Algorithms in Secondary Memory

External-memory merge-sort

Team Members:

Ankush Sharma

René Gómez

INFO-H-417: Database Systems Architecture

Supervised by: Stijn Vansummeren

Table of content

1. Introduction and Environment

The objective of this project is to perform an external multi way merge sort on files of 32 bit integers and examine its performance under different parameters. The experiment will be divided into two parts.

In the first part, we experiment with four different mechanisms to read and write data on secondary memory. They are :

1. Read/Write one integer at a time from/to disk
2. Equip our Input and output streams with the default buffer size provided by the programming language of our choice, which is Java.
3. Equip our Input and output streams with a buffer size of our own choosing
4. Use memory mapped files to Read/Write data from/to disk

In the second part, we perform a d-way merge-sort algorithm. We will run multiple benchmarks on our stream implementations and the d-way merge-sort algorithm using different parameters to see if our expectations match with the actual results. To read and write data from/to disk, the second part will use the most performant streams as evaluated by the benchmark of the first implementation.

The environment used for this project is :

- Java 8 and Gradle build tool
- We use the slf4j and logback as logging libraries to debug our program. These libraries are declared in build.gradle
- We wrote our own benchmark in Java instead of using a benchmarking library. The code for benchmark can be found in the benchmark folder.
- For reading we randomly generated the files of different size using the output streams to write the integers directly in bytes. We selected the N parameter to generate 1MB, 4MB, 32MB and 128MB files. They were enough to evaluate the performance of the algorithms implemented. The k-streams were always performed on k distinct files.
- There are a lot of factors that are going to affect the tests. It depends on how the operative system is storing the file, the disk's block size, how fragmented the disk is, the buffers in the kernel space and some other factors. We ran the test with the same parameters 3 different times and took as final measure the average of such times.
- As a supporting library for benchmarking, we make use for Apache commons io help us with basic operations like copying files.

The specification of the machine for benchmark is as follows :
Model : MacBook Pro
Processor : 2.2 GHz Intel Core i7
Memory : 16 GB 1600 MHz DDR3
Secondary storage : 256 GB SSD (Block Size = 4096 Bytes)

2. Observations on streams

This chapter focuses on the on our expectations with the different stream implementations and their experimental observations.

For the first part of the experiment we created different classes for input and output streams. In order to keep consistency, the input and output streams implement two different interfaces with the operations ***open***, ***read_next*** and ***end_of_stream*** for the input streams and ***create***, ***write*** and ***close*** for the output streams. The instructions provided in the project were followed to create each type of stream. The implementation is described in the next sections.

To summarize, the streams manage all the operations to read and write to disk, like opening files in the right mode, reading or writing integers, and then closing.

2.1. Expected behavior

2.1.1. Basic streams - StreamA

Implementation

First we created the basic streams without any buffer and named them ReadStream/WriteStream since they mimic the read/write systems calls. In the result analysis, we will refer this type of streams as *streamA*.

Expected behavior

This streams do not have buffers so we don't expect them to have a good performance. This can be derived from the fact that to read N integers it would be necessary to make N read/write system calls producing a high level of I/Os.

$$Cost = N$$

Similarly, since it is required to do experiments with ***k*** streams on ***k*** distinct files, we will have this cost multiply by ***k***.

$$Cost = k * N$$

As an example, considered it is required to load 10^6 integers we will have to make *k**10^6 I/Os.

2.1.2. FStreams - StreamB

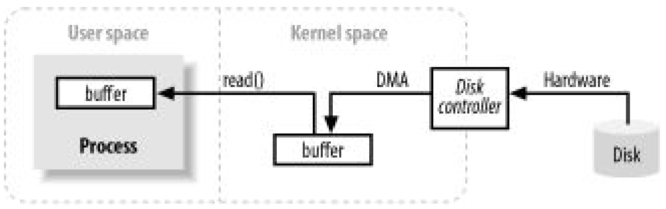
Implementation

These are the streams for mimicking the fread and fwrite functions, consequently we named them FReadStream and FWriteStream. In Java, the these classes can be implemented using BufferedInputStream/BufferedOutputStream. In this experiment, these classes are called FReadStream/FWriteStream. In the results analisis we will refer this type of streams as streamB.

The mechanism to fill a buffer is as follows :

- The process makes a read() call
- The kernel issues a command to the Disk Controller to fetch the data. Using Direct Memory Access, the controller writes the data directly into a temporary buffer in the kernel space

- The kernel copies the data from its temporary buffer and populates the buffer as requested by the process



Source

In conclusion, data is being copied twice, once from the disk controller to kernel space, and then from kernel space to the process buffer.

Expected behavior

BufferedInputStream comes with a default buffer size of 8192 bytes, or 2048 integers. Since we can now read/write more data from/to disk, we expect this implementation to be better than FRead/FWrite. In the next example we estimate the IO cost of loading integers in I/Os

$$Cost = k * \frac{N}{BufferSize} = k * \frac{10^6}{2048} \cong 489 * k$$

2.1.3. Buffered streams - StreamC

Implementation

This implementation is essentially the same as FStreams, the only difference being that now we can equip our streams with a buffer size of our choice. In this experiment, these classes are called BufferedReadStream/BufferedWriteStream. In the results analisis we will refer this type of streams as streamC.

Expected behavior

For a buffer size of 8192 bytes, we expect our performance *identical* to FStreams implementation. However, now that we can set the value for the buffer we could test prefetching data, which consists on loading data that is going to be needed in next steps. Since we know we need to access the file in sequence we speculate we could increase the performance (if by chance it happens to be stored in contiguous blocks) by loading integers, for instance, two times the buffer size. This may allow us to increase the performance as described in Chapter 13 of TCB (Garcia-Molina, Ullman, and Widom 2013) but we don't know for how much or when it could start to be useless.

Here we estimate the cost of loading 10^6 integers with a buffer size of 16384 bytes (4096 Integers).

$$Cost = \frac{N}{BufferSize} * k = \frac{10^6}{4096} * k \cong 245 * k$$

2.1.4. Memory mapping - StreamD

Implementation

Finally, we implemented the memory mapping and called them MemMapReadStream and MemoryMappedWriteStream. In the results analysis we will refer this type of streams as streamD.

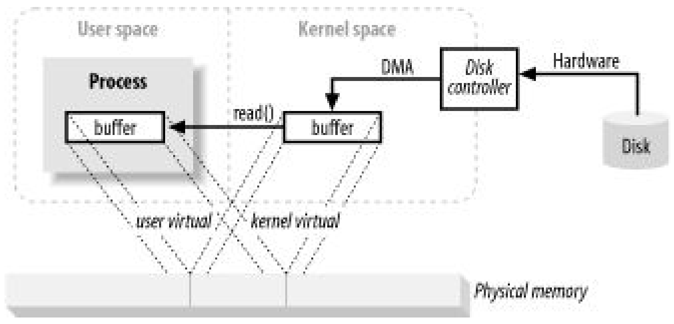
We briefly explain how Memory mapping works, but we need to first understand how Virtual Memory works, which we do in the following section.

Virtual Memory

A process only cares about having access to data, and in practice this access should be as fast as possible. Theoretically, we can scale up and keep on adding more RAM. But RAM is expensive, and very limited.

Virtual Memory is a technique used by the Operative System to give the illusion of having more "memory" available to processes than is actually available. It does so by mapping the *actual physical addresses* to *virtual address (“Wikipedia” n.d.)*. The processes themselves are not aware if the data they want access to resides in RAM or secondary storage. The operating system divides the virtual address space into chunks called Pages, which are usually 4 KB in size.

Memory Mapped Files



Source

A memory mapped file is a portion of virtual memory that is mapped directly to a portion of a file in the file system.

There are advantages and disadvantages to using memory-mapped I/O. The main advantages include (SQLite Documentation n.d.):

1. I/O intensive operations can be faster since content does need to be copied between kernel space and user space
2. The application may need less RAM since it shares pages with the operating-system page cache and does not always need its own copy of working pages.

There are, however, some disadvantages to using memory mapping:

1. The program implementing memory mapping has no control over the memory mapped portion of the file is kept in memory and for how long. The mechanism of handling Memory mapping is handled entirely by the kernel and the hardware, thus making it opaque to the process itself. The pages that are not being accessed often by the program will be swapped to disk
2. The operating system must have a unified buffer cache in order for the memory-mapped I/O extension to work correctly

Implementation

For memory mapping (streamsD) the buffer size is the size of the portion of the file we map in memory, starting at the beginning of the file and moving for the next section when we exhausted the first one and so on.

Expected behavior
In the formula we are considering a map with the same size of the buffered streams. By research we know this still is going to be better due to the mechanism of accessing the data in the kernel space, but here we don't know for how much.

$$Cost = \frac{N}{MapSize} * k = \frac{10^6}{4096} \cong 245$$

Expected performance
In this project we expect to have the best performance with these streams, mainly with *big* Ns.
Since the buffer saves the operating system from the hassle of copying the buffers twice, we expect a better performance as compared with other stream implementations.

2.2. Experimental Observations

For the purpose of reading the following plots/graphs please consider we named the streams with a r/w when presenting reading/writing times and letters for the different types. rStreamA is therefore time for reading using streams with no buffer, rStreamB for reading streams with fixed buffer size, and so on. Similarly, wStreamA is time for writing using streams with no buffer and so on.

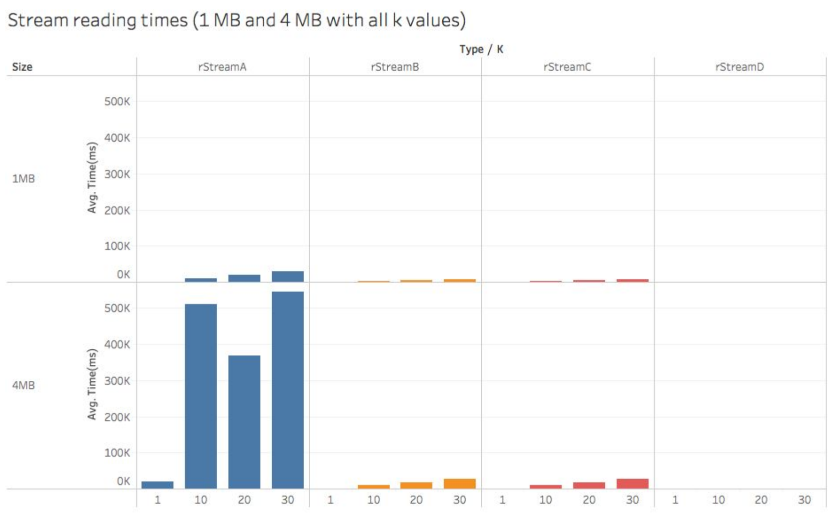
The reading and writing times are presented separately. In this experiment we used different values of k (1, 10, 20 and 30) and for each value of N we present the different values of K, since they together represent the change of the size of data to be processed.

2.2.1. Small size of N.

We started with a experiment with small N (1MB and 4 MB) to discard the slower streams. For 1 MB and 4MB we tested all the streams for each value of k (1, 10, 20 and 30) and fixed the buffer size in 8192 bytes.

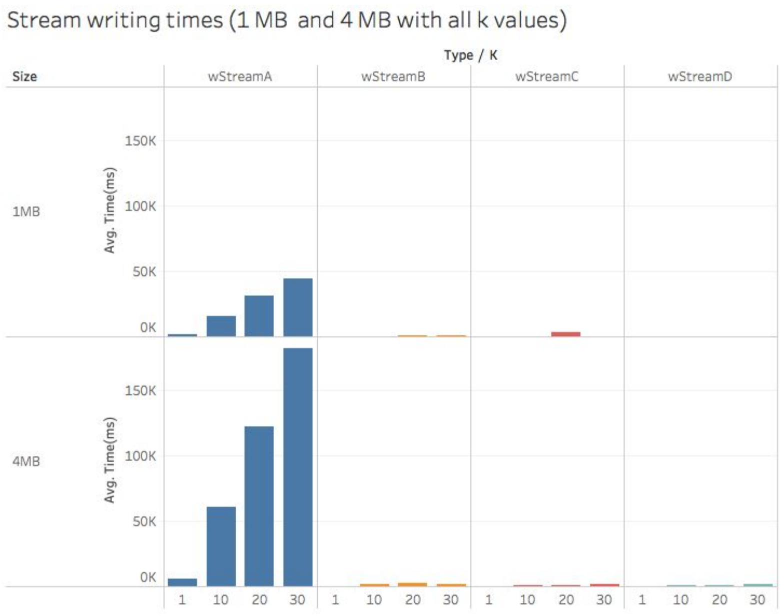
Notice that with this small change of N the files produced a big increase in the time for the first type of streams. However, these results are consistent with what was expected.

The reading times are presented in the following figure. In the x-axis for each type of stream we present the times for the different values of k (1, 10, 20 and 30). In the y-axis we present the measurements for each size of N. This allows to easily compare the results with different parameters, moving horizontally or vertically. The color of the bars helps to identify the type of streams.



StreamsA do have also the worst performance when writing. From here, we decided to discard the first type of streams.

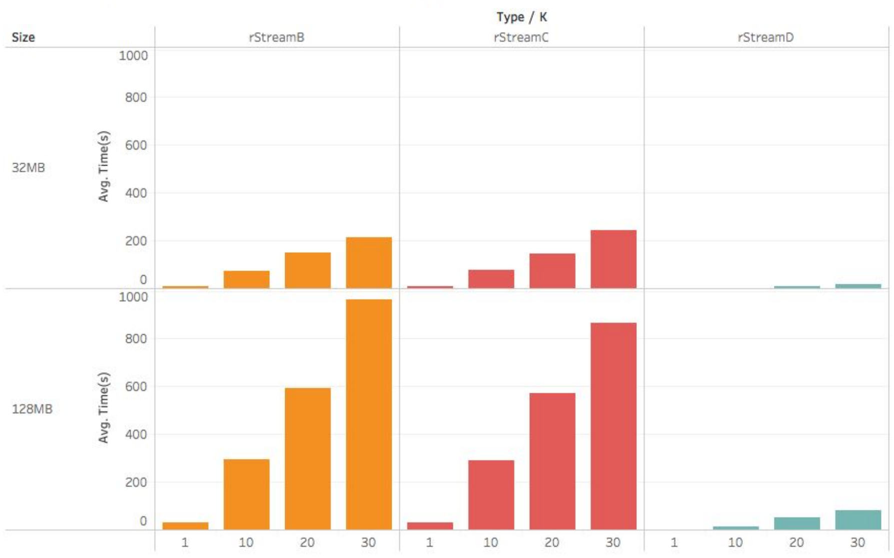
The writing times are presented in the following figure:



2.2.2. Increasing N

We decided to increase the size of N to 32MB and 128MB. We are now going to analyze the times for the first type of streams (streamA). The results are presented in the next figure:

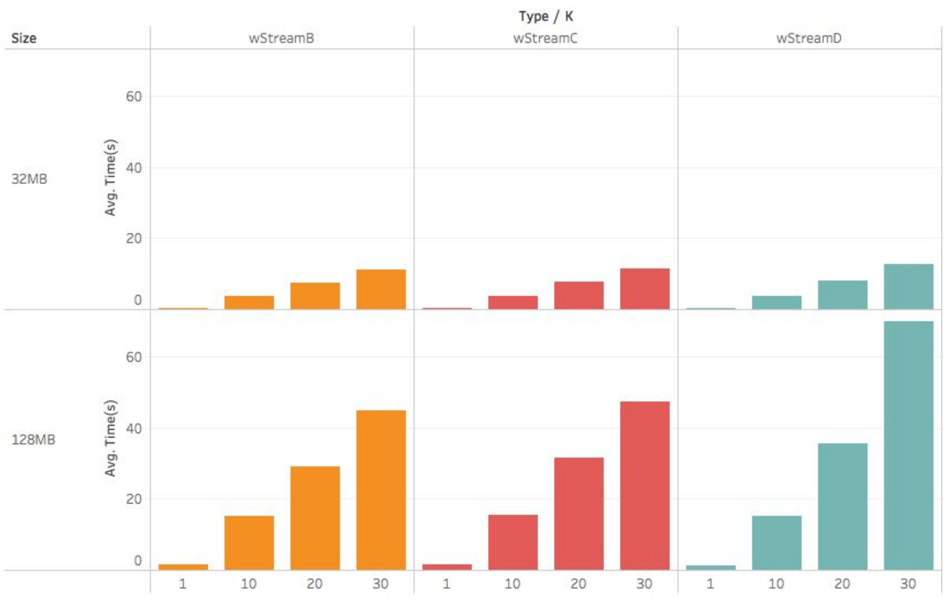
Stream reading times (32 MB and 128 MB with all k values)



Here we can see that the streams B and C perform similarly with the size of the buffer fixed in 8192 bytes. In this experiment you can hardly see the times for the StreamsD, those are the streams using memory mapping. In the reading experiments the StreamsD are faster than the conventional buffered streams.

However, when writing time is considered the streams perform in a different way. The results for writing are presented in the next figure:

Stream writing times (32 MB and 128 MB with all k values)



The buffered streams perform better for writing. We have fixed the value of the buffer in 8192 bytes in the figure to consider the streamsB. However, the pattern is similar with different buffer size for streamsC and streamsD. In the next section we analyze what happens with different buffer sizes.

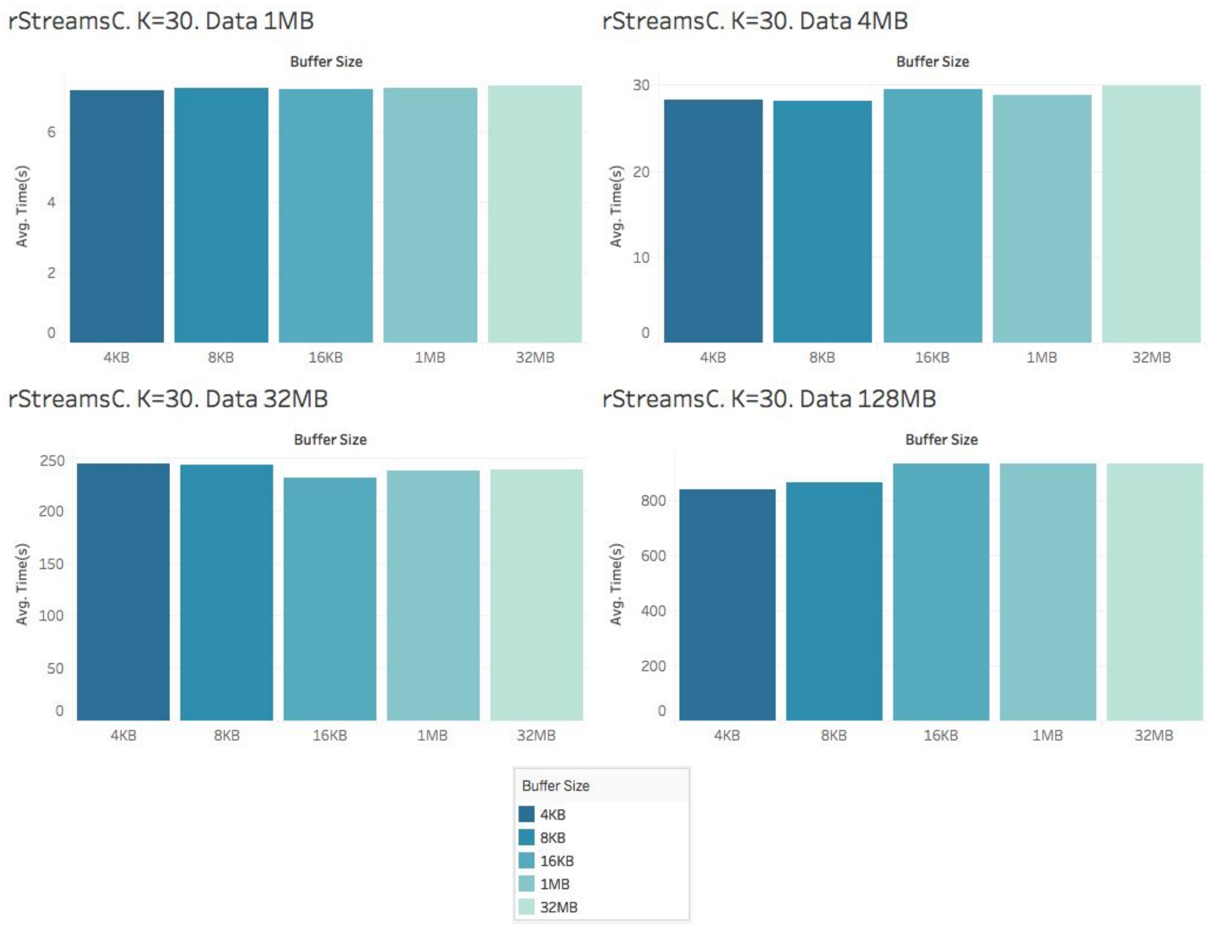
2.2.3. Modifying the buffer size

As mentioned before, streamsB and streamsC have similar performance with a buffer of 8192 Bytes. Nevertheless, we wanted to see what happens if we increased the value of the buffer for streamsC and streamsD to test buffered streams vs memory mapped streams.

For this experiment we tested all values of N and k, but in the following figure we present the results of k=30 to facilitate the reading of the results, the other values of k behave simirately. We used the buffer size of 4KB (to match the block size of the operative system and see if it has any influence), 8KB, 16KB, 1MB and 32MB.

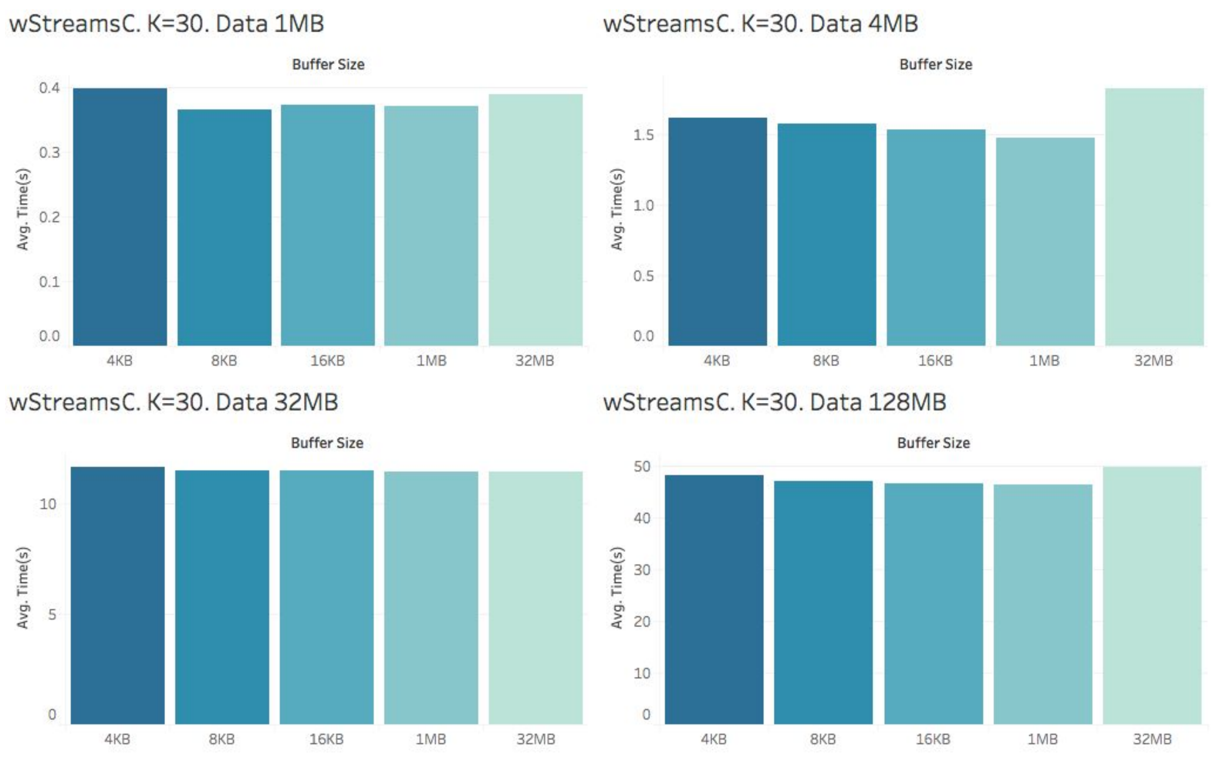
Buffered Streams (StreamsC)

The reading times with different buffer size are presented in the following image:



Reading. Different buffer size for streamsC. Fixed k=30.

It is evident that for all sizes of N the buffer size has roughly the same performance. When reading and writing there are some variations but they are not big enough to be considered as performance improvement. We thought the buffer size could improve the performance, however the results show the time reach the more consistent performance between 4KB and 8KB (the default value for the Java libraries).
The writing times with different buffer size are presented in the following image:

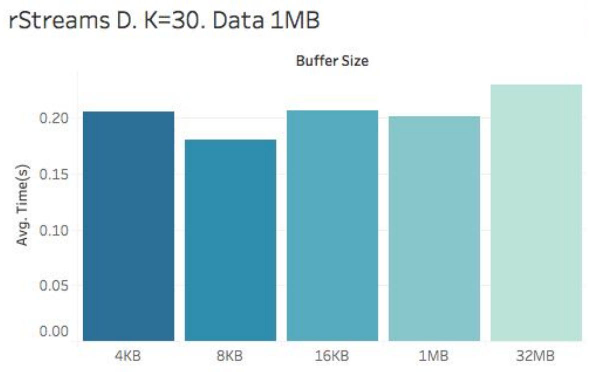


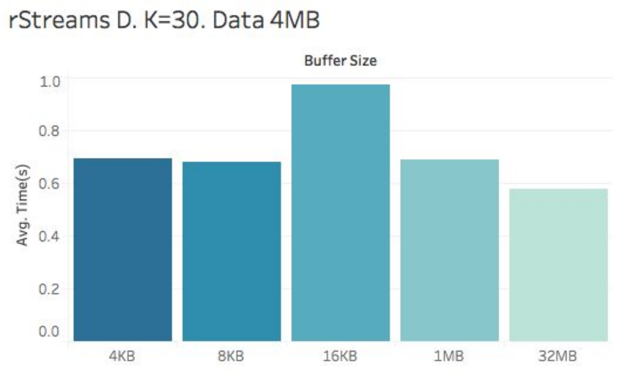
Writing. Different buffer size for streamsC. Fixed k=30.

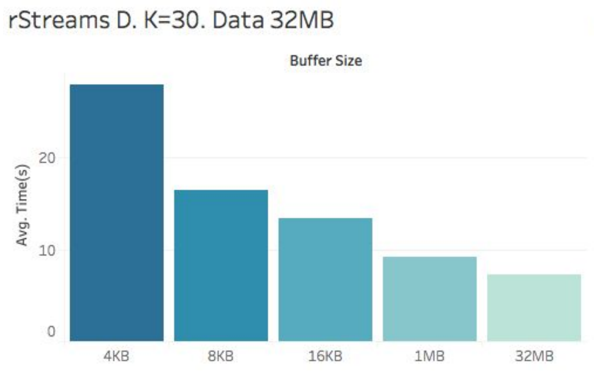
Memory Mapping (StreamsD)

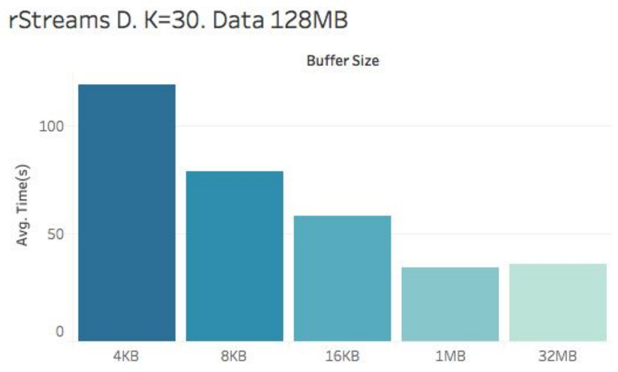
When the memory mapping streams are considered, the results of increasing the buffer size to determine the file region mapped to memory show an improvement in performance when reading. The buffer size for small data has some variations, but you should notice it is in the order of milliseconds, and could be attributed to variations in the entire system. For greater values of N (32MB and 128MB) it's possible to see a reduction in reading time but there is not a big difference between using a map size of 1MB and 32MB.

The results are presented in the next figure:

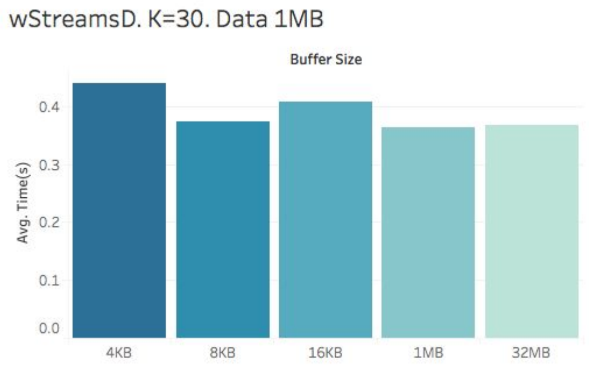


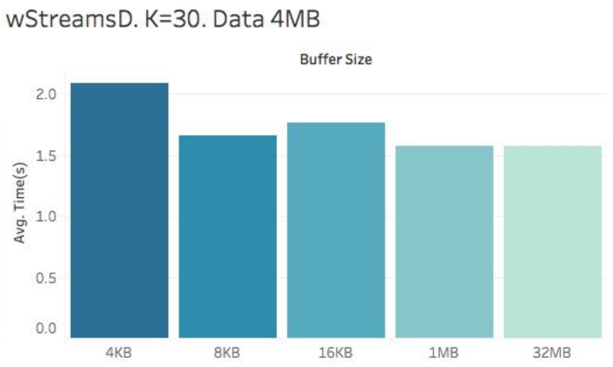


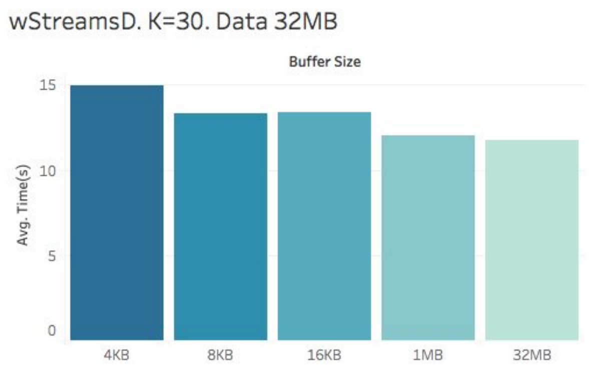


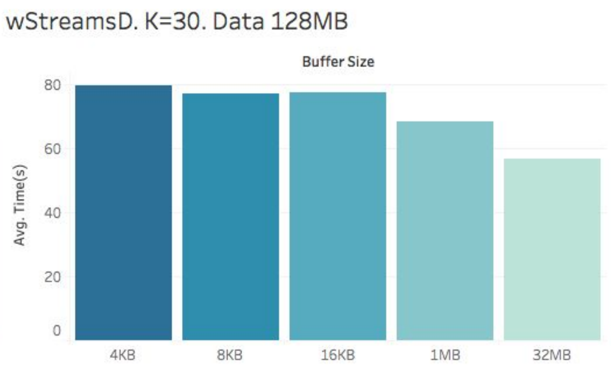


Nevertheless, for writing with streamsD the changes in the results of increasing the buffer size are too small to be considered improvement. Regardless of the buffer size, the streamsC were better than streamsD when writing.









2.3. Expected vs Experimental Observations

For the basic streams the expected behavior was confirmed by the experiments. For the buffered streams we also expected a better performance, but we thought we could obtain a better performance with bigger buffer size. With the results and some research we now understand the mechanism of buffering in the OS and how copying in a big buffer could decrease the performance of an application.

From the buffer experiments, in certain way we confirmed the convenience of the buffer size selected by the default in the Java environment. Here we cite some paragraphs in one of the websites visited to understand this situation (Coffey n.d.):

«As a general rule of thumb, you should always wrap an input stream in a BufferedInputStream except where you know there's other buffering going on. For example, if you are calling the multi-byte reads on a FileInputStream and reading a reasonably large number of bytes (say, a few K) at a time, then adding an extra BufferedInputStream probably won't give you much benefit and could even slow down your I/O slightly due to the extra buffer copying.»

Memory mapping streams

The results of increasing the buffer size to determine the file region mapped to memory in StreamsD show a improvement in performance when reading. According to the research, memory mapping implies that the process no longer has to make read calls directly to the kernel, therefore data does not need to be copied between kernel space and user space. Even though we didn't expect such difference in performance with the rest of the streams when reading.
Besides, we are accessing the entire files in sequence, this means all the mapped regions are read before the next page is required, this ensures the minimum number of *page faults*.

3. Observations on multi-way merge sort

In this chapter we discuss the algorithms we used to perform a d-way multi merge-sort on files of 32 bit integers.

The parameters passed to our algorithm are:

1. N : The size of the file, in number of 32 bit integers.
2. M : The memory available during the first sorting phase(explained below). We assume that M is the memory available for the buffers itself, and not the programming environment as a whole. Also we consider important to keep one of the buffers for the output, since we need to write the sublists.
3. d : The number of streams to merge

We now explain our algorithm briefly step by step:

1. We convert the memory M into buffers multiplying M by 4 bytes.
2. We start reading from the file. Based on our experimental observations while benchmarking stream implementations, we saw that MemMapReadStream with a buffer size of 1MB(1024 * 1024 performed the best. We use MemMapReadStream to read from the file, and allocate it a size of M buffers.
3. We start reading the file, and divide the file into ⌈N/M⌉ streams, each of size M
4. An empty Queue is initialized to store references to sublists that will be created
5. We take the first stream to read the firsts M integers from the specified file, once in memory we perform an in memory sort using merge-sort, which guarantees maximum time complexity of O(nlogn). We make use of Java's inbuilt Collection.sort() method for this. Similarly, we perform in memory merge-sort on every stream
6. The sorted sublists are then sent to the output buffer with buffer size of M. We use the FWriteStream class for this
7. A reference to these sublists are also stored in the queue. For simplicity, in the first pass we name our files in ascending order with which they are created. For instance, the first sublist is called "1", the second sublist is called "2" and so on
8. Now we have ⌈N/M⌉ sublists, and we start to perform our merge
9. For our d-way merge over ⌈N/M⌉ sublists, we make use of PriorityQueue. We starting reading the first element out of d sublists, and using a PriorityQueue we find the minimum integer value. This value is written to the output buffer immediately, and is replaced by the next integer from that sublist
10. For each d-sublist that are merged, they are popped from the queue, and the merged file reference is moved to the end of the queue, to be merged later. For example, let's say that we have queue as [1,2,3,4,5,6,7,8,9,10]. At the beginning the values in this queue are the names of the file, where 1 is the first sublist with name 1, 2 is the second sublist with name 2 and so on. If d = 3, we merge 1, 2 and 3, and create a new file with a random name . The new file goes to the end of the queue. We keep doing this till we are left with only one file, fully sorted and merged.

3.1. Expected behaviour

For the multiway merge sort we use Memory mapping. We followed the instructions to write the program to partition, sort and then merge the files.

Our experiment were conducted on files of sizes 1MB, 4MB, 32MB, and 128MB. These equate to files with 250000, 1000000, 8000000 and 32000000 integers respectively. . We believe that these file sizes were sufficient to benchmark our algorithm. For each of these files sizes, we expect that the bigger the value of **M**, the better our results should be. This is because we would have less number of streams to merge, and thus less Disk I/O's. Therefore, since we believe that **M** is the most crucial parameter, instead of testing for fixed values of **M** for each file of size **N**, we varied the parameter **M** as:

- $\lceil N/1000 \rceil$
- $\lceil N/100 \rceil$
- $\lceil N/10 \rceil$
- $\lceil N/2 \rceil$
- N

Our motivation for taking $M = N$ was to see what would happen if we allocate enough memory to the algorithm to load everything in memory, sort in memory, and then output only 1 sublist. We call this case *inMemory*. We expect this to be the fastest implementation for these files.

Similar to M, we also vary the values of **d**. The minimum value for d is 2 and the maximum value is $\lceil N/M \rceil$. These values vary with value of **M**.

We were also curious to see what would happen if we decide to perform the algorithm on a big file of size 1GB, and the number of streams that we on the sorted sublist in parallel. We share our findings in the "Experimental observations" section.

Cost estimation:
For this we use the cost explanation provided in the class' slides (Vansummeren n.d.) changing the values were is necessary. Notice we have M memory available and we need at least one buffer to write the sublists in files:

In the first pass we read M blocks from the input file and we sort these using the java implementation of merge-sort, once sorted we write the resulting sublist to disk. After the **first** pass we hence have *N/M* sorted sublists of *M* blocks each:

$$\frac{N}{M * d^0}$$

Notice we are in the 1st pass and we introduced the term d^0, we are going to use it to facilitate the calculation of the costs, while keeping consistency with the number of passes. In the second pass we read again ***d*** blocks from these sublists and merge them into larger sorted lists of size *M*d*. Hence we get:

$$\frac{N}{M * d} \text{ sorted sublists, and after the third pass } \frac{N}{M * d^2}$$

We repeat until we obtain a single sorted sublist. Assuming we have h passes overall, the value h satisfies the equation:

$$d^{h-1} = \frac{N}{M}$$

The number of passes is determined by:

$$\left\lceil \log_d \left(\frac{N}{M}\right) + 1 \right\rceil$$

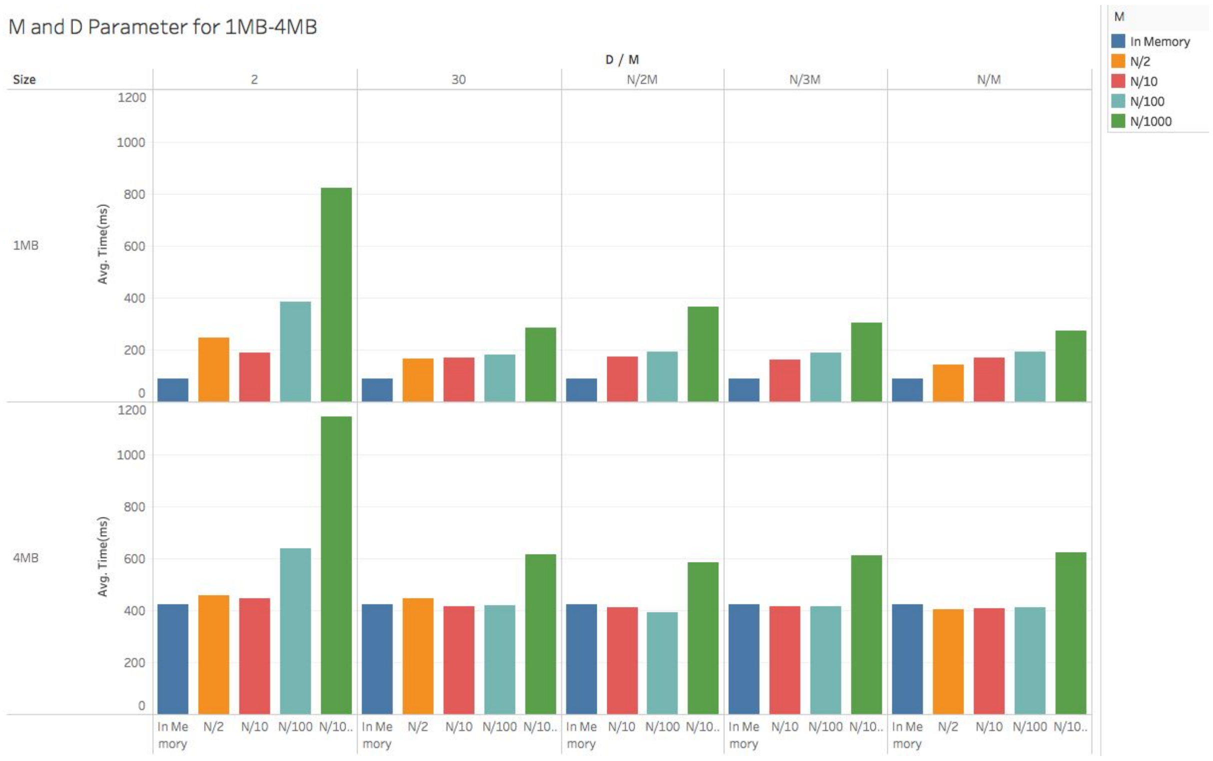
Also, since we are reading and writing in each pass N integers, the total cost is approximately:

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

3.2. Experimental observations

3.2.1. 1MB and 4MB

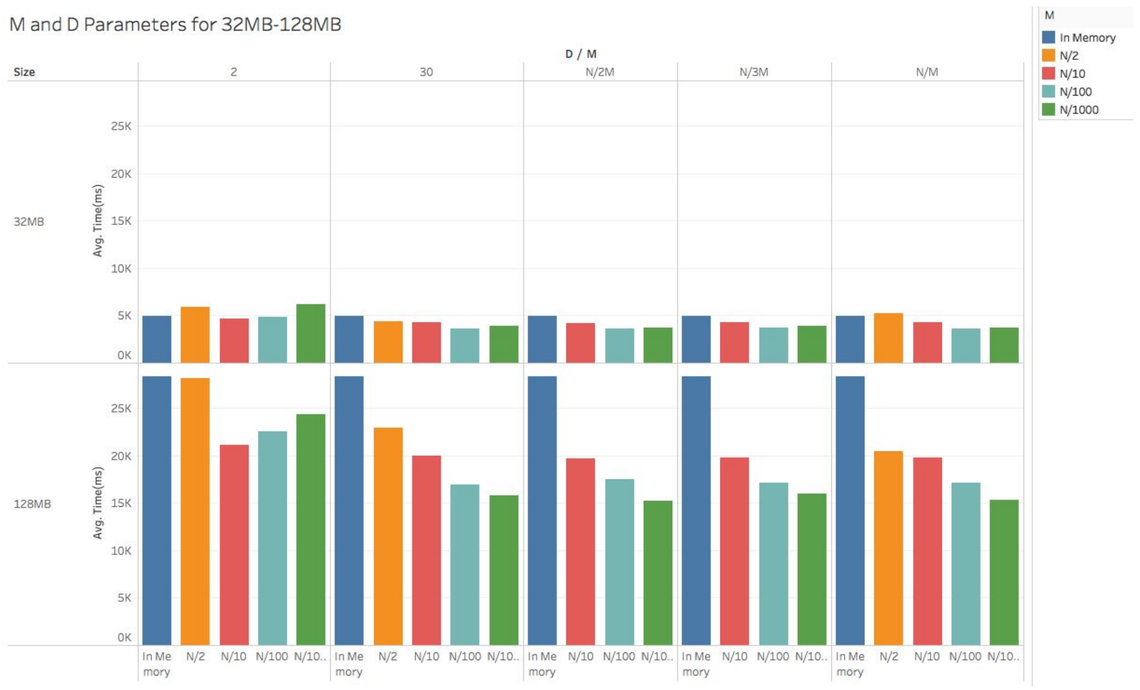
We present the results of different values for parameters M and d for N of 250.000 integers or 1MB and 10^6 integers or 4MB. We included the time of in memory sorting (bars in blue) to facilitate comparison. We used different values of M and d.



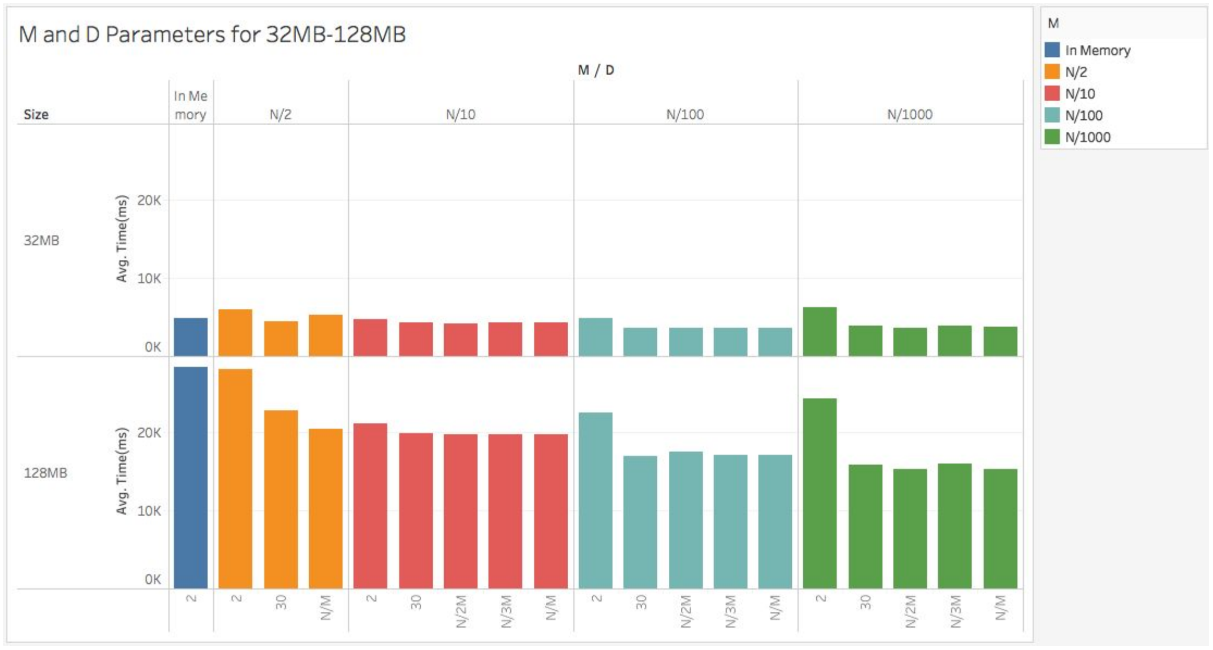
For the smallest file of 1MB, in memory performs the best as expected. For the multiway merge sort the worst performance is with a small M and a small d, this follows the logic we got from the experiments with streams. With small M a lot of sublists are created and with small d the number of required merges increases, even though N is equal, this means more processing time like opening and closing streams, loading small quantities of data to memory and so on. Here we can see that when there are a lot of streams to merge, a good value for d is 30, the one mandatory for the experiment with streams.

3.2.2. 32MB and 128MB

We present the result for bigger Ns (files of 32MB and 128MB). Again, we included the time of in memory sorting (bars in blue) to facilitate comparison.
We could say that for 32MB there is a plateau respect the parameter M. However, the results are different with the file of 128MB. You can see how the performance is better having M as a moderate parameter (green bars).



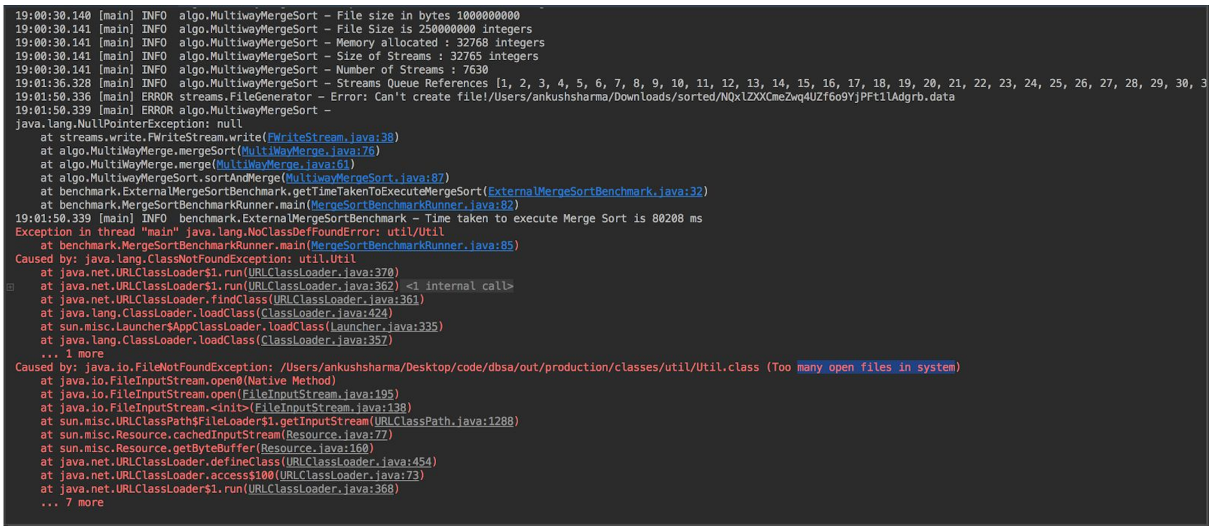
In the next figure we inverted the parameters D and M to M and D analize D more easily.



As mentioned before d does not have big changes. We see the worst case is to have d=2, this is because again it means merging operations increase with all the cost it implies. In the same way, notice d=30 is good enough and secure enough to have as a merging parameter.

3.2.3. 1GB File

To perform external multi way merge sort on 1GB file (N=250 million integers), we went with parameters $M = 32678$ and $d = 30$, (which was the best value of M for 128MB file). This resulted in $\lceil N/M \rceil = 7630$ streams. When conducted the experiment, we ran into the following error:



```
19:00:30.140 [main] INFO  algo.MultiwayMergeSort - File size in bytes 1000000000
19:00:30.141 [main] INFO  algo.MultiwayMergeSort - File Size is 250000000 integers
19:00:30.141 [main] INFO  algo.MultiwayMergeSort - Memory allocated : 32768 integers
19:00:30.141 [main] INFO  algo.MultiwayMergeSort - Size of Streams : 32765 integers
19:00:30.141 [main] INFO  algo.MultiwayMergeSort - Number of Streams : 7630
19:01:36.328 [main] INFO  algo.MultiwayMergeSort - Streams Queue References [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 3
19:01:50.336 [main] ERROR streams.FileGenerator - Error: Can't create file!/Users/ankushsharma/Downloads/sorted/NQxlZXXCmeZwq4UZf6o9YjPFt1lAdgrb.data
19:01:50.339 [main] ERROR algo.MultiwayMergeSort -
java.lang.NullPointerException: null
    at streams.write.FWriteStream.write(FWriteStream.java:38)
    at algo.MultiWayMerge.mergeSort(MultiWayMerge.java:76)
    at algo.MultiWayMerge.merge(MultiWayMerge.java:61)
    at algo.MultiwayMergeSort.sortAndMerge(MultiwayMergeSort.java:87)
    at benchmark.ExternalMergeSortBenchmark.getTimeTakenToExecuteMergeSort(ExternalMergeSortBenchmark.java:32)
    at benchmark.MergeSortBenchmarkRunner.main(MergeSortBenchmarkRunner.java:82)
19:01:50.339 [main] INFO  benchmark.ExternalMergeSortBenchmark - Time taken to execute Merge Sort is 80208 ms
Exception in thread "main" java.lang.NoClassDefFoundError: util/Util
    at benchmark.MergeSortBenchmarkRunner.main(MergeSortBenchmarkRunner.java:85)
Caused by: java.lang.ClassNotFoundException: util.Util
    at java.net.URLClassLoader$1.run(URLClassLoader.java:370)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:362) <1 internal call>
    at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:335)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 1 more
Caused by: java.io.FileNotFoundException: /Users/ankushsharma/Desktop/code/dbsa/out/production/classes/util/Util.class (Too many open files in system)
    at java.io.FileInputStream.open0(Native Method)
    at java.io.FileInputStream.open(FileInputStream.java:195)
    at java.io.FileInputStream.<init>(FileInputStream.java:138)
    at sun.misc.URLClassPath$FileLoader$1.getInputStream(URLClassPath.java:1288)
    at sun.misc.Resource.cachedInputStream(Resource.java:77)
    at sun.misc.Resource.getByteBuffer(Resource.java:160)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:454)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
    ... 7 more
```

In the next section we attempt to interpret this error.

3.3. Expected vs Experimental Observations

Our expectation of in memory being the fastest implementation for files of size 128MB was not met. We understand we were stressing the memory with this quantity of data but it is a good experiment to see how the external-memory merge-sort operates. Our theory as to why this is happening is probably because the input buffers are constantly being garbage collected after they are used up, which results in a lot of garbage collection pauses.
Furthermore, this experiment showed that we can reach a great performance with proper values of M, balancing the number of Disk IO's and the capacity of processing in memory.

On the 1GB file, we realized that each operating system imposes a limit on the number of files that can be "opened" by a process. This is done so as to limit the number of resources a process can use at one particular point of time. In our case, we were trying to create 7630 streams, i.e 7630 sorted files. On the machine under the test, the limit of open files was 256. This was verified by the command:

```
ulimit -a
```

4. Conclusions

This project allowed us to understand the real impact of I/Os operations. We experimented with different types of streams and learned the trade-offs of using different mechanisms for accessing data in secondary memory.

We understood the restrictions in which Database Systems operate and how with a proper algorithm you can overcome such restrictions. We also understood how complex and valuable is the task of optimizing physical resources to keep acceptable performance with restrictions of memory, open input/output streams and so on. Besides, now we now that increasing physical resources not necessarily means improving results. We saw how virtual memory and memory mapping can dramatically increase read operations by avoiding buffer copies. We had a glimpse of how important and complex it is for a database management system to manage data on a file system.

5. Appendix

In this section we briefly explain how to run the code. Our entry class into the code is called *MultiwayMergeSort* . It takes in the parameters :

1. file : The file of integers that has to be sorted and merged. It is passed via a Java File object
2. memory : The memory, in number of 32 bit integers allocated for the algorithm
3. D : the nunber of files that have to be merged. It should always be greater than one

The code to run :

```
final File file = new File("/path/to/file");
final int memory = 4096;
final int d = 2;
final MultiwayMergeSort m = new MultiwayMergeSort(file, memory, d);
m.sortAndMerge();
```

The directories to store sorted sublist and file extension are as follows :

```
public class Constants {

    public static final String SORTED_DIR = "./src/main/resources/sorted/";
    public static final String SORTED_EXT = ".data";

}
```

Sequential File

With this project we understood how critical it is for files to be stored sequentially to guarantee optimal read and write performances. We also would like to consider the case when there is no fragmentation on disk and the file created is stored in a contiguous blocks. Up till now in our experiment, we cannot be guaranteed that the file of integers that we created are actually stored sequentially on disk. If in file is stored sequentially, we wanted to modify our implementation slightly to consider what could be the buffer size for reading and writing to disk.

We used the following command to get the block size used in the system under test.

```
diskutil info / | grep "Allocation Block Size"
```

On the systems under test, the result is 4096 bytes. Since all the parameters are given in number of integers (N) and the system has a block size of 4096 bytes each block stores 1024 integers. If you have M of this buffers then you can have a block based cost.

$$BufferSize = \frac{BlockSize}{IntegerSize} = \frac{4096Bytes}{4Bytes} = 1024$$

Since we are now dealing with blocks and not only single integer values, we can divide our memory M into K buffers via the following formulas :

$$bufferSize = Util.getOsBlockSize()$$

$$
\begin{aligned}
&memoryInBytes = m * 4\\
&K = \lceil memoryInBytes / bufferSize \rceil\\
&inputStreamBufferSize = (K - 1) * bufferSize\\
&outputBufferSize = memoryInBytes - inputStreamBufferSize
\end{aligned}
$$

After dividing M into K buffers, we allocate $K - 1$ buffers to read from the disk in the sorting phase, and 1 buffer to the output. For the merging phase, we again use $K - 1$ buffers to read data, and 1 buffer for output.

REFERENCES

Coffey, Neil. n.d. “How Big Should My Input Buffer Be?” Accessed January 6, 2019. https://www.javamex.com/tutorials/io/input_stream_buffer_size.shtml.

Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. 2013. *Database Systems: Pearson New International Edition: The Complete Book*. Pearson Higher Ed.

SQLite Documentation. n.d. “Memory-Mapped I/O.” Accessed January 5, 2019. https://www.sqlite.org/mmap.html.

Vansummeren, Stijn. n.d. “INFO-H-417 : Database Systems Architecture [Université Libre de Bruxelles - Service CoDE - Laboratoire WIT].” Accessed January 6, 2019. http://cs.ulb.ac.be/public/teaching/infoh417.

“Wikipedia.” n.d. Accessed January 6, 2019. Memory-mapped file https://en.wikipedia.org/wiki/Memory-mapped_file.