

# Database Systems Architecture

Project: External merge-sort



***Submitted to:***

Prof. [Stijn Vansummeren](#)

[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
2	Observations on streams .....	4
2.1	Expected Behavior .....	4
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) .....	7
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	Expected Behavior .....	<b>Error! Bookmark not defined.</b>
3.1.1	Algorithm description .....	18
3.1.2	Performance expectations .....	<b>Error! Bookmark not defined.</b>
3.2	Experimental Observations .....	19
3.2.1	N – varying on number of integers .....	19
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	Discussion of expected behavior vs experimental observations .....	23
4	Overall Conclusion .....	23
5	Bibliography .....	<b>Error! Bookmark not defined.</b>
6	Java IO: BufferedInputStream .....	<b>Error! Bookmark not defined.</b>
7	Annex .....	25

# 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 and read/write implementations. 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 4<sup>th</sup> IO implementation as suggested in a [stackoverflow answer](#)<sup>1</sup>. Other than this, some notable usage from the standard library is:

- `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 produces 32-bit unsorted signed integer binary files, taking *N* (number of integers) as a parameter. 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.

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 implementation, we have tested with file which contained 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 ENVIROMENT 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
  - SanDisk SD8TB8U2
  - SSD

---

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

## 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. What is more, 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 Input Stream (named `InStream`) and general 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).

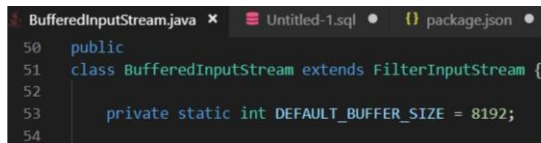
Since there is no buffer introduced, we are expecting slow performance, as we need to access the secondary memory each time we request an integer. In other words, we require as many I/O's as tuples (in our case integers) exist in our input. With high number of I/O's the performance is poor, especially 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` between `DataInputStream` and the `InputStream`, meaning that `BufferedInputStream` wraps `InputStream` and `DataInputStream` wraps 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 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 window. The title bar shows three tabs: 'BufferedInputStream.java', 'Untitled-1.sql', and 'package.json'. The code in the 'BufferedInputStream.java' tab is as follows:

```
50 public
51 class BufferedInputStream extends FilterInputStream {
52
53     private static int DEFAULT_BUFFER_SIZE = 8192;
54 }
```

Figure 1 Default buffer size of Java SDK 1.8

Considering that we need one pass for reading and another one for writing, the cost formula is the following:

$$Cost(Impl2) = k * 2 * roundup(N/B)$$

And in our case, it is going to be:

$$Cost(Impl2) = k * 2 * roundup(N/2048)$$

#### 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 that we cannot avoid. The formula just gives the number of IO requests that our code will execute.

### 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 should expect an improvement of the performance in comparison with the second implementation as we increase the buffer size. However, as later noticed,

the performance does not keep on growing as we increase the buffer size. This happens because, when you increment the buffer size, you are incrementing the size of the array that is going to keep the numbers, but if the cache size of your machine and the block size that your hard disk reads in is less than the size of this array, the increment of the buffer size does not have any improvement in the performance.

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

*Figure 2 Specifying buffer size within BufferedInputStream*

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

$$Cost(Impl2) = k * 2 * roundup(N/B)$$

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.



## 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 to 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 3 below.

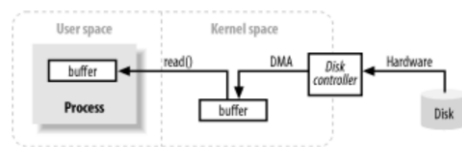


Figure 3 how data moves from disk to process memory<sup>2</sup>

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 are requesting so much memory that exceeds RAM spatial capabilities, as it does not have the required available space. Then the memory manager takes some data that is in RAM (that you do not currently need) and copies it to the swap space on the hard disk (to be fetched later upon request). This process 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. maps the file to process' address space, so the process can address the file directly and no copies are required.

<sup>2</sup> <https://howtodoinjava.com/java/io/how-java-io-works-internally-at-lower-level/>

Memory paging explanation...

Resources used to study upon memory mapping technology:

[https://en.wikipedia.org/wiki/Memory-mapped\\_file](https://en.wikipedia.org/wiki/Memory-mapped_file)

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

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

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

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

#### 2.1.4.2 Implementation details

For Memory Mapped read implementation we created class `InputStream4`, extending `InStream` as usual. As mentioned for this implementation we used 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;
4.     fc = new RandomAccessFile(path, "rw").getChannel();
5.     fileSize = fc.size();
6.     mem = fc.map(FileChannel.MapMode.READ_ONLY, byteSkip, min(fileSize -
7.     byteSkip, bsize));
8.     memPos = byteSkip;
9.     runningPos = 0;
10. }
```

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;
9.     memPos += 4;
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.

On close...

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

Commented [SP1]: Giannis help pls :P

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 spatialtemporal locality when referencing values, meaning that we read/write integers that are close to each other in memory.

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

## 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 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 2048 integers (8192 bytes) and third and fourth implementations is of 4000 Integers, around 16000 Bytes.

The next table shows the time of the first experiment. We can see that the worst implementation is the first, only with 100.000 integers, it spends 23 seconds. The second and third implementation start to spend more time that the fourth one with 100.000 items, but the difference is considerably more visible with 1.000.000 where the fourth one is less than a second ant the others are around 6 seconds.

Both buffered implementations exhibit the same behavior in this test, probably because the buffer of the 3<sup>rd</sup> implementation is just about twice the default size of the 2<sup>nd</sup> implementation. We will examine in more detail how these implementations differ in the next experiment that we try different buffer size values.

Implementation\ number of N	1.000	10.000	100.000	1.000.000	10.000.000
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

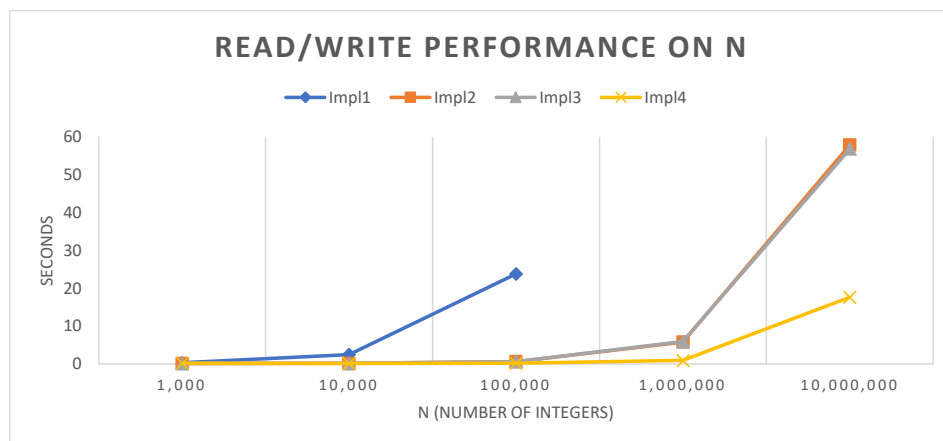


Figure 4: IO performance of the different implementation with a variety of input sizes (N)

### 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 is 10.000.000. We have analyzed the third and the fourth implementation, but we have the result of the last experiment of the second implementation with 10.000.000 of integers in order to verify that the difference of the buffer size in the previous experiment is insignificant.

In the previous experiment we analyzed the behavior of each implementation with different number of integers and we could see that the fourth implementation is better with a big number of integers than the other. Therefore, we know in this experiment that the fourth implementation is going to get better results, since we have chosen a big number of integers (10.000.000), so here we are interested in the curve of both implementations.

We can see that the buffer size is important for the time, since if compare the second and the third implementation, they have the same time (more or less) with the same buffer size (more or less), but if we decrease the buffer size of the third implementation, this is going to spend significantly more time. However, if we increase the buffer size is going to be an insignificant improvement, so there is a point where despite you increase more the buffer you are not going to get a better result. We do not know why exactly this happen, maybe this is because java buffer stream does not allow a buffer size bigger than 8192 bytes (we do not know exactly the limit size); there is a point where the time related with the buffer size is insignificant (you do not have so much I/O's) or something related with the cache size.

Implementation \ Buffer size	40	400	4.000	40.000	400.000	4.000.000
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

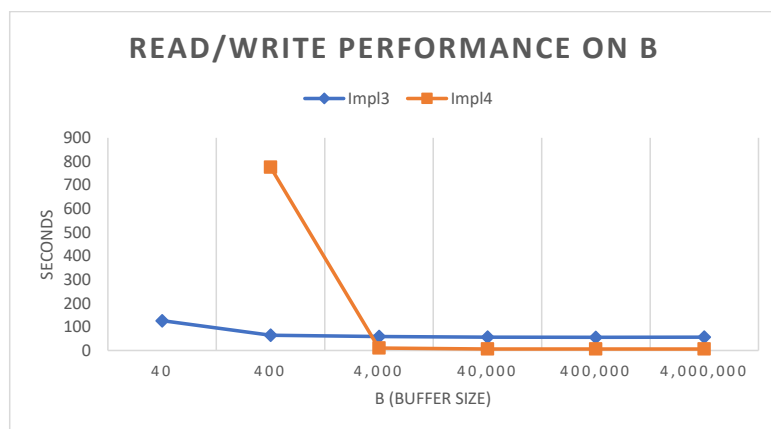


Figure 5: IO performance with varying buffersize

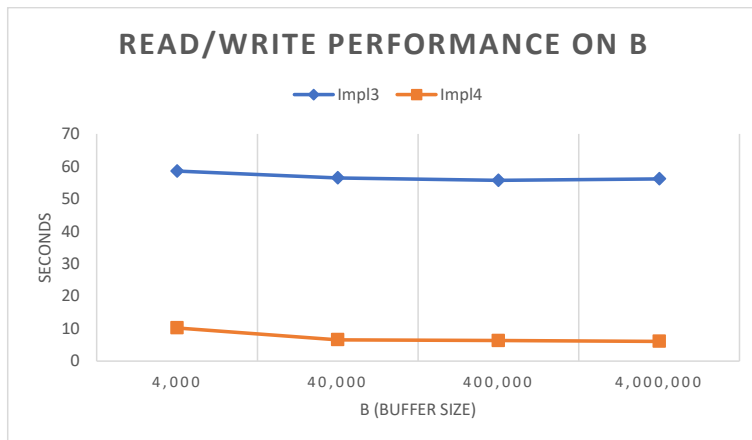


Figure 6: IO performance with varying buffersize (zoomed in after saturation)

### 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 bad, is very slow and its time grows quickly. However, we do not have apparently differences between the second and third implementation, and we can see that the fourth one is the best.

Moreover, there is another interesting thing. With the second and third implementation, the curve starts to grow a fast when the number of streams is bigger than 8 and with the fourth implementation the point is when the number of streams is bigger than 16.

Implementatio n\ streams	2	4	8	16	30
Impl1	1,576	3,084	6,127	12,934	23,619
Impl2	0,097	0,135	0,207	0,360	0,663
Impl3	0,096	0,135	0,204	0,352	0,660
Impl4	0,088	0,080	0,101	0,111	0,160

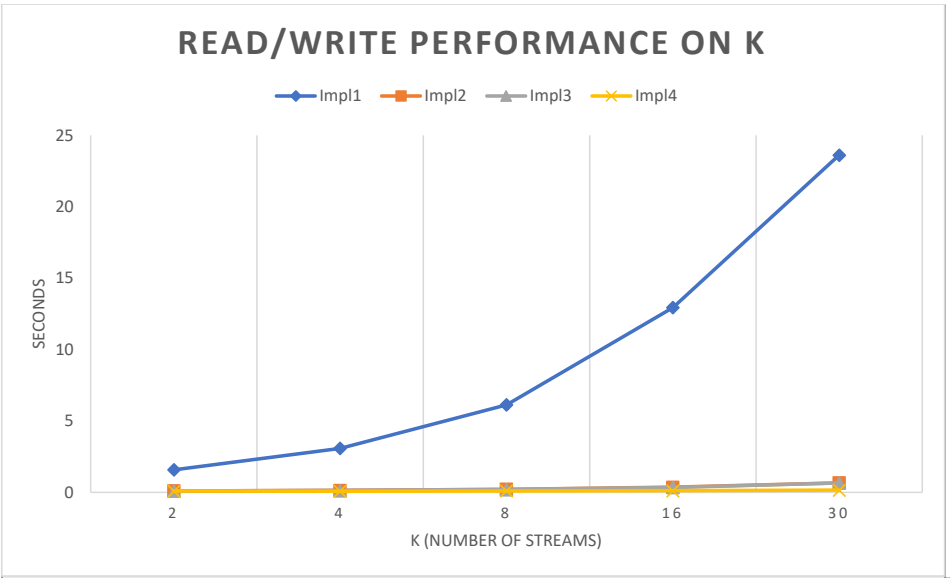


Figure 7: Read/Write performance with varying the number of streams (all implementations except for 1st one)

Commented [PI2]: Different plots for 1 and 2,3,4

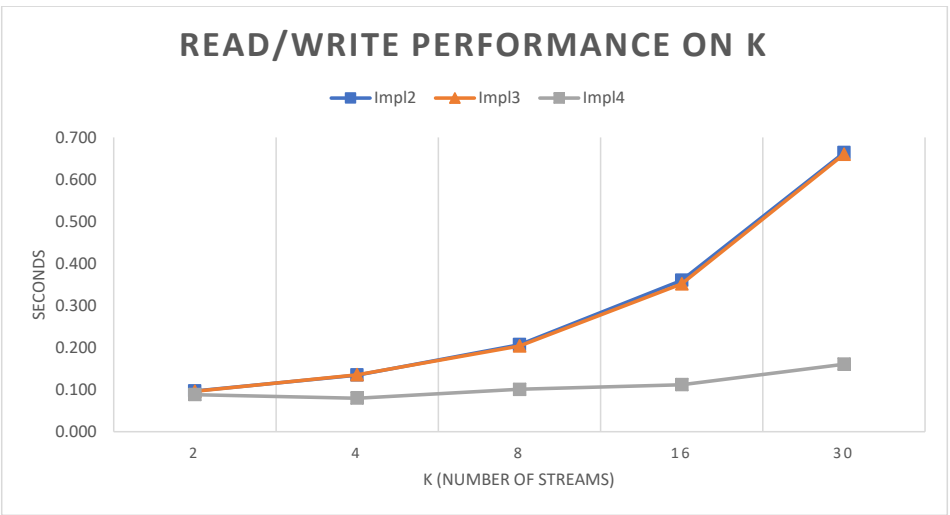


Figure 8: Read/Write performance with varying the number of streams (all implementations except for 1st one)

Commented [PI3R2]:

Commented [SP4R2]: It's right below already 😊



## 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.

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.

As a result, 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 give an overview of the algorithm, we analyze the expected IO cost and give our expectations of 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.

$N=10$  integers,  $M=5$  streams,  $d=2$  streams to merge



Figure 9 Example of algorithm execution

Firstly, the splits the input ( $N$ ) into  $\text{roundup}(N/M)$  segments, each of them having at most  $M$  integers and thus completely fit into memory. Then, it loads those segments one-by-one (stream-by-stream) into the main memory and sorts them there before writing them back to disk in a sorted manner. Up until now we have constructed  $d$  sorted segments, all stored in disk.

Now we can fetch as many streams as fit in memory (maximum being  $M$  streams, as each stream requires one spot) and perform a synchronous merging using the multi-way mergesort.

The algorithm accepts  $d$  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.

The algorithm starts by getting 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 streams points into a text file under `final_output` folder.

### 3.1.2 Cost calculation

*Explain what kind of performance behavior you expect (before running any experiment) from the implementation. Hereto, define a cost function that estimates the total number of I/O's that need to be done, in function of  $N$ ,  $d$ , and  $M$ .*

For the calculation of the cost of the external mergesort, we will ignore any speedup we can get from buffering. We have  $N/M$  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 internal 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.

## 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, 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 of 4000 Integers ( $B$ ), around 16000 Bytes, the memory is going to be of 1000 integers ( $M$ ) and the number of streams to merge ( $d$ ) is 1000.

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

N	10000	100000	1000000	10000000	100000000
External Merge sort Time	0,098	0,218	0,678	5,787	53,186

Below we showcase the results of our experiments into plots.

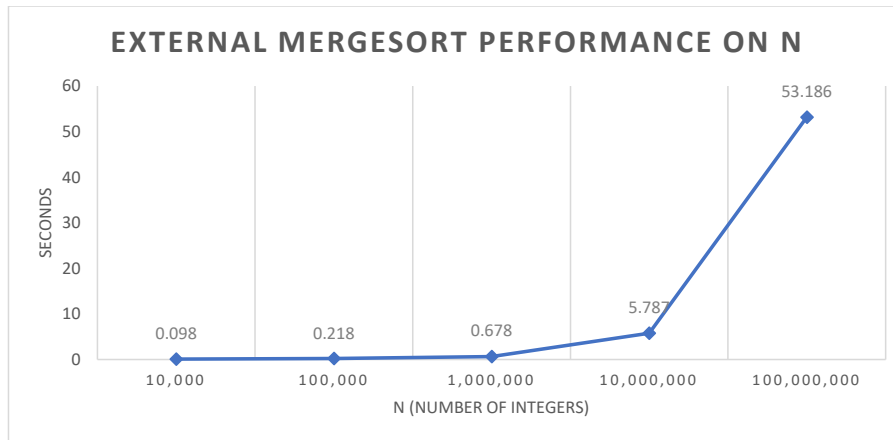


Figure 10: Effect of number of integers on performance of external mergesort algorithm

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.

### 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 of 4000 Integers (B), around 16000 Bytes, 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 is 1000.

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	10	50	100	500	1000	10000	100000
External Merge sort time	7,882	2,253	1,632	0,949	0,757	0,584	0,515

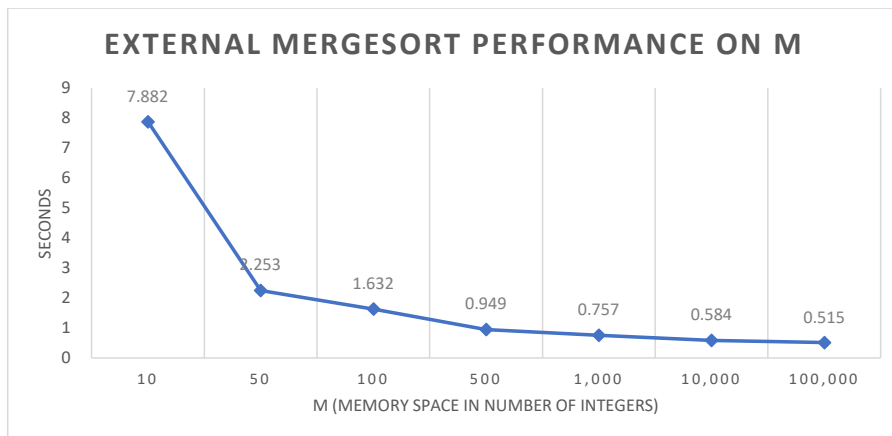


Figure 11: Effect of parameter M on performance of external mergesort algorithm

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 of 4000 Integers (B), around 16000 Bytes, the number of integers is going to be 1.000.000 since we have seen in the first experiment that with 1.000.000 of integers the execution time is good for a big size of number; and the memory space in number of integers, that we have chosen 1000 since with more memory space the improvement of the execution time is not worth it.

The next table shows the execution times of the experiment depending on the number of streams. We can observe that if we increase the number of streams, we are going to reduce the execution time. However, we should analyze until which point the increment of the number of streams is worth it, since we can see that with 100 of streams the execution time is 0,761 and if we increase the stream to 1000 (multiplying by 10) the execution time is 0,686, the improvement is just of 0.08 seconds.

Number of Streams	2	4	10	20	50	100	1000
External Merge sort Time	1,382	1,118	0,893	0,871	0,763	0,761	0,686

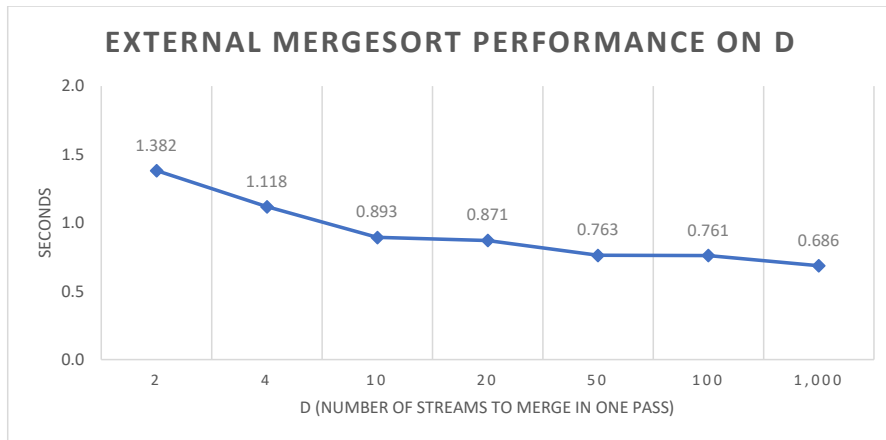


Figure 12: Effect of parameter  $d$  on performance of external mergesort

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 best value for this parameter is  $\min(M, N/M)$ . When  $d = N/M$ , the merging of the streams can happen in only one pass.

### 3.3 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  $d = N/M$ , at which point the merging can happen in only one pass.

## 4 OVERALL CONCLUSION

---

*Summarize what you have learned in this project.*

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 seconds part of the project, we implemented an external mergesort algorithm and tested it with different parameters. We identified good values of 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.

## 5 BIBLIOGRAPHY

---

- Jakob Jenkov, "Optimal Buffer size for an input Stream" . JAVA IO: BuferedInputStream.  
[accessed 02/01/1996]. <http://tutorials.jenkov.com/java-io/bufferedinputstream.html#optimal-buffer-size>



## 6 ANNEX

	<b>N</b>	<b>B=16384</b>		<b>k=30</b>		
	1000	10000	100000	1000000	10000000	
Impl1	0.323	2.441	23.767			
Impl2	0.103	0.149	0.641	5.789	57.829	
Impl3	0.128	0.153	0.648	5.901	56.784	
Impl4	0.163	0.125	0.180	0.915	17.622	

	<b>B</b>	<b>N=10m</b>		<b>k=30</b>			
	40	400	4000	40000	400000	4000000	40000000
Impl3	126.252	64.738	58.557	56.453	55.698	56.152	
Impl4		775.960	10.208	6.571	6.313	6.070	6.830565

	<b>k</b>	<b>B=16384</b>		<b>N=100000</b>		
	2	4	8	16	30	
Impl1	1.576	3.084	6.127	12.934	23.619	
Impl2	0.097	0.135	0.207	0.360	0.663	
Impl3	0.096	0.135	0.204	0.352	0.660	
Impl4	0.088	0.080	0.101	0.111	0.160	

---

	<b>N</b>	<b>B=4000</b>		<b>M=1000</b>	<b>d=1000</b>			
	10000	100000	1000000	10000000	100000000			
External Mergesort	0.098	0.218	0.678	5.787	53.186			

	<b>M</b>	<b>B=4000</b>		<b>N=1000000</b>	<b>d=100</b>			
	10	50	100	500	1000	10000	100000	
External Mergesort	7.882	2.253	1.632	0.949	0.757	0.584	0.515	

	<b>d</b>	<b>B=4000</b>		<b>N=1000000</b>	<b>M=1000</b>			
	2	4	10	20	50	100	1000	
External Mergesort	1.382	1.118	0.893	0.871	0.763	0.761	0.686	