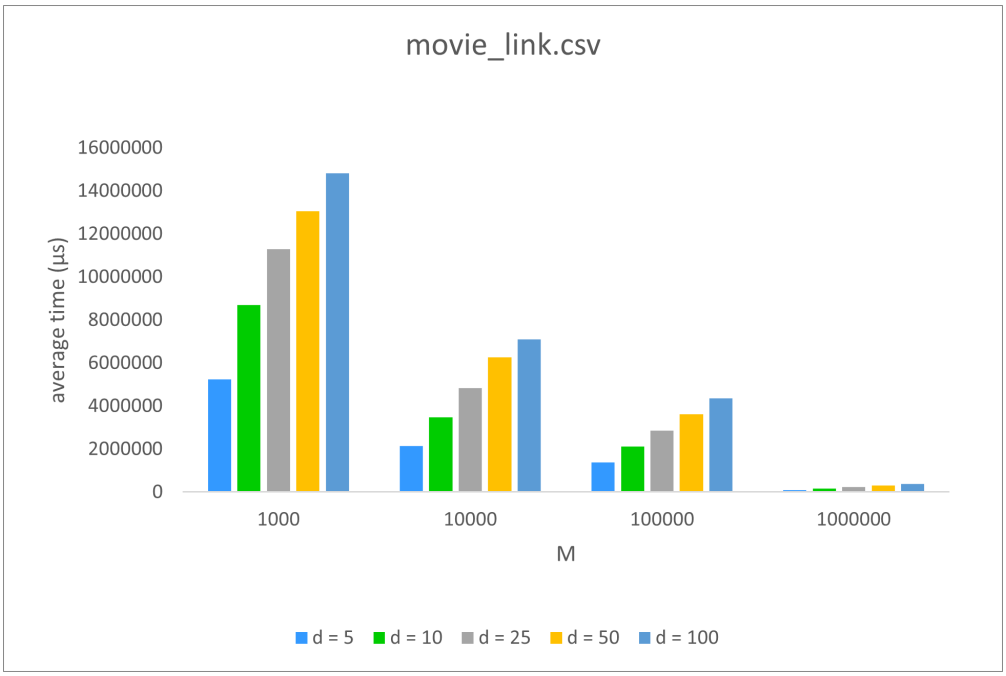

Figure 43: Merge sort on movie_link.csv

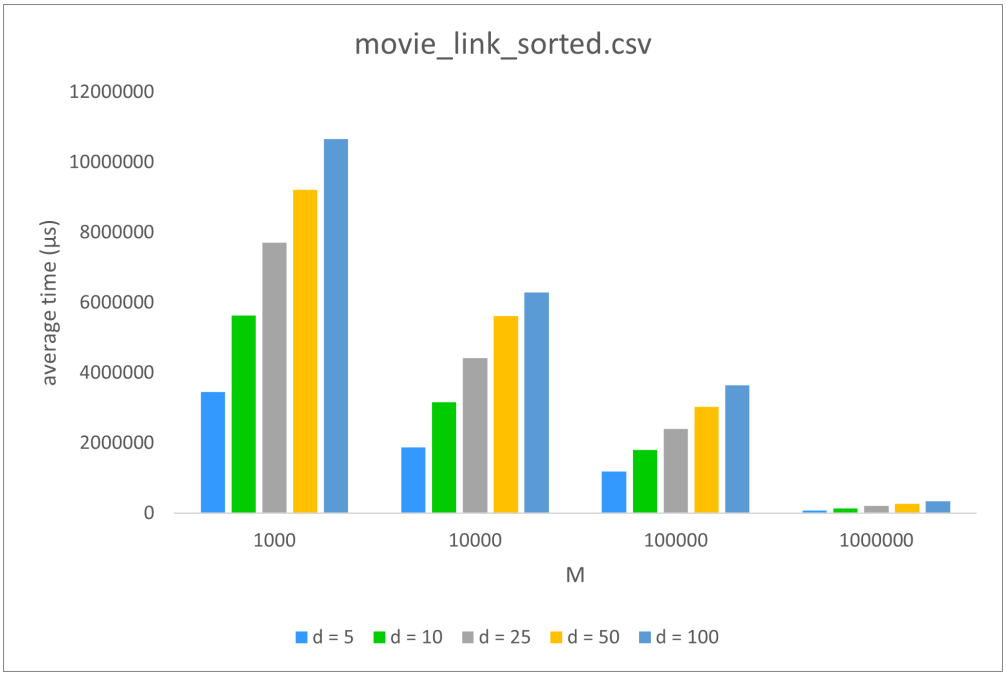

Figure 44: Merge sort on movie_link.csv already sorted

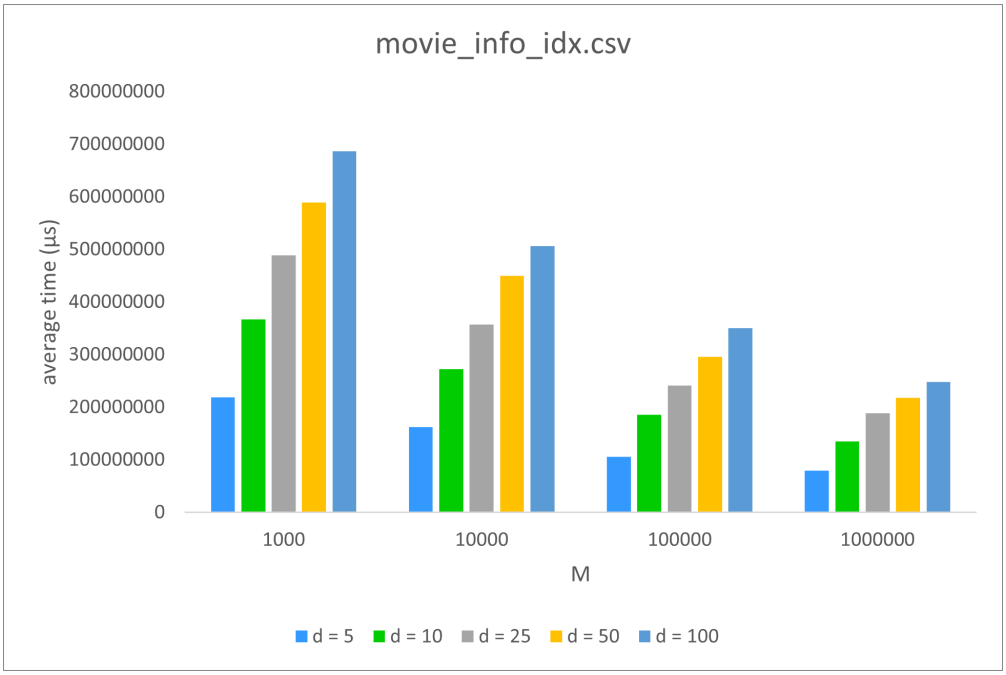

Figure 45: Merge sort on movie_info_idx.csv

7.3 Discussion of expected behavior vs Experimental observations

Firstly, we can see that no matter the size of the file, the results are pretty consistent. Indeed, the same pattern is seen on every graph.
Secondly, we see that having the file already sorted improves the performance slightly even though it does not change the number of I/Os. This might be because the sorting and priority queue are optimised for this specific case and thus saved some CPU cycles. Other tests were made that testify this conclusion but their graphs were not included in the report.
Thirdly, we observe that the average time decreases when increasing M. This was expected because when we increase M, the number of files created during the first phase is smaller ($\lceil \frac{N}{M} \rceil$) and thus there are less files to merge so less I/Os to do.
Finally,we can see that, counter intuitively, the time increases when d is increased. The expecting result would have been to have a decreasing time as d defines the number of files merged at once, which means that a bigger d results in less passes to do, as well as less files to create because we merge more files into one. One hypothesis that we have is based on our implementation. Indeed, we chose to open and close a file each time we read a line in the merging phase. This leads to more I/Os to do when reading and thus might have influenced the results.
In conclusion, the best parameters are a high value for M and a small value for d no matter the size of the file. However, we must be careful that the parameter chosen is still in the scope of the file as it would not make sens to choose M=1000 for a file containing 100 bytes.

8 Overall conclusion

In conclusion, this project gave us the opportunity to implement different types of input and output methods and to implement them into different algorithms. It made us aware of their limitations and differences.

We saw that the results can diverge widely from the expected behaviour. This is expected since the only metric we used to evaluate the cost of a method was the number of I/Os. There are however many other factors that we could take into account, like Operating System optimizations, overheads, as well as the fact that our implementations might call functions that are quite expensive which goes beyond the scope of this course. The cost of such operations is not negligible because the files can be quite big and those operations might be done in a loop.

On another page, we could think of some improvements for our implementation. For example, in the third method, we could have made the buffer used for reading/writing an attribute of the class which is allocated memory at construction, so that it does not need to be allocated every time we use `readln` or `writeln`. That would have meant that for a given instance of In/OutputStream3, the size of the buffer is fixed.
Another example would have been to not open and close each time we read a line in the merge sort algorithm but have a vector of open streams instead. We could also have used pointers instead of strings for the addresses seeing that pointers take less space in memory. Other various optimisation could have been added with more time to tidy up the code, for example, passing addresses instead of copying strings. As a last word, this project taught us the importance of optimizing reading and writing methods since different implementations can greatly affect the overall performances. So when choosing to work with a certain implementation, one should, in addition to making a prediction of the IO costs, take into account the additional costs that could slow down the performance.

Appendix A : Sequential reading

8.1 role_type.csv

Method	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
1	10	160	153.5	1.04235
2		160	7.25	22.069
3		160	51.75	3.09179
4		160	46.25	3.45946
1	50	160	145.5	1.09966
2		160	7	22.8571
3		160	37	4.32432
4		160	37.75	4.23841
1	1000	160	133	1.20301
2		160	6.75	23.7037
3		160	35	4.57143
4		160	47.5	3.36842
1	4096	160	132.25	1.20983
2		160	6.25	25.6
3		160	33.25	4.81203
4		160	43	3.72093
1	8192	160	131.75	1.21442
2		160	7.75	20.6452
3		160	34.25	4.67153
4		160	36.25	4.41379
1	12288	160	163.25	0.980092
2		160	7	22.8571
3		160	32.5	4.92308
4		160	36.5	4.38356
1	40960	160	128	1.25
2		160	5.75	27.8261
3		160	37.75	4.23841
4		160	40.75	3.92638
1	409600	160	120.75	1.32505
2		160	6.25	25.6
3		160	37.75	4.23841
4		160	32.25	4.96124
1	4096000	160	120.75	1.32505
2		160	6.25	25.6
3		160	37.75	4.23841
4		160	31.25	5.12

8.2 movie_link.csv

Method	Size of Buffer	Char read	Average Time (μs)	Average Speed (`MB/s`)
1	10	656584	456578	1.43805
2		656584	6045.5	108.607
3		656584	150439	4.36445
4		656584	43217.2	15.1926
1	50	656584	468697	1.40087
2		656584	5967.25	110.031
3		656584	77461.2	8.47629
4		656584	42799.2	15.341
1	1000	656584	470699	1.39491
2		656584	9270.25	70.827
3		656584	84834.8	7.73956
4		656584	42993.2	15.2718
1	4096	656584	475004	1.38227
2		656584	6747.5	97.3077
3		656584	87256	7.5248
4		656584	42268	15.5338
1	8192	656584	479470	1.3694
2		656584	6436	102.017
3		656584	88858.5	7.3891
4		656584	44277.8	14.8288
1	12288	656584	469499	1.39848
2		656584	6223.5	105.501
3		656584	93248	7.04127
4		656584	42205	15.557
1	40960	656584	467546	1.40432
2		656584	6237.75	105.26
3		656584	140060	4.68788
4		656584	42281	15.5291
1	409600	656584	467591	1.40418
2		656584	6456.75	101.69
3		656584	525304	1.24991
4		656584	41909.8	15.6666
1	4096000	656584	472179	1.39054
2		656584	6293.75	104.323
3		656584	615006	1.06761
4		656584	41184.2	15.9426

8.3 movie_info_index.csv

Method	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
1	10	35335875	2.56975e+07	1.37507
2		35335875	356193	99.2044
3		35335875	7.68817e+06	4.59614
4		35335875	3.62888e+06	9.7374
1	50	35335875	2.51259e+07	1.40635
2		35335875	365812	96.5957
3		35335875	3.67679e+06	9.61053
4		35335875	3.63948e+06	9.70905
1	1000	35335875	2.50876e+07	1.4085
2		35335875	355655	99.3543
3		35335875	3.78681e+06	9.33132
4		35335875	3.33514e+06	10.595
1	4096	35335875	2.49892e+07	1.41405
2		35335875	345178	102.37
3		35335875	3.93931e+06	8.97007
4		35335875	3.34531e+06	10.5628
1	8192	35335875	2.50255e+07	1.41199
2		35335875	358363	98.6037
3		35335875	4.1701e+06	8.47363
4		35335875	2.90295e+06	12.1724
1	12288	35335875	2.49034e+07	1.41892
2		35335875	342854	103.064
3		35335875	4.30377e+06	8.21045
4		35335875	2.70966e+06	13.0407
1	40960	35335875	2.42875e+07	1.4549
2		35335875	343296	102.931
3		35335875	6.43252e+06	5.49332
4		35335875	2.1946e+06	16.1013
1	409600	35335875	2.45672e+07	1.43834
2		35335875	337104	104.822
3		35335875	3.12962e+07	1.12908
4		35335875	2.12136e+06	16.6571
1	4096000	35335875	2.37328e+07	1.48891
2		35335875	338793	104.299
3		35335875	4.892e+08	0.072232
4		35335875	1.99704e+06	17.6941

8.4 aka_name.csv

Method	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
1	10	73004383	4.88613e+07	1.49411
2		73004383	574022	127.181
3		73004383	1.15109e+07	6.34221
4		73004383	9.62466e+06	7.58514
1	50	73004383	4.9618e+07	1.47133
2		73004383	602188	121.232
3		73004383	3.93414e+06	18.5566
4		73004383	1.03233e+07	7.07181
1	1000	73004383	4.90066e+07	1.48969
2		73004383	590656	123.599
3		73004383	2.68561e+06	27.1835
4		73004383	9.22372e+06	7.91485
1	4096	73004383	4.89393e+07	1.49173
2		73004383	597961	122.089
3		73004383	2.81233e+06	25.9587
4		73004383	1.11035e+07	6.57491
1	8192	73004383	4.88801e+07	1.49354
2		73004383	618682	118
3		73004383	2.94162e+06	24.8178
4		73004383	6.4253e+06	11.362
1	12288	73004383	4.87781e+07	1.49666
2		73004383	590331	123.667
3		73004383	3.07428e+06	23.7468
4		73004383	4.38814e+06	16.6368
1	40960	73004383	4.92503e+07	1.48231
2		73004383	596180	122.454
3		73004383	4.54358e+06	16.0676
4		73004383	2.61458e+06	27.9221
1	409600	73004383	4.97755e+07	1.46667
2		73004383	591604	123.401
3		73004383	2.07901e+07	3.5115
4		73004383	1.79044e+06	40.7747
1	4096000	73004383	4.93641e+07	1.4789
2		73004383	590272	123.679
3		73004383	3.60731e+08	0.202379
4		73004383	1.66878e+06	43.7471

9 Appendix B : Random reading

9.1 role_type.csv

Method1-2

Method	Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
1	10	/	70	62.5	1.12
2		/	70	20.75	3.37349
1	50	/	370	286	1.29371
2		/	370	85.75	4.31487
1	1000	/	7315	5687.5	1.28615
2		/	7315	1682	4.34899
1	10000	/	73183	54934.5	1.33219
2		/	73183	15967.2	4.58332
1	100000	/	731815	531412	1.37711
2		/	731815	157258	4.65361
1	1000000	/	7318183	5.49156e+06	1.33262
2		/	7318183	1.59059e+06	4.60093

Method3

Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
10	10	70	41	1.70732
10	50	70	30	2.33333
10	1000	70	30.5	2.29508
10	4096	70	30.25	2.31405
10	8192	70	30.25	2.31405
10	12288	70	30	2.33333
10	40960	70	36.75	1.90476
10	409600	70	34.5	2.02899
50	10	370	169.25	2.18612
50	50	370	144	2.56944
50	1000	370	146.25	2.52991
50	4096	370	142	2.60563
50	8192	370	144.75	2.55613
50	12288	370	143	2.58741
50	40960	370	165	2.24242
50	409600	370	165.25	2.23903
1000	10	7315	3293	2.22138
1000	50	7315	3164	2.31195
1000	1000	7315	2970.5	2.46255
1000	4096	7315	3184.5	2.29706
1000	8192	7315	3105.75	2.35531
1000	12288	7315	2789.5	2.62233
1000	40960	7315	3269.75	2.23717
1000	409600	7315	3296	2.21936
10000	10	73183	30807	2.37553
10000	50	73183	26845.2	2.72611
10000	1000	73183	27309.5	2.67976
10000	4096	73183	26759.8	2.73482
10000	8192	73183	27350	2.6758
10000	12288	73183	27184.5	2.69209
10000	40960	73183	30319	2.41377
10000	409600	73183	30867.5	2.37088
100000	10	731815	322830	2.26687
100000	50	731815	293150	2.49639
100000	1000	731815	294571	2.48434
100000	4096	731815	297153	2.46275
100000	8192	731815	278488	2.62782
100000	12288	731815	286948	2.55034
100000	40960	731815	315556	2.31913
100000	409600	731815	319678	2.28922
1000000	10	7318183	3.13371e+06	2.33531
1000000	50	7318183	2.8297e+06	2.5862
1000000	1000	7318183	2.87517e+06	2.54531
1000000	4096	7318183	2.82396e+06	2.59146
1000000	8192	7318183	2.82991e+06	2.58601
1000000	12288	7318183	2.84067e+06	2.57622
1000000	40960	7318183	3.24558e+06	2.25482
1000000	409600	7318183	3.20734e+06	2.2817

Method4

Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
10	4096	70	39.5	1.77215
10	8192	70	65.5	1.0687
10	12288	70	105.5	0.663507
10	40960	70	134	0.522388
10	409600	70	166.5	0.42042
10	4096000	70	192.5	0.363636
50	4096	370	112.5	3.28889
50	8192	370	215.25	1.71893
50	12288	370	322.5	1.14729
50	40960	370	446	0.829596
50	409600	370	557	0.664273
50	4096000	370	664.25	0.557019
1000	4096	7315	1844.5	3.96584
1000	8192	7315	3874.25	1.88811
1000	12288	7315	5832.75	1.25413
1000	40960	7315	7695	0.950617
1000	409600	7315	9909.25	0.738199
1000	4096000	7315	11668.5	0.626902
10000	4096	73183	16741	4.37148
10000	8192	73183	33045.2	2.21463
10000	12288	73183	49362.5	1.48256
10000	40960	73183	65601.8	1.11556
10000	409600	73183	82164.8	0.890686
10000	4096000	73183	98861.2	0.74026
100000	4096	731815	165223	4.42925
100000	8192	731815	328613	2.22698
100000	12288	731815	491962	1.48754
100000	40960	731815	657343	1.11329
100000	409600	731815	821072	0.891292
100000	4096000	731815	984529	0.743315
1000000	4096	7318183	1.63722e+06	4.46989
1000000	8192	7318183	3.27157e+06	2.2369
1000000	12288	7318183	4.90476e+06	1.49206
1000000	40960	7318183	6.53949e+06	1.11908
1000000	409600	7318183	8.17248e+06	0.895467
1000000	4096000	7318183	9.80653e+06	0.746256

9.2 movie_link.csv

Method1-2

Method	Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (`MB/s`)
1	10	/	113	1321.75	0.0854927
2		/	113	156.75	0.720893
1	50	/	578	4432.75	0.130393
2		/	578	109	5.30275
1	1000	/	11583	11126.5	1.04103
2		/	11583	1878.75	6.16527
1	10000	/	113998	84319.5	1.35198
2		/	113998	17737.5	6.42695
1	100000	/	1137784	831384	1.36854
2		/	1137784	185252	6.14181
1	1000000	/	11376150	8.27612e+06	1.37458
2		/	11376150	1.89287e+06	6.00999

Method3:

Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
10	10	113	1157.5	0.0976242
10	50	113	31.75	3.55906
10	1000	113	288	0.392361
10	4096	113	849.75	0.13298
10	8192	113	874.75	0.12918
10	12288	113	402	0.281095
10	40960	113	461.5	0.244854
10	409600	113	413	0.273608
50	10	578	511.75	1.12946
50	50	578	150.75	3.83416
50	1000	578	287.25	2.01218
50	4096	578	204.25	2.82987
50	8192	578	259.25	2.22951
50	12288	578	233.25	2.47803
50	40960	578	296	1.9527
50	409600	578	977.75	0.591153
1000	10	11583	4234.5	2.73539
1000	50	11583	2922	3.96407
1000	1000	11583	2996	3.86615
1000	4096	11583	3008.5	3.85009
1000	8192	11583	3205	3.61404
1000	12288	11583	3314.75	3.49438
1000	40960	11583	5100	2.27118
1000	409600	11583	16632.5	0.696408
10000	10	113998	35796.5	3.18461
10000	50	113998	27601.5	4.13014
10000	1000	113998	29970.2	3.80371
10000	4096	113998	30629.2	3.72187
10000	8192	113998	32580	3.49902
10000	12288	113998	34282.5	3.32525
10000	40960	113998	50597.2	2.25305
10000	409600	113998	171090	0.666304
100000	10	1137784	357370	3.18377
100000	50	1137784	275682	4.12715
100000	1000	1137784	298449	3.81233
100000	4096	1137784	310131	3.66872
100000	8192	1137784	330036	3.44745
100000	12288	1137784	341869	3.32813
100000	40960	1137784	501054	2.27078
100000	409600	1137784	1.7207e+06	0.661233
1000000	10	11376150	3.68261e+06	3.08915
1000000	50	11376150	2.90322e+06	3.91846
1000000	1000	11376150	3.01468e+06	3.77359
1000000	4096	11376150	3.12281e+06	3.64292
1000000	8192	11376150	3.30451e+06	3.44261
1000000	12288	11376150	3.52795e+06	3.22458
1000000	40960	11376150	5.12416e+06	2.2201
1000000	409600	11376150	1.73949e+07	0.653993

Method4:

Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
10	4096	113	1374.25	0.0822267
10	8192	113	1443.5	0.078282
10	12288	113	1512.25	0.0747231
10	40960	113	1577.25	0.0716437
10	409600	113	1628.5	0.069389
10	4096000	113	1664.25	0.0678985
50	4096	578	3705.25	0.155995
50	8192	578	4133.75	0.139825
50	12288	578	4561.75	0.126706
50	40960	578	4968.25	0.116339
50	409600	578	5261.25	0.10986
50	4096000	578	5431.25	0.106421
1000	4096	11583	11685.8	0.991207
1000	8192	11583	19914.8	0.581629
1000	12288	11583	28062.5	0.412757
1000	40960	11583	36003.5	0.321719
1000	409600	11583	41596	0.278464
1000	4096000	11583	43418	0.266779
10000	4096	113998	82164	1.38744
10000	8192	113998	162948	0.699597
10000	12288	113998	244619	0.466023
10000	40960	113998	323284	0.352625
10000	409600	113998	380144	0.299881
10000	4096000	113998	396810	0.287286
100000	4096	1137784	822867	1.38271
100000	8192	1137784	1.6402e+06	0.693685
100000	12288	1137784	2.46093e+06	0.462339
100000	40960	1137784	3.31075e+06	0.343663
100000	409600	1137784	3.9241e+06	0.289948
100000	4096000	1137784	4.09539e+06	0.277821
1000000	4096	11376150	8.58353e+06	1.32535
1000000	8192	11376150	1.72094e+07	0.661042
1000000	12288	11376150	2.53172e+07	0.449344
1000000	40960	11376150	3.32236e+07	0.342411
1000000	409600	11376150	3.90094e+07	0.291626

9.3 movie_info_idx.csv

Method1-2:

Method	Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (`MB/s`)
1	10	/	161	1413.25	0.113922
2		/	161	144.5	1.11419
1	50	/	700	2563	0.273117
2		/	700	130.25	5.37428
1	1000	/	13334	42811	0.311462
2		/	13334	2140.5	6.22939
1	100000	/	1366693	1.10556e+06	1.2362
2		/	1366693	220214	6.20619
1	1000000	/	13686406	1.02318e+07	1.33764
2		/	13686406	2.28794e+06	5.98197

Method3:

Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
10	10	161	1343.25	0.119859
10	50	161	162.75	0.989247
10	1000	161	38	4.23684
10	4096	161	984.5	0.163535
10	8192	161	1130.25	0.142446
10	12288	161	1315.25	0.12241
10	40960	161	1483.75	0.108509
10	409600	161	2222.25	0.0724491
50	10	700	4157	0.168391
50	50	700	301.75	2.3198
50	1000	700	1033.5	0.67731
50	4096	700	2954	0.236967
50	8192	700	2175	0.321839
50	12288	700	1419.5	0.493131
50	40960	700	2230.75	0.313796
50	409600	700	7888	0.0887424
1000	10	13334	29967	0.444956
1000	50	13334	3908	3.41198
1000	1000	13334	9258.25	1.44023
1000	4096	13334	20923.2	0.637281
1000	8192	13334	17411.8	0.765805
1000	12288	13334	17961.5	0.742366
1000	40960	13334	26406	0.504961
1000	409600	13334	50095.2	0.266173
10000	10	135023	42562.2	3.17237
10000	50	135023	29937.5	4.51016
10000	1000	135023	31477	4.28958
10000	4096	135023	35198	3.8361
10000	8192	135023	40502.8	3.33367
10000	12288	135023	46198	2.9227
10000	40960	135023	79035.2	1.70839
10000	409600	135023	510214	0.26464
100000	10	1366693	404202	3.38121
100000	50	1366693	293186	4.66152
100000	1000	1366693	316719	4.31516
100000	4096	1366693	353188	3.86959
100000	8192	1366693	405242	3.37254
100000	12288	1366693	431502	3.16729
100000	40960	1366693	711938	1.91968
100000	409600	1366693	4.44466e+06	0.307491
1000000	10	13686406	4.04211e+06	3.38596
1000000	50	13686406	2.91944e+06	4.68802
1000000	1000	13686406	3.15445e+06	4.33876
1000000	4096	13686406	3.5223e+06	3.88564
1000000	8192	13686406	4.10283e+06	3.33584
1000000	12288	13686406	4.35625e+06	3.14178
1000000	40960	13686406	7.36856e+06	1.8574
1000000	409600	13686406	4.5096e+07	0.303495

Method4:

Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
10	4096	161	1738.5	0.0926086
10	8192	161	1803.25	0.0892832
10	12288	161	1867.25	0.0862231
10	40960	161	1931.5	0.0833549
10	409600	161	1997.75	0.0805907
10	4096000	161	2079.5	0.0774225
50	4096	700	4992.25	0.140217
50	8192	700	5440.5	0.128665
50	12288	700	5880	0.119048
50	40960	700	6348.25	0.110267
50	409600	700	6787.75	0.103127
50	4096000	700	7284.5	0.0960944
1000	4096	13334	72547.5	0.183797
1000	8192	13334	111760	0.119309
1000	12288	13334	152542	0.0874117
1000	40960	13334	192171	0.0693861
1000	409600	13334	231522	0.0575927
1000	4096000	13334	270476	0.0492984
10000	4096	135023	2.17986e+06	0.0619413
10000	8192	135023	3.74577e+06	0.0360468
10000	12288	135023	5.41903e+06	0.0249165
10000	40960	135023	6.95107e+06	0.0194248
10000	409600	135023	8.42055e+06	0.0160349
10000	4096000	135023	9.78345e+06	0.0138012
100000	4096	1366693	1.72324e+07	0.0793096
100000	8192	1366693	3.43302e+07	0.0398102
100000	12288	1366693	5.09909e+07	0.0268027
100000	40960	1366693	6.82642e+07	0.0200207
100000	409600	1366693	8.38536e+07	0.0162986
100000	4096000	1366693	9.89228e+07	0.0138158

9.4 aka_name.csv

Method1-2

Method	Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
1	10	/	519	1395.25	0.371976
2		/	519	107	4.85047
1	50	/	2335	3828.5	0.609899
2		/	2335	125.5	18.6056
1	1000	/	42352	80818.5	0.524038
2		/	42352	2346.75	18.0471
1	10000	/	414410	612599	0.676478
2		/	414410	27428.2	15.1089
1	100000	/	4136008	3.31139e+06	1.24902
2		/	4136008	260496	15.8774
1	1000000	/	41366418	2.97426e+07	1.39081
2		/	41366418	2.60466e+06	15.8817

Method3

Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (MB/s)
10	10	519	1266.75	0.40971
10	50	519	47.75	10.8691
10	1000	519	160.25	3.23869
10	4096	519	1048.75	0.494875
10	8192	519	1193.25	0.434947
10	12288	519	1021.25	0.508201
10	40960	519	1045.75	0.496295
10	409600	519	2029.5	0.255728
50	10	2335	2335.75	0.999679
50	50	2335	243.75	9.57949
50	1000	2335	511.75	4.56277
50	4096	2335	1320.5	1.76827
50	8192	2335	1550.5	1.50597
50	12288	2335	1420	1.64437
50	40960	2335	2170.75	1.07567
50	409600	2335	7574.75	0.308261
1000	10	42352	38285.5	1.10622
1000	50	42352	3863	10.9635
1000	1000	42352	11363	3.72718
1000	4096	42352	19466.2	2.17566
1000	8192	42352	23836.8	1.77675
1000	12288	42352	26063.2	1.62497
1000	40960	42352	37575.5	1.12712
1000	409600	42352	71636.5	0.591207
10000	10	414410	87675.2	4.72665
10000	50	414410	36102.5	11.4787
10000	1000	414410	38411.8	10.7886
10000	4096	414410	38927.2	10.6458
10000	8192	414410	45947.8	9.01916
10000	12288	414410	49888.2	8.30677
10000	40960	414410	84680.2	4.89382
10000	409600	414410	529255	0.783006
100000	10	4136008	792907	5.21626
100000	50	4135907	403918	10.2395
100000	1000	4136008	363288	11.3849
100000	4096	4136008	376107	10.9969
100000	8192	4136008	442439	9.34819
100000	12288	4136008	464703	8.90032
100000	40960	4136008	800616	5.16604
100000	409600	4136008	4.83943e+06	0.854648
1000000	10	41366418	8.39483e+06	4.92761
1000000	50	41169284	3.92972e+06	10.4764
1000000	1000	41366418	3.57859e+06	11.5594
1000000	4096	41366418	3.84413e+06	10.7609
1000000	8192	41366418	4.2927e+06	9.63646
1000000	12288	41366418	4.73725e+06	8.73216
1000000	40960	41366418	7.45427e+06	5.54936
1000000	409600	41366418	4.63398e+07	0.892677

Method4

Number of Jumps	Size of Buffer	Char read	Average Time (μs)	Average Speed (`MB/s`)
10	4096	519	822.5	0.631003
10	8192	519	899.5	0.576987
10	12288	519	977.75	0.530811
10	40960	519	1067.25	0.486297
10	409600	519	1153.5	0.449935
10	4096000	519	1256.25	0.413134
50	4096	2335	3210.25	0.727358
50	8192	2335	3781.75	0.617439
50	12288	2335	4270.75	0.546742
50	40960	2335	5538.5	0.421594
50	409600	2335	6035.25	0.386894
50	4096000	2335	6616	0.352932
1000	4096	42352	86435.5	0.489984
1000	8192	42352	131164	0.322893
1000	12288	42352	176402	0.240088
1000	40960	42352	221162	0.191498
1000	409600	42352	263989	0.160431
1000	4096000	42352	307045	0.137934
10000	4096	414410	3.21337e+06	0.128964
10000	8192	414410	4.96655e+06	0.0834402
10000	12288	414410	6.71052e+06	0.0617553
10000	40960	414410	8.44286e+06	0.0490841
10000	409600	414410	1.01785e+07	0.0407143
10000	4096000	414410	1.1872e+07	0.0349065
100000	4096	4136008	3.02669e+07	0.136651
100000	8192	4136008	5.84444e+07	0.0707682
100000	12288	4136008	8.85391e+07	0.0467139
100000	40960	4136008	1.19701e+08	0.0345528
100000	409600	4136008	1.47756e+08	0.0279922
100000	4096000	4136008	1.75753e+08	0.0235331

10 Appendix C : Combined reading and writing

10.1 Rrmerge 2 on 2 same files

File	Method	Size Buffer	Char output	Average time (μs)
role_type.csv	1	/	320	1339
	2	/	320	335
	3	10	320	518
	4	10	320	353
	1	/	320	1257
	2	/	320	292
	3	50	320	341
	4	50	320	369
	1	/	320	1999
	2	/	320	260
	3	1000	320	449
	4	1000	320	608
	1	/	320	2670
	2	/	320	381
	3	4096	320	401
	4	4096	320	328
	1	/	320	1201
	2	/	320	271
	3	8192	320	352
	4	8192	320	329
	1	/	320	1343
	2	/	320	216
	3	12288	320	315
	4	12288	320	324
	1	/	320	1186
	2	/	320	221
	3	40960	320	379
	4	40960	320	313
	1	/	320	1152
	2	/	320	227
	3	409600	320	614
	4	409600	320	301

File	Method	Size Buffer	Char output	Average time (μs)
movie_link.csv	1	/	1313168	3651614
	2	/	1313168	23063
	3	10	1313168	545784
	4	10	1313168	301270
	1	/	1313168	3610771
	2	/	1313168	25136
	3	50	1313168	216449
	4	50	1313168	286250
	1	/	1313168	3494728
	2	/	1313168	23312
	3	1000	1313168	212286
	4	1000	1313168	307432
	1	/	1313168	3576182
	2	/	1313168	23890
	3	4096	1313168	212055
	4	4096	1313168	306801
	1	/	1313168	3505860
	2	/	1313168	23431
	3	8192	1313168	208023
	4	8192	1313168	297718
	1	/	1313168	3398476
	2	/	1313168	24482
	3	12288	1313168	230669
	4	12288	1313168	321061
	1	/	1313168	3387484
	2	/	1313168	23812
	3	40960	1313168	283335
	4	40960	1313168	299099
	1	/	1313168	3391524
	2	/	1313168	23241
	3	409600	1313168	697715
	4	409600	1313168	298850

File	Method	Size Buffer	Char output	Average time (μs)
complete_cast.csv	1	/	4828990	12411336
	2	/	4828990	74003
	3	10	4828990	2276634
	4	10	4828990	1285467
	1	/	4828990	12656803
	2	/	4828990	72478
	3	50	4828990	840975
	4	50	4828990	1221745
	1	/	4828990	12548431
	2	/	4828990	78521
	3	1000	4828990	864905
	4	1000	4828990	1262774
	1	/	4828990	12552458
	2	/	4828990	77638
	3	4096	4828990	872434
	4	4096	4828990	1288118
	1	/	4828990	12554633
	2	/	4828990	73925
	3	8192	4828990	932256
	4	8192	4828990	1258461
	1	/	4828990	12649844
	2	/	4828990	70304
	3	12288	4828990	930406
	4	12288	4828990	1247958
	1	/	4828990	12941807
	2	/	4828990	71554
	3	40960	4828990	1180438
	4	40960	4828990	1251819
	1	/	4828990	12291212
	2	/	4828990	74413
	3	409600	4828990	3025855
	4	409600	4828990	1287552

File	Method	Size Buffer	Char output	Average time (μs)
movie_info_idx.csv	2	/	70671750	1478215
	3	10	70671750	27894869
	4	10	70671750	15202748
	2	/	70671750	1924469
	3	50	70671750	10300273
	4	50	70671750	14813247
	2	/	70671750	1491256
	3	1000	70671750	10415975
	4	1000	70671750	15402639
	2	/	70671750	1524011
	3	4096	70671750	10325006
	4	4096	70671750	15633158
	2	/	70671750	2023669
	3	8192	70671750	10710631
	4	8192	70671750	15758569
	2	/	70671750	1454837
	3	12288	70671750	11013499
	4	12288	70671750	15083186
	2	/	70671750	1470467
	3	40960	70671750	14122433
	4	40960	70671750	15630809
	2	/	70671750	2106305
	3	409600	70671750	34475763
	4	409600	70671750	15566355

10.2 Rrmerge 2 on 3 same files

File	Method	Size Buffer	Char output	Average time (μs)
role_type.csv	1	/	480	1696
	2	/	480	164
	3	10	480	442
	4	10	480	289
	1	/	480	2283
	2	/	480	254
	3	50	480	619
	4	50	480	593
	1	/	480	2852
	2	/	480	270
	3	1000	480	407
	4	1000	480	434
	1	/	480	1756
	2	/	480	262
	3	4096	480	433
	4	4096	480	385
	1	/	480	1644
	2	/	480	236
	3	8192	480	420
	4	8192	480	384
	1	/	480	1634
	2	/	480	284
	3	12288	480	379
	4	12288	480	415
	1	/	480	1970
	2	/	480	231
	3	40960	480	463
	4	40960	480	378
	1	/	480	2020
	2	/	480	364
	3	409600	480	902
	4	409600	480	433

File	Method	Size Buffer	Char output	Average time (μs)
movie_link.csv	1	/	1969752	5133446
	2	/	1969752	35889
	3	10	1969752	821311
	4	10	1969752	439140
	1	/	1969752	4980784
	2	/	1969752	37759
	3	50	1969752	296918
	4	50	1969752	467129
	1	/	1969752	4963741
	2	/	1969752	34607
	3	1000	1969752	313602
	4	1000	1969752	455755
	1	/	1969752	4979760
	2	/	1969752	44641
	3	4096	1969752	306188
	4	4096	1969752	473351
	1	/	1969752	5060405
	2	/	1969752	36042
	3	8192	1969752	326180
	4	8192	1969752	449895
	1	/	1969752	5038850
	2	/	1969752	36334
	3	12288	1969752	335917
	4	12288	1969752	473610
	1	/	1969752	4962676
	2	/	1969752	37668
	3	40960	1969752	422492
	4	40960	1969752	470426
	1	/	1969752	5079148
	2	/	1969752	34209
	3	409600	1969752	1074457
	4	409600	1969752	465673

File	Method	Size Buffer	Char output	Average time (μs)
complete_cast.csv	1	/	7243485	19062157
	2	/	7243485	109766
	3	10	7243485	2574748
	4	10	7243485	1961916
	1	/	7243485	18841289
	2	/	7243485	113544
	3	50	7243485	1310793
	4	50	7243485	1926348
	1	/	7243485	19127019
	2	/	7243485	112842
	3	1000	7243485	1345991
	4	1000	7243485	1959620
	1	/	7243485	18621111
	2	/	7243485	107121
	3	4096	7243485	1281649
	4	4096	7243485	1908558
	1	/	7243485	18517134
	2	/	7243485	109642
	3	8192	7243485	1373210
	4	8192	7243485	1948762
	1	/	7243485	18632256
	2	/	7243485	112690
	3	12288	7243485	1377925
	4	12288	7243485	1914101
	1	/	7243485	18615192
	2	/	7243485	106932
	3	40960	7243485	1827891
	4	40960	7243485	1916897
	1	/	7243485	18481936
	2	/	7243485	104086
	3	409600	7243485	4551256
	4	409600	7243485	1914906

10.3 Rrmerge 2 on different files

File	Method	Size Buffer	Char output	Average time (μs)
complete_cast.csv + movie_link	1	/	3071079	7895996
	2	/	3071079	54804
	3	10	3071079	1116919
	4	10	3071079	675056
	1	/	3071079	7900000
	2	/	3071079	50029
	3	50	3071079	539465
	4	50	3071079	696699
	1	/	3071079	7924510
	2	/	3071079	50390
	3	1000	3071079	537901
	4	1000	3071079	671461
	1	/	3071079	8084774
	2	/	3071079	48857
	3	4096	3071079	525720
	4	4096	3071079	663250
	1	/	3071079	7957482
	2	/	3071079	52281
	3	8192	3071079	552778
	4	8192	3071079	660325
	1	/	3071079	8646913
	2	/	3071079	53169
	3	12288	3071079	570149
	4	12288	3071079	681557
	1	/	3071079	7917910
	2	/	3071079	52605
	3	40960	3071079	744821
	4	40960	3071079	660813
	1	/	3071079	7917127
	2	/	3071079	55941
	3	409600	3071079	1888436
	4	409600	3071079	656748

File	Method	Size Buffer	Char output	Average time (μs)
complete_cast.csv + movie_link.csv + movie_info_idx.csv	2	/	38406954	792405
	3	10	38406954	15350614
	4	10	38406954	7340729
	2	/	38406954	782948
	3	50	38406954	5512477
	4	50	38406954	6991852
	2	/	38406954	722394
	3	1000	38406954	6059921
	4	1000	38406954	6963222
	2	/	38406954	736921
	3	4096	38406954	5359242
	4	4096	38406954	6948607
	2	/	38406954	717171
	3	8192	38406954	5652685
	4	8192	38406954	7040473
	2	/	38406954	743486
	3	12288	38406954	5854558
	4	12288	38406954	7059220
	2	/	38406954	747608
	3	40960	38406954	7350950
	4	40960	38406954	6713169
	2	/	38406954	717767
	3	409600	38406954	18540182
	4	409600	38406954	6979821

10.4 Rrmerge 3 on 2 same files

File	Method	Size Buffer	Char output	Average time (μs)
role_type.csv	1	/	320	1517
	2	/	320	493
	3	10	320	478
	4	10	320	390
	1	/	320	1354
	2	/	320	314
	3	50	320	394
	4	50	320	559
	1	/	320	1583
	2	/	320	335
	3	1000	320	537
	4	1000	320	423
	1	/	320	1672
	2	/	320	333
	3	4096	320	400
	4	4096	320	479
	1	/	320	1778
	2	/	320	363
	3	8192	320	395
	4	8192	320	396
	1	/	320	1509
	2	/	320	358
	3	12288	320	550
	4	12288	320	538
	1	/	320	2041
	2	/	320	332
	3	40960	320	442
	4	40960	320	396
	1	/	320	1232
	2	/	320	283
	3	409600	320	688
	4	409600	320	414

File	Method	Size Buffer	Char output	Average time (μs)
movie_link.csv	1	/	1313168	3632892
	2	/	1313168	178382
	3	10	1313168	726686
	4	10	1313168	470993
	1	/	1313168	3669392
	2	/	1313168	175293
	3	50	1313168	367287
	4	50	1313168	456428
	1	/	1313168	3518457
	2	/	1313168	174194
	3	1000	1313168	373140
	4	1000	1313168	447930
	1	/	1313168	3524330
	2	/	1313168	174743
	3	4096	1313168	383061
	4	4096	1313168	462632
	1	/	1313168	3692744
	2	/	1313168	171817
	3	8192	1313168	375932
	4	8192	1313168	470046
	1	/	1313168	3608029
	2	/	1313168	173280
	3	12288	1313168	392591
	4	12288	1313168	491198
	1	/	1313168	3489194
	2	/	1313168	169906
	3	40960	1313168	452262
	4	40960	1313168	474249
	1	/	1313168	3471938
	2	/	1313168	169872
	3	409600	1313168	858312
	4	409600	1313168	465863

File	Method	Size Buffer	Char output	Average time (μs)
complete_cast.csv	1	/	4828990	13233895
	2	/	4828990	741607
	3	10	4828990	2459073
	4	10	4828990	1948176
	1	/	4828990	13046060
	2	/	4828990	715898
	3	50	4828990	1561069
	4	50	4828990	2030518
	1	/	4828990	13035672
	2	/	4828990	720886
	3	1000	4828990	1571546
	4	1000	4828990	2091435
	1	/	4828990	13202006
	2	/	4828990	718591
	3	4096	4828990	1594599
	4	4096	4828990	1917832
	1	/	4828990	13073859
	2	/	4828990	713980
	3	8192	4828990	1621057
	4	8192	4828990	2061156
	1	/	4828990	13415672
	2	/	4828990	716971
	3	12288	4828990	1688430
	4	12288	4828990	1959116
	1	/	4828990	13760470
	2	/	4828990	711627
	3	40960	4828990	1944963
	4	40960	4828990	1998099
	1	/	4828990	13123854
	2	/	4828990	712131
	3	409600	4828990	3785101
	4	409600	4828990	1990072

File	Method	Size Buffer	Char output	Average time (μs)
movie_info_idx.csv	2	/	70671750	8286781
	3	10	70671750	36103897
	4	10	70671750	23666381
	2	/	70671750	8544077
	3	50	70671750	17166865
	4	50	70671750	23250521
	2	/	70671750	8104264
	3	1000	70671750	17059675
	4	1000	70671750	24102798
	2	/	70671750	8125462
	3	4096	70671750	17394090
	4	4096	70671750	22917640
	2	/	70671750	7972873
	3	8192	70671750	17692230
	4	8192	70671750	24351618
	2	/	70671750	8219409
	3	12288	70671750	17868881
	4	12288	70671750	23084578
	2	/	70671750	8045782
	3	40960	70671750	20952594
	4	40960	70671750	23885778
	2	/	70671750	8035687
	3	409600	70671750	39782043
	4	409600	70671750	23106885

10.5 Rrmerge 3 on 3 same files

File	Method	Size Buffer	Char output	Average time (μs)
role_type.csv	1	/	480	1835
	2	/	480	484
	3	10	480	649
	4	10	480	499
	1	/	480	2025
	2	/	480	447
	3	50	480	706
	4	50	480	541
	1	/	480	2319
	2	/	480	614
	3	1000	480	559
	4	1000	480	574
	1	/	480	2220
	2	/	480	410
	3	4096	480	573
	4	4096	480	938
	1	/	480	1906
	2	/	480	366
	3	8192	480	457
	4	8192	480	503
	1	/	480	1734
	2	/	480	323
	3	12288	480	543
	4	12288	480	505
	1	/	480	2046
	2	/	480	338
	3	40960	480	562
	4	40960	480	462
	1	/	480	2077
	2	/	480	360
	3	409600	480	896
	4	409600	480	511

File	Method	Size Buffer	Char output	Average time (μs)
movie_link.csv	1	/	1969752	5342705
	2	/	1969752	261586
	3	10	1969752	1014654
	4	10	1969752	663426
	1	/	1969752	5387218
	2	/	1969752	268388
	3	50	1969752	569534
	4	50	1969752	699101
	1	/	1969752	5559017
	2	/	1969752	254193
	3	1000	1969752	553424
	4	1000	1969752	687691
	1	/	1969752	5334745
	2	/	1969752	252179
	3	4096	1969752	556544
	4	4096	1969752	698843
	1	/	1969752	5335826
	2	/	1969752	250616
	3	8192	1969752	564385
	4	8192	1969752	727126
	1	/	1969752	5503235
	2	/	1969752	255014
	3	12288	1969752	600743
	4	12288	1969752	696351
	1	/	1969752	5497600
	2	/	1969752	261551
	3	40960	1969752	701419
	4	40960	1969752	698905
	1	/	1969752	5292055
	2	/	1969752	254576
	3	409600	1969752	1301485
	4	409600	1969752	701347

File	Method	Size Buffer	Char output	Average time (μs)
complete_cast.csv	1	/	7243485	19570425
	2	/	7243485	1089960
	3	10	7243485	3719006
	4	10	7243485	3012544
	1	/	7243485	19686829
	2	/	7243485	1086367
	3	50	7243485	2372376
	4	50	7243485	2955734
	1	/	7243485	19723999
	2	/	7243485	1080730
	3	1000	7243485	2353257
	4	1000	7243485	2983796
	1	/	7243485	19726798
	2	/	7243485	1074650
	3	4096	7243485	2428024
	4	4096	7243485	2997118
	1	/	7243485	19627251
	2	/	7243485	1072769
	3	8192	7243485	2428319
	4	8192	7243485	2967171
	1	/	7243485	19486348
	2	/	7243485	1078165
	3	12288	7243485	2504305
	4	12288	7243485	3037940
	1	/	7243485	19363765
	2	/	7243485	1071072
	3	40960	7243485	2929495
	4	40960	7243485	3050189
	1	/	7243485	19407340
	2	/	7243485	1083088
	3	409600	7243485	5702114
	4	409600	7243485	2956474

10.6 Rrmerge 3 on different files

File	Method	Size Buffer	Char output	Average time (μs)
complete_cast.csv + movie_link.csv	1	/	3071079	8274161
	2	/	3071079	463958
	3	10	3071079	1580702
	4	10	3071079	1211766
	1	/	3071079	8153835
	2	/	3071079	455332
	3	50	3071079	966923
	4	50	3071079	1239758
	1	/	3071079	8218349
	2	/	3071079	466273
	3	1000	3071079	981742
	4	1000	3071079	1205758
	1	/	3071079	8286123
	2	/	3071079	451885
	3	4096	3071079	1006301
	4	4096	3071079	1256176
	1	/	3071079	8254554
	2	/	3071079	455925
	3	8192	3071079	1016518
	4	8192	3071079	1303060
	1	/	3071079	8354001
	2	/	3071079	456345
	3	12288	3071079	1022534
	4	12288	3071079	1263246
	1	/	3071079	8338483
	2	/	3071079	454474
	3	40960	3071079	1230777
	4	40960	3071079	1231334
	1	/	3071079	8256636
	2	/	3071079	451433
	3	409600	3071079	2337796
	4	409600	3071079	1246304

File	Method	Size Buffer	Char output	Average time (μs)
complete_cast.csv + movie_link.csv + movie_info_idx.csv	2	/	38406954	4441264
	3	10	38406954	19517475
	4	10	38406954	12110963
	2	/	38406954	5082965
	3	50	38406954	9902497
	4	50	38406954	12133872
	2	/	38406954	4369080
	3	1000	38406954	9316416
	4	1000	38406954	12058527
	2	/	38406954	4390063
	3	4096	38406954	9452219
	4	4096	38406954	11918230
	2	/	38406954	4353490
	3	8192	38406954	9559068
	4	8192	38406954	12045170
	2	/	38406954	4752290
	3	12288	38406954	9659682
	4	12288	38406954	12060575
	2	/	38406954	4339714
	3	40960	38406954	11386064
	4	40960	38406954	12073390
	2	/	38406954	4308542
	3	409600	38406954	23004871
	4	409600	38406954	12063048

11 Appendix D : Merge sort

11.1 movie_link.csv

d	M	Average time (μs)
5	1000	5236025
10	1000	8699274
25	1000	11280304
50	1000	13063482
100	1000	14817575
5	10000	2135367
10	10000	3472138
25	10000	4844945
50	10000	6260643
100	10000	7098899
5	100000	1370754
10	100000	2118912
25	100000	2854249
50	100000	3606811
100	100000	4352112
5	1000000	82890
10	1000000	159863
25	1000000	236924
50	1000000	313797
100	1000000	389166

11.2 movie_link.csv already sorted

d	M	Average time (μs)
5	1000	3445921
10	1000	5627263
25	1000	7703617
50	1000	9208573
100	1000	10664717
5	10000	1866760
10	10000	3157596
25	10000	4416958
50	10000	5610440
100	10000	6281586
5	100000	1185321
10	100000	1795246
25	100000	2401876
50	100000	3031986
100	100000	3639162
5	1000000	71538
10	1000000	138603
25	1000000	203034
50	1000000	269653
100	1000000	333754

11.3 complete_cast.csv

complete_cast.csv: for each method we mesured initial file size 2414495 and output file size 2414495.

d	M	Average time (μs)
5	1000	20852225
10	1000	35940108
25	1000	44097325
50	1000	49904848
100	1000	55588455
5	10000	9752929
10	10000	17119286
25	10000	22178716
50	10000	27113926
100	10000	31960066
5	100000	4848086
10	100000	9787121
25	100000	12317736
50	100000	14890729
100	100000	17477266
5	1000000	2745860
10	1000000	5563283
25	1000000	8336591
50	1000000	11064772

11.4 movie_info_idx.csv

d	M	Average time (μs)
5	1000	218445713
10	1000	367100757
25	1000	488176029
50	1000	589359849
100	1000	686303061
5	10000	162135690
10	10000	272643108
25	10000	357517926
50	10000	449317369
100	10000	505707007
5	100000	105600523
10	100000	185394542
25	100000	240651925
50	100000	295717068
100	100000	349967642
5	1000000	78802805
10	1000000	134427844
25	1000000	188023874
50	1000000	217611579
100	1000000	247930266

References

[1] *Memory mapping*
https://www.clear.rice.edu/comp321/html/laboratories/lab10/

[2] *Wikipedia : Mmap*
https://en.wikipedia.org/wiki/Mmap

[3] *Wikipedia : Memory Mapped file*
https://en.wikipedia.org/wiki/Memory-mapped_file

[4] https://dannythorpe.com/2004/03/19/the-hidden-costs-of-memory-mapped-files/

[5] https://sasha-f.medium.com/why-mmap-is-faster-than-system-calls-24718e75ab37

[6] Database System Architecture , *Stijn Vansummeren*