Université libre de Bruxelles

INFO-H 417 Database System Architecture

Winter Semester 2017-2018

Database System Architecture Group 11 Assignment

Authors:
Dagoberto Herrera
Matricule ULB: 000455058
Sergio José Ruiz
Matricule ULB: 000458874
Yue Wang
Matricule ULB: 0004544710

Supervisors:
Prof. Stijn Vansummeren

January 7, 2018

Contents

1 Introduction

The purpose of this assignment is to examine the performance of an external-memory merge-sort algorithm under different parameters. The project is made up of two parts. The first section explores and evaluates four different ways to read data from, and write data to secondary memory (read & write, freed & fwrite, fread & fwrite with buffer size, and read & write through memory mapping). In the second section, the external multi-way merge-sort algorithm uses the most performant stream implementation to read and write data to disk. This algorithm was implemented according to the specifications described by Garcia-Molina, Ullman & Widom (2016). Finally, the differences and similarities between the expected and observed behavior are discussed, as well as the most convenient parameters for varying input sizes.

In the first phase where the performance of the streams is tested, a system with the following characteristics was used.

Machine type	PC
CPU	Intel Core i7 4GHz Quad Core
Hard disk type	SSD
Operating system	Windows 8.1
Total memory available	8GB
Programming language	Java

Figure 1: First phase machine information

In the second phase where the external merge sort is implemented, a system with the following characteristics was used.

Machine type	Macbook Pro 2015
CPU	Intel Core i5 2,7GHz Quad Core
Hard disk type	SSD
Operating system	macOS High Sierra
Total memory available	8GB
Programming language	Java

Figure 2: First phase machine information

The following libraries were used:

- java.util.PriorityQueue;
- java.util.Queue;

- java.nio.MappedByteBuffer;
- java.nio.channels.FileChannel;
- java.util.ArrayList;
- java.io.File;
- java.io.IOException;
- java.io.RandomAccessFile;
- import java.io.DataOutputStream;
- import java.io.FileOutputStream;

It was decided to use two systems because the first experiment required more speed (especially for stream 1) and Windows also failed to manipulate files with the multiway merge sort. It would be expected that the relative efficiency of the streams would not change from one system to another. In other words, this does not affect the selection of the best stream that is used in the second phase.

For the purposes of stream testing, 30 different files (k) were generated, each with 64,000,000 integers MB (N). While files of different size (N) were prepared for the External Multi-way Merge Sort Test. Regarding the sizes of the files, N = 1,000,000 integers was taken as a basis to multiply or divide by powers of 2.

For each experiment, the average execution time of 10 repetitions was calculated to level the fluctuations associated with other processes. The System.currentTimeMillis() method was used to time the execution of the tests in milliseconds. java.util.Random was used to generate a stream of pseudorandom 32-bit integers between Integer.MIN_VALUE and Integer.MAX_VALUE. This implementation is contained in the class RandomSample.java.

2 Observations on Stream

2.1 Expected Behavior

To create a cost formula (I/Os) for each of the streams, the following parameters were defined:

- C: cost, in I/Os
- N: size of the input file, in number of 32-bit integers
- k: the number of streams
- B: buffer size, in number of 32-bit integers (applies for stream 3 and 4)

2.1.1 Stream 1 – read & write

In this case, the program must read once and write once each integer in the file. The cost and time are directly proportional to N and k. The estimated cost function for read & write system calls is

$$C = 2 * N * k$$

2.1.2 Stream 2 – fread & fwrite

BufferedInputStream and BufferedOutputStream have an default buffer of 8,192 bytes -8KB- (Vorontsov, 2013), which is equivalent to (8,192x8/32) 2,048 integers. The read() method reads a buffer of 2,048 integers at a time waiting to be used. Meanwhile, the write() method writes buffers of 2,048 integers to disk, one by one. This does not implicitly means that the time that it takes to read the data is 2048 times faster. Because there is a buffer, there are internal operations to access it. The estimated cost function for fread & fwrite system calls is

$$C = 2 * (N/2,048) * k$$

2.1.3 Stream 3 – fread & fwrite with buffer size

The difference between this implementation and the second lies in the fact that now the size of the buffer is not constant. B is a new variable that represents the size of the buffer and is incorporated in the cost function

$$C = 2 * (N/B) * k$$

The cost is expected to be directly proportional to the value of N and k and inversely proportional to the size of buffer B.

2.1.4 Stream 4 – read & write through memory mapping

Memory mapping is a technique that treats files on disk in the same way as they were located in the main memory (Pournelle & McKean, 2004). Memory-mapped files are segments of virtual memory which have been assigned to direct byte-for-byte correlation with all or part of a file on disk (Stoica, 2016). One benefit of using memory mapping is that the system performs all data transfers in pages of data. Internally all pages of memory are managed by the virtual-memory manager which you decide when to page should be paged to disk. Since the virtual-memory manager performs all disk operations reading or writing memory one page at a time. It has been optimized to make it as fast as possible reducing the number of times the hard disk head moves. Another benefit of using memory mapping is that all of the current I / O interaction now occurs in RAM in the form of standard memory addressing, which makes execution transparent to the application and other disk transactions can win in terms of performance (Kath, 1993). In this case we define B as the number of integers that are mapped to internal memory. The estimated cost function is

$$C = 2 * (N/B) * k$$

From the cost formulas we have the hypothesis that:

1. Streamer 1 will be the slowest.
2. Streamer 3 will be faster than streamer 2 when the buffer of the first exceeds the value of 2048.
3. The streamer 4 it must be the fastest.

2.2 Experimental observations

Figure 3 contains the results of an experiment that tests the four streams with k = 1, expressed as the duration of the execution of reading + writing in milliseconds. There are four groups of columns N (N1 = 125,000, N2 = 1,000,000, N3 = 8,000,000, N3 = 64,000,000). Within each group N are the streams S1, S2, S3 and S4. Meanwhile in the vertical axis, incremental values of the buffer of the streams 3 and 4 are shown. To define the potential values of the size of the buffer (B), measured in number of elements, 2,048 were taken as a basis to multiply or divide by powers of 2. Thus B values ranging from 256 to 33,554,432 were generated.

In the scenario in which the value of k is increased with small and large sizes of N. Again, stream 1 is the slowest. The streams 2 and 3 have an equivalent performance and about 200 times faster than the streamer 1. The streamer reduces its performance with multiple k, but still remains superior to the other implementations, about 500 times faster than the streamer 1 .

The first conclusion is that the duration of the four streams is directly proportional to the size of the input N. For example, when going from an input of 8,000,000 to an input of 64,000,000, the execution time is approximately 8

B	N1=125,000				N2=1,000,000				N3=8,000,000				N4=64,000,000			
	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
256	1117	50	32	240	8174	36	39	1406	64623	224	315	11300	505508	1771	2555	88753
512	1117	50	5	23	8174	36	36	30	64623	224	286	216	505508	1771	3256	3445
1,024	1117	50	5	3	8174	36	33	16	64623	224	264	135	505508	1771	2072	2062
2,048	1117	50	5	2	8174	36	31	11	64623	224	238	88	505508	1771	1940	680
4,096	1117	50	5	2	8174	36	30	8	64623	224	232	66	505508	1771	1871	520
8,192	1117	50	4	2	8174	36	30	8	64623	224	229	56	505508	1771	1837	433
16,384	1117	50	5	2	8174	36	29	6	64623	224	246	51	505508	1771	1817	389
32,768	1117	50	5	1	8174	36	29	7	64623	224	233	50	505508	1771	1819	375
65,536	1117	50	4	1	8174	36	30	6	64623	224	232	48	505508	1771	1842	375
131,072	1117	50	5	1	8174	36	33	6	64623	224	229	47	505508	1771	1912	358
262,144	1117	50	5	1	8174	36	33	6	64623	224	246	46	505508	1771	1918	355
524,288	1117	50	6	1	8174	36	32	6	64623	224	248	48	505508	1771	1955	353
1,048,576	1117	50	8	1	8174	36	37	6	64623	224	261	48	505508	1771	2087	363
2,097,152	1117	50	12	1	8174	36	36	6	64623	224	249	48	505508	1771	1961	354
4,194,304	1117	50	24	1	8174	36	43	6	64623	224	254	47	505508	1771	2111	360
8,388,608	1117	50	31	1	8174	36	64	8	64623	224	345	47	505508	1771	2798	366
16,777,216	1117	50	73	1	8174	36	95	8	64623	224	407	46	505508	1771	2992	356
33,554,432	1117	50	138	1	8174	36	162	7	64623	224	453	47	505508	1771	3134	356

Figure 3: Stream experiment results

times longer. S1 (N4) / S1 (N3) = 7.8, S2 (N4) / S2 (N3) = 7.9, average S3 (N4) / S3 (N3) = 8.1, average S4 (N4) / S4 (N3) = 8.5.

Figure 3 has a color scale to track the effect of changing the buffer size on the execution time of streams 3 and 4. For stream 3 it is observed that increasing the buffer can improve the execution time to a certain point where performance rather begins to deteriorate. With stream 4 a poor performance is observed when the amount of mapped elements is small, as the value of B increases, the execution time is reduced to a certain point where the performance does not improve, although the execution does not deteriorate when the size of B is high.

To evaluate the effect of the number of streams (k). A small size N (N1 = 1,000,000) and a large size N (N2 = 64,000,000) were tested. For each scenario, 4 values of k were evaluated (k1 = 1, k2 = 10, k3 = 20, k4 = 30). The buffer size of streams 3 and 4 was defined from the optimal values that were obtained in the previous phase of the experiment with k = 1. (In this case the tests were not repeated because the execution of stream1 is very long). The results of this experiment are shown in Figure 4. As with the case of the size of N, increasing k also increases the execution time. For streams 1 and 2 that relationship is quite linear, both for N1 and N2. In the particular case of stream 4, the change from k = 1 to k = 10 substantially reduces performance and stabilizes in subsequent jumps.

When it's time to decide which is the stream with the best performance. Figure 5 shows the average execution time of the four streamers for different values of N with k = 1 and the optimal buffers for streams 3 and 4. Clearly stream 1 is the slowest. Streams 2 and 3 have a more or less equivalent performance in the order of about 280 times faster with respect to stream 1. And stream 4 is the fastest, 1300-1400 times faster than stream 1.

K	N1=1,000,000				N2=64,000,000			
	S1	S2	S3	S4	S1	S2	S3	S4
1	8,174	36	29	6	505,508	1,771	1,817	353
10	80,785	421	355	235	5,149,346	24,207	41,135	15,990
20	162,417	630	771	253	10,719,548	60,616	62,740	14,908
30	238,518	996	984	351	15,872,509	85,166	84,407	35,660

Figure 4: Test Streamer varying variable k

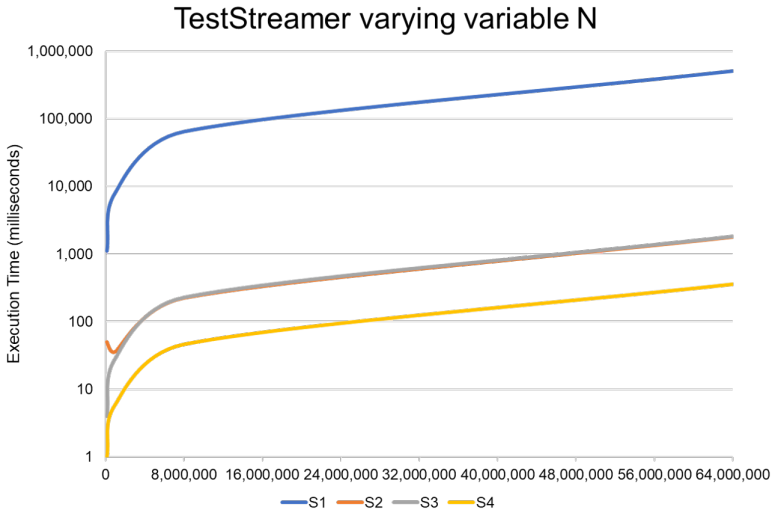

Figure 5: Test Streamer varying variable N

2.3 Discussion of expected behavior vs experimental observations

In general terms, it is confirmed that the execution time of the read / write streamers is directly proportional to N and k. The hypotheses that were raised previously are retaken.

a) streamer 1 will be the slowest:
This hypothesis was correct. In fact, stream 1 is the one that presents a more predictable behavior with respect to the comparison between the cost function and the execution time.

b) streamer 3 will be faster than streamer 2 when the buffer of the first exceeds the value of 2,048:
This hypothesis was incorrect: increasing the size of the buffer in stream 3 does not represent a benefit in terms of execution time with respect to stream 2.

Streams 2 and 3 were equivalent in the different scenarios.

c) The streamer 4 it must be the fastest, except when the number of elements mapped is very small:
This hypothesis was correct. Memory mapping turned out to be more efficient than the rest of the streams in all scenarios. This is because this technique allows to treat the mapped segments as if they were in the primary memory.

3 Observations on multi-way merge sort

3.1 Expected behavior

The program takes an input file as input and the following parameters:

- C: cost, in I/Os
- N: size of the input file, in number of integers
- M: the size of the main memory available during the first sort phase, in number of 32-bit integers
- d: the number of streams to merge in one pass in the later sort phases

In the first part, the input file is splitted into N/M streams. If M is bigger than the remainder numbers, we take the remainders. Then, each stream created is intermediately sorted by a priority queue, which is a main-memory Java sorting implementation that uses a heap. Then, batches of (at least) M size are written to disk. After the first pass, there are N/M sorted streams into the same file. The cost in this case is two times the access to the file, one for reading, one for writing. This job is done by the class called HeapSort.

$$C = 2 * N$$

Then, we split the file into N/M and we keep In the following passes we keep reading batches (parts of the file with size M). References to those streams are added to a priority queue. The content of the streams is never copied entirely in the memory. In successive iterations, a merge is made by copying and then eliminating d streams from the queue and saving the output immediately at the end of the queue, until the queue is of size 1. Since the queue does not store elements in memory, it stores the references to the streams. Each element must be read and written (2N) from one file to another file during the merge passes. The number of passes can be represented as log (d) N / M. The resulting cost of the merge phase is

$$2N * log_d(N/M)$$

The sum of the cost of generating N / M sorted streams and then merge them is:

$$2N + 2N * log_d(N/M)$$

3.2 Experimental observations

In this section a series of experiments is carried out, using stream 4. For the buffer size of stream 4, B = 524,288 was chosen, which was the value with the best performance for input N = 64,000,000. The input of the experiments are the files containing randomly generated 32-bit integers, which were created with the class RandomSample.java. Different values of N, M, and d are tested.

a) Varying N (number of integers) It is appropriate to use d = 2, to guarantee multiple passes of the merge phase especially when the size of N is small.

The figure 6 shows the relationship between the size of N and the execution time.

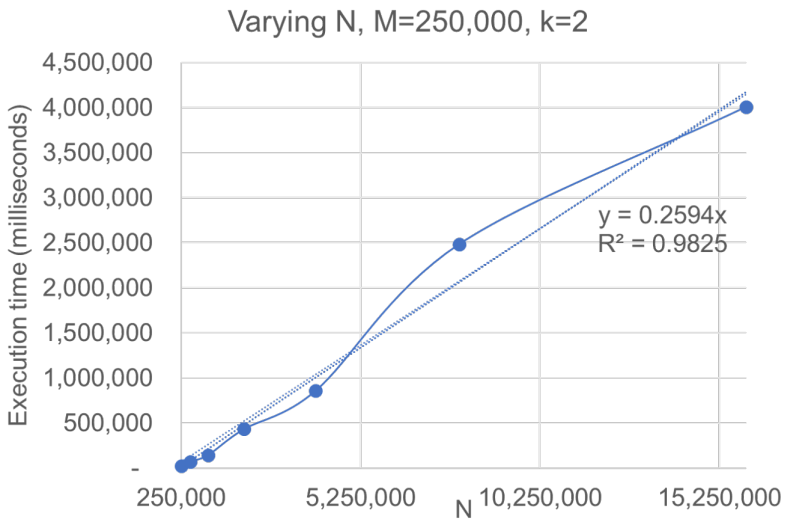

Figure 6: External multi-way merge sort varying variable N

N	Time Execution
250,000	22,676
500,000	74,244
1,000,000	146,273
2,000,000	433,633
4,000,000	859,663
8,000,000	2,485,375
16,000,000	4,009,375
M=250,000 k=2	

b) Varying M (number of integers) It is use d = 2, to guarantee multiple passes of the merge phase especially when the size of N is small.

The figure 7 shows the relationship between the size of M and the execution time with an small N=1,000,000.

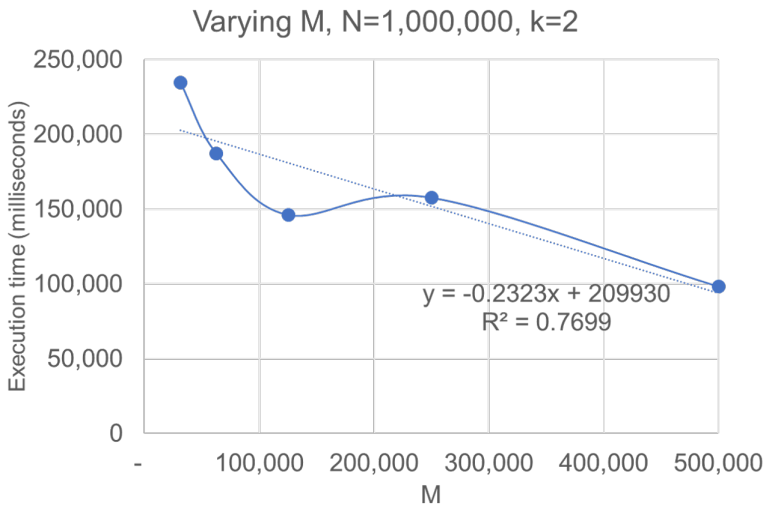

Figure 7: External multi-way merge sort varying variable M with N=1,000,000

M	Time Execution
31,250	234,676
62,500	187,406
125,000	146,273
250,000	157,781
500,000	98,470
M=1,000,000 k=2	

The figure 8 shows the relationship between the size of M and the execution time with an big N=16,000,000.

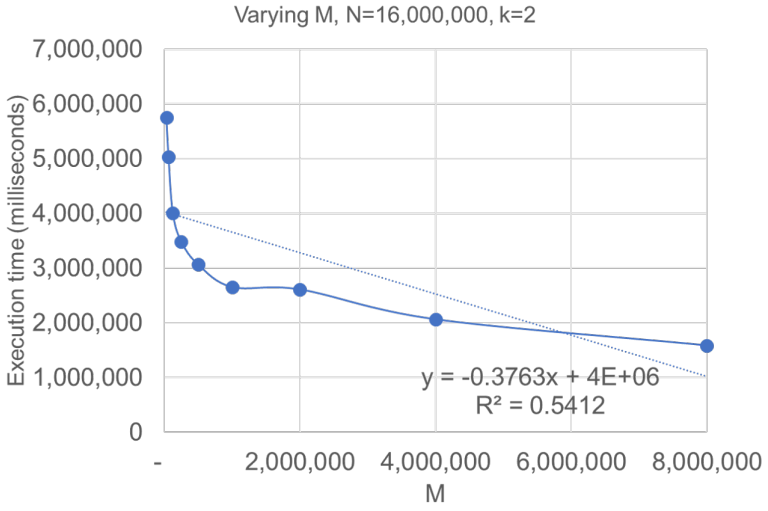

Figure 8: External multi-way merge sort varying variable M with N=16,000,000

M	Time Execution
31,250	5,752,713
62,500	5,027,378
125,000	4,009,375
250,000	3,483,022
500,000	3,064,456
1,000,000	2,652,740
2,000,000	2,609,709
4,000,000	2,064,103
8,000,000	1,587,206
M=16,000,000 k=2	

c) Varying d

The figure 9 and 10 show the relationship between the size of d and the execution time with an small N1=1,000,000 and N2=16,000,000 respectively.

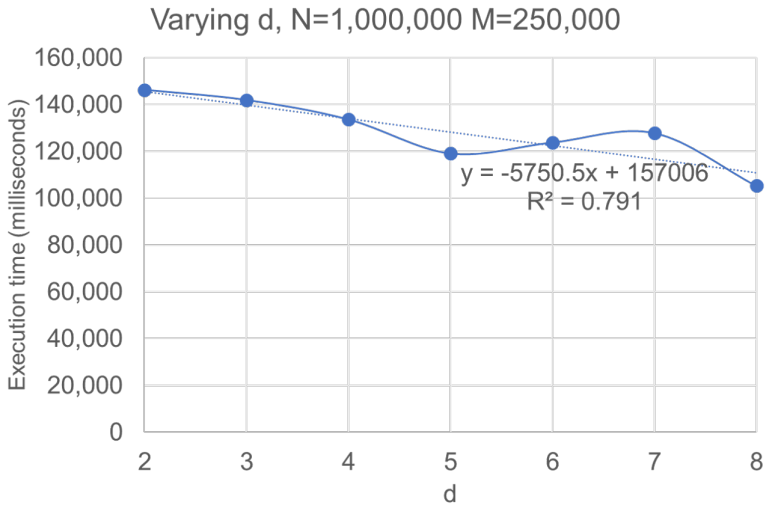

Figure 9: External multi-way merge sort varying variable d with N=1,000,000

d	Time Execution
2	146,273
3	141,924
4	133,617
5	119,100
6	123,771
7	127,772
8	105,318
M=1,000,000 N=125,000	

.

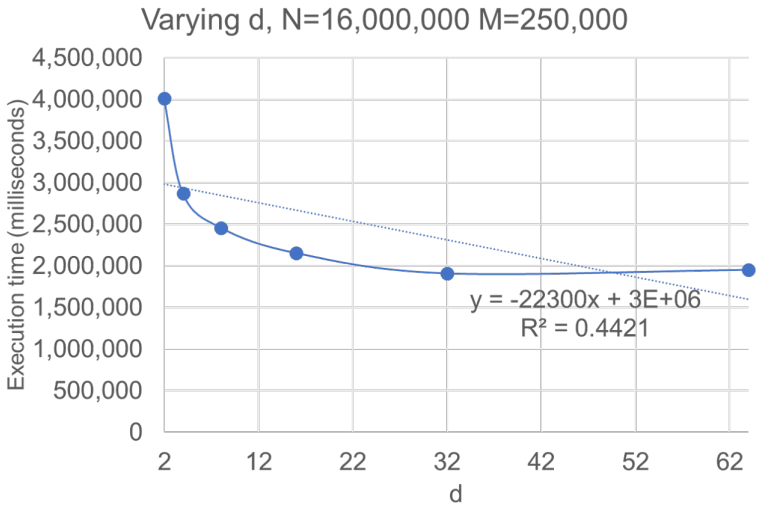

Figure 10: External multi-way merge sort varying variable d with N=16,000,000

d	Time Execution
2	4,009,375
4	2,869,975
8	2,457,831
16	2,153,043
32	1,909,681
64	1,951,737
M=16,000,000 N=125,000	

d) Good choices M and d for various input sizes

2		4		8	
31,250	234,676	31,250	203,284	31,250	165,121
62,500	187,406	62,500	131,184	62,500	137,449
125,000	146,273	125,000	133,617	125,000	105,318
250,000	157,781	250,000	98,974	250,000	111,143
500,000	98,470	500,000	114,562	500,000	106,670
N=1,000,000					

Figure 11: External multi-way merge sort varying variables M and d with N=16,000,000

2		16		64	
31,250	5,752,713	31,250	2,138,049	31,250	1,928,723
62,500	5,027,378	62,500	2,177,200	62,500	1,935,218
125,000	4,009,375	125,000	2,153,043	125,000	1,951,737
250,000	3,483,022	250,000	1,626,129	250,000	1,542,391
500,000	3,064,456	500,000	1,949,553	500,000	1,570,572
1,000,000	2,652,740	1,000,000	1,525,232	1,000,000	1,611,005
2,000,000	2,609,709	2,000,000	1,503,604	2,000,000	1,593,884
4,000,000	2,064,103	4,000,000	1,470,676	4,000,000	1,444,522
8,000,000	1,587,206	8,000,000	1,473,338	8,000,000	1,475,854
N=16,000,000					

Figure 12: External multi-way merge sort varying variables M and d with N=16,000,000

e) External multiway merge sort vs. heapsort

In this test, we compare sorting a file with heapsort and with the external multi-way merge-sort. For executing this test, we need to choose a M value so the file is entirely fit into memory.

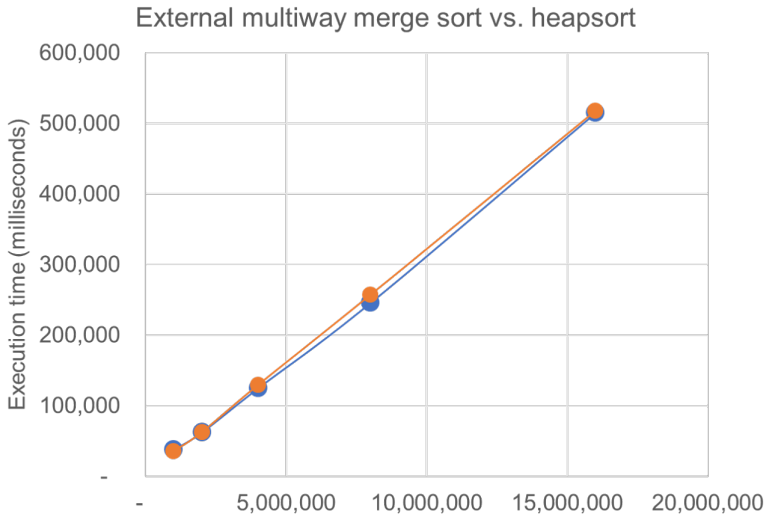

Figure 13: External multiway merge sort (orange) vs. heapsort (blue)

	TIME EXECUTION	
N	HEAPSORT	EXTERNAL
16,000,000	514,902	518,237
8,000,000	246,041	257,568
4,000,000	125,149	129,692
2,000,000	62,485	62,751
1,000,000	37,893	35,620

3.3 Discussion of expected behavior vs experimental observations

In general terms, it is confirmed that the execution time of the external merge sort is directly proportional to N, and inversely proportional to M and d.

After executing the benchmark and analyze the expected behavior, we can see that there are three critical points int the performance of the problem:

First of all, we have the size of the file N (this is, the amount of numbers that is contained in the original file). The bigger amount of numbers we have to deal with, the bigger is the I/O cost. This increment has a mostly linear component that can be traced to $N * log_d N$. So, incrementing N will impact negatively to the cost and the time of execution.

Secondly, we got M, which represents the size of each batch in which is splitted the original file in the first pass. This batches will be merged lately in the merge phase. The bigger M is, the less amount of Streams will be merged lately. Because of this, the bigger amount of streams we have, the bigger will be the executions of the merge (this is, if d has always the same value during

the execution). The results of increasing the size of M for N1 = 1,000,000 and N2 = 16,000,000 are in line with the expectations of the effect -$log_d M$

Lastly, we got the size of the buffer d. The bigger the value of d, the bigger is the amount of streams that will be merged in parallel in the merge phase. Choosing a big d will increment the performance.

As we can see in the results, in all three cases we mentioned, our results are in line with the expected behavior. Also, we want to mention that we used the default settings of Java Virtual Machine. This is, all the resource limit of JVM are configured by default.

4 Overall conclusion

This project consisted in the implementation of an external multi-way merge sort using Java to determine the effect that different parameters have on the performance of the algorithm measured at runtime.

In the first part four input / output streams were tested, to determine which of them has the best performance. In general terms, the value of N (number of integers) and k (number of streams) is directly proportional to the execution time. Streams 3 and 4 incorporate an additional variable B, number of buffer elements and number of elements mapped respectively. It is possible to reduce the execution time by increasing the size of the buffer to a certain point where improvements in performance are no longer observed. Stream 4, which consists of read and write through memory mapping, was the best implementation.

For the External Multi-Way Merge-sort we can see that, as for the streamers, N impacts directly to the performance. Incrementing N will impact negatively for both the cost and the time of execution. Other variables to take into account are the size of the buffer d, because is the number of streams that will be merged in parallel in the merge phase. Choosing a big d will increment the performance. Finally, M, which represents the size of the streams. These streams will be merged lately in the merge phase. The total amount of these streams is calculated dividing N by M, so with a big value of M we will get less amount of executions of the merge-sort.

When comparing our external merge-sort implementation with a main-memory sorting algorithm (heapsort). It is observed that both alternatives have a similar performance.

5 Reference

- Garcia-Molina H., Ullman J.D., & Widom J. (2016). Database Systems: The Complete Book (second, international edition). Prentice Hall.
- Vorontsov, M. (2013). Java Performance Tuning Guide: java.io.BufferedInputStream and java.util.zip.GZIPInputStream. Retreived from: http://java-performance.info/java-io-bufferedinputstream-and-java-util-zip-gzipinputstream/
- Pournelle, J.,& McKean, E. (2004). 1001 Computer Words You Need to Know. Oxford University Press.
- Stoica, I. (2016). Section 10: Intro to I/O and File Systems. Retrieved from https://inst.eecs.berkeley.edu/ cs162/fa16/static/sections/section10.pdf
- Kath, R. (1993). Managing Memory-Mapped Files. Retrieved from https://msdn.microsoft.com/en-us/library/ms810613.aspx