INFO-H-417

Database Systems Architecture

2019-2020

Algorithms in Secondary Memory

Project Assignment

Group 11

JUNQI KUANG	000493883
YALEI LI	000493070
HAN JIANG	000493910

ULB UNIVERSITÉ LIBRE DE BRUXELLES

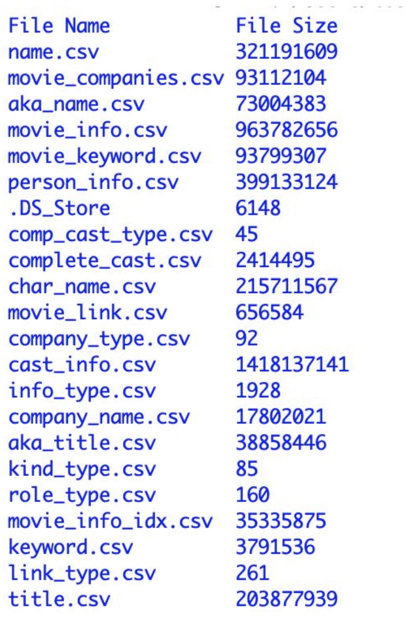
TABLE OF CONTENT

1. Introduction

This project is aimed to develop different external memory algorithms and experiment with their performances under different conditions. To be more precise, two major tasks are performed in this work: 1) different reading and writing methods targeting secondary memory are developed based on the requirements, and; 2) an external memory merge-sort algorithm is implemented given the dataset. Both tasks are conducted under various sets of parameters, and we managed to discover the algorithm and parameter setting to optimize secondary memory performance.

2. General environment

The programming implementation of the project is written in Python, and experimented with the real-world dataset of the IMDB movie database. The examined dataset contains CSV files in different sizes, which allows a multitude of testing purposes and real-world experience. For simplicity, the text files are mainly used to conduct reading, writing, and sorting tests. Figure 2.1 shows the general information for the dataset, and file size is read in bytes. The test result is generated based on the file sizes instead of the file contents.



```
File Name           File Size
name.csv            321191609
movie_companies.csv 93112104
aka_name.csv        73004383
movie_info.csv      963782656
movie_keyword.csv   93799307
person_info.csv     399133124
.DS_Store           6148
comp_cast_type.csv  45
complete_cast.csv   2414495
char_name.csv       215711567
movie_link.csv      656584
company_type.csv    92
cast_info.csv       1418137141
info_type.csv       1928
company_name.csv    17802021
aka_title.csv       38858446
kind_type.csv       85
role_type.csv       160
movie_info_idx.csv  35335875
keyword.csv         3791536
link_type.csv       261
title.csv           203877939
```

Figure 2.1. File Size Information

To complete the given tasks, besides the standard libraries (i.e. *io, os, time, mmap*), the table below reveals the list of external libraries we used to conduct this project.

External library	Usage
pandas, numpy	Assist in result processing
matplotlib.pyplot	Visually plot the result

The documented observations are generated under a macOS operating system with the following specifications:

Machine: MacBook Pro (Retina, 13-inch, Early 2015

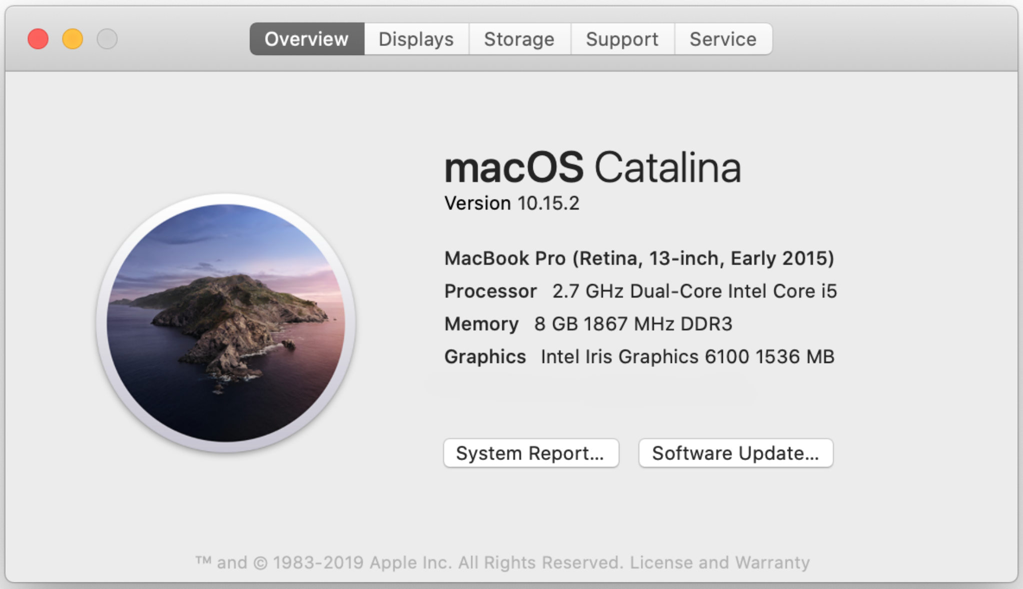
Operating System: macOS Catalina (Version: 10.15.1 (19B88))

Processor: 2.7 GHz Dual-Core Intel Core i5

Memory: 8 GB 1867 MHz DDR3

Hard Disk: APPLE SSD SM0128G

Detailed system information is demonstrated in Figure 2.1 and 2.2.

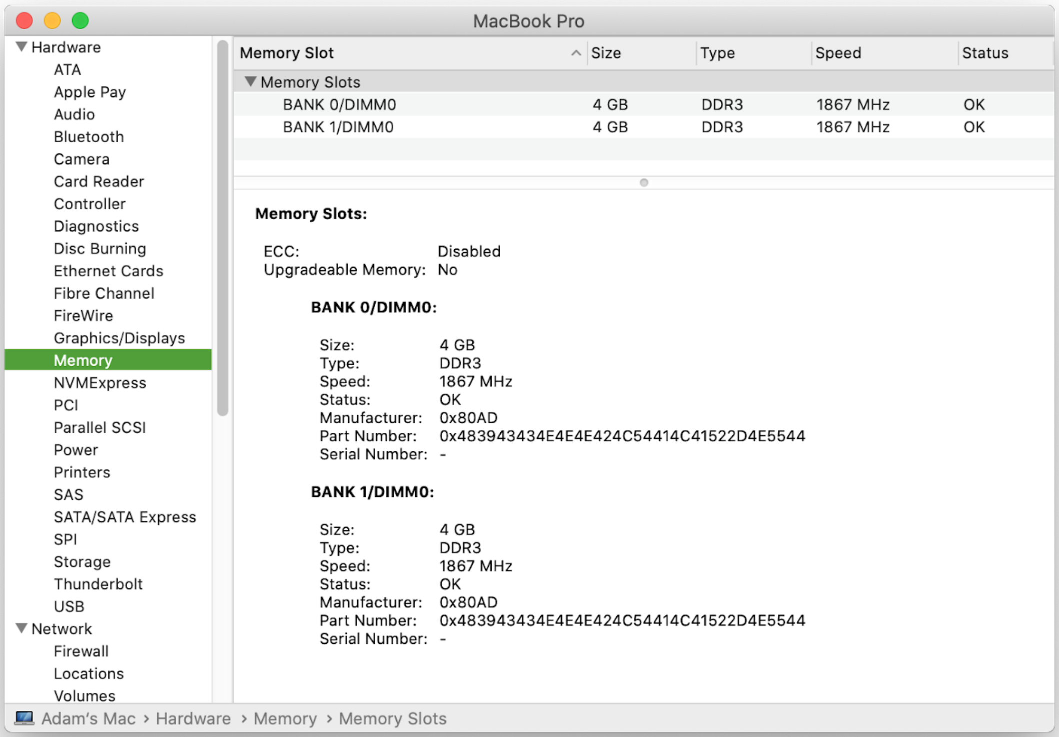


macOS Catalina
Version 10.15.2

MacBook Pro (Retina, 13-inch, Early 2015)
Processor 2.7 GHz Dual-Core Intel Core i5
Memory 8 GB 1867 MHz DDR3
Graphics Intel Iris Graphics 6100 1536 MB

™ and © 1983-2019 Apple Inc. All Rights Reserved. License and Warranty

Figure 2.1 Testing Environment Overview



Memory Slot	Size	Type	Speed	Status
Memory Slots				
BANK 0/DIMM0	4 GB	DDR3	1867 MHz	OK
BANK 1/DIMM0	4 GB	DDR3	1867 MHz	OK

```
Memory Slots:

  ECC:                  Disabled
  Upgradeable Memory:   No

        BANK 0/DIMM0:

          Size:           4 GB
          Type:           DDR3
          Speed:          1867 MHz
          Status:         OK
          Manufacturer:   0x80AD
          Part Number:    0x483943434E4E4E424C54414C41522D4E5544
          Serial Number:  -

        BANK 1/DIMM0:

          Size:           4 GB
          Type:           DDR3
          Speed:          1867 MHz
          Status:         OK
          Manufacturer:   0x80AD
          Part Number:    0x483943434E4E4E424C54414C41522D4E5544
          Serial Number:  -
```

Figure 2.2. Testing Memory Information

3. Implementations of I/O mechanisms

The external-memory merge sort implementation in the later part requires basic operations of reading data from, and write data to disk. Therefore, the first step is to develop *stream* classes to read and write text files (i.e. *input streams* and *output streams).* The following states the general project requirements for the *streams*:

For *input stream,* supporting operations should include:

- *open*: open an existing file for reading,
- *readln*: read the next line from the stream,
- *seek(pos)*: move the file cursor to *pos* so that a subsequent *readln* reads from position *pos* to the next end of line,
- *end_of_stream*: a boolean operation that returns true if the end of stream has been reached.

For *output stream,* the following operations are included:

- *create*: create a new file,
- *writeln*: write a string to the stream and terminate this stream with the newline character,
- *close*: close the stream.

The sections below give detailed explanations of the designated designs for the four distinct I/O mechanisms. For clarification, three parameters are defined in the following part, where:

N: number of bytes to be read or written on each stream,

k: number of streams open at a time,

B: buffer size in internal memory.

3.1. Implementation 1

The first mechanism is to read and write the text files **one character** at a time using the *read* and *write* system calls. In the Python environment, we mimic the behaviour by setting the *buffering* parameter of the open call to 0 when opening the file and then read/write each character with a loop operation. Major code in Python is written as `f = open(file, 'rb', buffering=0).`

Given the processing feature, each time a 32-bits character is read/write from/to the input/output stream, one I/O operation will be consumed. Therefore, the estimated cost may be: $\boldsymbol{C = k * N}$**.** According to the cost formula, it is expected that the performance may decrease when the file size increases and the processing streams increase. On the condition that either one parameter is fixed (i.e. N or k), the cost will proportionally increase with the increase of the other parameter.

3.2. Implementation 2

The second mechanism is to include an **application-default buffering mechanism** during reading and writing. The required ***fgets*** can be mimicked in Python by opening the file with the default buffering setting, and reading/writing line by line through the file with the *f.readline()* function.

For method 2, it is implied a buffering mechanism, which enables read/write each line within the buffer at a time. It makes use of the available internal memory capacity, which otherwise constrains the performance given the testing environment. Supposing the default buffer size as B_{sys}, each stream requires *N/B* times to process the whole file. It can be known through `io.DEFAULT_BUFFER_SIZE` that the default buffer size in Python 3.7.0 is 8192, and therefore the estimated cost may be: $C = k * \frac{N}{8192}$.

It is expected that with a larger default buffer size, the cost will be lower. Under the condition that *B* and *N* are fixed, the increase of *k* will decrease the performance. Intuitively, method 2 should be able to outperform method 1 with the buffer mechanism.

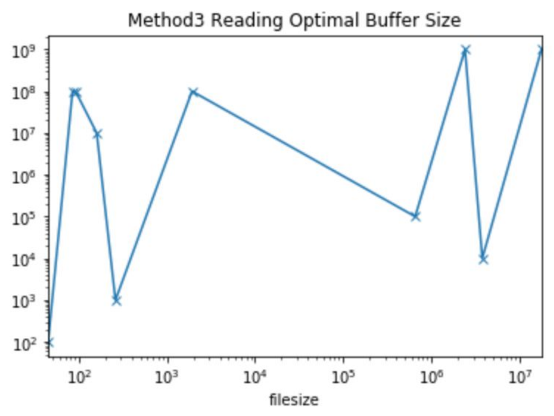
3.3. Implementation 3

The third method is rather similar to the first one, except for deploying **a buffer with the size *B*** in internal memory. This task is accomplished by pre-defining the buffering size when opening the file. To be specific, the buffer can be viewed as a byte array, where each character is first read/write to/from the buffer. Once the buffer is empty/full, the next buffer_size characters will be loaded/written from/into the input/output stream.

In Python, we specify the buffer size when open the reading/written file: `open(file, 'rb', buffering=B)`.Given the buffer size of *B*, characters loaded into buffer are in the block of *B,* and thus each stream will consume *N/B* I/Os. Overall, the estimated cost can be summarized as $\boldsymbol{C = k * \frac{N}{B}}$. It is expected that the larger the buffer size is, the better the performance is. If the *B* is set over 8192, it should be expected to outperform method 2 given the larger buffer size.

Optimal Buffer Size Determination

In order to horizontally compare the performance later, the relatively optimal buffer size needs to be determined in advance. We iterate method 3 over testing files 50 times with different magnitude of the buffer size (from 10 to 10^9), and manage to generate the optimal buffer size for each file size with a minimum running time (details shown in *BufferDetermination.py*). Figure 3.3.1 and 3.3.2 demonstrates the results for reading, writing, and the total running time separately. (Given the restraint of the testing system, file size larger than 2*10**7 bytes is not tested for writing.)



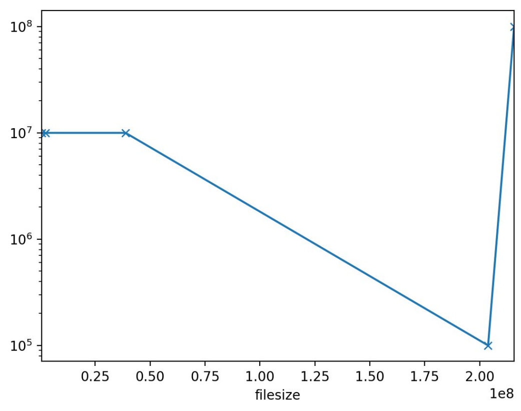
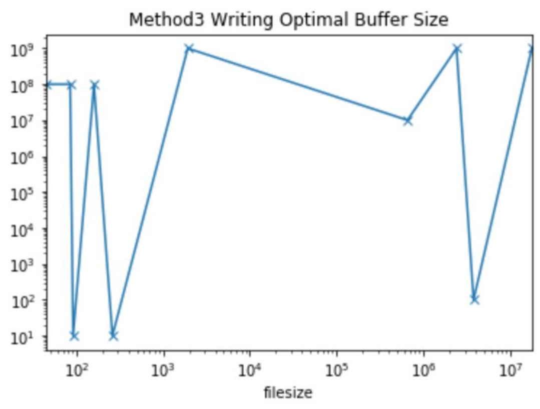


Figure 3.3.1. Method 3 Optimal Buffer Size for Reading in Different File Sizes



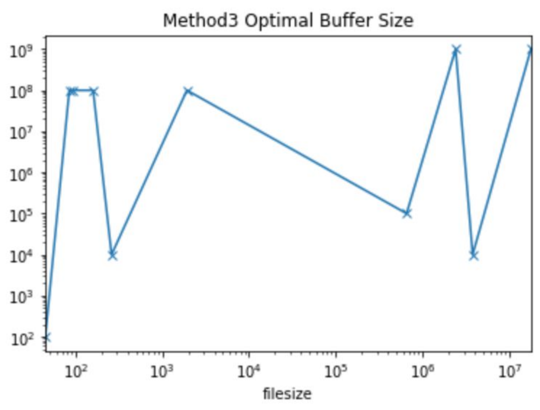


Figure 3.3.2 Method 3 Optimal Buffer Size for Writing (Left) & Total (Right)

As revealed above, for method 3, the buffer size varies for different files. We therefore take the mode as our optimal buffer size in the later test, which is **10^7** byte for reading and **10^8** for writing.

3.4. Implementation 4

3.4.1. Introduction of memory mapping

The last mechanism is featured in ***memory mapping*** operation to read and write data through mapping and unmapping *B* characters to internal memory. In nature, memory mapping can be regarded as a subclass of byte buffer (i.e. mapped byte buffer). Different from the implementation 3, it maps the buffered portion of a file, or an entire file (given adequate buffer size) on disk directly to virtual memory through a series of addresses within the application's address space [1]. Taking the advantage of virtual memory capacities, the mapped buffer can be expected with faster file reads and writes with reduced program complexity [2]. The estimated cost formula is $\boldsymbol{C} = \boldsymbol{k} * \frac{N}{B}$.

Figure 3.4.1 demonstrates the mapped buffer operation in comparison with standard I/O operation (Figure 3.4.2). The figures reveal the simplified operation of the memory-mapped buffer, where it replaces the conventional read() and write() system call by directly accessing data through process buffer space. Memory-mapped buffer is therefore expected to outperform the rest mechanism and facilitates frequent access of small files by reading into memory once. However, there may also emerge some problems for a memory-mapped buffer in practice. In this project, high memory usage often lead to application crashes. An additional file close clause is also required for this implementation since a mapped buffer will remain valid until it is garbage-collected [4].

In Python, the memory mapping operation can be done through *mmap* module, and specifying the demanded buffer size in the *mmap* statement.

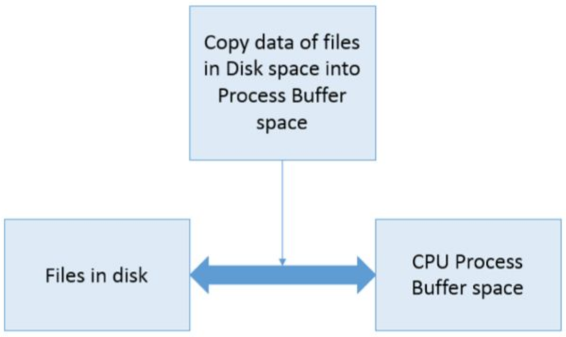


Figure 3.4.1. Mapped Buffer Operation [3]

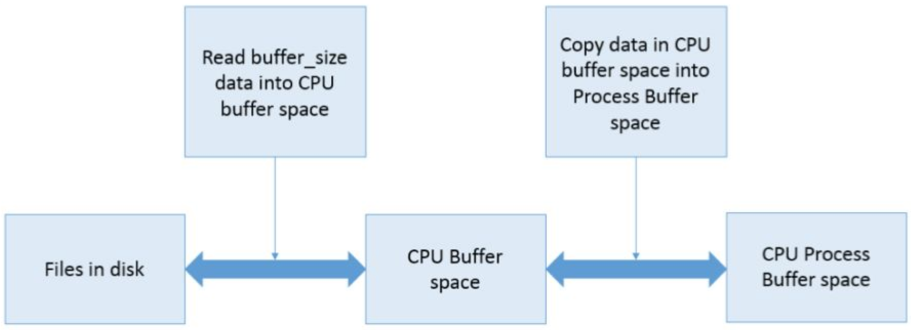

Read buffer_size data into CPU buffer space
Copy data in CPU buffer space into Process Buffer space
Files in disk
CPU Buffer space
CPU Process Buffer space

Figure 3.4.2. Standard I/O Operation [3]

3.4.2. Optimal Buffer Size Determination

After the same buffer size tests as method 3, figures below indicates the results for method 4.

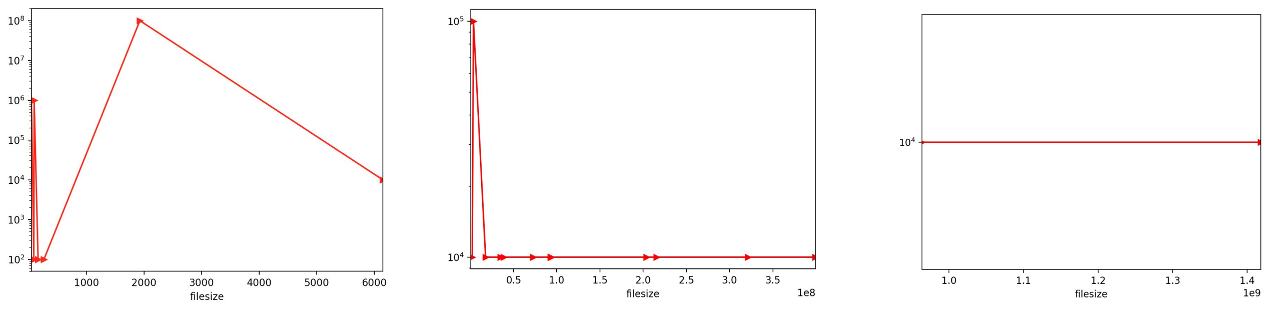


Figure 3.4.3. Method 4 Optimal Buffer Sizes for Reading

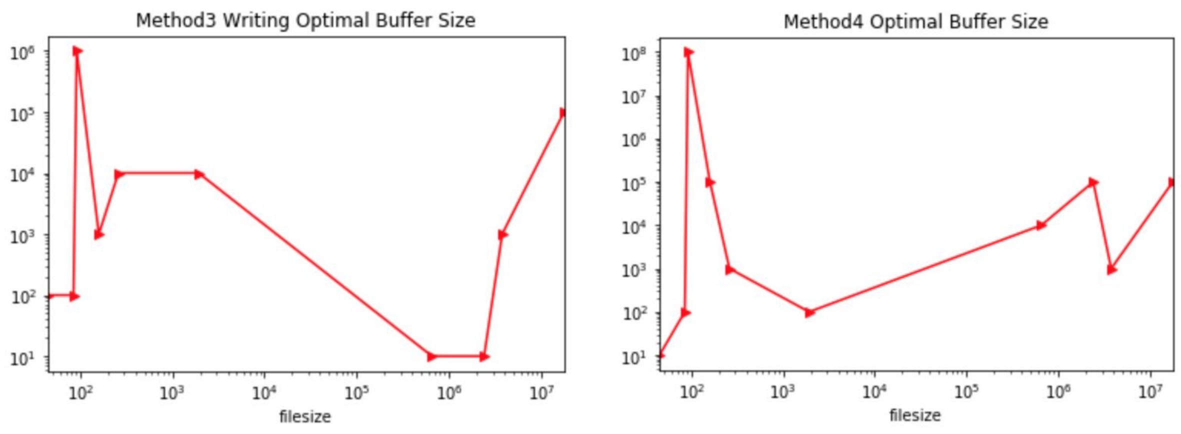


Figure 3.4.4 Method 4 Optimal Buffer Size for Writing (Left) & Total (Right)

For method 4, the buffer size fluctuates over the small files, but generally comes to an optimal value as 10^4 byte for reading (Figure 3.4.3). For writing, the optimal buffer size varies over different size, but within the range of 10^3 to 10^4 (Figure 3.4.4). Taking the overall running time, we take **10^4** as the optimal buffer size for method 4.

3.5. Running Times Determination

Due to the complicated computing environment, the tested running time for each I/O mechanism varies every time. Therefore, we need to decide the suitable number of times for each test. Figure 3.5.1 shows the trial average reading time over the running times (tested on *link_type.csv*). Therefore, we take 50 iterations to level out the fluctuations, where the methods tend to generate more stable results.

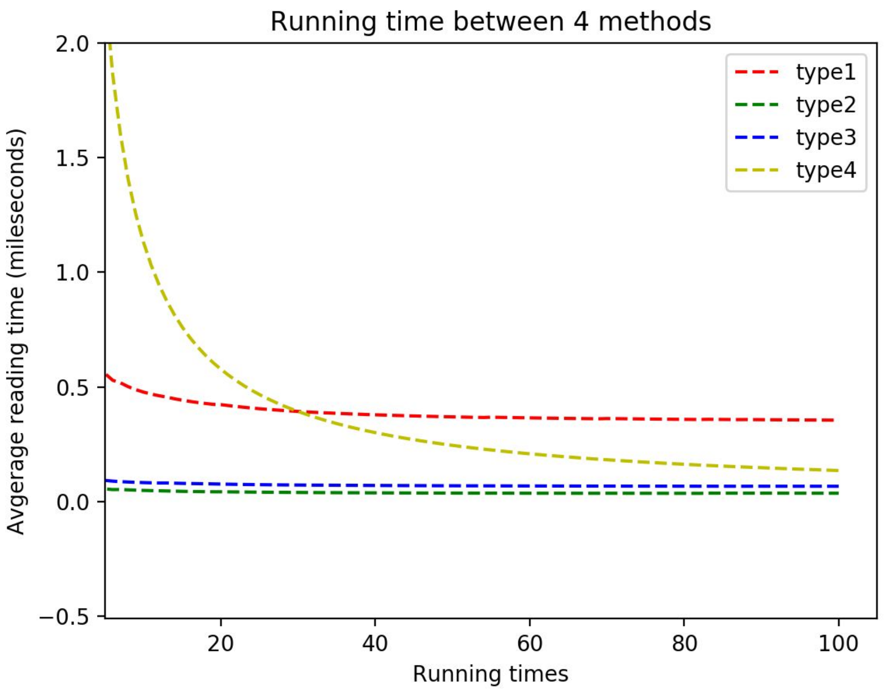


Figure 3.5.1. Performance Over the Number of Execution Times

4. Experiment 1.1: Sequential Reading

4.1. Expectations

As briefly discussed above, the four reading mechanisms feature differently, and therefore the estimated I/O costs vary. For sequential reading, the input stream can be viewed as 1 (i.e. *k = 1*). With previous cost analysis, method 1 is expected to underperform the rest, while method

2 to 4 may vary due to other factors of the testing environment. It is expected that the reading performance decreases with the increase of file size *N*.

4.2. Result

Given the significant range in file sizes, we run the reading rest in two batches for better visual illustration. The optimal buffer sizes for method 3 and 4 are set according to the previous experiment (i.e. Method 3: 10^7; Method 4: 10^4). Figure 4.2.1 shows the results for small files, and figure 4.2.2 for larger files.

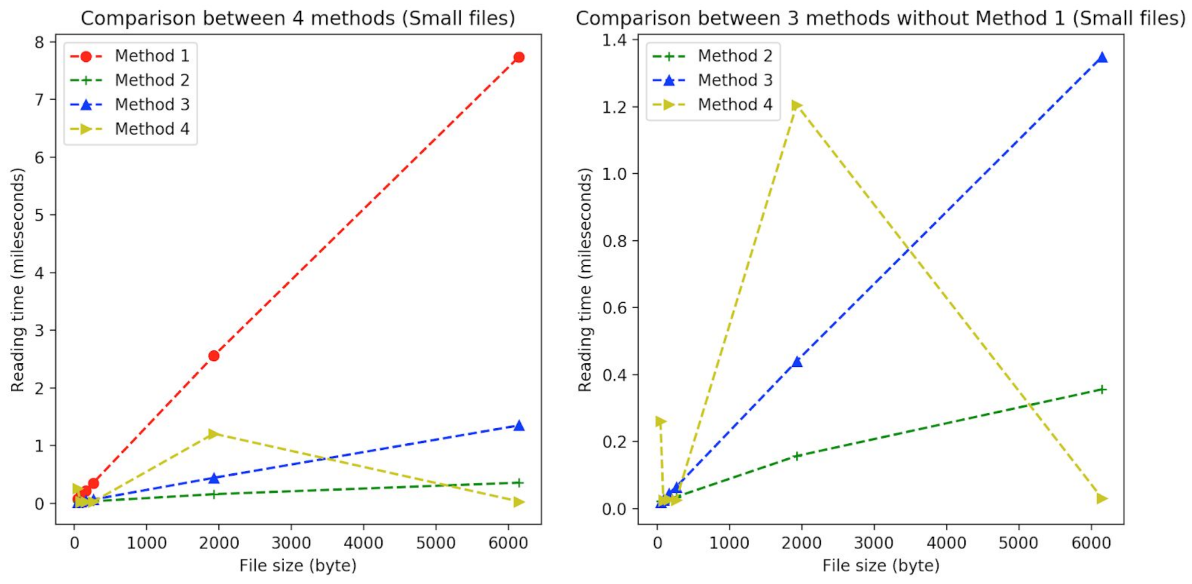


Figure 4.2.1 Sequential Reading of Small Files Comparison (Left: 4 methods; Right: 3 methods)

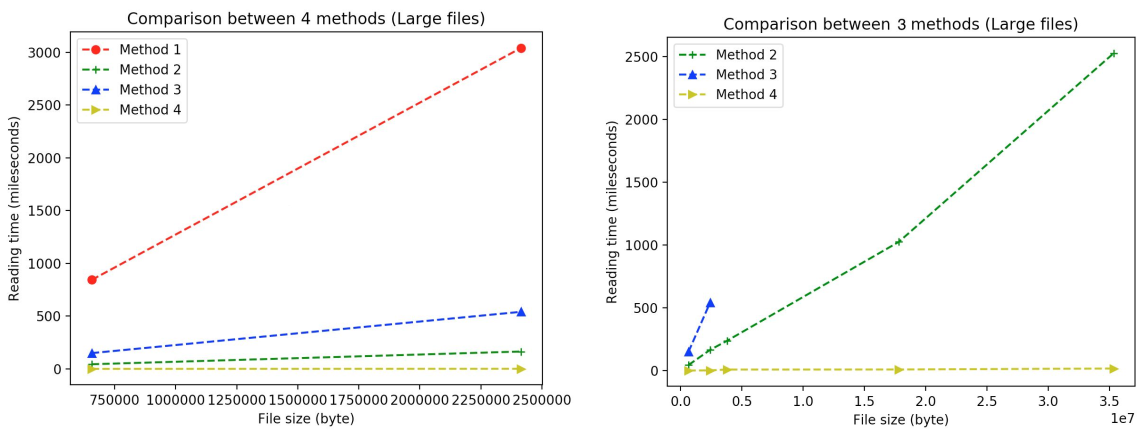


Figure 4.2.1 Sequential Reading of Big Files Comparison (Left: 4 methods; Right: 3 methods)

In accordance with previous expectations, method 1 takes the longest to perform the task regardless of the file sizes. Performance of method 4 fluctuates when reading smaller files but gradually outperform the others for large files. Thus, the best method for sequential reading may vary between method 2 and method 4 depends on the processing file size.

In addition, it is out of expectation that method 3 costs more than method 2, since we anticipate a larger buffer size of method 3 may outperform the default buffer size in method 2. We may suspect that the extra time could be resulted from the inside buffer reading process given the testing environment.

5. Experiment 1.2: Random Reading

5.1. Expectations

This experiment conducts *random reading* for the four implementations. Given the same random number, it is expected that the buffer-equipped methods may outperform method 1. To be specific, method 4 should be the best mechanism to access a specific position due to the simplified process design. In terms of the file size *N* and the number of iteration *j,* it is assumed that *N* may not influence the result much while *j* may make an impact.

5.2. Optimal buffer size

Given the different nature of the experiment, the optimal buffer sizes for method 3 and 4 may vary. As revealed in Figure 5.2, it is suggested that **100 for method 3** and **10^4 for method 4**.

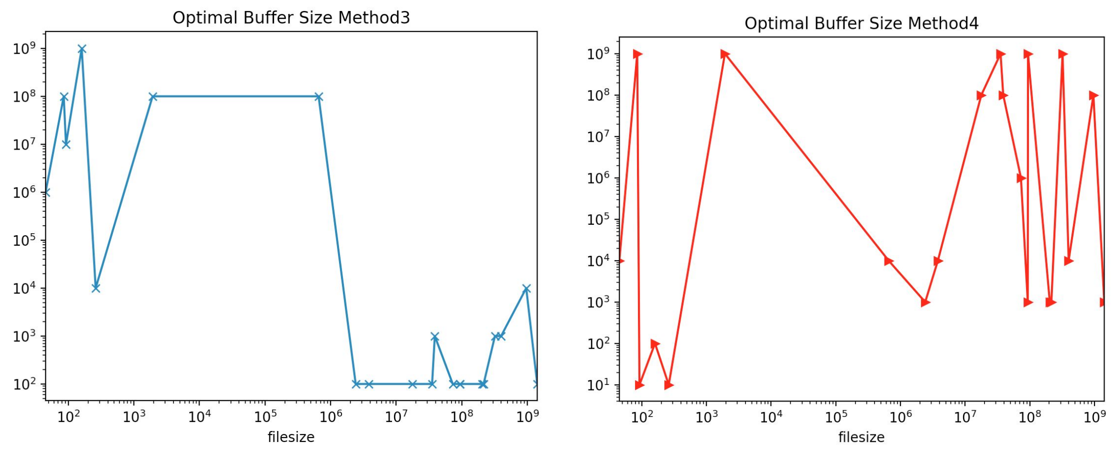
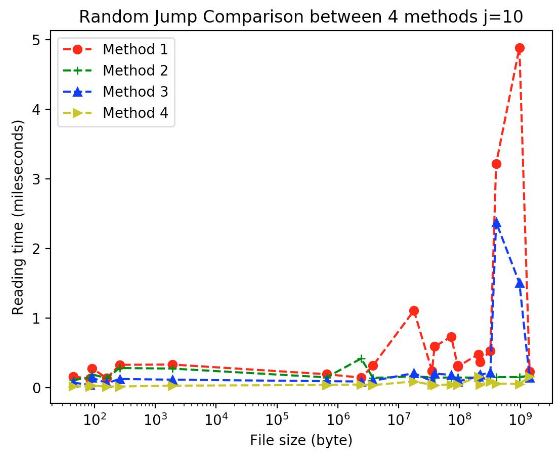


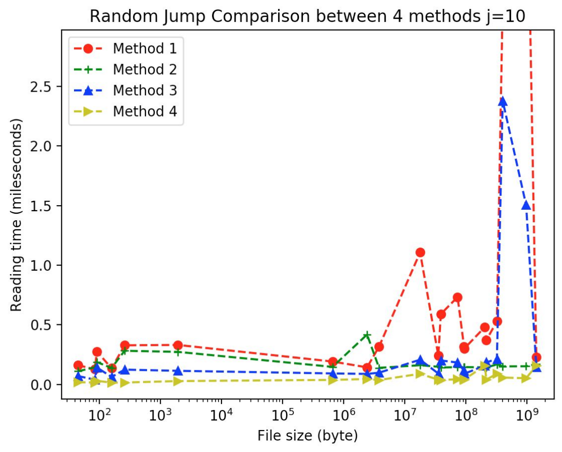
Figure 5.2. Optimal Buffer Size Results for Method 3 & 4 for Different Files

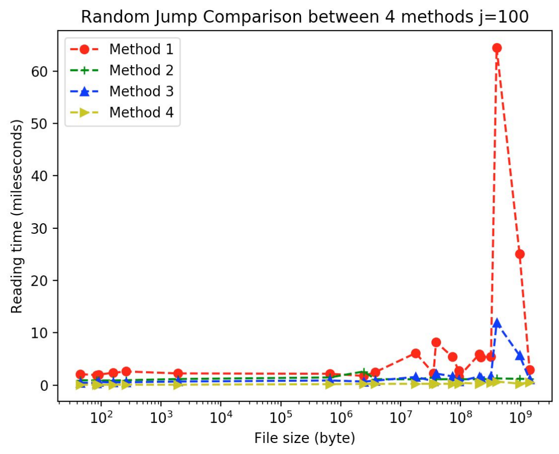
5.3. Results

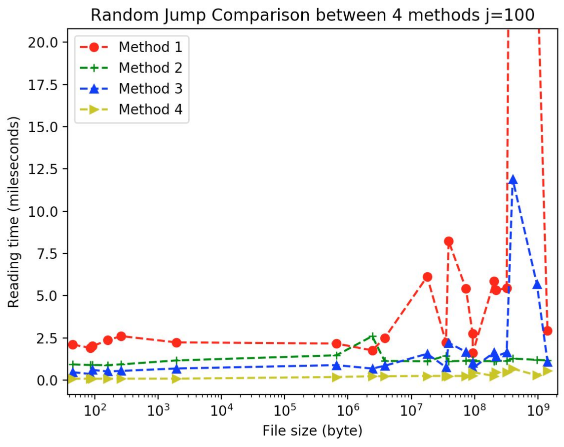
As expected, method 4 maintains a competitive advantage in this experiment. As mentioned in section 3.4.1, the memory-mapped buffer entitles faster access to files in comparison to the standard buffer operation. With the set of optimal buffer size, it results stably over the file sizes.

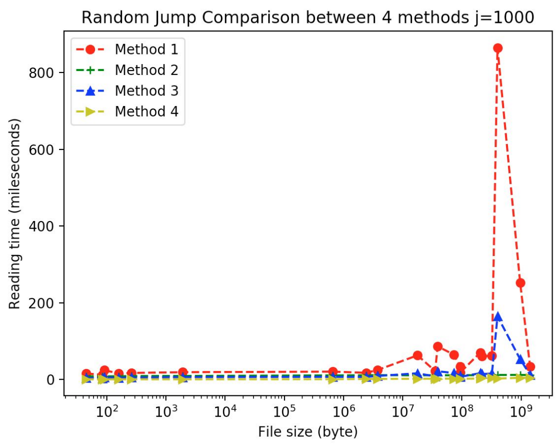
As shown in the graphs, the file sizes make less difference for the I/O mechanisms. The number of random jumps performed proportionately increase the execution time, which is as expected. However, the performance over large files varies for each method. It is speculated that the content structure for each file may be different, which can be significant impact during the reading line process. Method 3 outperforms method 2 in this experiment, which is different from the experiment 1.1. Possible reason could be that the counting line length process takes less time which reveals the advantage of a bigger buffer of method 3.











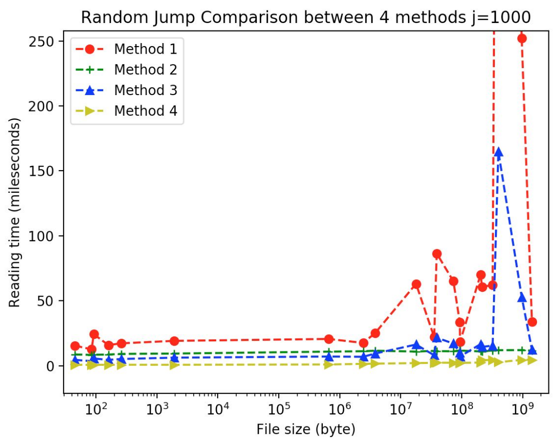


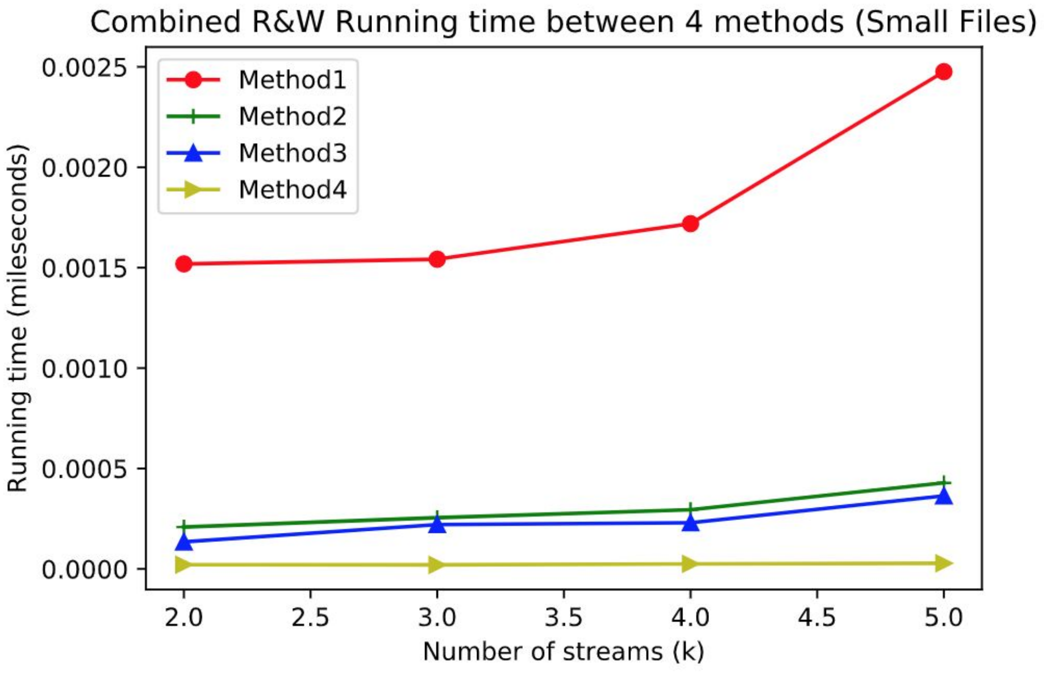
Figure 5.3. Results for Random Jump between 4 Methods for Different Files with Different Iteration (*j*)

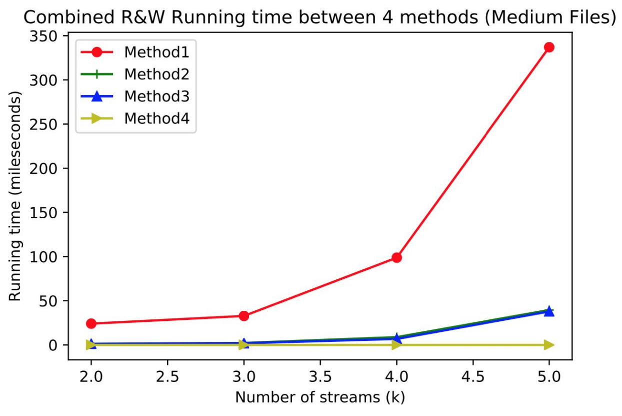
6. Experiment 1.3: Combined Reading & Writing

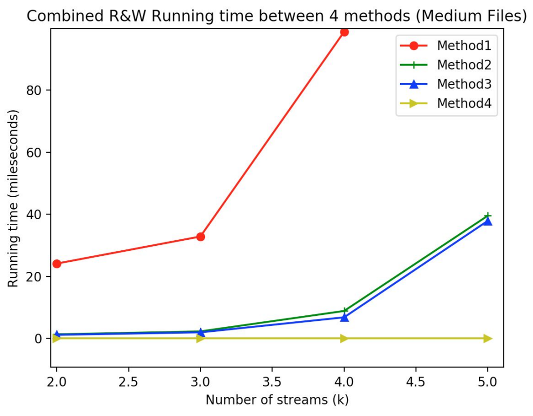
In this section, we try to identify the best combination of reading and writing methods from the previous part. Based on experiment 1.1 and 1.2, method 2 and 4 both appear to be the top performer, and thus we use these two reading methods together with all the possible writing means.

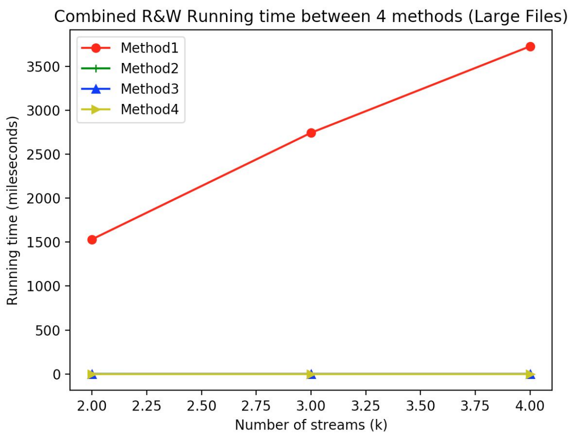
It is anticipated that *k* may influence greatly on the performance given the estimated cost formula before. The optimal buffer size is taken as discussed in section 3, where 10^8 for method 3 writing and 10^4 for method 4. Figures below demonstrate our preliminary results.

Reading: method 2; Writing: all:









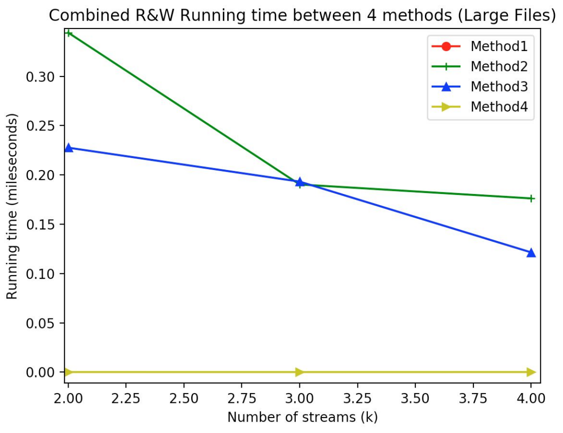


Figure 6.1. Results for Method 2 Reading & All Means Writing over Different Files

Reading: method 4; Writing: all

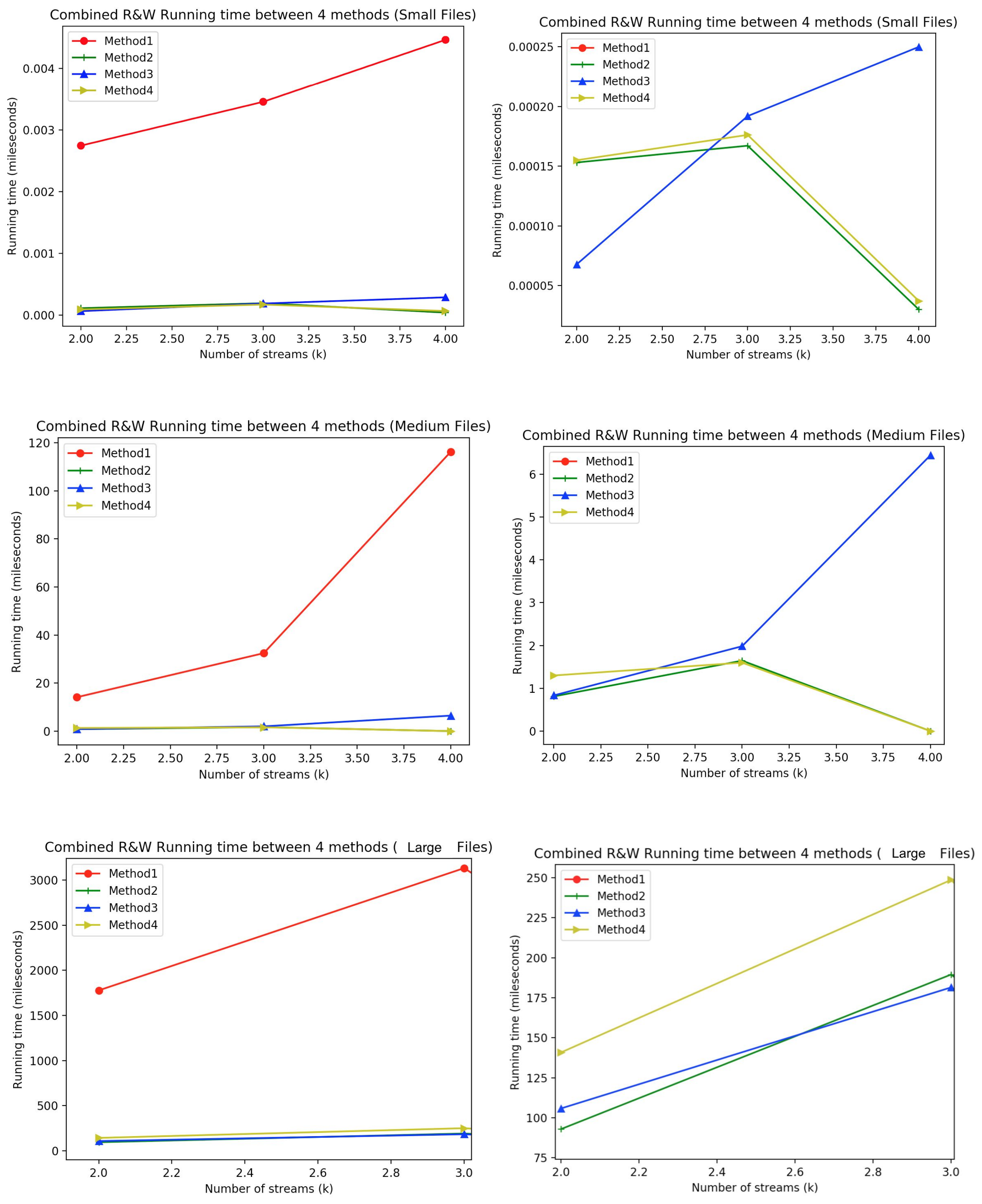


Figure 6.2. Results for Method 4 Reading & All Means Writing over Different Files

Taking all results as a whole, it is explicit that the combination of **method 2 reading and method 4 writing** is the solution for this task, especially when dealing with large files. However, it is surprising that the combinations of method 4 underperform the ones of method

2. We may think that when files get larger in size, it may take a significant amount of the virtual address space to process the commands. Mapping files under large size often lead to out-of-memory error in practice, which we believe to be the main reason why method 4 fails in this experiment.

7. Multi-way merge

7.1. Expectations

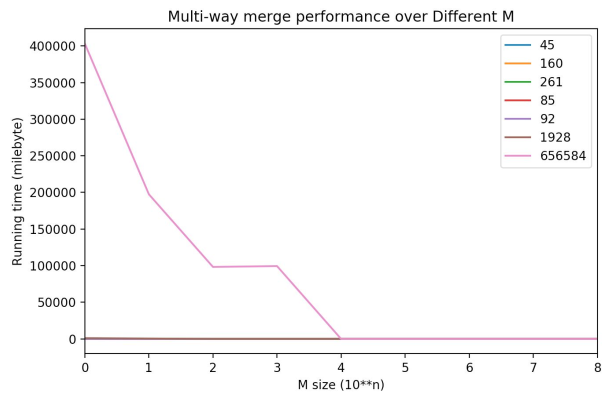
Based on the previous experiment, the combination of method2 reading and method4 writing stands out as our solution for I/O optimization. In this section, an *extsort* program is conducted to perform the external-memory merge-sort algorithm. Four parameters are employed here: an input file f from the IMDB dataset, a positive integer k as the demanded kth column to be sorted of the file, a positive integer M as the buffer size to perform sorting, and a positive integer d as the size during the merging.

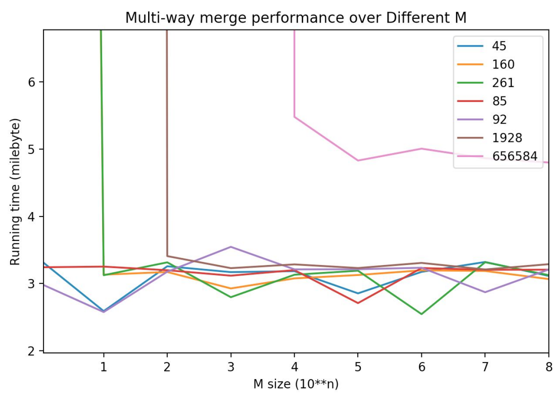
The estimated cost formula can be recalled from Lecture 6. During the sorting process, f is loaded by M and sorted each time, which will result in N/M sublists. After that, we further load and sort the N/M sublists based on M capacity. The sorting process continues until a single sorted queue is generated. The cost for the sorting part can then be expressed as $C= 2N [\log_M N]$. If the M is set big enough, the last sorting process can be avoided, which lead to an optimized cost formula as $C_{sort} = 2N [(\log_M N)- 1]$. In addition, we need to merge the separate files based on the sorted references. The merging process can be summarized as $M*d$. Therefore, the estimated cost formula can be viewed as: $\boldsymbol{C = 2N [(\log_M N)- 1] + M*d}$.

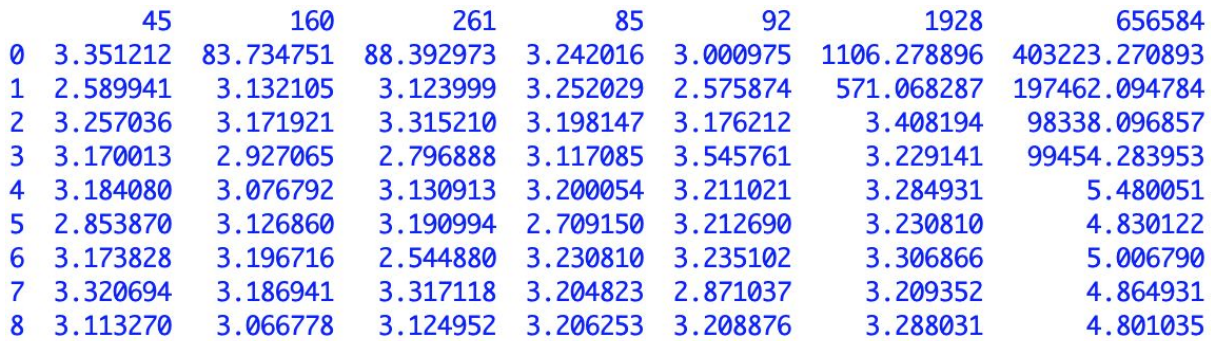
For expectation, the f size (i.e. N) is vital given the cost formula. We may anticipate the performance will decrease with the increase of the file size. Besides, M is believed to be bigger as to better perform the sorting task. d may also influence the performance, which is expected to be negatively related to the merge.

7.2. Experimental Results & Discussions

We first set d fixed in order to examine the impact of M on different files. However, given the system limitations on memory capacity, we may not be able to perform on larger files.



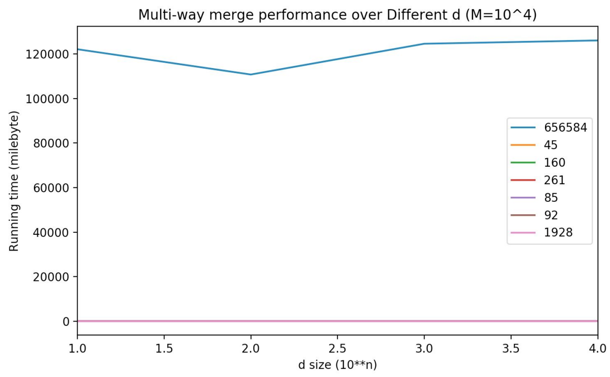


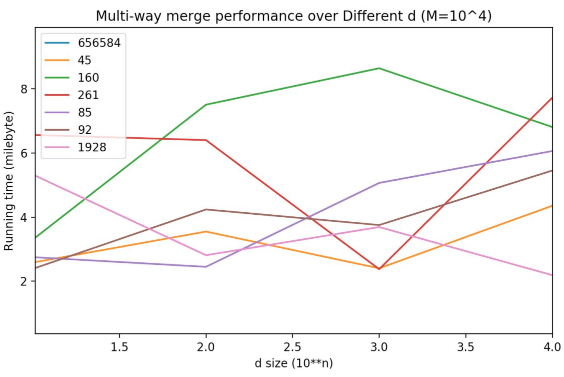


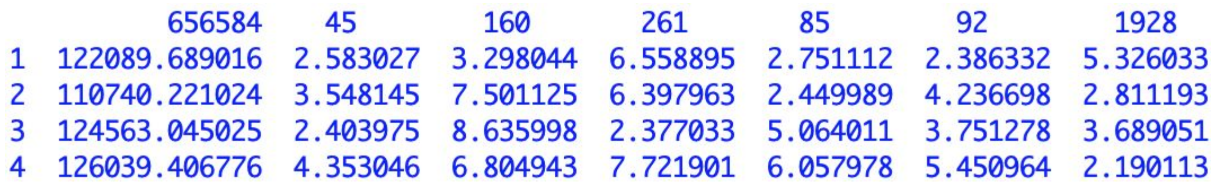
```
         45         160         261        85        92         1928         656584
0  3.351212   83.734751   88.392973  3.242016  3.000975  1106.278896  403223.270893
1  2.589941    3.132105    3.123999  3.252029  2.575874   571.068287  197462.094784
2  3.257036    3.171921    3.315210  3.198147  3.176212     3.408194   98338.096857
3  3.170013    2.927065    2.796888  3.117085  3.545761     3.229141   99454.283953
4  3.184080    3.076792    3.130913  3.200054  3.211021     3.284931       5.480051
5  2.853870    3.126860    3.190994  2.709150  3.212690     3.230810       4.830122
6  3.173828    3.196716    2.544880  3.230810  3.235102     3.306866       5.006790
7  3.320694    3.186941    3.317118  3.204823  2.871037     3.209352       4.864931
8  3.113270    3.066778    3.124952  3.206253  3.208876     3.288031       4.801035
```

Figure 7.2.1. Multi-way merge performance over different M size (d=M)

As shown in Figure 7.2.1, it is obvious that the *M* size does not influence the performance much after reaching the optimal value for each file. The optimal buffer size grows as the file size increases. It is suspected that the advantages of a bigger buffer may be offset by the merge and write phase, which therefore leads to an optimal value.







```
         656584        45       160       261        85        92      1928
1  122089.689016  2.583027  3.298044  6.558895  2.751112  2.386332  5.326033
2  110740.221024  3.548145  7.501125  6.397963  2.449989  4.236698  2.811193
3  124563.045025  2.403975  8.635998  2.377033  5.064011  3.751278  3.689051
4  126039.406776  4.353046  6.804943  7.721901  6.057978  5.450964  2.190113
```

Figure 7.2.2. Multi-way merge performance over different *d* size (M=10^4)

Secondly, we set *M* to 10^4 as the optimal value and examine further on the impact of *d.* Surprisingly, the performance over *d* varies for each file, but within a certain range given the level of file size. It can be speculated that after setting the optimal buffer size (i.e. *M*), *d* may not play a significant role as *M.*

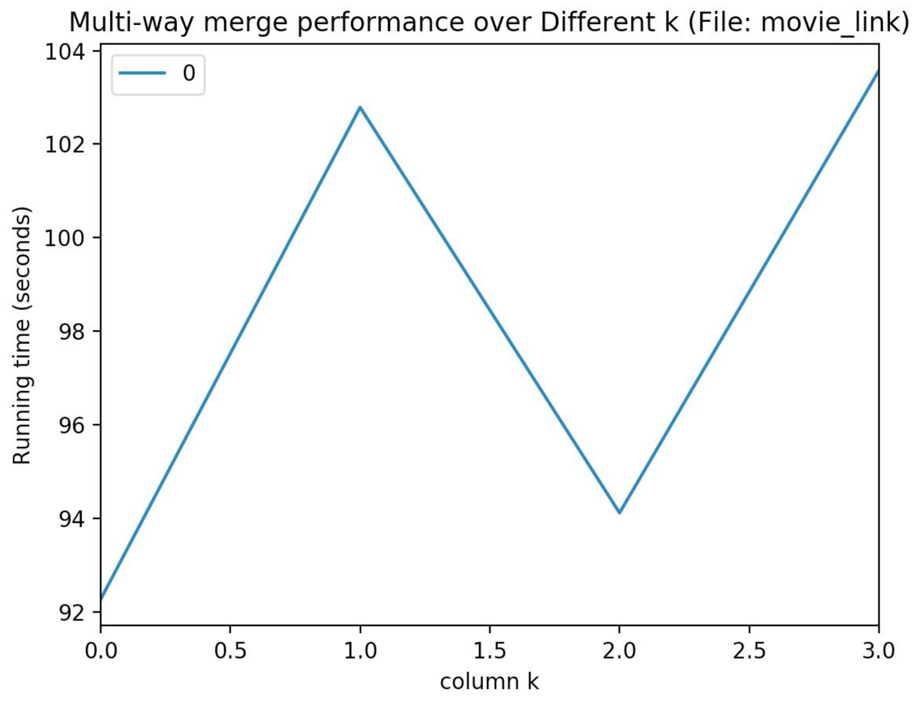


Figure 7.2.3. Figure 7.2.2. Multi-way merge performance over different *k* column (*movie_link.csv*)

Last but not least, we test whether the selected sorting column *k* will affect the performance based on a single file. As shown in Figure 7.2.3, the performance fluctuates over different column selection but within a reasonable range. Given the fact that column[0] is a pre-defined index (which is sorted), it can be viewed that the pre-sorted column does not affect the performance much. It can be concluded that the sorting criteria do not affect much in the merge-sort process since the *N, M, d* are the main factors in our estimated formula.

In general, for this experiment, we may conclude that *N, M, d* are the dominant factors affecting the performance. With the increase of file size *N* level, the execution time rises sharply. For a certain magnitude of *N*, there may exist an optimal buffer size *M* to minimize the overall performance.

8. Conclusion

This project provides an opportunity to reflect book knowledge and earn hands-on experience over the secondary memory operations. Given the scarce resource of memory in the modern era, it is vital for database administrators to determine the optimal methods to access files wisely and effectively. In this report, we present four different read/write methods and perform a series of experiments in optimizing the merge-sort algorithm in practice. To be specific, this work has identified that the combination of default buffer reading (method 2) and memory-mapped buffer writing (method 4) outperform the rest options. For the merge-sort algorithm, we believe the file size and the buffer size matter to the overall performance.

However, being limited to the testing environment and time constraints, this report lacks certain experiments over the large-size files and therefore the comparison is absent. The attached programming codes may sometimes seem redundant and over-simplified. Despite its preliminary results, this study offers some insight into the basic-level operations and further investigation and experimentation into buffer setting is strongly recommended.

Reference

1. Overview of Memory-Mapping- MATLAB & Simulink. (2020). Retrieved from https://www.mathworks.com/help/matlab/import_export/overview-of-memory-mapping.html
2. What is memory mapped I/O? - Quora. (2020). Retrieved from https://www.quora.com/What-is-memory-mapped-I-O
3. How does memory mapping a file have significant performance increases over the standard I/O system calls?. (2020). Retrieved from https://unix.stackexchange.com/questions/474926/how-does-memory-mapping-a-file-have-significant-performance-increases-over-the-s
4. MappedByteBuffer (Java Platform SE 7). (2020). Retrieved from https://docs.oracle.com/javase/7/docs/api/java/nio/MappedByteBuffer.html