

Database Systems Architecture

Project: External merge-sort



Submitted to:

Prof. [Stijn Vansummen](#)

[INFO-H-417 : Database Systems Architecture](#)

Submitted by:

Carlos Martinez Lorenzo (000477671)

Ioannis Prapas (000473813)

Sokratis Papadopoulos (000476296)

January 2019

Contents

1	Introduction and Environment	3
1.1	Data generation for Testing	3
1.2	Environment Setup	3
1.3	Benchmark Runs	4
2	Observations on Streams	5
2.1	Expected Behavior	5
2.1.1	Read/Write Implementation 1	6
2.1.2	Read/Write Implementation 2 (Default Buffer)	7
2.1.3	Read/Write Implementation 3 (Variable buffer size)	8
2.1.4	Read/Write Implementation 4 (Memory mapping)	9
2.2	Experimental Observations	13
2.2.1	N – varying on number of integers	13
2.2.2	B – varying on buffer size	14
2.2.3	k – varying on number of streams	15
2.3	Discussion of Expected Behavior VS Experimental Observations	17
3	Observations on multi-way merge sort	18
3.1	Theoretical View	18
3.1.1	Algorithm Description	18
3.1.2	Cost Calculation	19
3.1.3	Expected Behavior	19
3.2	Experimental Observations	20
3.2.1	N – varying on number of integers	20
3.2.2	M – varying on memory size	20
3.2.3	d – varying on number of streams sorted in one pass	21
3.3	Comparison with Internal Sorting	23
3.4	Discussion Of Expected Behavior VS Experimental Observations	23
4	Conclusion	24

1 INTRODUCTION AND ENVIRONMENT

The main outcome of this assignment is an external-memory merge-sort algorithm, developed in Java and thoroughly tested under different parameters. As a first task, we programmed four (4) different ways of reading and writing 32-bit integers from and to secondary memory into different input and output stream classes, supporting all basic operations. We experimented with reading and writing streams, testing them out by creating multiple number of streams, small and bigger files and a variety buffer sizes.

Apart from the standard java libraries, we used `sun.misc.Cleaner` to unmap a mapped file for the 4th IO implementation as suggested in a stackoverflow answer¹. Other than this, some notable usage from the standard library are:

- `java.util.PriorityQueue` (as a minimum heap)
- `java.util.Collections.sort` (for in-memory sorting)

1.1 DATA GENERATION FOR TESTING

Generating data is implemented within our program. Our file generator class (`GenerateFile`) produces 32-bit unsigned signed integer binary files, taking `N` (number of integers) as a parameter and utilizing `Random` java class it produces positive and negative integers based on `nextBoolean()` (to decide the sign: +/-) and `nextInt()` (to produce the random number, from 0 to `Integer.MAX_VALUE`). It outputs two files:

- A binary file that lists all randomly produced integers, using our output stream implementation.
- For `N` (number of integers produced) less or equal to 1 thousand, it also creates a corresponding non-binary human-readable file, enabling us to visually check on the way our read/writes and our external merge sort algorithm work. To do that it uses the `PrintWriter` java class.

For read/write testing we have created files from 1 thousand to 10 million integers (`N`), experimenting with 2 to 30 streams (`k`) and buffer size (`B`) varying from 10 to 10 million.

For external merge sort experiments, we have used files containing from 10 thousand to 100 million integers (`N`), memory (`M`) varying from 10 to 100 thousand and `d` (number of streams to merge in one pass) varying from 2 to 1000.

1.2 ENVIRONMENT SETUP

A machine with the following setup has been used for running the experiments:

- Operating System: Ubuntu Linux 16.04
- Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
 - 4 processors (each with 2 CPU cores, 3072 KB cache and 3100 MHz)
 - 64-bit
- RAM: 8 GB
 - DDR4
- Hard disk

¹ <https://stackoverflow.com/questions/5989267/truncate-memory-mapped-file>

- SanDisk SD8TB8U2
- SSD

1.3 BENCHMARK RUNS

To run our experiments, we used perf-stat tool². More specifically, we ran it with the following parameters:

```
1. perf stat -d -d -d -r 5
```

This means that we want to run the script 5 times (-r 5) and get as many details as possible (-d -d -d) for the run. As a result, we get an output like the following:

```

4 Performance counter stats for 'java Main 1 1 16384 1000 30' (5 runs):
5
6      313.419995 task-clock (msec)    #    0.972 CPUs utilized          ( +-  2.01% )
7      219 context-switches          #    0.698 K/sec                  ( +-  1.68% )
8      17 cpu-migrations              #    0.054 K/sec                  ( +- 11.20% )
9      2'546 page-faults              #    0.008 M/sec                  ( +-  0.33% )
10     946'652'545 cycles               #    3.020 GHz                    ( +-  0.37% ) (29.96%)
11     1'167'895'230 instructions        #    1.23 insn per cycle          ( +-  2.82% ) (37.76%)
12     228'346'474 branches             #    728.564 M/sec                ( +-  2.32% ) (37.08%)
13     2'823'948 branch-misses          #    1.24% of all branches        ( +-  2.31% ) (37.56%)
14     372'273'349 L1-dcache-loads       #   1187.778 M/sec                ( +-  1.10% ) (38.26%)
15     6'944'541 L1-dcache-load-misses   #    1.87% of all L1-dcache hits   ( +-  2.51% ) (38.41%)
16     1'279'067 LLC-loads              #    4.081 M/sec                  ( +-  2.53% ) (30.11%)
17     119'461 LLC-load-misses           #    9.34% of all LL-cache hits    ( +-  7.53% ) (29.72%)
18 <not supported> L1-icache-loads
19     31'347'721 L1-icache-load-misses   ( +-  2.69% ) (30.12%)
20     353'425'987 dTLB-loads           #   1127.643 M/sec                ( +-  2.74% ) (31.63%)
21     73'782 dTLB-load-misses          #    0.02% of all dTLB cache hits ( +-  4.87% ) (32.81%)
22     455'631 iTLB-loads              #    1.454 M/sec                  ( +-  3.58% ) (32.72%)
23     88'207 iTLB-load-misses          #   19.36% of all iTLB cache hits ( +- 29.25% ) (31.62%)
24 <not supported> L1-dcache-prefetches
25 <not supported> L1-dcache-prefetch-misses
26
27     0.322578237 seconds time elapsed ( +-  2.99% )
28

```

Figure 1 Output of a benchmark run

For the purpose of this report, we focus on the time elapsed (on line 27), however we also used rest measures to get a better understanding of the results. The scripts we used to perform the experiments discussed on the rest of the report are available in the scripts/ directory of our repository³. The analytical results we got are all stored in the results/ directory.

² <http://man7.org/linux/man-pages/man1/perf-stat.1.html>

³ <http://wit-projects.ulb.ac.be/rhocode/INFO-H-417/2018-2019-1/project-abmartin-iprapas-sopapado>

2 OBSERVATIONS ON STREAMS

In this section we are explaining how we programmed the four different read/write implementations and discuss on the expected performance. Then we are testing out all implementations with various parameters and we present and discuss the outcome.

2.1 EXPECTED BEHAVIOR

To build sustainable and extensible read/write implementations we created a class for each read/write way, all of them inheriting from a relevant abstract superclass. To make our code cleaner and implementation-independent we also created another class which has one attribute: the abstract superclass, being able to create (according to each run) the instance of the relevant read/write implementation required. To visualize a bit the code, here is the general abstract Input Stream (named `InStream`) and Output Stream (`OutStream`) classes:

```
public abstract class InStream {  
  
    protected String path;  
    protected InputStream is;  
    protected DataInputStream ds;  
  
    public abstract void open() throws IOException;  
    public abstract void open(int pos) throws IOException;  
    public abstract int read_next () throws IOException;  
    public abstract boolean end_of_stream() throws IOException;  
    public abstract void close() throws IOException;  
}  
  
public abstract class OutStream {  
  
    protected OutputStream os;  
    protected DataOutputStream dos;  
    protected String path;  
  
    public abstract void create() throws IOException;  
    public abstract void create(int skip) throws IOException;  
    public abstract void write (int element) throws IOException;  
    public abstract void close() throws IOException;  
}
```

As you can see, it includes all functions required for its children-classes to implement (open, read_next, end_of_stream, close). Plus, we added open(int) to be able to open files in specific position (by skipping some amount of bytes). The same goes for output stream having create, write, close, with the addition of create(int) to write into specific position of files. All attributes are protected, so that they can be easily referenced by their subclasses.

Plus, here is part of the code for the implementation-independent `ReaderStream` class that is getting initialized accordingly, based on the specified implementation (`WriterStream` is identical and thus not shown). As you can see, according to what the implementation number is, it creates the relevant instance of the corresponding read/write class.

```
1. public ReaderStream(int implementation, String filepath, int buffersize) {  
2.     switch (implementation) {  
3.         case 1:  
4.             is = new InputStream1(filepath);  
5.             break;  
6.         case 2:  
7.             is = new InputStream2(filepath);  
8.             break;  
9.         case 3:  
10.            is = new InputStream3(filepath, buffersize);  
11.            break;  
12.         case 4:  
13.            is = new InputStream4(filepath, buffersize);  
14.            break;  
15.         default:  
16.            System.out.println("Please select implementation among [1,4]");
```

```
17.     }  
18. }
```

Finally, here is an example on how to create an instance of an implementation, where IMPLEMENTATION can be any of 1,2,3,4.

```
1. ReaderStream rs = new ReaderStream(IMPLEMENTATION, infile, BUFFERSIZE);  
2. WriterStream ws = new WriterStream(IMPLEMENTATION, outfile, BUFFERSIZE);
```

We have used these classes instances throughout different implementations.

2.1.1 Read/Write Implementation 1

We created class InputStream1 extending InStream and OutputStream1 extending OutStream to carry out the first implementation. This implementation is based on a DataInputStream instance, wrapping the InputStream. As a result, functions are working on integer-by-integer fashion instead of byte-by-byte (which would be the case given the absence of DataInputStream wrapper).

Since there is no buffer introduced, we are expecting slow performance, as we need to access the secondary memory each time we request an integer. In other words, we require as many I/O's as tuples (in our case integers) exist in our input. With high number of I/O's the performance is poor, especially as the input file is getting bigger. To support this mathematically, we hereto define a formula calculating number of I/O's needed for each input file (where N: number of integers in input file, k: number of streams used and B: buffer size). For k streams we need to read and write one integer at a time. As we need 1 I/O for each integer reading and 1 I/O for each integer writing, the I/O cost formula is the following:

$$Cost(Impl1) = 2 * k * N$$

2.1.2 Read/Write Implementation 2 (Default Buffer)

We have created class `InputStream2` extending `InStream` and `OutputStream2` extending `OutStream` to carry out the second implementation. This implementation is similar to the previous one, but we are going to use a `BufferedInputStream` to wrap `InputStream` and `DataInputStream` to wrap the `BufferedInputStream`. As a result, functions are working on integer-by-integer fashion instead of byte-by-byte (which would be the case given the absence of `DataInputStream` wrapper) and since there is a buffer involved we can read groups of integers all together from secondary memory (as many as buffer size allows), instead of reading them one by one.

Since there is a buffer with default size introduced, we are expecting an improvement on performance in comparison with the first implementation. We no longer require as many I/O's as tuples (integers) exist in our input. In this case we are reading batch of integers at a time keeping them within the buffer, resulting in much less I/O's. More specifically, the I/O's required is going to be the number of times we load the buffer with the next group of integers, so it is sensitive to the size of the buffer. The default buffer size used by Java is defined within the SDK. We used SDK 1.8 and default buffer size is defined as 8192 bytes. Since we are reading 32-bit integers (4 bytes), the buffer size fits 2048 integers.

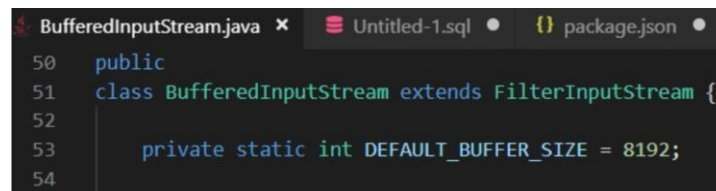
A screenshot of a code editor with a dark background. The editor shows a file named 'BufferedInputStream.java' with line numbers 50 through 54. The code is as follows: line 50: 'public', line 51: 'class BufferedInputStream extends FilterInputStream {', line 52: ' private static int DEFAULT_BUFFER_SIZE = 8192;', line 53: '}', line 54: '}'. The tabs at the top show 'BufferedInputStream.java', 'Untitled-1.sql', and 'package.json'.

Figure 2 Default buffer size of Java SDK 1.8

Considering that we need one pass for reading and another one for writing, and we implement it on `k` streams, the cost formula is the following:

$$Cost(Impl2) = 2 * k * \lceil N/B \rceil$$

And in our case, using SDK 1.8, it is going to be:

$$Cost(Impl2) = 2 * k * \lceil N/2048 \rceil$$

2.1.3 Read/Write Implementation 3 (Variable buffer size)

We have created class `InputStream3` extending `InStream` and `OutputStream3` extending `OutStream` to carry out the third implementation. This implementation is like the previous one, with the only difference being that we are using a `BufferedInputStream` with a varying buffer size instead of the default one.

Since we are going to modify the buffer size, we expect an improvement of the performance in comparison with the second implementation as we increase the buffer size. However, as later noticed, bigger buffer sizes did not produce greater performance. This happens because, as the buffer size is incremented, so as the size of the array that is going to store the numbers. But, if the cache size of the machine and the block size of the hard disk are smaller, there is a performance limit, independent of the how big the buffer may be⁴.

```
public BufferedInputStream(InputStream in, int size) {
    super(in);
    if (size <= 0) {
        throw new IllegalArgumentException("Buffer size <= 0");
    }
    buf = new byte[size];
}
```

Figure 3 Specifying buffer size within `BufferedInputStream`

The cost formula is the following, similar to implementation 2:

$$Cost(Impl3) = 2 * k * \lceil N/B \rceil$$

Where k is the number of streams used, N the number of integers, B the buffer size and $CacheSize$ is the size of the cache of the machine.

Note

The above formula wrongly suggests that increasing the buffer size will decrease the IO cost. But this is not correct, as no matter how many bytes we buffer in one go, in the end we have to transfer some bytes from secondary memory to main memory. This is an operation that has a cost we cannot avoid. The formula just gives the number of IO requests that our code will execute.

⁴ <http://tutorials.jenkov.com/java-io/bufferedinputstream.html#optimal-buffer-size>

2.1.4 Read/Write Implementation 4 (Memory mapping)

For the last implementation we have utilized the memory mapping functionality of Java, built within java.nio library. Before describing our implementation, we will focus on explaining the concept of memory mapping, as it is working on different mentality comparing to regular file I/O streams that we implemented above.

2.1.4.1 What is memory mapping?

To begin with memory mapping, let's dive a bit more into how traditional read and write calls work in a lower level and then show how it differs. A process requests a read call to fill its buffer, which results to disk controller fetching the requested data from disk and importing it into kernel buffer through DMA. Then kernel copies the data from its buffer to process buffer and then the process can use it as requested. An illustration of the above process is displayed on the figure below.

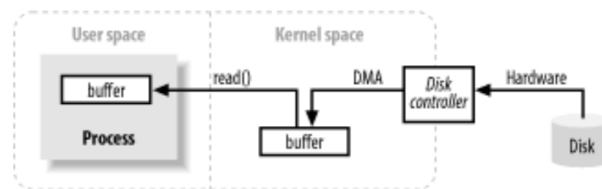


Figure 4 how data moves from disk to process memory. Image taken from article⁵.

Now, before diving into memory mapping, we need to understand the concept of virtual memory, which is used in memory mapping. Virtual memory is providing processes the impression that they have more RAM space than they actually do, letting them to behave like all data they require are already in memory, but instead majority of them are stored on disk and the transfer between the two is made upon request.

As a matter of fact, processes constantly request space in RAM to execute. At some points in time, the running processes' RAM space requests exceeds the total RAM space available. Then the memory manager takes some data that is in RAM (that is not currently needed) and copies it to the swap space into hard disk (to be fetched later upon request). This swap needs to happen very carefully because disk access is way slower than RAM access. So, the operating system memory manager is really good and has been highly optimized at this job: moving items from RAM to swap space on disk. And that is the exact mechanism that is being used by Memory Mapping in order to read and write files from secondary memory.

Memory mapping assigns a region of virtual address space to be logically associated with the whole or part of the file content. That way the program acts like the whole file (or part of it, depending on if given buffer size fits the whole file or not) is loaded into memory. So, instead of issuing system calls (like read or write), it lets this job for the operating system, as it knows how to do it best.

Memory mapping is using a "lazy" I/O mechanism that prevents this extra (above mentioned) copy of requested data from kernel buffer to process buffer. Instead, the disk controller through DMA fills a buffer that is simultaneously visible to both kernel and user space.

⁵ <https://howtodoinjava.com/java/io/how-java-io-works-internally-at-lower-level/>

Resources used to study upon memory mapping technology and base our explanation: Wikipedia⁶, Unix.Stackexchange⁷, HowToDoInJava⁸, Java NIO e-book⁹, YouTube channel (Jacob Sorber)¹⁰.

2.1.4.2 Implementation details

For Memory Mapped read implementation we created class `InputStream4`, extending `InStream` as usual. Plus, we imported Java library `java.nio`. Hereby you can find the attributes of the class.

```
1. private final int bsize;           //buffer size given
2. private FileChannel fc;
3. private long fileSize;
4. private long memPos = 0;           //absolute memory position
5. private long runningPos = 0;       //relative memory position
6. private MappedByteBuffer mem;
```

To open a file we use a `FileChannel` (`fc`) where we initiate a `RandomAccessFile` for the given path, in order to get a channel for the file. Then memory mapped byte buffers are created with the `fc.map()` call, which occupies a space as big as given buffer size, except if the file size is smaller, thus we limit the space to the file size. Lastly, we initialize `memPos` and `runningPos` to zero, as we start from the beginning of the file.

```
1. @Override
2. public void open() throws IOException {
3.     fc = new RandomAccessFile(path, "rw").getChannel();
4.     fileSize = fc.size();
5.     mem = fc.map(FileChannel.MapMode.READ_ONLY, 0, min(fileSize, bsize));
6.     memPos = 0;
7.     runningPos = 0;
8. }
```

The second function we implemented is `open(int)` which opens a file at a specific place, by skipping a given number of integers. It works the same as above with the difference that the mapping starts from the specified position, which naturally also equals with the `memPos`.

```
1. @Override
2. public void open(int skip) throws IOException {
3.     int byteSkip = skip * 4; //as each 32-bit integer captures 4 bytes.
4.     fc = new RandomAccessFile(path, "rw").getChannel();
5.     fileSize = fc.size();
6.     mem = fc.map(FileChannel.MapMode.READ_ONLY, byteSkip, min(fileSize - byteSkip, bsize));
7.     memPos = byteSkip;
8.     runningPos = 0;
9. }
```

⁶ https://en.wikipedia.org/wiki/Memory-mapped_file

⁷ <https://unix.stackexchange.com/questions/474926/how-does-memory-mapping-a-file-have-significant-performance-increases-over-the-s>

⁸ <https://howtodoinjava.com/java7/nio/memory-mapped-files-mappedbytebuffer/>

⁹ [http://ebooks.bharathuniv.ac.in/gdlc1/gdlc1/Computer%20Science%20Books/\(ebook-pdf\)%20-%20O'Reilly%20-%20Java%20NIO.pdf](http://ebooks.bharathuniv.ac.in/gdlc1/gdlc1/Computer%20Science%20Books/(ebook-pdf)%20-%20O'Reilly%20-%20Java%20NIO.pdf)

¹⁰ <https://www.youtube.com/watch?v=m7E9piHcfr4>

Third function is `read_next()` where we read the next integer in line. In order to get next integer, we first check whether we have overcome the memory limit, so as to request a new memory mapped byte buffers this time starting from the current `memPos` and until as much buffer size or file size allows. For file size computation we need to calculate the distance of the end of file from the current memory position, in order to get the correct remaining part of the file.

```
1. @Override
2. public int read_next() throws IOException {
3.
4.     if (runningPos >= mem.limit()) {
5.         mem = fc.map(FileChannel.MapMode.READ_ONLY, memPos, min(bsize, fileSize - memPos));
6.         runningPos = 0;
7.     }
8.     runningPos += 4; //as next int is 4 bytes away
9.     memPos += 4;    //as next int is 4 bytes away
10.    int nextInt = mem.getInt();
11.    return nextInt;
12. }
```

The `end_of_stream()` function is quite straight forward as we simply check if the current memory position is out of bounds, by comparing it with file size and returning true/false accordingly.

```
1. @Override
2. public boolean end_of_stream() throws IOException {
3.     return (memPos >= fileSize);
4. }
```

Lastly, `close` function does not need to perform any action as `MappedByteBuffer` will be garbage-collected by Java.

Regarding write functionality, we have implemented `OutputStream4`, as usual extending `OutputStream` class. As implementation is similar with the read one, we will just focus on closing function which worth commenting upon. At first we tried to base solely on `System.gc()`, but the solution was not reliable (worked in one PC and not in another). That is why, to reliably implement the close function we made use of `sun.misc.Cleaner` library, in order to unmap the mapped file as suggested on a stackoverflow answer¹¹. After closing the file mapping, it truncates the file keeping data only up until the current memory position (which is the last position where an integer is written).

```
1. @Override
2. public void close() throws IOException {
3.     if (mem instanceof sun.nio.ch.DirectBuffer) {
4.         sun.misc.Cleaner cleaner = ((sun.nio.ch.DirectBuffer) mem).cleaner();
5.         cleaner.clean();
6.     } else {
7.         mem = null;
8.         System.gc();
9.     }
10.    fc.truncate(memPos); //FileChannel
```

¹¹ <https://stackoverflow.com/questions/5989267/truncate-memory-mapped-file>

```
11.     fc.close();  
12. }
```

Lastly, we discuss the cost formula for memory mapping. As we have studied, memory mapping is proven to perform impressively well. In our case, we expect a performance boost because we have spatiotemporal locality when referencing values, meaning that we read/write integers that are close to each other in memory.

$$Cost(Impl4) = 2 * k * \lceil N/B \rceil$$

2.2 EXPERIMENTAL OBSERVATIONS

On our experiments of reading and writing streams we have varied N (number of integers) from 1 thousand to 10 million, experimenting with 2 to 30 streams (k) and buffer size (B) varying from 10 to 10 million. In each implementation, we vary only one parameter. In this section we present and analyze the outcome of our experiments.

2.2.1 N – varying on number of integers

The first experiment is focused on N, the number of integers. The number of streams (k) is 30 and the buffer size (B) depends on the implementation, the first one does not have a buffer, the second one has the default buffer size of the SDK 1.8 of java that is 2,048 integers (8,192 bytes) and for the third and fourth implementations it is 4,096 bytes.

The next table displays all timings from the first experiment. We can see that the worst-performing implementation is the first one, spending already 23 seconds on 100,000 integers, while the rest are still performing under a second. We omitted the output of first implementation for 1 and 10 million numbers, as it was extremely high. The second and third implementation already spend more time than the fourth one on 100,000 items, but the difference is considerably higher on 1,000,000 where the fourth one completes in less than a second, while the second and third are completing in around 6 seconds. Finally, the performance difference is way more visible as the number of integers increase, as memory mapping is 3 times faster (17 seconds) on 10,000,000 than second and third (around 57 seconds).

number of N \ Implementation	1,000 (sec)	10,000 (sec)	100,000 (sec)	1,000,000 (sec)	10,000,000 (sec)
1	0.323	2.441	23.767	-	-
2	0.103	0.149	0.641	5.789	57.829
3	0.128	0.153	0.648	5.901	56.784
4	0.163	0.125	0.180	0.915	17.622

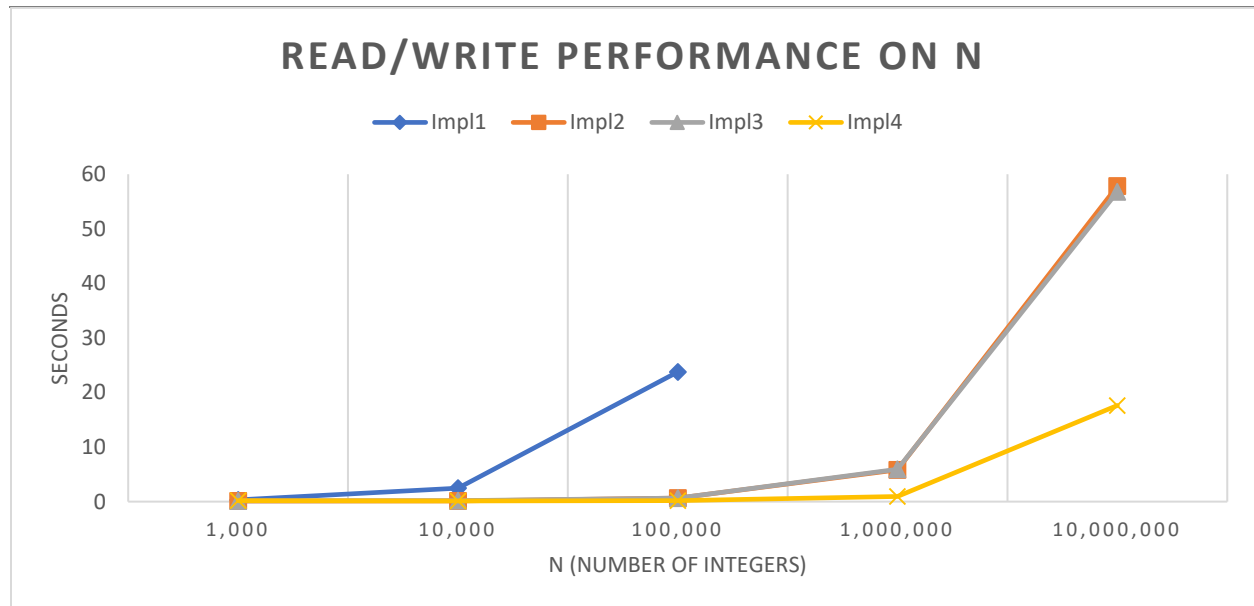


Figure 5: IO performance of the different implementation with a variety of input sizes (N)

As a conclusion, memory mapping outperforms the rest implementations and the difference is getting higher as the parameter N (number of integers) is growing. For large files it seems like memory mapping is the only one to maintain an acceptable performance.

2.2.2 B – varying on buffer size

The second experiment is focused on B, the buffer size. The number of streams (k) is 30 and the number of integers (N) is 10,000,000. We have analyzed the third and the fourth implementation on the variance of buffer size, and we also display the performance of the second implementation on 10,000,000 integers as a reference.

In this experiment we observe the importance of buffer size in performance. Comparing the second with the third implementation, they display a similar timing on same buffer size. However, the more we decrease the buffer size on the third implementation, the significantly more time it spends. When we instead increase the buffer size, the performance improvement we get is insignificant, due to the limitation explained on section 2.1.3. We also observe how poorly memory mapping performs on small buffer size and we briefly discuss it on section 2.3.

Buffersize \\Implementation	40 (sec)	400 (sec)	4,000 (sec)	40,000 (sec)	400,000 (sec)	4,000,000 (sec)
2	-	-	57.829	-	-	-
3	126.252	64.738	58.557	56.453	55.698	56.152
4	775.960	10.208	6.571	6.313	6.070	6.831

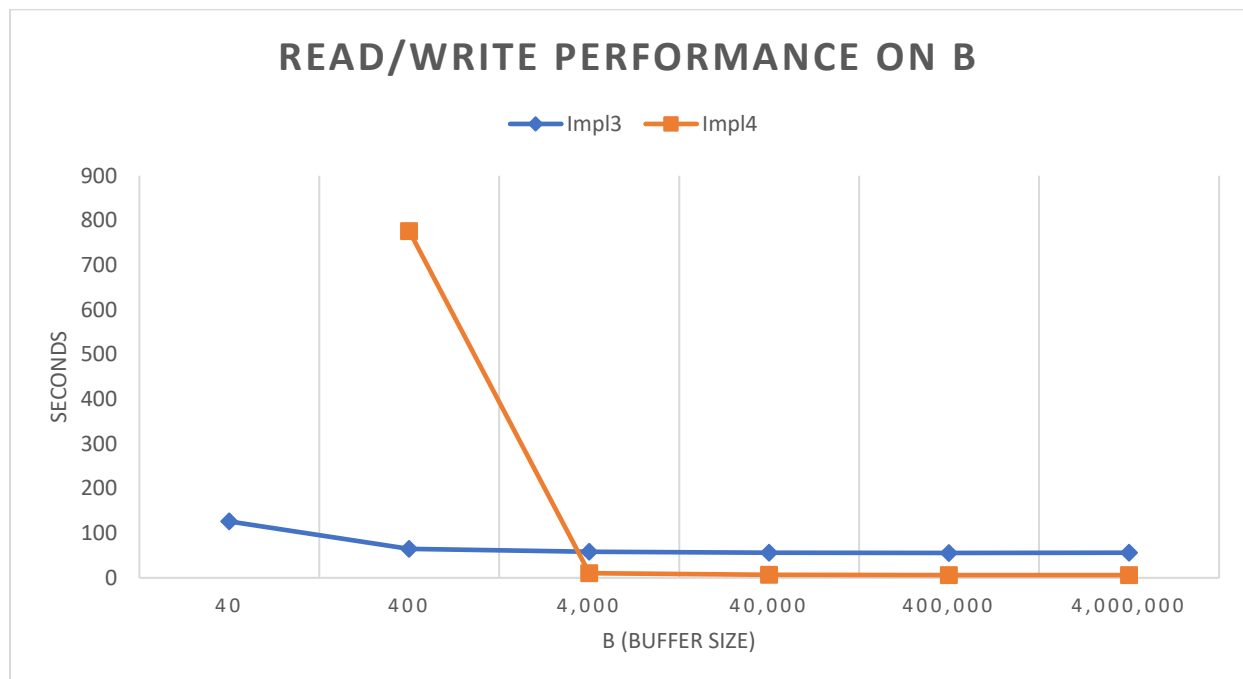


Figure 6: IO performance with varying buffersize

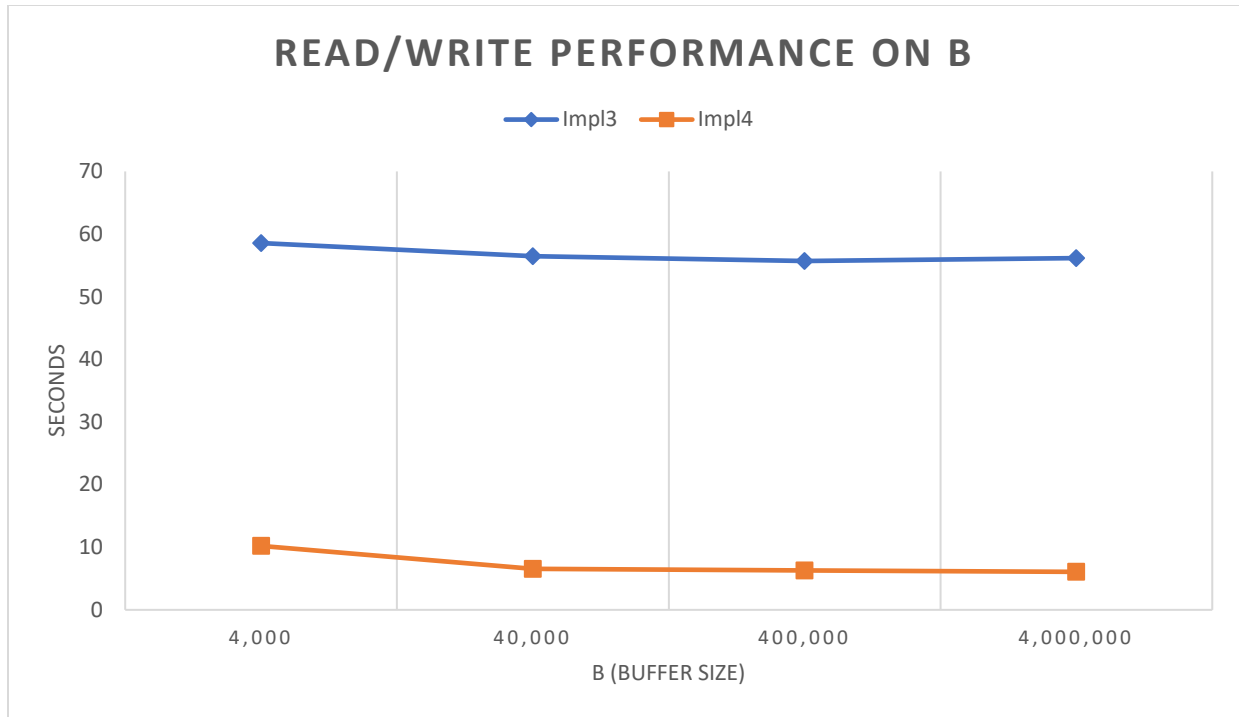


Figure 7: IO performance with varying buffersize (zoomed in after saturation)

2.2.3 k – varying on number of streams

The third experiment is focused on k, the number of streams. The buffer size is 16,384 (for the third and fourth implementations) and the number of integers is 100,000.

Here we can observe that the first implementation is considerably slow and its timings grow quickly. However, we do not have significant differences between the second and third implementation, and we can see that the fourth one is still performing the best.

Moreover, for the second and third implementation, the curve starts to grow with faster pace when the number of streams is bigger than 8, while the fourth implementation scales well, with performance remaining rather stable as k increases.

Streams\ Implementation	2 (sec)	4 (sec)	8 (sec)	16 (sec)	30 (sec)
1	1.576	3.084	6.127	12.934	23.619
2	0.097	0.135	0.207	0.360	0.663
3	0.096	0.135	0.204	0.352	0.660
4	0.088	0.080	0.101	0.111	0.160

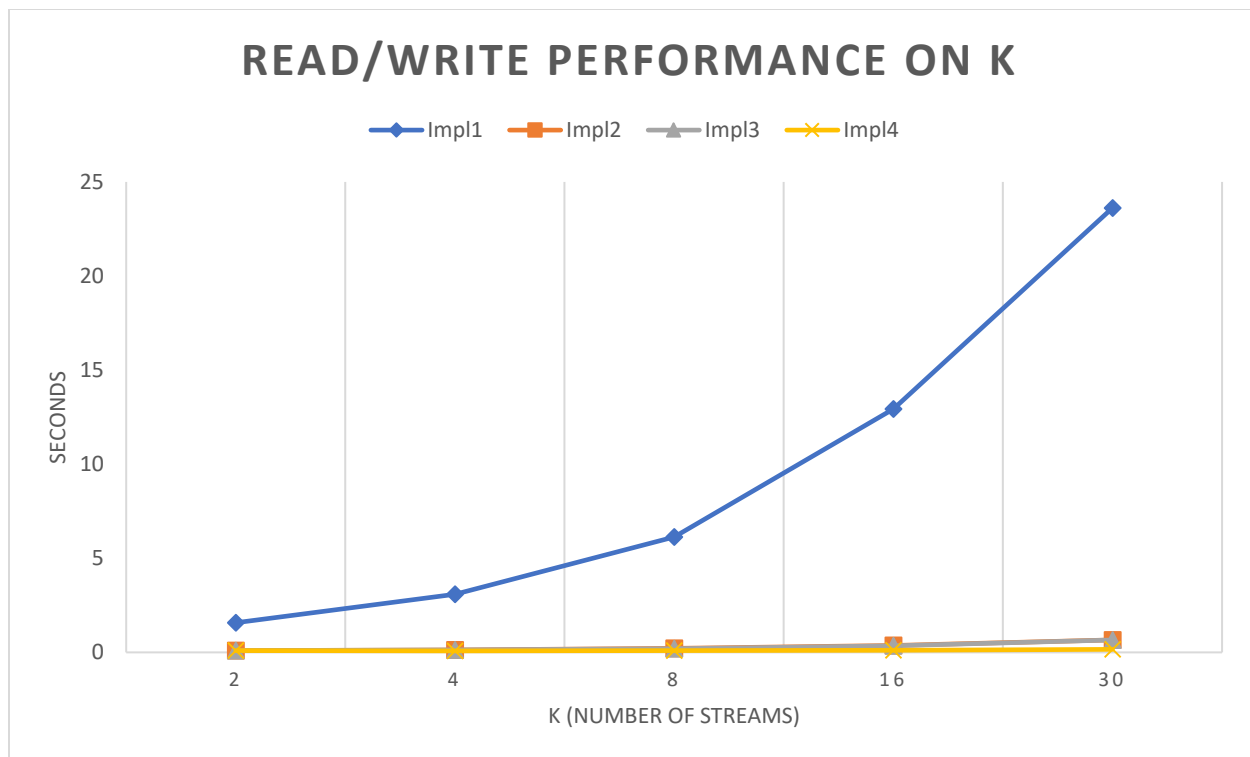


Figure 8: Read/Write performance with varying the number of streams

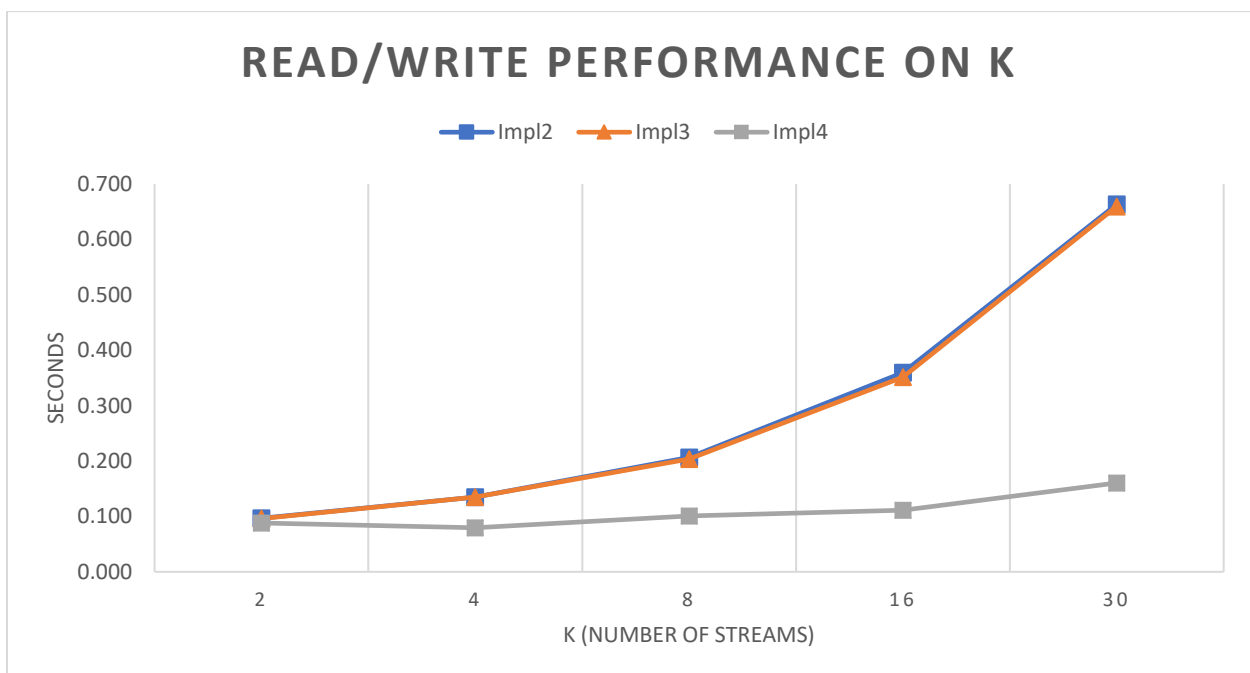


Figure 9: Read/Write performance with varying the number of streams (all implementations except for 1st one)

2.3 DISCUSSION OF EXPECTED BEHAVIOR VS EXPERIMENTAL OBSERVATIONS

After all the experiments, we can see that the first unbuffered implementation has the worst performance; an observation that goes in par with the expected behavior.

Then, we see that the difference of using a buffer reader with the default buffer size (second implementation) or with a variable buffer size (third implementation) is not very significant, with a buffer bigger than the default value. This is a bit counterintuitive and probably has to do with our system-specific details. To get a benefit from modifying the buffer size, it is perhaps smart to consider details like disk block size, memory page size and cache size. In our experiments, we chose the buffer size values arbitrarily and therefore we couldn't get better results. However, in general we notice that ever growing buffer size does not offer greater performance, as performance is limited by other factors, as explained in section 2.1.3. Plus, as expected, having a very small buffer significantly lowers performance.

At last, the fourth implementation scales the best as the map size gets bigger. After a certain value of map size, we can easily see that this implementation achieves superior performance than the previous ones. We cannot identify an optimal value for it, as we suspect that performance gets better as the mapping gets bigger. This is the expected behavior for our application, which has a lot of reads and writes with spatiotemporal locality. It is also worth mentioning that memory mapping performs very poorly on low buffer size. That happens because mapping has a great overhead cost, which is more important when the input is small.

As a result of our experiments, we decided to use the memory-map implementation for the IO operations needed for the next part, the external multi-way merge-sort algorithm.

3 OBSERVATIONS ON MULTI-WAY MERGE SORT

In this section we will analyze the external multi-way merge sort going through its algorithm details, its expected and actual performance, as well as a discussion over our observations.

3.1 THEORETICAL VIEW

In this chapter, we provide an overview of the algorithm, we analyze the expected IO cost and give our expectations on the effect of the different parameters.

3.1.1 Algorithm Description

Given N integers and M memory the external multi-way merge sort is outputting one file of sorted integers. As N can be big enough, the algorithm is utilizing disk to store the integers, since main memory can be insufficient.

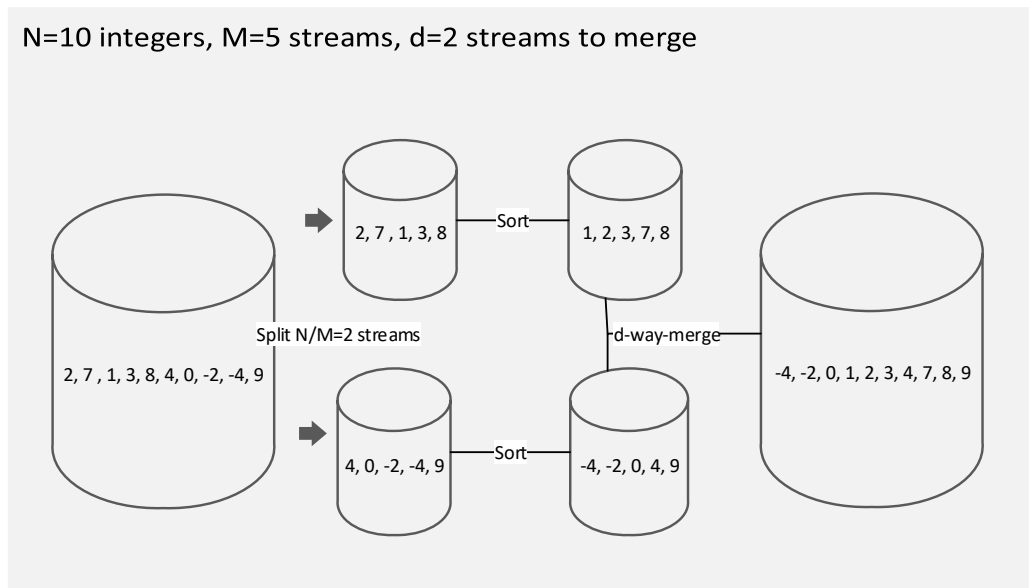


Figure 10 Example of algorithm execution

Firstly, the algorithm starts on **ExternalMergesort** class constructor, by splitting the input (N) into $\lceil N/M \rceil$ segments, each of them having at most M integers (in fact all of them list M integers, except the last segment that may have less than M) and thus completely fits into memory.

Then, using the *sortStream(stream, counter)* function, which takes as an input the current stream to sort and its identifying number, it loads those segments one-by-one (stream-by-stream) into the main memory and sorts them there using the Java default in-memory sort, before writing them back to disk in a sorted manner. Up until now we have constructed several sorted segments, all stored in disk.

The last job of **ExternalMergesort** class is to create a **MultiwayMerge** class instance and repeatedly call its *merged()* function to merge d streams at a time, until there are less than d streams left. Then, it is time for the final merge to occur, on *finalMerge()* function of the **MultiwayMerge** class.

```

1. MultiwayMerge mw = new MultiwayMerge(sortedStreams, d, B);
2. while (mw.getSize() > d) {
3.     mw.mergeD();
4. }
5. mw.finalMerge();

```

Inside those `mergeD()` and `finalMerge()` functions, we perform a synchronous merging using the multi-way mergesort. The algorithm accepts the d sorted lists of integer values and outputs one merged list into a file. To do that it uses a min priority queue, where it stores one integer from each stream synchronously and always outputs the minimum value.

More specifically, the algorithm gets the first value from each of the d streams and stores them all within the min priority queue (heap). As all streams are sorted, the heap now possesses the d globally minimum values, storing the minimum of them on root. We extract the root and write it through the output stream into a file and we insert a new value from the stream that min value belonged. We do that until all streams are empty.

To be able to know in which stream each number belongs, the heap accepts instances of custom-made **QueueItem** class implementing Comparable (to perform the sorting), where we store both the number and the stream it belongs to. Every time a new value needs to be fetched from a specific stream, it moves the pointer 4 bytes (size of integer) and reads the following 4 bytes to get the new value.

If the algorithm is getting executed for the last d streams (meaning it will output one and only stream -> our final output) then the output stream points into a text file under `final_output` folder, where all integers of the input will appear in sorted order.

3.1.2 Cost Calculation

For the calculation of the cost of the external mergesort, we will ignore any speedup we can get from buffering. We have $\lceil N/M \rceil$ streams and we merge d of them at a time. To merge them all we need to do $\log_d(\lceil \frac{N}{M} \rceil)$ passes and in each pass, we read and write N integers. Therefore, the total cost becomes:

$$Cost_{External_Mergesort}(N, M, d) = 2 * N * \left\lceil \log_d \left(\left\lceil \frac{N}{M} \right\rceil \right) \right\rceil$$

3.1.3 Expected Behavior

Analyzing this formula, we expect that:

- As the number of integers to sort (N) grows, the algorithm will take more time.
- As the memory (M) grows, the algorithm will take less time until $N = M$, at which point we can do in-memory sorting.
- As the streams to merge at once (d) grows, the algorithm will perform better until $d = N/M$, at which point the merging can happen in only one pass, which is optimal.
- Internal mergesort will significantly outperform external mergesort as I/O's number should be much lower.

3.2 EXPERIMENTAL OBSERVATIONS

To experiment with our algorithm implementation, we used files from 10 thousand to 100 million integers (N), memory (M) varying from 10 to 100 thousand and d (number of streams to merge in one pass) varying from 2 to 1 thousand.

For the implementation of the external merge sort we have used the fourth read/write implementation (memory mapping), so we need to specify the buffer size in each experiment.

3.2.1 N – varying on number of integers

The first experiment is focused on N, the number of integers. The buffer size is 4,000 (B), the memory fits 1000 integers (M) and the number of streams to merge in one pass (d) is 1,000.

The following table shows the execution times of the experiment, varying the number of integers (N).

N	10,000	100,000	1,000,000	10,000,000	100,000,000
External Mergesort Time (sec)	0.098	0.218	0.678	5.787	53.186

Below we showcase the results of our experiments into plots.

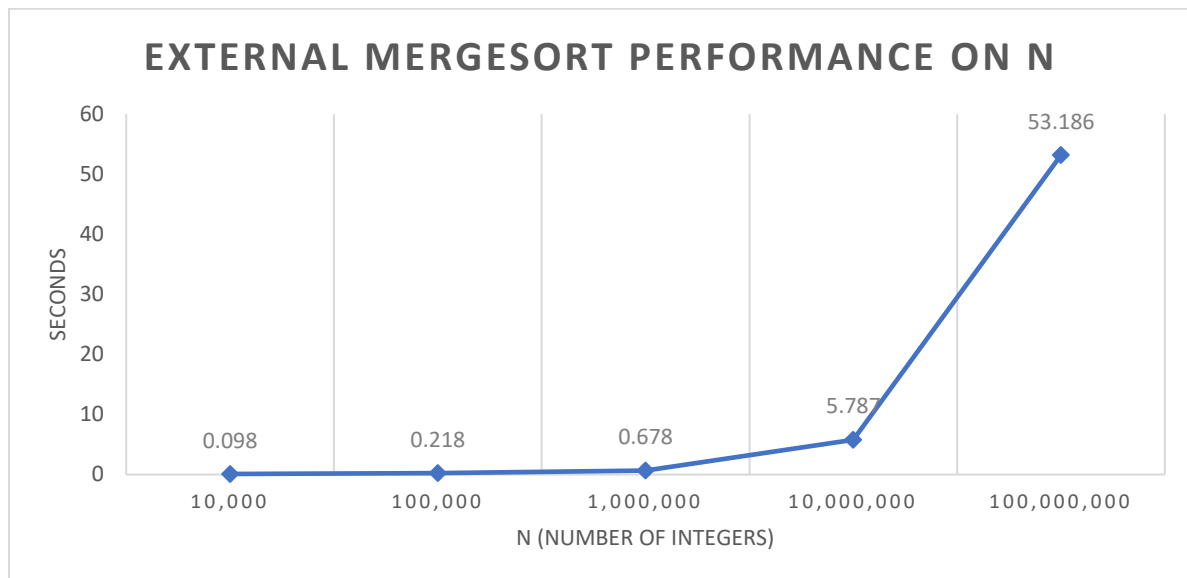


Figure 11: Effect of number of integers on performance of external mergesort algorithm

As expected, the external mergesort algorithm takes more time, as the size of the input grows. In the plot we can observe, that the scaling up is almost linear to the size of the input. As $d=1000$ and memory (M) fits 1000 integers, the first three executions (10 thousand, 100 thousand, 1 million) require one pass to sort the d streams and then one pass to merge all sorted streams, which showcase very high performance.

3.2.2 M – varying on memory size

The second experiment is focused on M, the memory space in number of integers. The buffer size is 4,000 (B), the number of integers is going to be 1,000,000 since we have seen in the previous experiment that

with 1,000,000 of integers the execution time is good for a big size of number; and the number of streams to merge in one pass (d) is 1,000.

The next table shows the execution times of the experiment depending on the memory space. We can observe in the plot that if we increase the memory space, we are going to reduce the execution time. However, with a memory space of 1,000 integers the execution time gets close to a horizontal asymptote, so in this case, increasing the memory space is not worth it.

Memory space (integers)	10	50	100	500	1,000	10,000	100,000
External Merge sort time (sec)	7.882	2.253	1.632	0.949	0.757	0.584	0.515

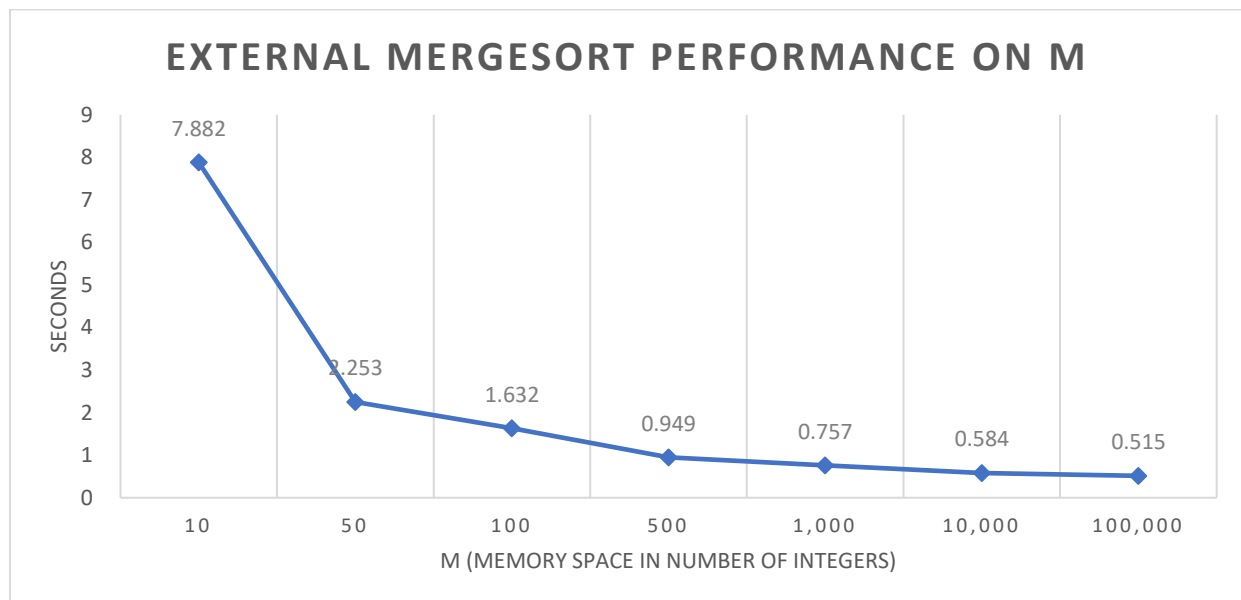


Figure 12: Effect of parameter M on performance of external mergesort algorithm

It is easy to see that the more memory we have, the better performance we get. This aligns to the expected behavior, because as M increases, the number of streams to create and merge also decreases, and both the in-memory sorting and the multiway merging are performed on bigger batches.

3.2.3 d – varying on number of streams sorted in one pass

The third experiment is focused on d, the number of streams to merge in one pass. The buffer size is 4,000 (B), the number of integers is going to be 1,000,000 and the memory space in number of integers (M) that we have chosen is 1,000 since with more memory space the improvement of the execution time is not significant (as proved on section 3.2.2).

The next table shows the execution times of the experiment depending on the number of streams. We observe that if we increase the number of streams, we are going to reduce the execution time. However, we should focus on determining until which point the incremental of the number of streams is worth it, since we can see that from one point onwards, increasing the number of streams to merge in one pass does not produce any significant performance increase.

Number of Streams	2	4	10	20	50	100	1,000
External Merge sort Time (sec)	1.382	1.118	0.893	0.871	0.763	0.761	0.686

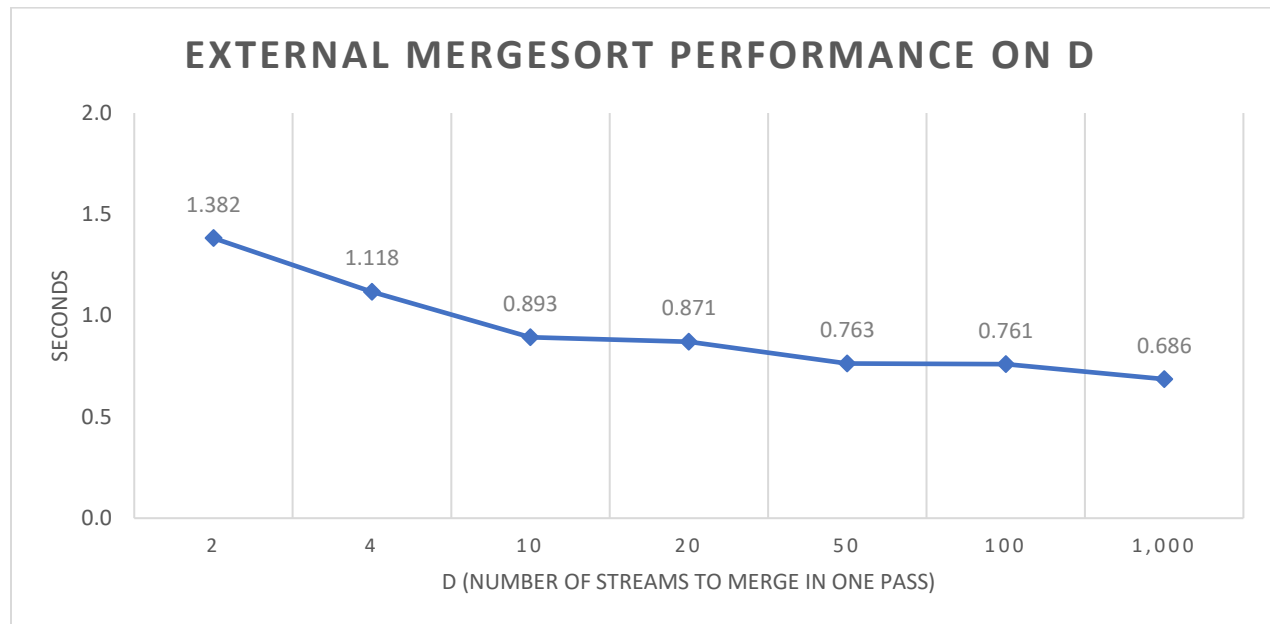


Figure 13: Effect of parameter d on performance of external mergesort

The more streams we can merge at a time, the better performance we get. This is exactly what we expected, and we can directly say that the optimal value for this parameter is $\min(M, \lceil N/M \rceil)$. When $d = \lceil N/M \rceil$, the merging of the sorted streams can happen in only one pass.

3.3 COMPARISON WITH INTERNAL SORTING

In this chapter we make a comparison of in-memory sorting and our implementation of external mergesort. As no IO operations with secondary memory are needed, rather than loading all numbers in memory and writing them back into disk ($2 * N$), we expect internal memory sorting to be much faster than its external memory counterpart. As the following table shows, this is also what happens in practice. Although, as the input grows, the difference becomes less and less evident as the cpu utilization of the external mergesort goes up.

N	10,000	100,000	1,000,000	10,000,000
External Mergesort time (sec)	0.098	0.218	0.678	5.787
Internal sort time (sec)	0.008	0.047	0.348	4.332

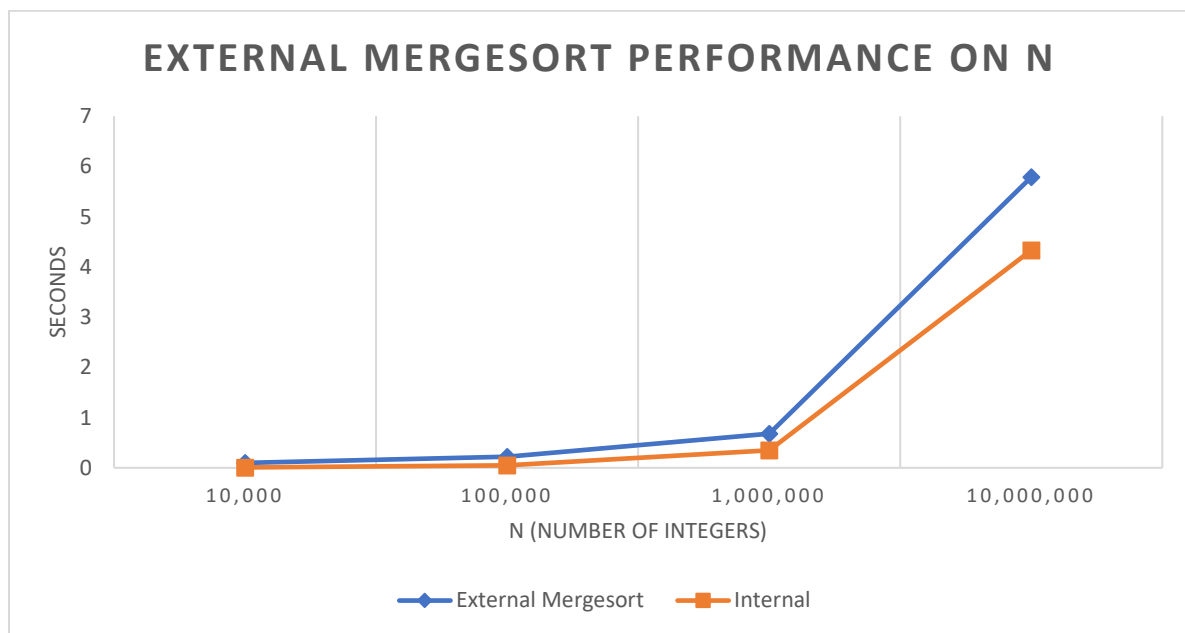


Figure 14 Comparing External mergesort with internal sort timings

3.4 DISCUSSION OF EXPECTED BEHAVIOR VS EXPERIMENTAL OBSERVATIONS

In these experiments, the experimental observations come in par with the expected behavior. To sum up our findings:

- As the number of integers to sort (N) grows, the algorithm will take more time.
- As the memory (M) grows, the algorithm will take less time until $N = M$, at which point we can do internal sorting.
- As the streams to merge at once (d) grows, the algorithm will perform better until $d = \lceil N/M \rceil$, at which point the merging can happen in only one pass.
- As expected, internal sorting outperforms external mergesort, however the difference was not as big as expected, especially for the 10 million input, because of external mergesort's cpu utilization.

4 CONCLUSION

In the first part of the project we have worked with various ways of reading and writing to secondary memory. We have seen how simple operations like this can dramatically affect the performance of external algorithms. Buffering is one idea, that increase IO performance and although we did not manage to benefit from varying buffer sizes bigger than the default, we understand that if IO is a critical part of an application, it is worth investigating. Then, we also learned about memory mapping, which gives an overhead time for small files, it is very performant for bigger files and especially in our application that has great spatiotemporal locality, meaning that we read and write integers that are close to each other in secondary memory.

In the second part of the project, we implemented an external mergesort algorithm in Java and tested it with different parameters. We identified good values for these parameters and understood that depending on the input it may be worth to dynamically change them.

In general, this project showed how it is in practice to work with the internals of a database system and how database architectural design decisions can seriously affect the performance of a system.