Université libre de Bruxelles

INFO-H-417 - Database System Architecture

Algorithms in Secondary Memory

Raymond Lochner and Aldar Saranov

Contents

January 8, 2017

1 Introduction

The purpose of this project implement an external memory sorting algorithm and measure its performance. This is needed if one wants to sort a file which is larger than the available memory. We do this by exploring several methods to read and write data to secondary memory - to find out which methods is suited best for this task - and implement a multi-way and external multi-way merge sort. Java is being used for this project.

2 Streams: Reading and Writing

In this section we will look at 4 different ways to read and write data to external memory, test each implementation with various parameters and conclude which method we found suitable best for our task.

To measure the performance of each implementation, we have created several test classes[1] and we take the time before and after the execution with the $System.nanoTime()$ command. Each of these test cases is being repeated 10 times to increase the accuracy which is being affected by the I/O influence of the operating system.

The tests are executed very simple. We write n-random integers to a file and then read the file until EOF is reached. Due to the operating system being smart and for the purposes of testing the real I/O speed, we are not allowed to read the same file twice. This is because of the lower workings of the operating system, where it keeps parts of already read data in memory and would hence make the tests meaningless. To show that this will give us good results, we can change the repeating factor between 1 and 20 and observe that the read times do not change drastically and stay in a margin of error.

The hardware specifics used for the tests can be found in appendix A and B.

2.1 One element at a time

For the first test we very simple read or write one integerto or from file[2]. We expect this technique to not be very fast as it reads only 4 bytes from the whole block of data from disk, which is 8192 bytes large. To read 2000 integers for example, this technique would need to read 8192 bytes 2000 times. Since we expect this to be very slow, we limit the testing on files up to 1 mb.

Integers	Size	Write Time [ns]	Read Time [ns]	$\frac{WriteTime}{Integers}$	$\frac{ReadTime}{Integers}$
25	100 bytes	1091905	195778	43676	7831
250	1 kb	1425048	571291	5700	2285
2500	10 kb	7395483	3784782	2958	1513
25000	100 kb	56178166	35476110	2247	1419
250000	1 mb	479258209	322050584	1917	1288

Table 1: Reading and Writing single integers

2.2 Buffering, using a library provided Buffered Stream

In the second implementation, we use $BufferedInputStream$ and $BufferedOutputStream$ library provided to read and write data[3]. This method reads a whole block of 8192 bytes from the hard disk and stores them in memory. When we call $readInt()$ the first time, it collects 8192 bytes (or 512 bytes - this is platform dependent), stores them in memory and returns to us the first 4 bytes. When we call it again after that, we don't have to access the disk again as the next 4 bytes were already collected and stored in RAM. This should make this implementation faster than the previous one.

[1] src/com/github/aldar_najim_raymond/test/
[2] src/com/github/aldar_najim_raymond/test/TestSimple.java
[3] src/com/github/aldar_najim_raymond/test/TestBuffered.java

Integers	Size	Write Time [ns]	Read Time [ns]	$\frac{WriteTime}{Integers}$	$\frac{ReadTime}{Integers}$
25	100 bytes	16188863	3798644	647554	151945
250	1 kb	1017139	139593	4068	558
2500	10 kb	3749788	1134005	1499	453
25000	100 kb	7617777	3798588	304	151
250000	1 mb	26790859	28273992	107	113
2500000	10 mb	253827116	184716897	101	73
25000000	100 mb	2181940186	463307342	87	18

Table 2: Buffered input and output using $BufferedInputStream$ and $BufferedOutputStream$

When we compare these results with the first implementation, we can clearly see that this method is a lot faster in terms of reading and writing.

2.3 Creating our own Buffered Stream

We create our own memory buffered stream by using the first technique. Instead of reading one integer, i.e. 4 bytes at a time, we indicate how many bytes we want to read from the hard drive and store them in memory. This is similar to the previous buffered stream method and we expect similar results.

Before we can test the speed of our own Memory Buffered implementation[4], we have to first test what value we should assign to the buffer size[5]. We do this by testing various buffer sizes and calculate the write and read time of a 100 mb file.

Buffer Size [bytes]	Write Time [ns]	Read Time [ns]	$\frac{WriteTime}{Integers}$	$\frac{ReadTime}{Integers}$
1024	2536980199	317583680	101	12
2048	2805871761	461848700	112	18
3072	2560823302	349881854	102	13
4096	2548883379	460483232	101	18
5120	2669493274	461499587	106	18
6144	2509305874	426169752	100	17
7168	2661079413	413066169	106	16
8192	2555655696	475487863	102	19
9216	2913873373	334538321	116	13
10240	2442408546	431839638	97	17
11264	2614165862	329610451	104	13
12288	2596385976	377040471	103	15
13312	2314436114	317969903	92	12
14336	2515973917	340552627	100	13
15360	2387622349	445647443	95	17
16384	2687217802	431713447	107	17
20480	2609755639	382593170	104	15
24576	2321382415	453519445	92	18
28672	2403891048	396418380	96	15
32768	2492095729	427299013	99	17
36864	2744283804	363239864	109	14
40960	2355269455	458182944	94	18
45056	2010202091	279064610	80	11
49152	1975515065	279877740	79	11
53248	2160770596	300076520	86	12
57344	2090669475	291854499	83	11
61440	2168303536	274332281	86	10

[4] src/com/github/aldar_najim_raymond/test/TestMemoryBuffered.java
[5] src/com/github/aldar_najim_raymond/test/TestMemoryBuffer_BufferSizes.java

65536	2159529124	275252526	86	11
1000000 (10 mb)	2076275394	279815200	83	11
10000000	2012123767	290502094	80	11
20000000	2204832371	282470064	88	11
30000000	2183227597	340751469	87	13
40000000	2439297657	402490405	97	16
50000000	2286560785	304337490	91	12
100000000 (100 mb)	2419916812	1308152865	96	52

Table 3: Testing different Buffer Sizes over a 100 mb file

This shows a similar performance with the Buffered Stream library when writing to file. Reading with higher buffer sizes seem to be faster. From these results, we can see that the read speeds do not get significantly faster with higher values and we can use a small buffer size for input streams with this implementation. The write speeds do get 20% faster with a larger buffer. Very large buffer sizes do not increase the read or write performance - from the results they even seem to slow down when getting larger than 20 mb. We conclude that we can use this implementation with many streams and let both the write and read buffer small and get good results. As the write stream is however the larger bottleneck, it is best to use a larger value for the write stream. From the results we set 45056 bytes as the write buffer size and 1024 as the read buffer size.

Integers	Size	Write Time [ns]	Read Time [ns]	$\frac{WriteTime}{Integers}$	$\frac{ReadTime}{Integers}$
25	100 bytes	12443939	4766953	497757	190678
250	1 kb	1981709	726459	7926	2905
2500	10 kb	1689943	1561509	675	624
25000	100 kb	10065860	1524894	402	60
250000	1 mb	29883041	6328236	119	25
2500000	10 mb	211476919	33549134	84	13
25000000	100 mb	2037684430	283550752	81	11

Table 4: Memory Buffer implementation with 45056 bytes write buffer and 1024 as read buffer

2.4 Memory mapping

Memory mapping is a technique where some user specified parts of external memory (hard-drive disk, GPU memory) is *„mapped“* into memory. Processes are able to access this portion of data in main memory which leads to a faster access time. Modifying data of the mapped portion modifies indirectly the actual source of the data, meaning the modified parts have to be written once changed to the original source.

Buffer Size [bytes]	Write Time [ns]	Read Time [ns]	$\frac{WriteTime}{Integers}$	$\frac{ReadTime}{Integers}$
1024	4730724983	1621612906	189	64
2048	4272506425	662096501	170	26
3072	4523560831	452007370	180	18
4096	3718064161	626180696	148	25
5120	3134060206	331441366	125	13
6144	3873352475	308028413	154	12
7168	3146199607	346544175	125	13
8192	3365698863	283990063	134	11
9216	2656665853	250897700	106	10
10240	2190130092	233776844	87	9
11264	2559233486	224366501	102	8
12288	1976972314	215491353	79	8
13312	2279427847	426658263	91	17

14336	2022304569	200938814	80	8
15360	1993880869	199851295	79	7
16384	2093839597	195146919	83	7
20480	2189921378	193777914	87	7
24576	1762869016	166971443	70	6
28672	1960698023	169691629	78	6
32768	2040090424	173852609	81	6
36864	2063261680	165659200	82	6
40960	2091542993	161275516	83	6
45056	2050219496	272949670	82	10
49152	2126054800	148670545	85	5
53248	2026096626	146748800	81	5
57344	1576313756	146695534	63	5
61440	1825208257	149669517	73	5
65536	1772204176	142746796	70	5
1000000	2178300225	131078129	87	5
10000000	1606189009	132031615	64	5
20000000	1581411619	132003751	63	5
30000000	1397448293	123978897	55	4
40000000	1457192809	122821312	58	4
50000000	1426037434	125239679	57	5
100000000	1455832366	121029906	58	4

Table 5: Testing different Buffer Sizes over a 100 mb file

As we can observe from these results, this method is slower with very small Buffer Sizes but is faster with larger values. It is different with the previous method in where very large values slow down the read and write speeds. We can observe that the Read Time per Integer is very fast starting at 4096 bytes as the Buffer Size and starts to even out at 32768 bytes.

We will hence use 32768 as the maximal buffer size for memory mapping with input streams and the write stream the remaining bytes for our algorithm.

Integers	Size	Write Time [ns]	Read Time [ns]	$\frac{WriteTime}{Integers}$	$\frac{ReadTime}{Integers}$
25	100 bytes	13247423	1042408	529896	41696
250	1 kb	3133945	283843	12535	1135
2500	10 kb	1902734	389320	761	155
25000	100 kb	4075581	660949	163	26
250000	1 mb	30471044	2289422	121	9
2500000	10 mb	150159888	7770379	60	3
25000000	100 mb	1693691625	67485740	67	2

Table 6: Memory mapped implementation with 32768 bytes as buffer

2.5 Read and Write speed conclusions

Our tests show that memory mapping is faster than every other implementation and will be used for both reading and writing for the implementation of the external memory merge sort algorithm.

A short inspection into the used hard-drive specifications in appendix B has to be made which show a Buffer Size of 8 MB and 4096 Bytes per Sector for the used hard drive. This would lead us to belief that a Buffer Size of 8192 bytes would be best suitable. However, the results of the tests show a different best suitable size. This is most likely due to the operating system and other processes interfering and accessing the hard drive.

2.6 Testing k input/output streams

As we need to use multiple input streams and a output stream for the external multi-way merge-sort, we need to know how many open streams we can have at a given time. First we will test how many open write streams we can have at any given time[6].

Listing 1: Testing simultaneously opened write streams

```
java.io.FileNotFoundException: 65522.txt (Too many open files)
        at java.io.FileOutputStream.open0(Native Method)
        at java.io.FileOutputStream.open(FileOutputStream.java:270)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:213)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:133)
        at com.github.aldar_najim_raymond.readerWriter.ReaderWriterSimple.
            <init>(ReaderWriterSimple.java:37)
        at com.github.aldar_najim_raymond.test.TestKStreams.
            main(TestKStreams.java:20)
```

As we can see, 65522 simultaneously opened write streams are being supported by the underlying system. We suspect that there is a shared maximum for both read and write streams, but we put this to the test as well[7].

Listing 2: Testing simultaneously opened read streams

```
java.io.FileNotFoundException: 65522.txt (Too many open files)
        at java.io.FileOutputStream.open0(Native Method)
        at java.io.FileOutputStream.open(FileOutputStream.java:270)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:213)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:133)
        at com.github.aldar_najim_raymond.readerWriter.ReaderWriterSimple.
            <init>(ReaderWriterSimple.java:37)
        at com.github.aldar_najim_raymond.test.TestKReadStreams.
            main(TestKReadStreams.java:22)
```

From this test we can see that read and write streams do indeed share the same maximum of 65522. This means for example that we could have 65522 read streams and 0 write streams or 32761 read and 32761 write streams.

For the purpose of the sorting algorithm, this gives us a limit of 65521 sorted input streams as we need 1 write stream to create the bigger sorted file. This seems a lot at first sight, however, if we have a very small amount of memory available during the initial splitting phase or a very large initial file, this seems not totally unreasonable and has to be taken into consideration while creating the algorithm.

3 External multi-way merge-sort

After studying various ways to read and write data to secondary memory and choosing the most suitable implementation for this task, we will now discuss the actual merge-sort algorithm.

To generate arbitrary large files with random integers, specified by a number represented by a *BigInteger*, we use the method created in [8].

[6] src/com/github/aldar_najim_raymond/test/TestKWriteStreams.java
[7] src/com/github/aldar_najim_raymond/test/TestKReadStreams.java
[8] src/com/github/aldar_najim_raymond/test/merge/FileCreator.java

3.1 Splitting a large file into sorted sub-files

Since we can not read files larger than the available memory and sort them there, we read a section of the large input file of unsorted integers into the main memory, sort them using $Arrays.sort()$[9] and store the sorted integers in a file. We then move on to the next section of the large file and repeat this process until we went through the entire file.[10]

After this stage we have n sorted files, of which we store the names in an $List < String >$. The $mergeFilePartly(StringfileName, intmemory)$ splits the given memory M (in bytes) into 3 even parts needed by:

1. Input-stream
2. Buffer to save the read integers and sort them in memory
3. Output-stream

3.2 Multi-way merge-sort

The multi-way merge-sort algorithm[11] takes as input: n sorted file references, the number d to represent the maximum stream value and the number M of maximal memory available.

The algorithm calls the merge method with maximal d files and merges them into one file. It sorts the lowest value of each stream in a Priority Queue[12]. It then writes the lowest value to the new file, deletes the value from the queue and inserts the next value of the InputStream in the queue. This is done until every InputStream has reached EOF.

We repeat this step until 1 sorted file remains.

3.3 Testing the effectiveness of our implementation

3.3.1 Different memory sizes

We will first test different memory sizes. From the stream tests, we concluded that 32768 bytes were a good middle-ground value for the read streams and write streams get faster with higher memory. We assume that k*32768 + 32768 will give us good results, i.e. where the buffer size is 393216 and we have 11 input streams. As the write stream is the bottleneck however, we expect that the total time needed will be faster with buffer sizes that are much larger than 393216. We will also look at the cost rate with higher buffer sizes and the memory cost associated with it - Buffer Size divided with the time taken.

Integers	Buffer Size	Max Streams	Time [ns]	$\frac{BufferSizes}{Time}$
250	100	11	39339189	393391.89
250	1000	11	1840457	1840.46
250	393216	11	867289	2.21
250	100000	11	640887	6.41
250	10000000	11	5133468	0.51
250000	100	11	33663730523	336637305.23
250000	1000	11	3129053871	3129053.87
250000	393216	11	122599070	311.79
250000	100000	11	118728615	1187.29
250000	10000000	11	86348500	8.63

[9] From the Oracle documentation: *„Sorts the specified array into ascending numerical order. Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.“*

[10] src/com/github/aldar_najim_raymond/test/merge/SingleFileMerge.java

[11] src/com/github/aldar_najim_raymond/test/merge/MultiWayMerge.java

[12] *java.util.PriorityQueue*

250000000	393216	11	311327742294	791747.39
250000000	100000	11	199338241337	1993382.41
250000000	10000000	11	181839057056	18183.91

Table 7: Testing different Buffer Sizes

From these results we can see that our initial assumption is partly correct. While higher buffer sizes do indeed give a better performance and have a better cost rate, the resources needed - a very high buffer size - might not be available. We can come to the conclusion that low buffer sizes are good with the formula of k*32768 + 32768, or when we have many input streams and a high buffer size.

3.3.2 Different d input-streams

We shall now test different input stream numbers with 393216 bytes as the Buffer Size. We expect that 11 streams is going to give us a good time compared to other streams, as we have previous found out that 393216 bytes is a good size for 11 streams.

Integers	Buffer Size	Max Streams	Time [ns]
2500	393216	2	11959008
2500	393216	5	2708230
2500	393216	10	2101940
2500	393216	11	2095814
2500	393216	15	2153601
2500	393216	20	12898534
2500	393216	30	3463888
2500	393216	50	2932659
2500	393216	100	3778971
2500	393216	1000	10917727
250000	393216	2	218909001
250000	393216	5	122922249
250000	393216	10	47400769
250000	393216	11	46019922
250000	393216	15	42925670
250000	393216	20	138095393
250000	393216	30	56352571
250000	393216	50	85354739
250000	393216	100	47112080
250000	393216	1000	141908889
25000000	393216	2	16639233421
25000000	393216	5	11862459901
25000000	393216	10	7876652490
25000000	393216	11	7638100291
25000000	393216	15	8441739295
25000000	393216	20	7867699817
25000000	393216	30	8311881713
25000000	393216	50	7370501047
25000000	393216	100	9747048143
25000000	393216	1000	21827527879

Table 8: Testing different input stream sizes with 393216 as the buffer size

The assumption that 11 streams were to perform good turned out to be correct. We can see the time taken is at the lowest point between 10 and 15 streams for small and large files

3.4 Suggested values

We will now suggest preferable values for the merge-sort. The formula of k*32768 + 32768 should be used when ha

- When using less than 2*32768 + 32768 = 98304 bytes of memory, d=2 streams should be used.
- Otherwise, one should calculate the number of streams by taking the available memory M in bytes and divide by M/32768 and subtract this number by one. This results in the formula of $d = M/32768 - 1$.

  For $M = 1000000 bytes, d = 1000000/32768 - 1 \approx 30$.

4 Conclusion

In this project studied and learned about 4 different ways to read and write data to a hard drive. Of these 4, the first 3 are almost the same in that they differ only by how many bytes they read at a time from the hard drive. The last one, memory mapping introduced us to a completely different technique. We learned that this technique is more performing than the others and will most likely be used in the future to do I/O in other projects.

From the experiments we learned that the optimal buffer size speed for reading and writing operations to a hard drive may differ than the actual value from the hard drive. On the tests used, the hard drive had a buffer Size of 8192 bytes, so the initial assumption is that this should be the value to use when reading and writing blocks. The tests showed however that this might not be necessary true and one should do these kind of testings with various buffer sizes when being bottle-necked by the IO operations and wanting a performance boost.

A DxDiag Hardware specification

Listing 3: DxDiag.exe System Information

```
------------------
System Information
------------------
       Time of this report: 1/3/2017, 17:09:22
             Machine name: TINAS-LAPTOP
               Machine Id: {2ACF1D60-89C6-4DD1-B0C5-1C3DCEAC9271}
         Operating System: Windows 10 Home 64-bit (10.0, Build 14393)
                           (14393.rs1_release_inmarket.161208-2252)
                 Language: German (Regional Setting: German)
       System Manufacturer: Acer
             System Model: Aspire V3-571G
                     BIOS: V2.04
                Processor: Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz
                           (8 CPUs), ~2.2GHz
                   Memory: 8192MB RAM
       Available OS Memory: 8008MB RAM
                 Page File: 6413MB used, 2873MB available
               Windows Dir: C:\WINDOWS
           DirectX Version: DirectX 12
       DX Setup Parameters: Not found
          User DPI Setting: Using System DPI
        System DPI Setting: 96 DPI (100 percent)
           DWM DPI Scaling: Disabled
                  Miracast: Available, with HDCP
Microsoft Graphics Hybrid: Supported
            DxDiag Version: 10.00.14393.0000 64bit Unicode

...

---------------------------
Disk & DVD/CD-ROM Drives
---------------------------
      Drive: C:
 Free Space: 575.2 GB
Total Space: 695.7 GB
File System: NTFS
      Model: TOSHIBA MQ01ABD075

      Drive: D:
      Model: MATSHITA DVD-RAM UJ8C0
     Driver: c:\windows\system32\drivers\cdrom.sys,
             10.00.14393.0000 (German),
             7/16/2016 12:41:53, 173056 bytes

...
```

B TOSHIBA MQ01ABD075 specification

Listing 4: Toshiba MQ01ABD075 specification

```
General
```

```
    Features          SilentSeek technology, shock sensor,
                      Silent HDD, Advanced Format technology,
                      S.M.A.R.T.
    Bytes per Sector  4096
    Interface         SATA 3Gb/s
    Buffer Size       8 MB
    Weight            3.95 oz
    Manufacturer      Toshiba
...

Performance
    Seek Time                   12 ms (average) / 22 ms (max)
    Drive Transfer Rate         300 MBps (external)
    Average Latency             5.55 ms
    Spindle Speed               5400 rpm
    Track-to-Track Seek Time    2 ms
...

Hard Drive
    Form Factor                 2.5" x 1/8H
    Interface Type              Serial ATA-300
    Spindle Speed               5400 rpm
    Hard Drive Type             internal hard drive
    Form Factor (Short)         2.5"
    Form Factor (metric)        6.4 cm x 1/8H
    Form Factor (Short)         6.4 cm
    Storage Interface           Serial ATA-300
    Interface                   Serial ATA-300
    Average Seek Time           12 ms
    Max Seek Time               22 ms
    Track-to-Track Seek Time    2 ms
    Average Latency             5.55 ms
    Data Transfer Rate          300 MBps
    Buffer Size                 8 MB
    Bytes per Sector            4096 Hz
    Non-Recoverable Errors      1 per 10^14
    Seek Errors                 1 per 10^6
```