ULB UNIVERSITÉ LIBRE DE BRUXELLES

BDMA

# Algorithms in Secondary Memory

Ariston Lim (000491325)
Hung Nguyen (000493176)
Julio Candela (000489735)

## Table of Contents

# I. Introduction and Environment

The objective of the report is to experiment with different methods of external memory algorithms and identify which of them has the best performance in different scenarios. The experiments consists on the following:

- Four implementations of read line and write line (By Character, by line with default buffer size, buffering and memory mapping)
- Sequential Reading
- Random Reading
- Sequential Reading and writing
- Merge Sort Algorithm

## 1. Machine Environment

- Operating System: Windows
- CPU: Core i7
- Processor: 2.3Ghz
- Memory: 12GB
- Hard Disk: 1 TB

## 2. Programming Language and Libraries

The programming language chosen for the implementation is Java with jre 1.8. Previously, Python was tried in the experiments as an alternative, but the performance was from 5 to 10 times slower compared to Java's implementation. As a consequence, Java seems to perform more efficiently for input and output streams which are the core of these experiments.

The Java project is composed by four packages:

- Dsa.read: Contains the four implementations of read streaming with methods for opening, readline, close, validate end of line and seek) in addition to the reader interface.
- Dsa.write: Contains the four implementations of write streaming with methods for opening, readline and close) in addition to the reader interface.
- Experiments: Intermediate class to perform the different experiments with each implementation: Experiments_Read (Sequential and Random Jump), Experiments_Write (Combined Streams) and Experiments_MergeSort (Multi-way merge)
- Main: Perform and measure the time of each experiment (sequential, random, combined and merge sort)

Additionally, the following external jars were added to the project:

- Apache-commons-codec-1.4.jar: Validate if the content of the write file is equal to the original file
- Commons-io-2.6.jar: Clean directory after external sort.

## 3. Experimentation details

In order to measure the time's performance in different datasets, csv files of wide range sizes were provided. These datasets range from 3.7 KB to 1.3 GB and represent relational tables with IMDB information about films, television, and others. For the purpose of the report, we have chosen 5 of them which represent very small, small, medium, large and very large files as depicted in the following table:

| **Name Files** | **Size (Bytes)** | **Number of columns** | **Category** |
|---|---|---|---|
| movie_link.csv | 656,584 bytes | 4 | Very Small |
| keyword.csv | 3,791,536 bytes | 3 | Small |
| aka_name.csv | 73,004,383 bytes | 8 | Medium |
| person_info.csv | 399,133,124 bytes | 5 | Large |
| cast_info.csv | 1,418,137,141 bytes | 7 | Very Large |

As part of the benchmarking among the four implementations and further experiments, the former 5 csv files were used for each experiment. In the case of the buffered and memory map read/write line implementations, different buffer sizes were tested. For each of these experiments, 5 simulations were run in order to get the average time excluding the max and min results (a consistent measure of the time of each experiment). The general details of the first 3 experiments are described below:

| **Stream type** | **Num Simulations** | **Files** | **Buffer Size Tuning (B)** | **Random exp.2 (j)** |
|---|---|---|---|---|
| Character | 5 | 5 csv | 1 | 10,50,100,500,<br>1000,1500,2000 |
| Default Line | 5 | 5 csv | 8192 (default by jdk) | |

| | | | | |
|---|---|---|---|---|
| Buffered | 5 | 5 csv | 10,128,1 024,4 096,<br>8 192,16 384, 65 536,<br>524 288, 4 194 304,<br>10 000 000 | |
| Memory Map | 5 | 5 csv | | |

The average time formula used is the following:

$$Consistent\ Average\ Time\ (CAT)\ =\ \frac{(\Sigma\ (times)\ -min-max)}{numsimulations-2}$$

In case of the external sort, the experiment adds three new parameters which are going to be tested using the best implementation which results from the previous experiments:

- Buffer Size (M): Max number of bytes that can be stored in memory
- Column number (k): num of the column which will be sorted
- Buffers to merge (d): Number of buffers which can be merged and sorted at the same time.

Additionally, the files will be tested in the scenario in which they are already ordered to see whether or not the performance varies. These files appear with the suffix "_Ord". The number of simulations to get an average time and the time formula remains the same.

| Name Files | Simulations | Col order(k) | Buffer Size(M) | Buffers to merge(d) |
|---|---|---|---|---|
| movie_link.csv | 5 | 1 | 50 000,<br>100 000,<br>500 000,<br>1 000 000,<br>5 000 000 | 10,25,50,100,200 |
| keyword.csv | 5 | 1 | | |
| aka_name.csv | 5 | 1 | | |
| movie_link_Ord.csv | 5 | 1 | | |
| keyword_Ord.csv | 5 | 1 | | |
| aka_name_Ord.csv | 5 | 1 | | |

# II. Observations on reading and writing

## 1. Implementation

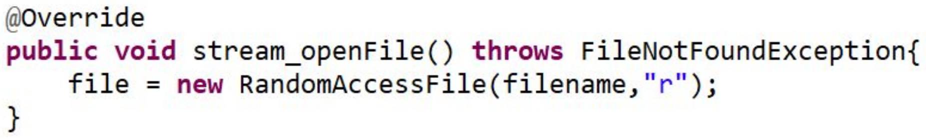
In this part, different mechanisms for reading and writing need to be first implemented. From the requirement, these mechanisms should read/write line by line from/into external files without loading the whole file into main memory.  There are many ways to implement stream reading and writing, but in this experiment, only these following types are implemented.

### a. Reading/Writing one character at a time

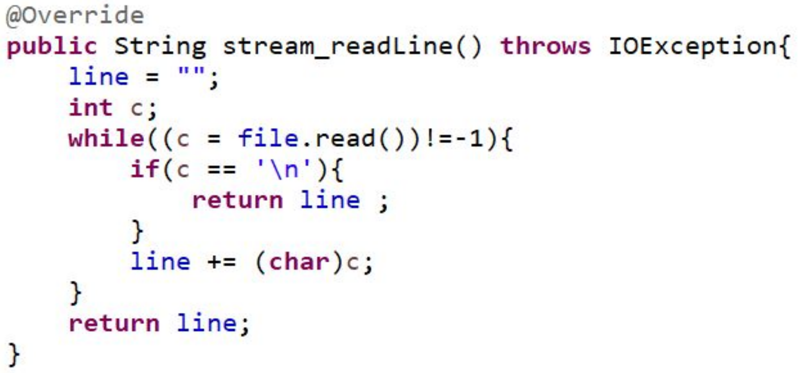
**Reading**

For this purpose, a *FileReader* instance can be used to create a stream connected to an input file. However, a *FileReader* instance does not have a *seek()* method which is needed for the experiment 1.2, so we decided to use an instance of *RandomAccessFile.* This instance treats the input file like a large array of bytes and provides a file pointer which acts like a cursor. The method *stream_openFile()* is implemented to create this *RandomAccessFile* instance.

The next bytes will be read from this stream by calling the *read()* method. These bytes keep passing through this stream and being stored in a String called *line* until an end of line character is found (these bytes need to be cast as character before storing). The variable *line* now contains a completed line from the input files. The method *stream_readLine()* returns this value of the variable *line.*



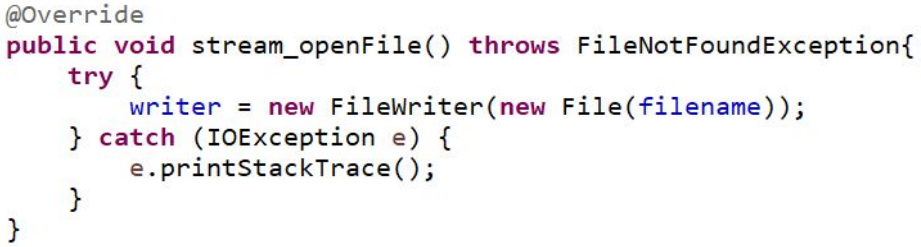
```
@Override
public void stream_openFile() throws FileNotFoundException{
    file = new RandomAccessFile(filename,"r");
}
```



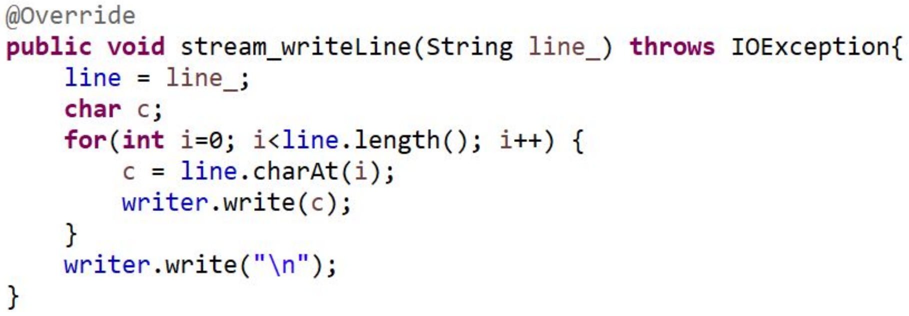
```
@Override
public String stream_readLine() throws IOException{
    line = "";
    int c;
    while((c = file.read())!=-1){
        if(c == '\n'){
            return line ;
        }
        line += (char)c;
    }
    return line;
}
```

### Writing

For writing by character, we use an instance of *FileWriter* class. This stream can be opened by calling method *stream_openFile().* The variable *line_* is the content of the next line that needs to be written to the output file. Each character from the input line is written to the writing stream by iterating over the input line. An end of line character is written at the end of the line. After calling this *stream_writeLine()* method, the input line would be written completely to the output file



```
@Override
public void stream_openFile() throws FileNotFoundException{
    try {
        writer = new FileWriter(new File(filename));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

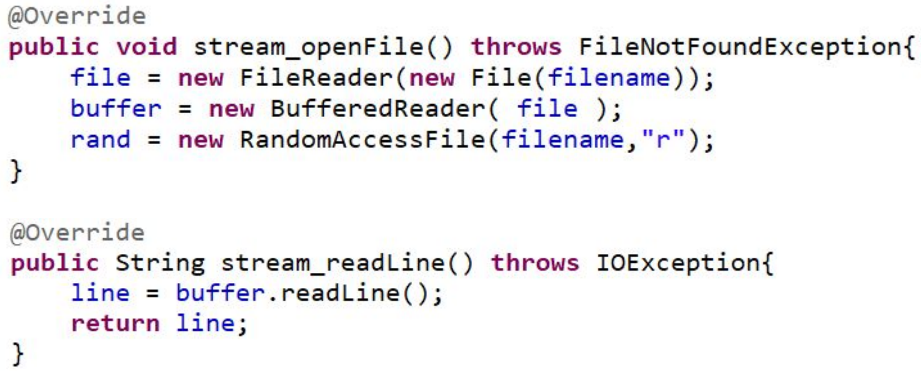


```
@Override
public void stream_writeLine(String line_) throws IOException{
    line = line_;
    char c;
    for(int i=0; i<line.length(); i++) {
        c = line.charAt(i);
        writer.write(c);
    }
    writer.write("\n");
}
```

## b. Reading/Writing using buffering mechanism

### Reading

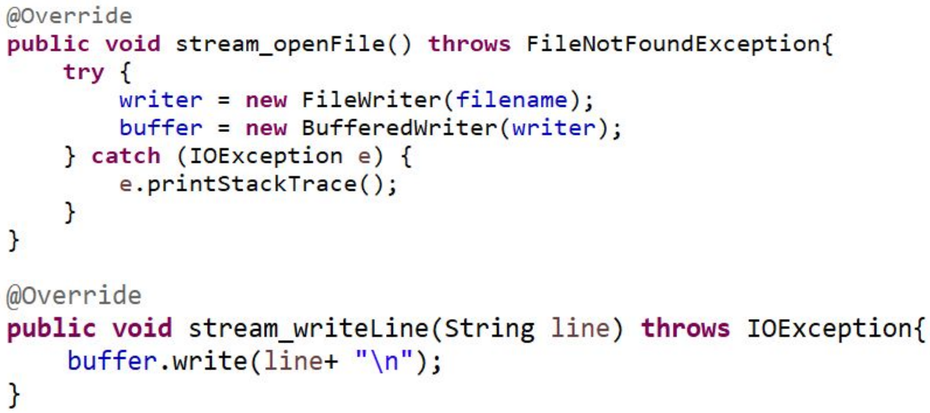
In this experiment, the reading stream is equipped with a buffering mechanism. First, we create an instance of *FileReader* and then wrap an instance of *BufferedReader* class around the *FileReader* instance, which would support buffering. Without buffering, each call of *readLine()* could cause bytes to be read from the physical disk, which can be very inefficient. The default buffer size is 8192 bytes, which is large enough for most purposes.



```
@Override
public void stream_openFile() throws FileNotFoundException{
    file = new FileReader(new File(filename));
    buffer = new BufferedReader( file );
    rand = new RandomAccessFile(filename,"r");
}

@Override
public String stream_readLine() throws IOException{
    line = buffer.readLine();
    return line;
}
```

**Writing**

Implementation of this writing mechanism is similar to the above reading mechanism. *The BufferedWrite* instance is used to wrap outside of the *FileWriter* instance to support buffering. The input line is written to the output file by calling the *write()* method of the *BufferedWriter* instance.

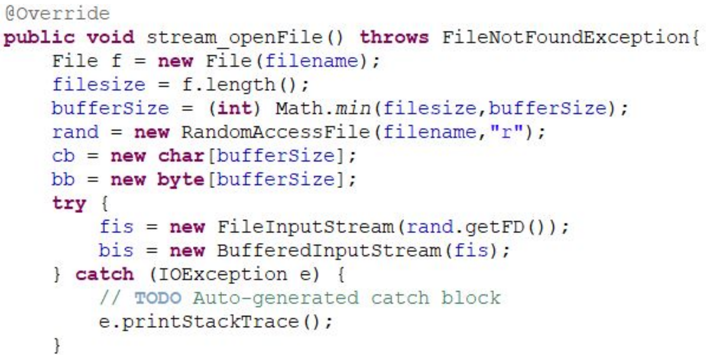


```
@Override
public void stream_openFile() throws FileNotFoundException{
    try {
        writer = new FileWriter(filename);
        buffer = new BufferedWriter(writer);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@Override
public void stream_writeLine(String line) throws IOException{
    buffer.write(line+ "\n");
}
```

**c. Reading/Writing with a buffer of size B in internal memory**

**Reading**

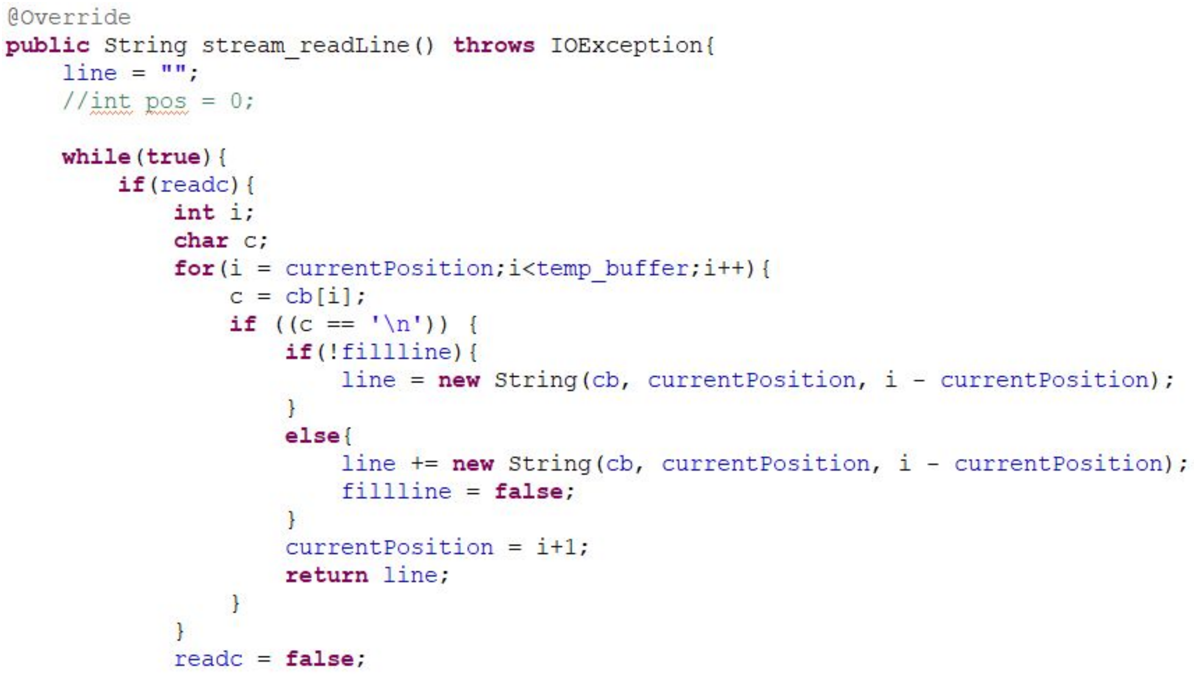
The implementation of the buffered reading streaming requires the following I/O java functions: *RandomAccessFile* (in order to perform the seek operation faster), *FileInputStream* which gets the information of the random access file and the *BufferedInputStream* which will perform as the reader. Additionally, it’s necessary to create temporal arrays for storing the current buffer in memory. This initialization is performed in the function *stream_openFile().*



```
@Override
public void stream_openFile() throws FileNotFoundException{
    File f = new File(filename);
    filesize = f.length();
    bufferSize = (int) Math.min(filesize,bufferSize);
    rand = new RandomAccessFile(filename,"r");
    cb = new char[bufferSize];
    bb = new byte[bufferSize];
    try {
        fis = new FileInputStream(rand.getFD());
        bis = new BufferedInputStream(fis);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
```

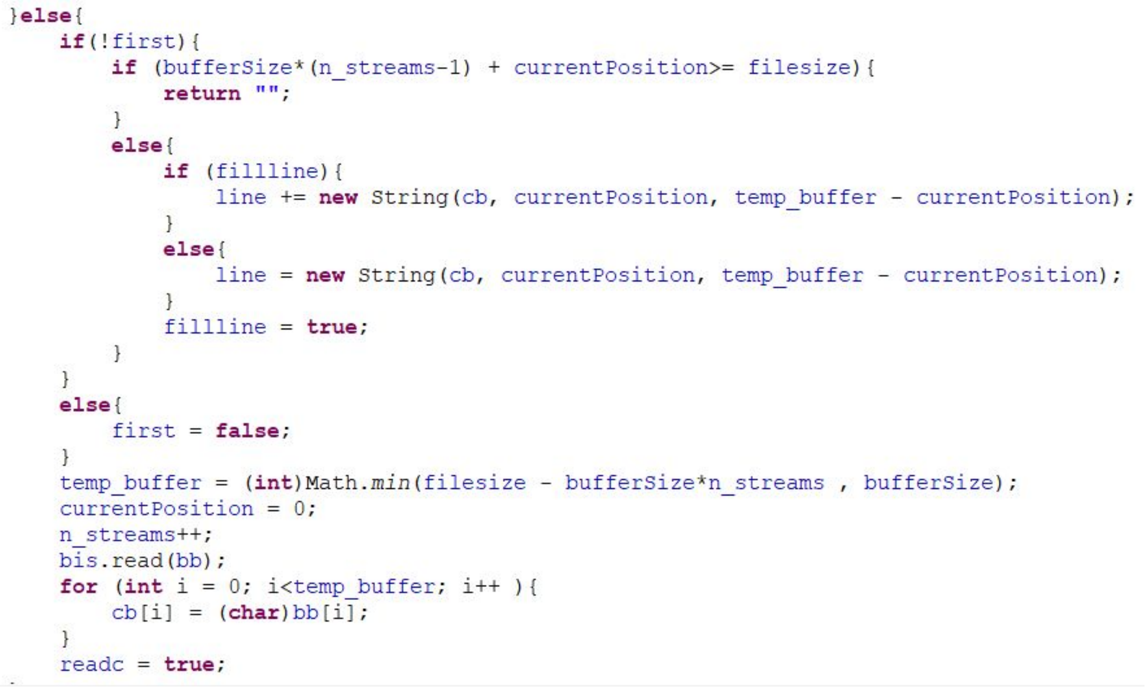
The stream_readline consists of reading lines until the buffer is empty or there is an end of file. For this purpose, some scenarios have been managed. First of all, the char array is swept from the current position (relative to the buffer size) until an end of line is found. When the char array has been swept completely and there is no and end of line, the partial line is stored and the function calls for the BufferedInputStream to read a new buffer and store it in the char array. Finally, the line is filled and completed. The process repeats until it reaches the size of the file.

- Read Line until end of file or end of line is found



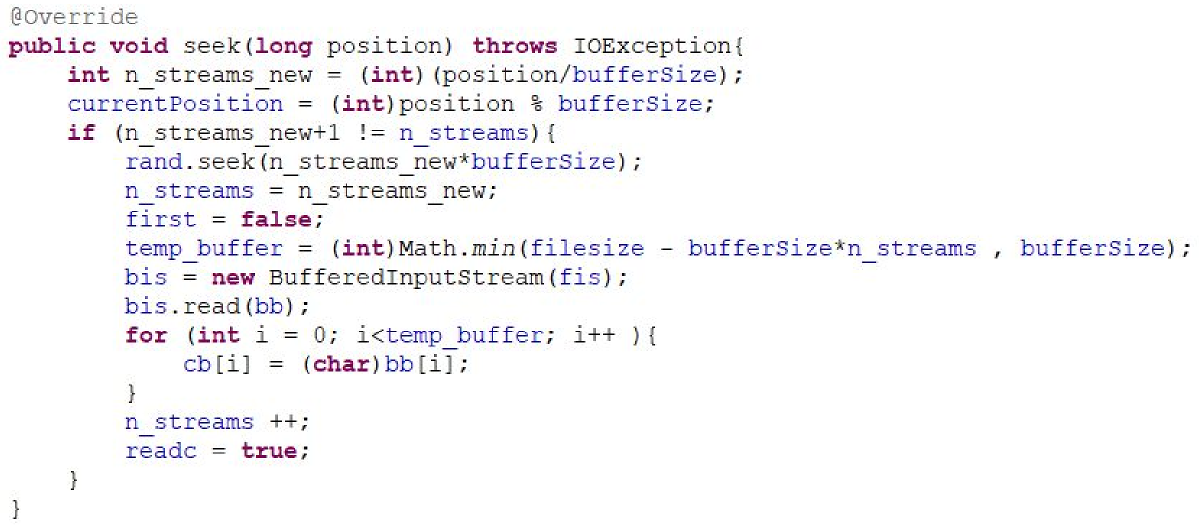
```
@Override
public String stream_readLine() throws IOException{
    line = "";
    //int pos = 0;

    while(true){
        if(readc){
            int i;
            char c;
            for(i = currentPosition;i<temp_buffer;i++){
                c = cb[i];
                if ((c == '\n')) {
                    if(!fillline){
                        line = new String(cb, currentPosition, i - currentPosition);
                    }
                    else{
                        line += new String(cb, currentPosition, i - currentPosition);
                        fillline = false;
                    }
                    currentPosition = i+1;
                    return line;
                }
            }
            readc = false;
```

- Fill the buffer with the next buffer size bytes



```
}else{
    if(!first){
        if (bufferSize*(n_streams-1) + currentPosition>= filesize){
            return "";
        }
        else{
            if (fillline){
                line += new String(cb, currentPosition, temp_buffer - currentPosition);
            }
            else{
                line = new String(cb, currentPosition, temp_buffer - currentPosition);
            }
            fillline = true;
        }
    }
    else{
        first = false;
    }
    temp_buffer = (int)Math.min(filesize - bufferSize*n_streams , bufferSize);
    currentPosition = 0;
    n_streams++;
    bis.read(bb);
    for (int i = 0; i<temp_buffer; i++ ){
        cb[i] = (char)bb[i];
    }
    readc = true;
```

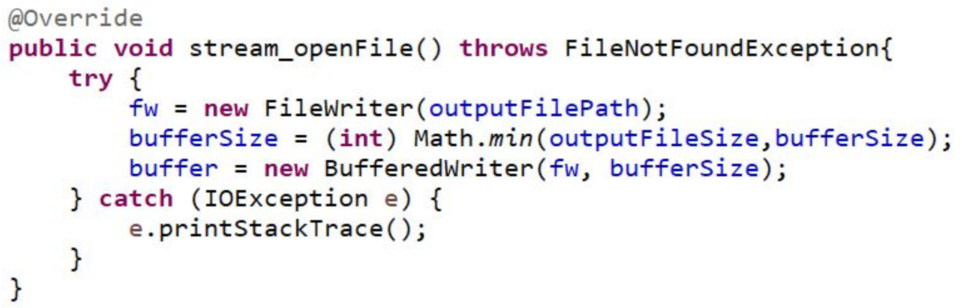
Additionally, it’s necessary a seek function for experiment 1.2 which checks if the new stream belongs to the current stream. If it’s true, it’s not necessary to load a new buffer, only to update the current position relative to the buffer size; otherwise, a new buffer is loaded starting from position with n_stream and the current position relative to the buffer size.



```
@Override
public void seek(long position) throws IOException{
    int n_streams_new = (int)(position/bufferSize);
    currentPosition = (int)position % bufferSize;
    if (n_streams_new+1 != n_streams){
        rand.seek(n_streams_new*bufferSize);
        n_streams = n_streams_new;
        first = false;
        temp_buffer = (int)Math.min(filesize - bufferSize*n_streams , bufferSize);
        bis = new BufferedInputStream(fis);
        bis.read(bb);
        for (int i = 0; i<temp_buffer; i++ ){
            cb[i] = (char)bb[i];
        }
        n_streams ++;
        readc = true;
    }
}
```

### Writing

This writing implementation is similar to the previous writing one (*Line Writing*) except that the buffer size here is specified. By testing a wide range of different values for buffer size we can choose the best one.



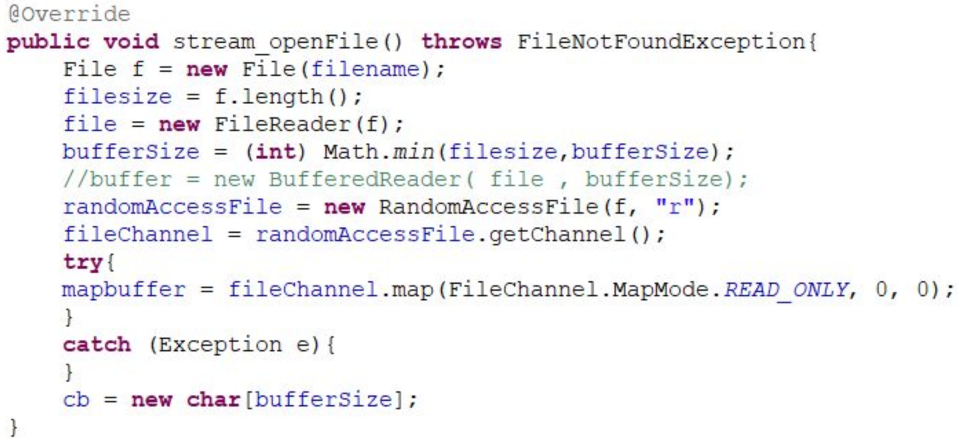
```
@Override
public void stream_openFile() throws FileNotFoundException{
    try {
        fw = new FileWriter(outputFilePath);
        bufferSize = (int) Math.min(outputFileSize,bufferSize);
        buffer = new BufferedWriter(fw, bufferSize);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## d. Reading/Writing using Memory Mapping

Memory mapping allows accessing file data directly from main memory, which is achieved by mapping byte-for-byte the whole file or a portion of file into memory [2]. The correlation between the memory and the file allows applications to treat the mapped portion as if it were primary memory. The application accessing the file only needs to deal with memory and the operating system takes care of loading the requested contents and writing changes into file. Java supports memory mapping with the *java.nio* package [3].

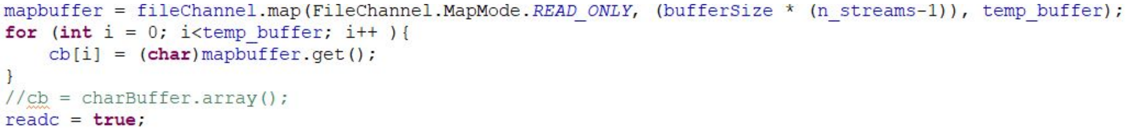
**Reading**

The implementation of the memory mapping reading streaming requires the following I/O java functions: *RandomAccessFile* (in order to get the channel for virtual memory access with *FileChannel)* and the *MappedByteBuffer* which will perform the virtual memory reader. Additionally, it's necessary to create temporal arrays for storing the current buffer in memory. This initialization is performed in the function *stream_openFile().*



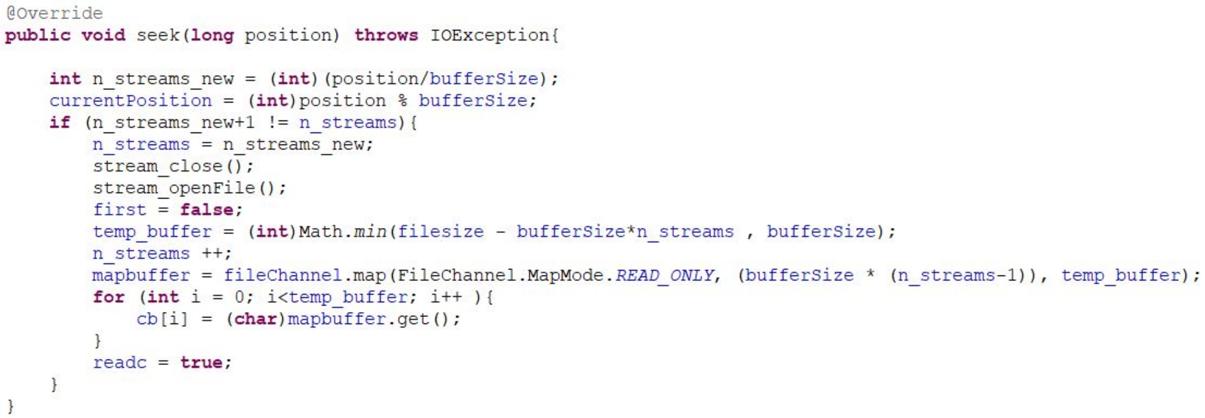
```
@Override
public void stream_openFile() throws FileNotFoundException{
    File f = new File(filename);
    filesize = f.length();
    file = new FileReader(f);
    bufferSize = (int) Math.min(filesize,bufferSize);
    //buffer = new BufferedReader( file , bufferSize);
    randomAccessFile = new RandomAccessFile(f, "r");
    fileChannel = randomAccessFile.getChannel();
    try{
    mapbuffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, 0, 0);
    }
    catch (Exception e){
    }
    cb = new char[bufferSize];
}
```

The implementation of the stream_readline is similar to the buffered reading. The only difference is the way to fill the char array after calling the map function.



```
mapbuffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, (bufferSize * (n_streams-1)), temp_buffer);
for (int i = 0; i<temp_buffer; i++ ){
    cb[i] = (char)mapbuffer.get();
}
//cb = charBuffer.array();
readc = true;
```

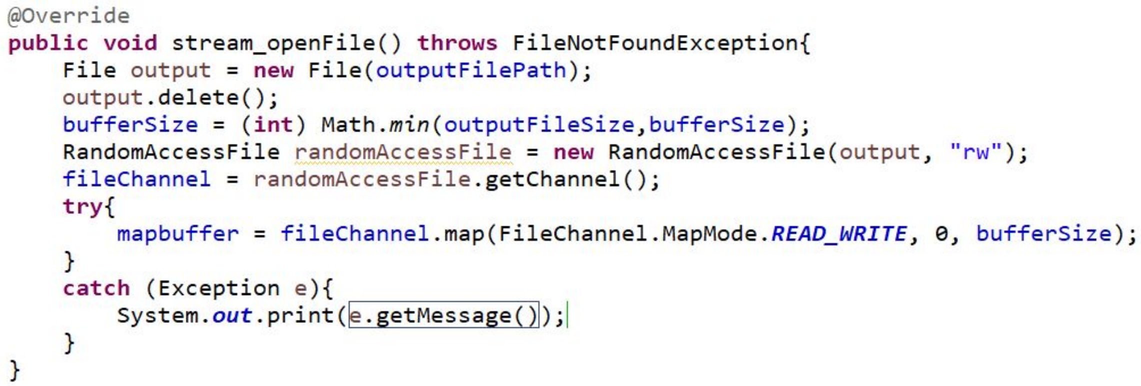
The seek implementation is performed directly with the offset and the buffer size of the map function only when the new position belongs to a different stream. The current position is also updated relative to the buffer size



```
@Override
public void seek(long position) throws IOException{

    int n_streams_new = (int)(position/bufferSize);
    currentPosition = (int)position % bufferSize;
    if (n_streams_new+1 != n_streams){
        n_streams = n_streams_new;
        stream_close();
        stream_openFile();
        first = false;
        temp_buffer = (int)Math.min(filesize - bufferSize*n_streams , bufferSize);
        n_streams ++;
        mapbuffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, (bufferSize * (n_streams-1)), temp_buffer);
        for (int i = 0; i<temp_buffer; i++ ){
            cb[i] = (char)mapbuffer.get();
        }
        readc = true;
    }
}
```

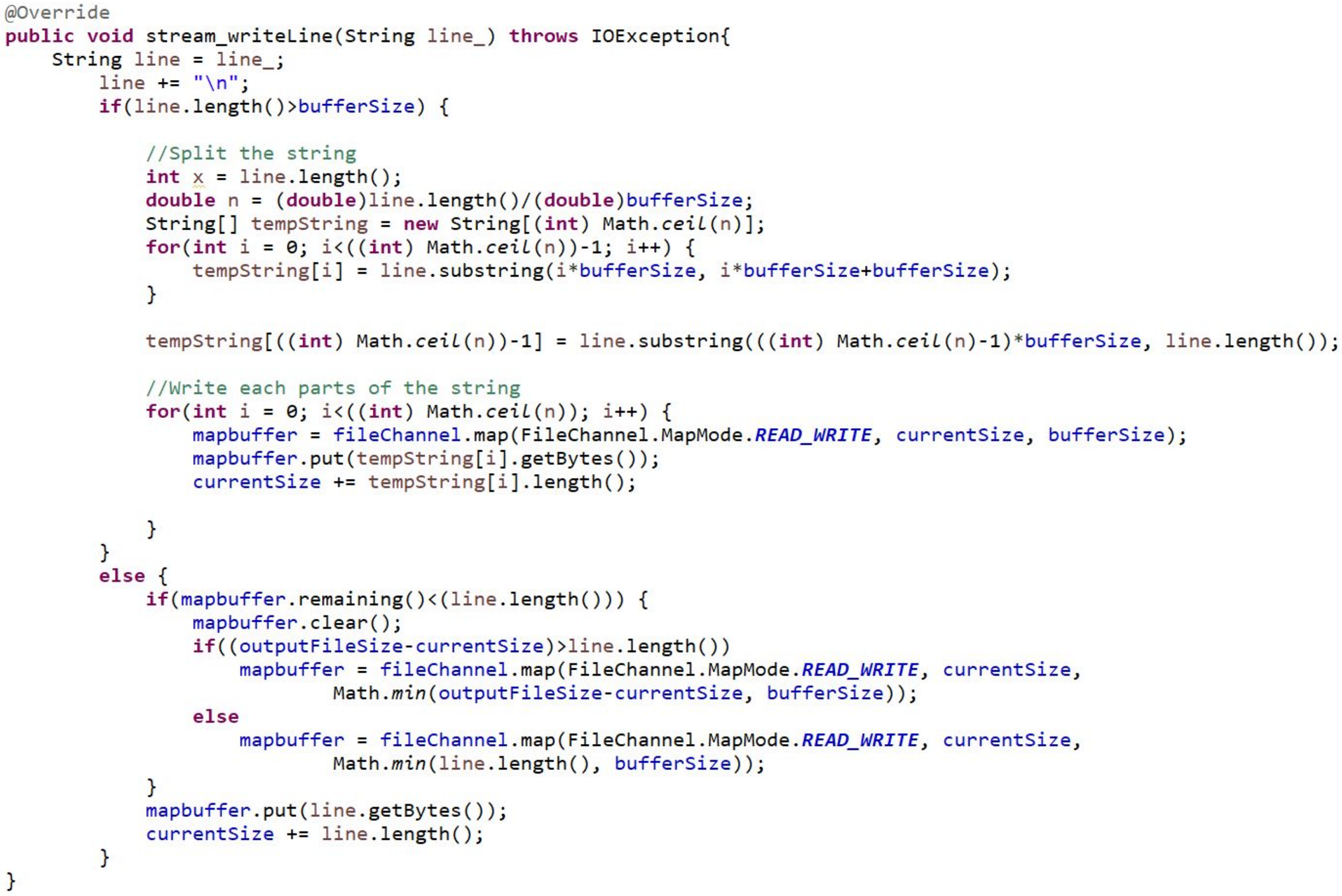
### Writing

*FileChannel* class provides a mechanism for memory mapping. We need to create an instance of *FileChannel* wrapping outside of a *RandomAccessFile* instance. The access mode is set to “rw” for reading and writing. The variable *mapbuffer*, which is an instance of the *MappedByteBuffer* class, represents the memory mapped writing stream. This can be achieved by calling the *map()* method from the *FileChannel* instance. MapMode is set to “*READ_WRITE*” and position is set to “0” which means we map from the beginning of the input file. The buffer size is specified by users. The following lines of code implement this stream.



```
@Override
public void stream_openFile() throws FileNotFoundException{
    File output = new File(outputFilePath);
    output.delete();
    bufferSize = (int) Math.min(outputFileSize,bufferSize);
    RandomAccessFile randomAccessFile = new RandomAccessFile(output, "rw");
    fileChannel = randomAccessFile.getChannel();
    try{
        mapbuffer = fileChannel.map(FileChannel.MapMode.READ_WRITE, 0, bufferSize);
    }
    catch (Exception e){
        System.out.print(e.getMessage());
    }
}
```

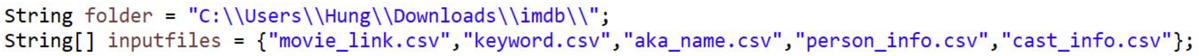
Next, we want to write to this stream line by line. This can be achieved by calling the *put()* method. During writing to this buffer, we need to create a new mapped buffer every time the buffer is full.



```java
@Override
public void stream_writeLine(String line_) throws IOException{
    String line = line_;
        line += "\n";
        if(line.length()>bufferSize) {

            //Split the string
            int x = line.length();
            double n = (double)line.length()/(double)bufferSize;
            String[] tempString = new String[(int) Math.ceil(n)];
            for(int i = 0; i<((int) Math.ceil(n))-1; i++) {
                tempString[i] = line.substring(i*bufferSize, i*bufferSize+bufferSize);
            }

            tempString[((int) Math.ceil(n))-1] = line.substring(((int) Math.ceil(n)-1)*bufferSize, line.length());

            //Write each parts of the string
            for(int i = 0; i<((int) Math.ceil(n)); i++) {
                mapbuffer = fileChannel.map(FileChannel.MapMode.READ_WRITE, currentSize, bufferSize);
                mapbuffer.put(tempString[i].getBytes());
                currentSize += tempString[i].length();

            }
        }
        else {
            if(mapbuffer.remaining()<(line.length())) {
                mapbuffer.clear();
                if((outputFileSize-currentSize)>line.length())
                    mapbuffer = fileChannel.map(FileChannel.MapMode.READ_WRITE, currentSize,
                            Math.min(outputFileSize-currentSize, bufferSize));
                else
                    mapbuffer = fileChannel.map(FileChannel.MapMode.READ_WRITE, currentSize,
                            Math.min(line.length(), bufferSize));
            }
            mapbuffer.put(line.getBytes());
            currentSize += line.length();
        }
}
```

## 2. Experimental observations regarding experiment 1.1

In this experiment, we are going to test sequential reading with four different implementations of reading and these following files are input files.

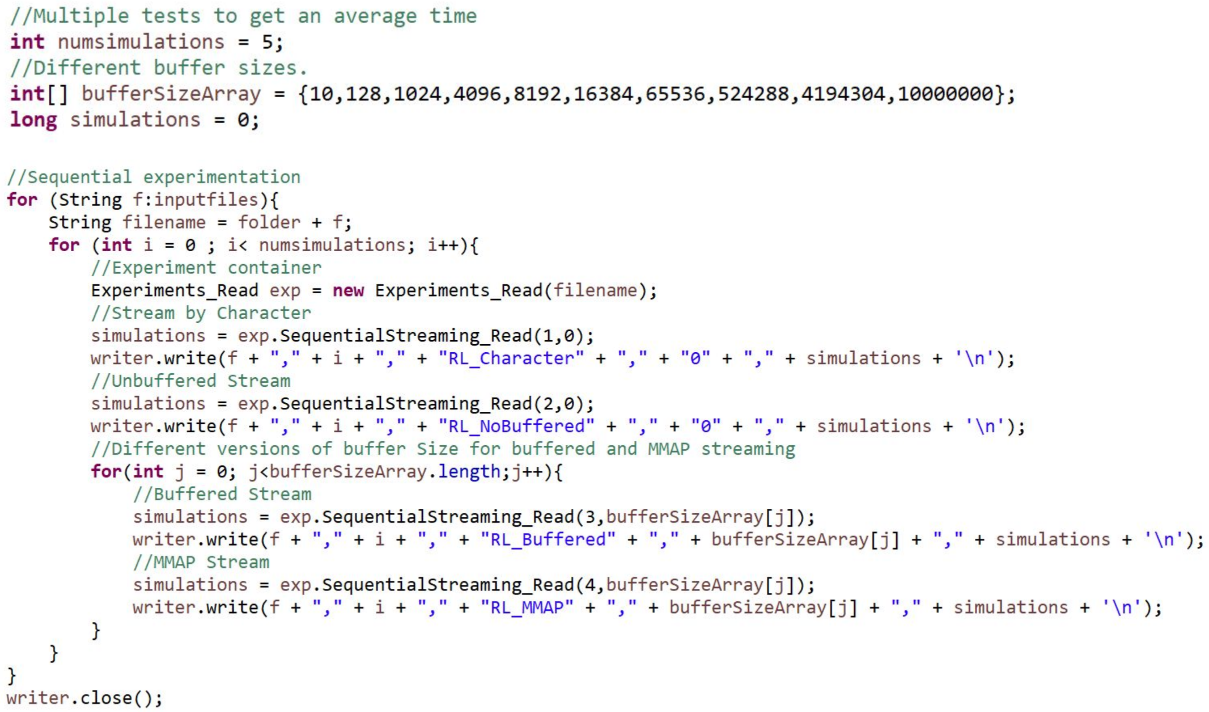


```java
String folder = "C:\\Users\\Hung\\Downloads\\imdb\\";
String[] inputfiles = {"movie_link.csv","keyword.csv","aka_name.csv","person_info.csv","cast_info.csv"};
```

We run the experiments in 5 iterations with different values of buffer size to find the optimal reading mechanism. Each implementation is tested as follows.

- *exp.SequentialStreaming_Read(1, 0):* Reading character by character
- *exp.SequentialStreaming_Read(2, 0):* Reading with default buffering mechanism
- *exp.SequentialStreaming_Read(3, bufferSizeArray[j]):* Reading with a specific buffer size

- *exp.SequentialStreaming_Read(4, bufferSizeArray[j]):* Reading using Memory Mapping with a specific buffer size



```java
//Multiple tests to get an average time
int numsimulations = 5;
//Different buffer sizes.
int[] bufferSizeArray = {10,128,1024,4096,8192,16384,65536,524288,4194304,10000000};
long simulations = 0;

//Sequential experimentation
for (String f:inputfiles){
    String filename = folder + f;
    for (int i = 0 ; i< numsimulations; i++){
        //Experiment container
        Experiments_Read exp = new Experiments_Read(filename);
        //Stream by Character
        simulations = exp.SequentialStreaming_Read(1,0);
        writer.write(f + "," + i + "," + "RL_Character" + "," + "0" + "," + simulations + '\n');
        //Unbuffered Stream
        simulations = exp.SequentialStreaming_Read(2,0);
        writer.write(f + "," + i + "," + "RL_NoBuffered" + "," + "0" + "," + simulations + '\n');
        //Different versions of buffer Size for buffered and MMAP streaming
        for(int j = 0; j<bufferSizeArray.length;j++){
            //Buffered Stream
            simulations = exp.SequentialStreaming_Read(3,bufferSizeArray[j]);
            writer.write(f + "," + i + "," + "RL_Buffered" + "," + bufferSizeArray[j] + "," + simulations + '\n');
            //MMAP Stream
            simulations = exp.SequentialStreaming_Read(4,bufferSizeArray[j]);
            writer.write(f + "," + i + "," + "RL_MMAP" + "," + bufferSizeArray[j] + "," + simulations + '\n');
        }
    }
}
writer.close();
```

## Expected Behavior

Character Streaming is expected to be the slowest mechanism since it needs to read the character from the physical disk every time we call the read*()* method. The cost formula for Character Streaming is as follows:

*I/Os cost estimation = N*

with N = file size

The Line Streaming with the default size of buffer (8192) would perform better than Character Streaming since it stores characters in a buffer for later use. There is a chance that the needed characters are already loaded in the memory so it does not require any I/Os. The cost formula for this mechanism is as follows:

*I/Os cost estimation = N / 8192*

with N = file size, 8192 is the default buffer size

The Buffered Streaming and the MMap Streaming are expected to have the highest performance when the buffer size is big enough. The cost formula for these two implementations is as follows:

$$\textit{I/Os cost estimation} = N / B$$

with N = file size, B = buffer size

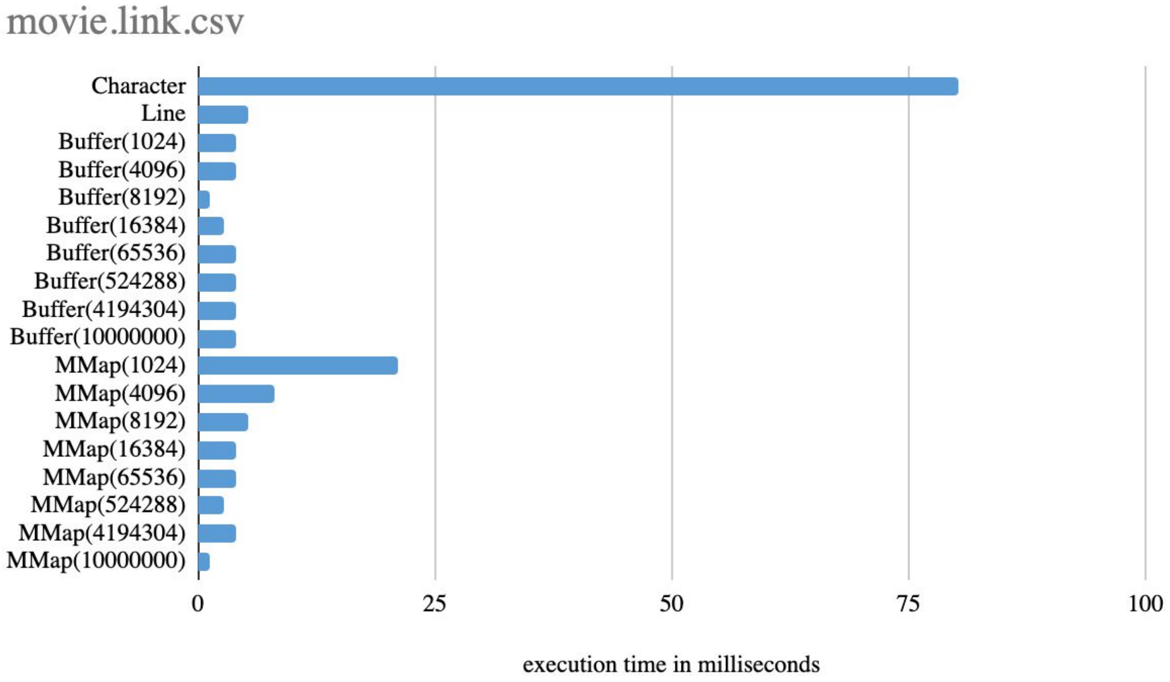
**Results**

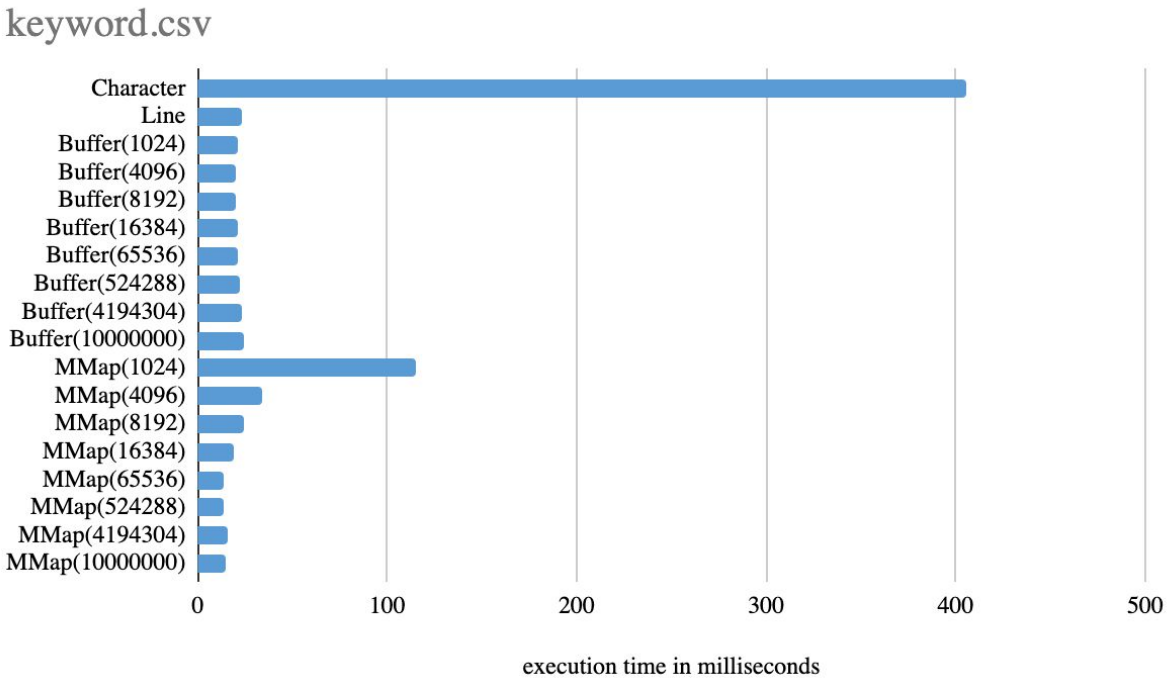
The results obtained confirmed the expected behavior described before. The observations can be divided according to the size of the files:

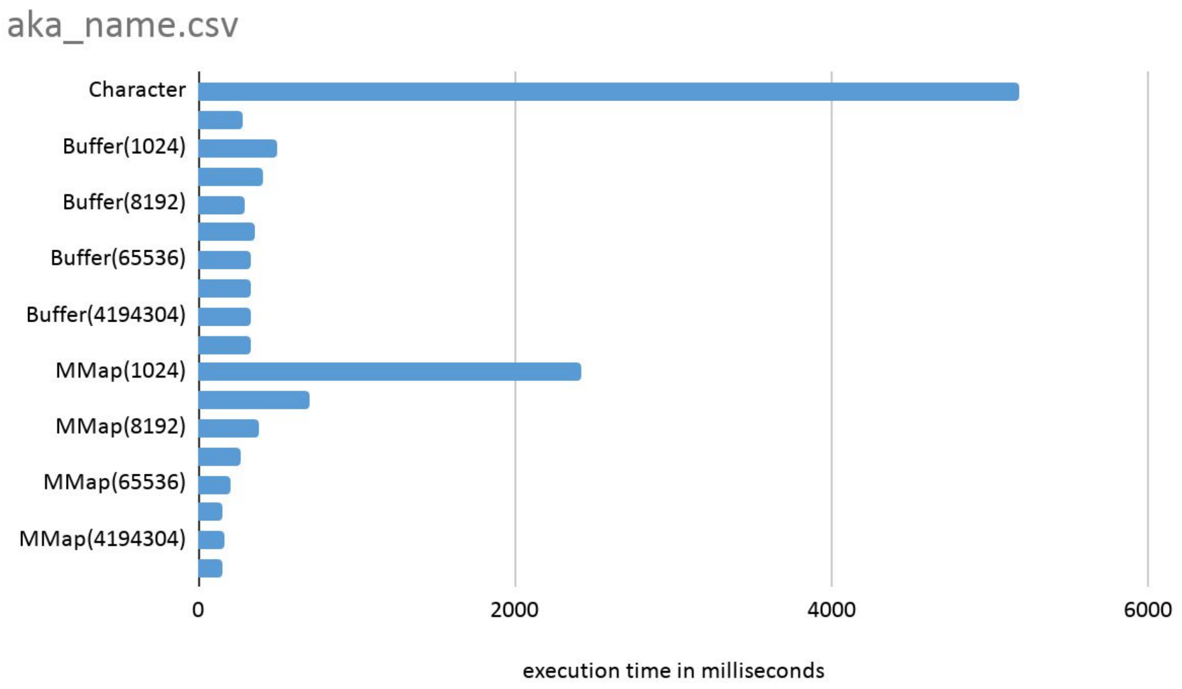
- Small files (movie_link.csv and keyword.csv): Buffered Reading and MMap reading with big buffer size have better performance than the other
- Medium and big files (aka_name.csv, person_info.csv, cast_info.csv): MMap reading outperform the other implementations, and the bigger buffer size means faster reading

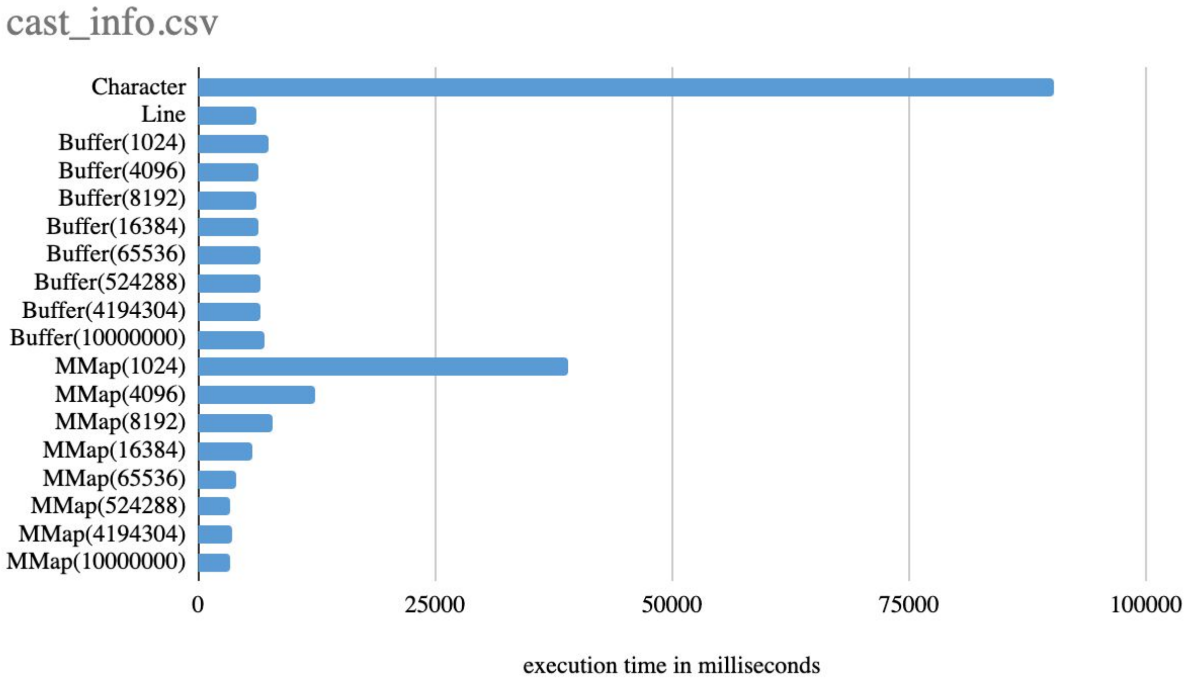
The best implementation for reading in this experiment is MMap with buffer size of 10000000 bytes. Character reading is the worst implementation for reading. For example, it took 121 seconds to read *person_info.csv* compared to the best one MMap with buffer size of 10000000 bytes which took only nearly 1 second.

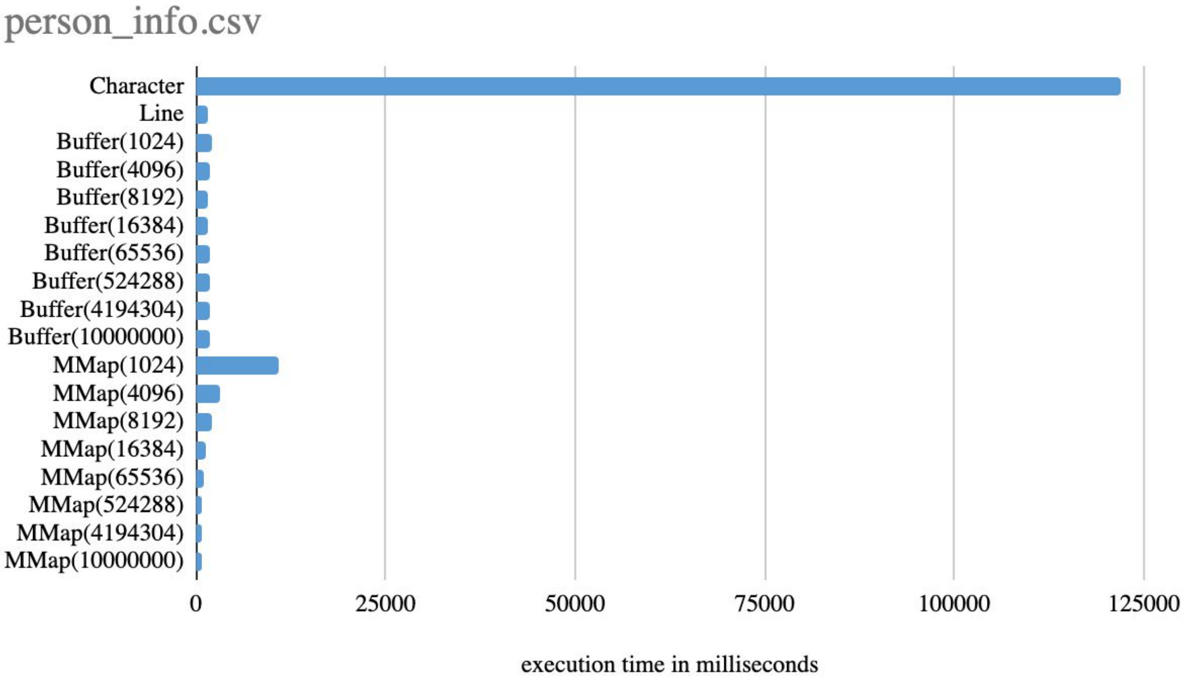
These following charts show the reading time on average (in milliseconds) for each file with different implementations.


movie.link.csv
Character
Line
Buffer(1024)
Buffer(4096)
Buffer(8192)
Buffer(16384)
Buffer(65536)
Buffer(524288)
Buffer(4194304)
Buffer(10000000)
MMap(1024)
MMap(4096)
MMap(8192)
MMap(16384)
MMap(65536)
MMap(524288)
MMap(4194304)
MMap(10000000)
0
25
50
75
100
execution time in milliseconds


keyword.csv
Character
Line
Buffer(1024)
Buffer(4096)
Buffer(8192)
Buffer(16384)
Buffer(65536)
Buffer(524288)
Buffer(4194304)
Buffer(10000000)
MMap(1024)
MMap(4096)
MMap(8192)
MMap(16384)
MMap(65536)
MMap(524288)
MMap(4194304)
MMap(10000000)
0
100
200
300
400
500
execution time in milliseconds


aka_name.csv
Character
Buffer(1024)
Buffer(8192)
Buffer(65536)
Buffer(4194304)
MMap(1024)
MMap(8192)
MMap(65536)
MMap(4194304)
0
2000
4000
6000
execution time in milliseconds


cast_info.csv
Character
Line
Buffer(1024)
Buffer(4096)
Buffer(8192)
Buffer(16384)
Buffer(65536)
Buffer(524288)
Buffer(4194304)
Buffer(10000000)
MMap(1024)
MMap(4096)
MMap(8192)
MMap(16384)
MMap(65536)
MMap(524288)
MMap(4194304)
MMap(10000000)
0
25000
50000
75000
100000
execution time in milliseconds

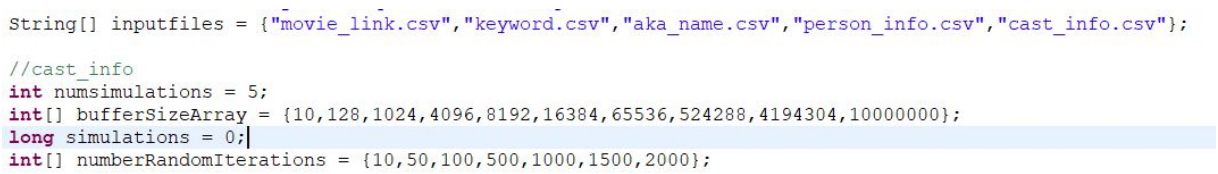


**Conclusions and Observations**

- The best implementation for sequential reading is MMAP with the highest buffer size. It has been shown that the performance improves when the buffer size increases at the maximum. Otherwise, the lowest the buffer size is, the implementation gets worse since it has to call many times to the mapping function.
- In the case of the buffered reading, the best performance is gotten with 8,192 in almost all cases which is the default value for the read line.
- The character streaming performs too many I/Os thereby increasing the execution time

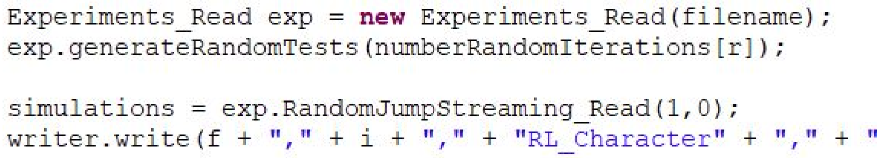
## 3. Experimental observations regarding experiment 1.2

The experiment 1.2, also called “randjump”, allows us to test the four implementations in a different scenario in which we are going to read lines from (j) random positions of the document. These random positions go from 0 to the filesize of the document and they are created at the beginning so that the implementations are tested with the same positions (as a warning, the relative positions can affect the performance). The parameters that are going to be tested are:



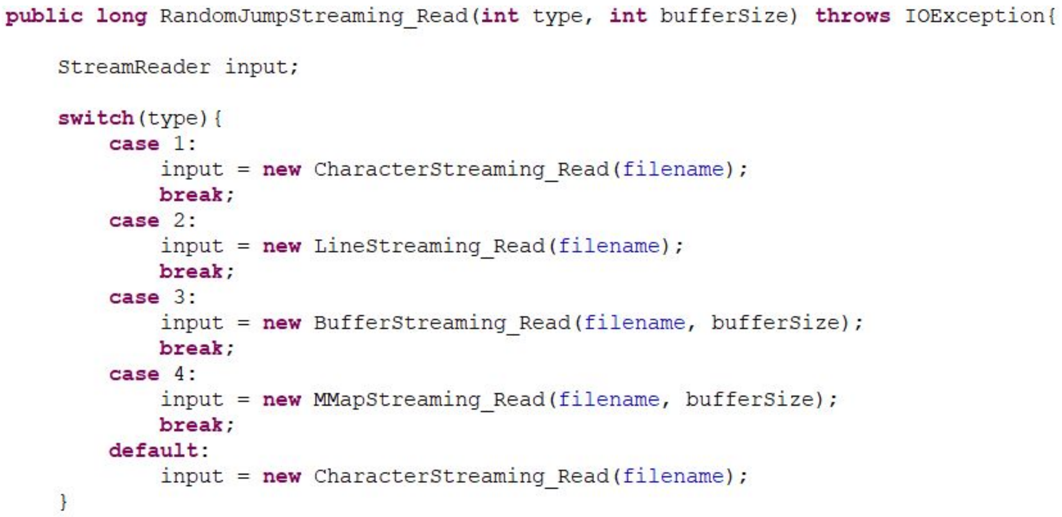
```
String[] inputfiles = {"movie_link.csv","keyword.csv","aka_name.csv","person_info.csv","cast_info.csv"};

//cast_info
int numsimulations = 5;
int[] bufferSizeArray = {10,128,1024,4096,8192,16384,65536,524288,4194304,10000000};
long simulations = 0;
int[] numberRandomIterations = {10,50,100,500,1000,1500,2000};
```

The Experiments_Read class contains the algorithm used for this experiment, so we invoke it using the file as a parameter. As explained before, the (j) random positions (r, in the code below) are generated by generateRandomTest in the first instance for a selected file in the class attribute “randomNumbers”. Finally, the experiment is run measuring the times for each implementation. The times are written in the results file.



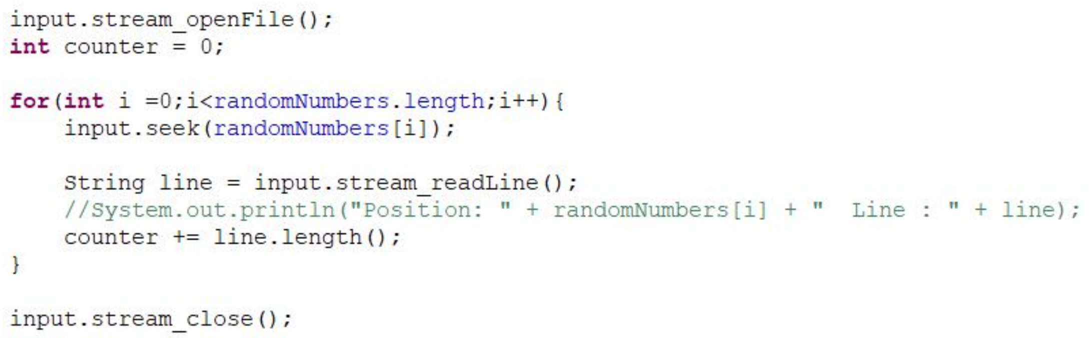
```
Experiments_Read exp = new Experiments_Read(filename);
exp.generateRandomTests(numberRandomIterations[r]);

simulations = exp.RandomJumpStreaming_Read(1,0);
writer.write(f + "," + i + "," + "RL_Character" + "," + "
```

The RandomJumpStreaming_Read firstly creates a reader interface according to each of the four implementations



```
public long RandomJumpStreaming_Read(int type, int bufferSize) throws IOException{

    StreamReader input;

    switch(type){
        case 1:
            input = new CharacterStreaming_Read(filename);
            break;
        case 2:
            input = new LineStreaming_Read(filename);
            break;
        case 3:
            input = new BufferStreaming_Read(filename, bufferSize);
            break;
        case 4:
            input = new MMapStreaming_Read(filename, bufferSize);
            break;
        default:
            input = new CharacterStreaming_Read(filename);
    }
```

After opening the selected file, for each random position in randomNumbers(array of positions to seek), it performs a seek operation, depicted before, and then reads until the next end of line. It stores the sum of the lengths of the partial lines for comparison and validation with others methods. At the end, it closes the stream



```
input.stream_openFile();
int counter = 0;

for(int i =0;i<randomNumbers.length;i++){
    input.seek(randomNumbers[i]);

    String line = input.stream_readLine();
    //System.out.println("Position: " + randomNumbers[i] + "  Line : " + line);
    counter += line.length();
}

input.stream_close();
```

### Expected Behavior

From the previous experiment, we realized that MMAP performs better in a sequential reading; however, we are going to read from different parts of the document in this experiment. As a consequence, either allocating a huge space of virtual memory (sometimes expensive) or calling repeatedly to the map channel (mmap with small buffersizes performs bad in the previous experiment) can be a priori counterproductive. This implementation may be affected by the buffer size and file size since it will determine the probability of calling to the map channel.

$$Probability\ of\ call\ map\ = 1\ -\ \frac{Buffer\ Size}{File\ Size}$$

If this probability is too big, the program will call the map for each number thereby weakening the performance.

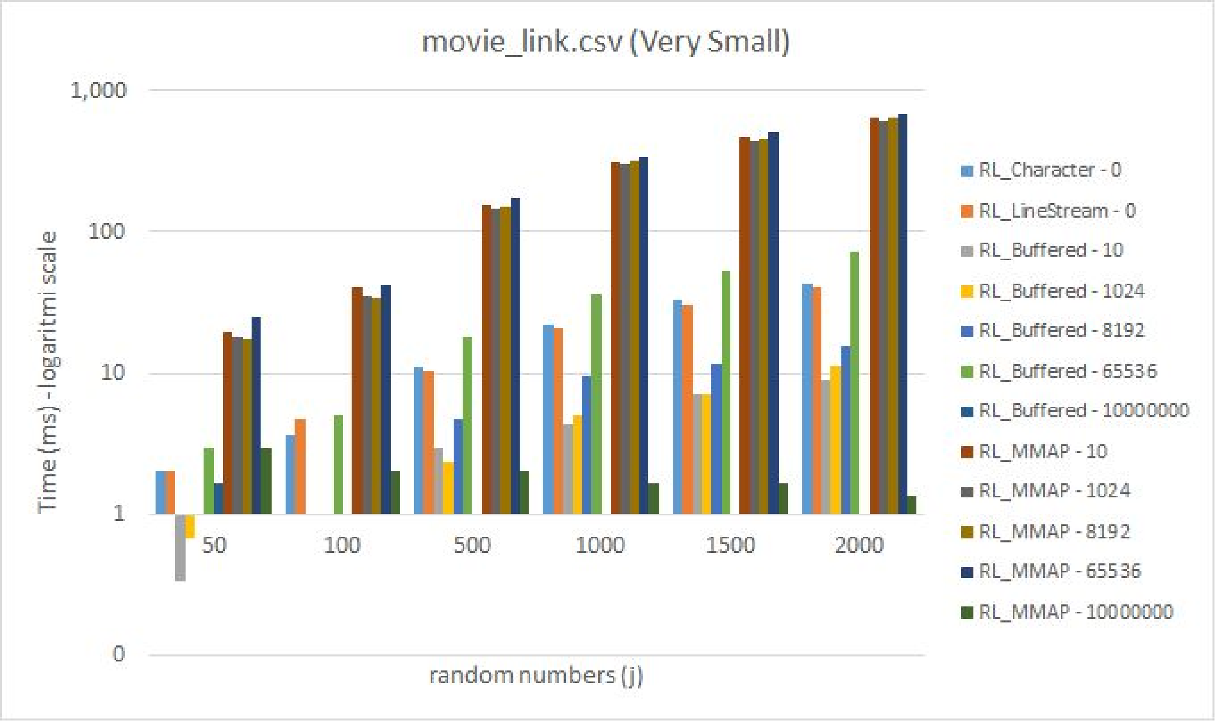
In the case of the buffer, it’s expected that the performance may be affected by the buffer size and the file size. If the buffer size is almost the same as the file size,the probability to require an I/O operation will be less, so the implementation can read from what is stored in main memory. The previous formula would apply in the buffered stream somehow.

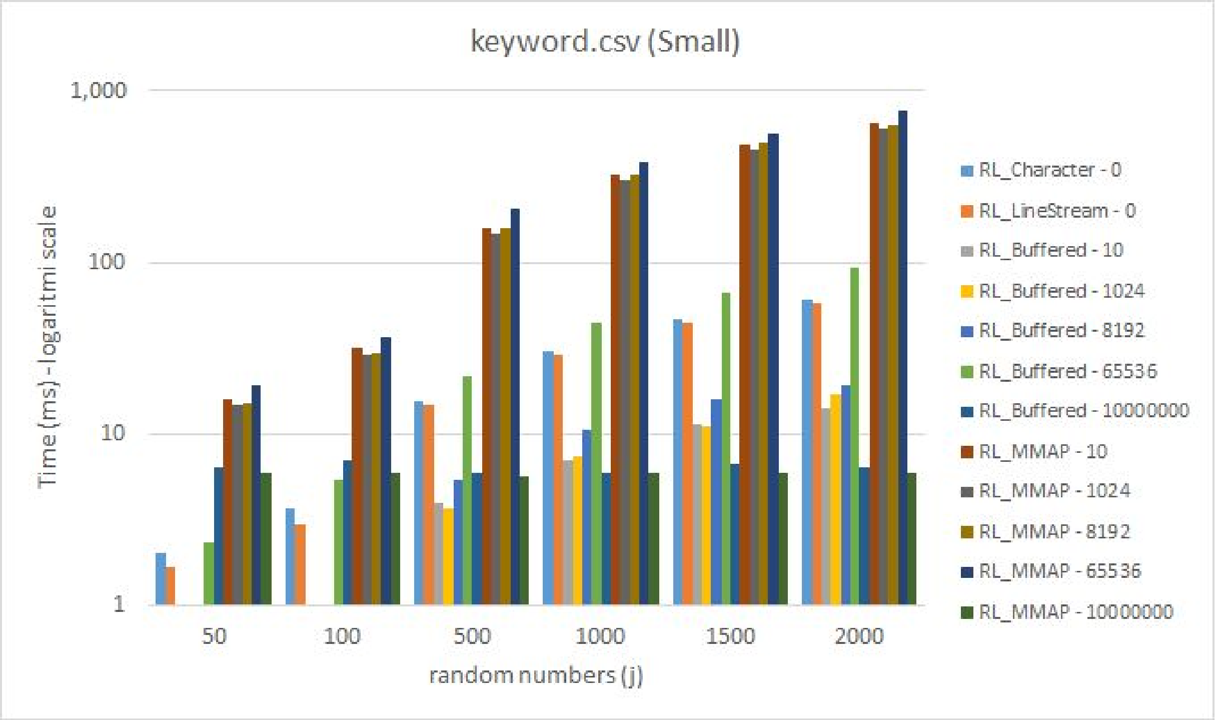
Otherwise, the results must be steady in the character and default line implementation. The character may be a bit slower, especially for files with long lines, but not as in the sequential experiment.

### Results

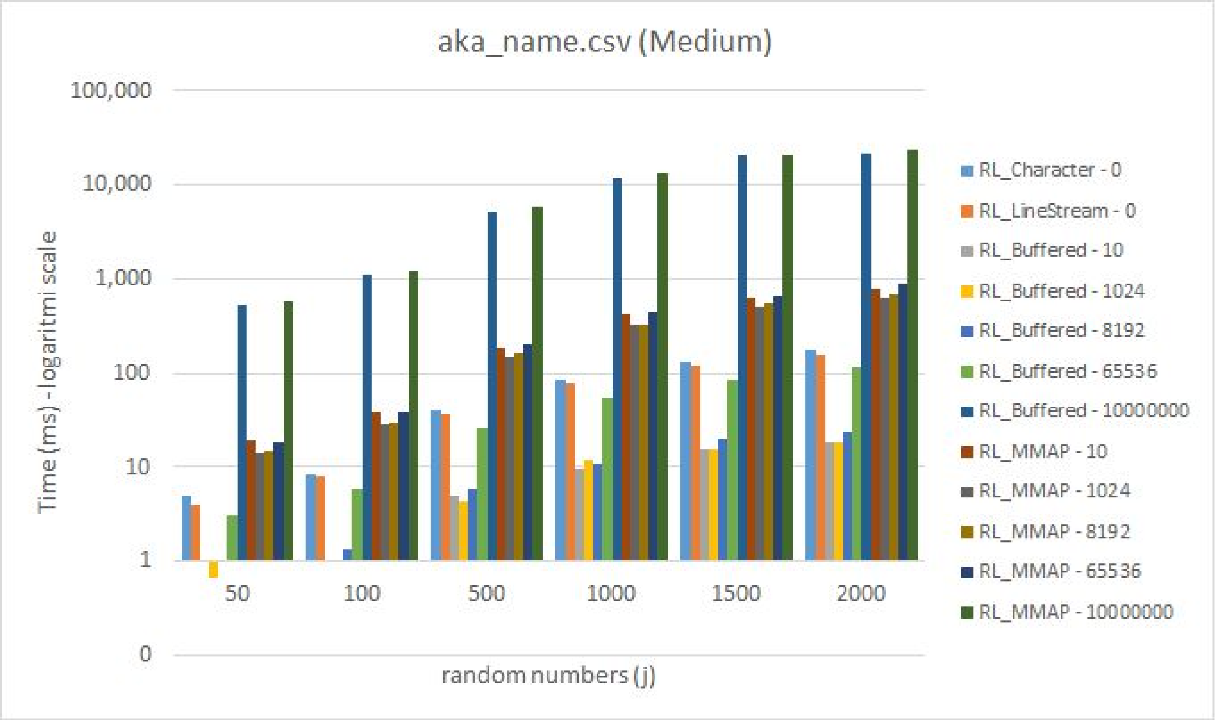
The results obtained **(in logarithmic scale)** represent somehow the expected behavior described before. The observations can be divided according to the size of the files:

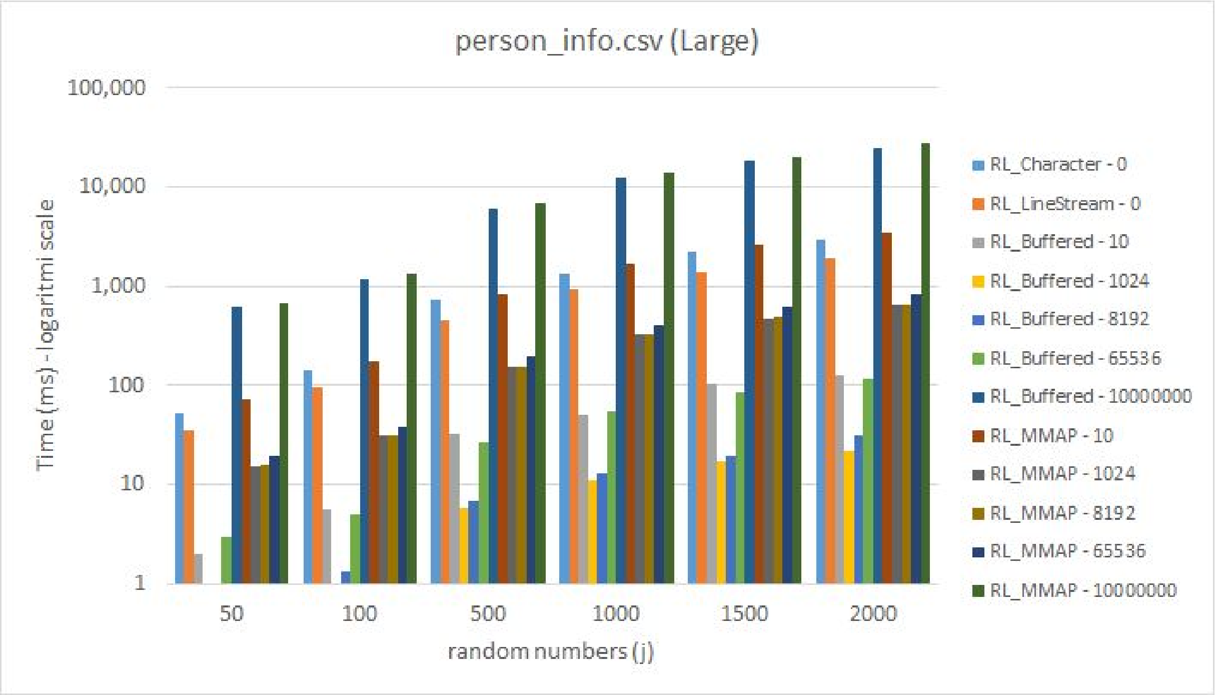
- Smaller Files (movie_link.csv and keyword.csv): Character and Default Line streaming has similar performance. They are good with a low random number of positions. Otherwise, Buffered and MMAP tend to have a similar “devious” behavior when the buffer size is so big, almost equal to the file; the performance is outstanding. But when MMAP has low buffer sizes, its performance is really bad. Unlike MMAP, Buffered has the best performance when the buffer size is similar to the length of a line (10 - 10 124).

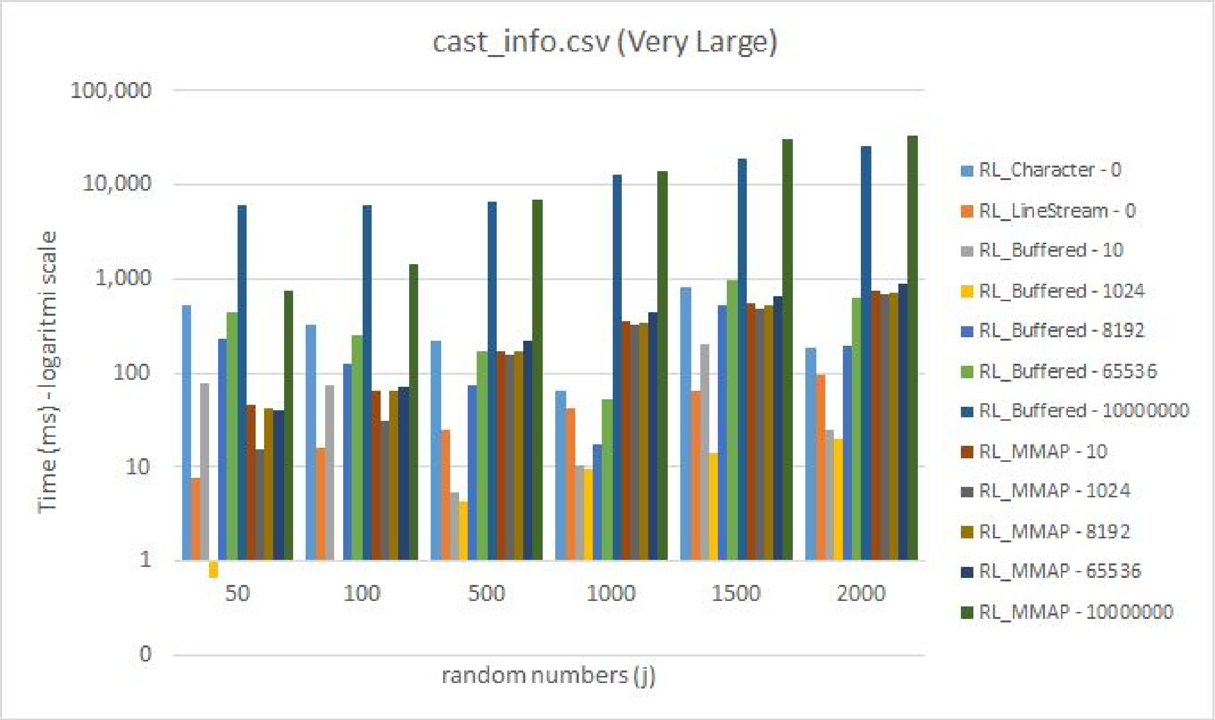




- Medium and Bigger Files: When working with bigger files, the performance of the character streaming tends to be slower than the default line.Since the buffer sizes are lower than the file sizes, MMAP shows the worst performance with the biggest buffer size as well as buffered streaming. The key difference is when working with buffered streaming and small buffer sizes. This experiment shows that these parameters are the optimal.


aka_name.csv (Medium)
100,000
10,000
1,000
100
10
1
0
Time (ms) - logaritmi scale
50
100
500
1000
1500
2000
random numbers (j)
RL_Character - 0
RL_LineStream - 0
RL_Buffered - 10
RL_Buffered - 1024
RL_Buffered - 8192
RL_Buffered - 65536
RL_Buffered - 10000000
RL_MMAP - 10
RL_MMAP - 1024
RL_MMAP - 8192
RL_MMAP - 65536
RL_MMAP - 10000000

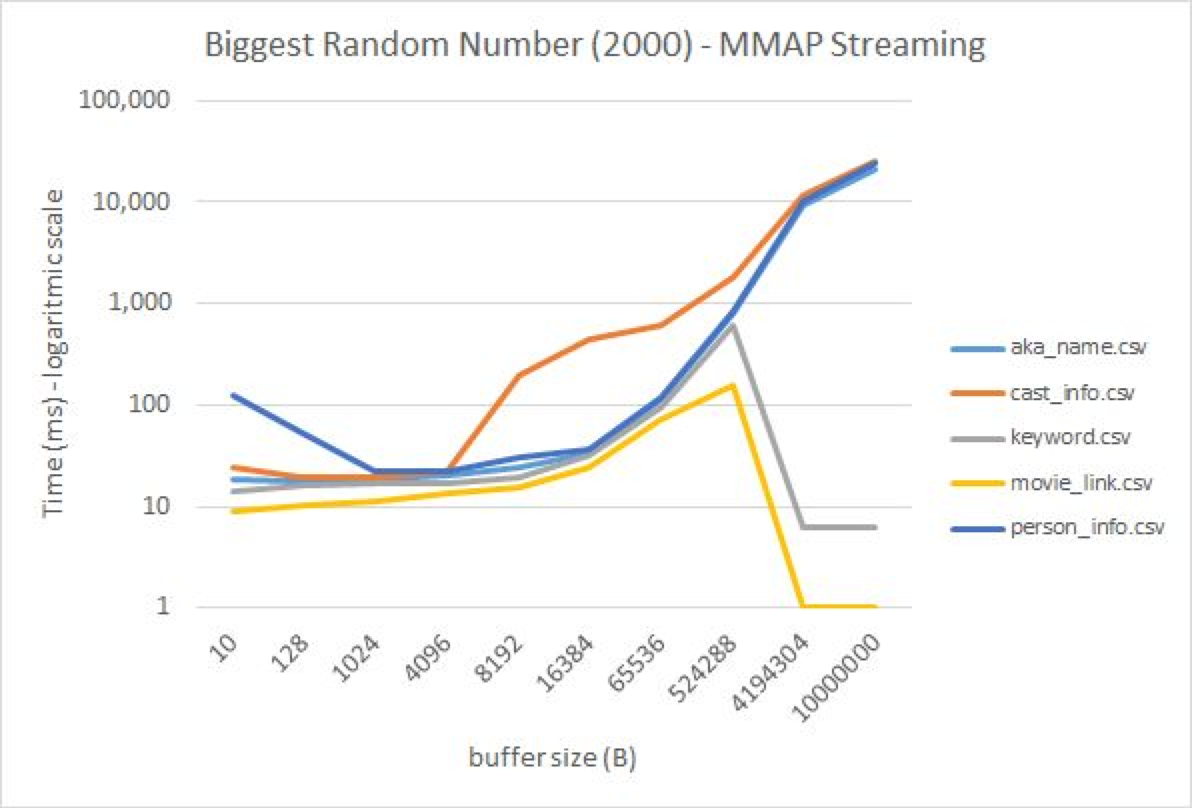

person_info.csv (Large)
100,000
10,000
1,000
100
10
1
Time (ms) - logaritmi scale
50
100
500
1000
1500
2000
random numbers (j)
RL_Character - 0
RL_LineStream - 0
RL_Buffered - 10
RL_Buffered - 1024
RL_Buffered - 8192
RL_Buffered - 65536
RL_Buffered - 10000000
RL_MMAP - 10
RL_MMAP - 1024
RL_MMAP - 8192
RL_MMAP - 65536
RL_MMAP - 10000000

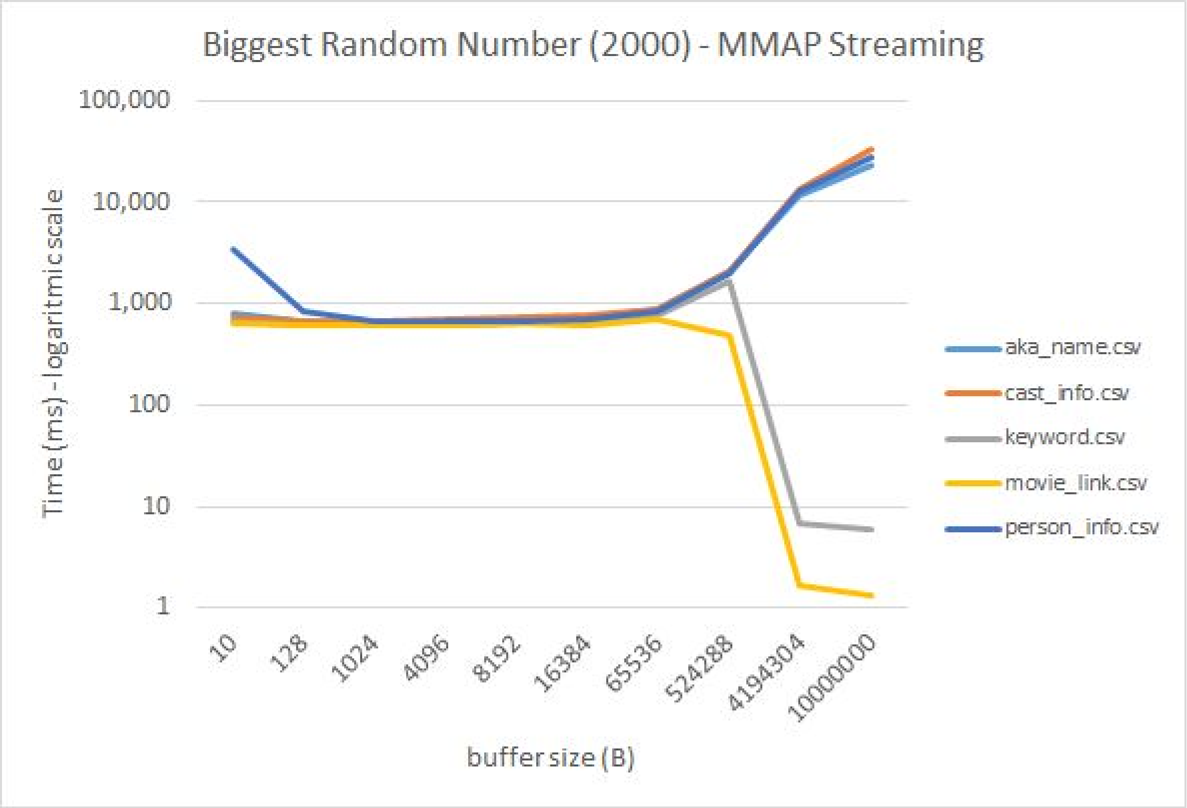


Additionally, The next graphs shows the evolution of each file when the buffer size increases in a logarithmic scale for either buffered streaming and mmap streaming with the biggest random number of positions:

- For big files as cast_info, the best results in buffered streaming occurs when buffer sizes is equal to 128 or 1024

- For small files as movie_link, the best results, in either MMAP or buffered, occur when buffer sizes are the highest.


Biggest Random Number (2000) - MMAP Streaming
100,000
10,000
1,000
100
10
1
Time (ms) - logaritmic scale
10
128
1024
4096
8192
16384
65536
524288
4194304
10000000
buffer size (B)
aka_name.csv
cast_info.csv
keyword.csv
movie_link.csv
person_info.csv



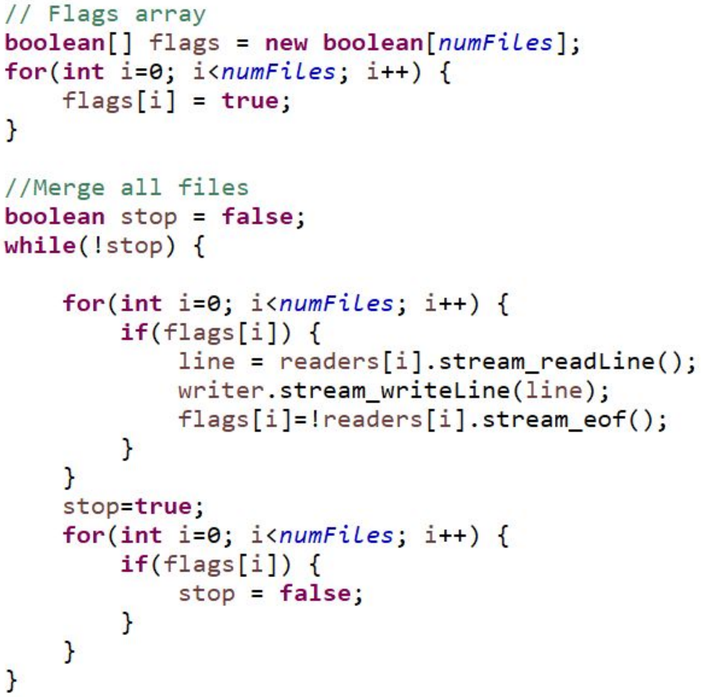
### Conclusions and Answers

- The best readline streaming for random jump is the buffered with low buffer size (almost the same as the average line length) because It will only perform **(j) I/Os** with the most precise buffer of bytes for a line in memory. As shown in the results, bigger buffer sizes will load lots of useless bytes when the probability of calling the buffer is high.
- MMAP and Buffered streamings with highest buffer sizes are good options when the buffer size is almost equal to the file size since the probability to call the functions is less. Otherwise, calling many times MMAP can be counterproductive for the performance.
- Character and default line performs good, but default line gets faster when the file and lines are bigger.
- Comparison with Sequential: While buffered streaming with low buffer size is optimal for random jumping, previous experiments have shown that it’s not a good option for sequential readings which are part of the next experiment.

## 4. Experimental observations regarding experiment 1.3.

In this experiment, we implement *rrmerge* for merging a number of files in a round robin manner. Two types of reading used in this experiment are Line Reading and MMap Reading (with buffer size of 10 000 000 bytes). The reason why we choose these implementations is that in this experiment input files are read sequentially and they are the best ones from the previous sequential experiments.

Four types of writing are implemented, such as: Character Writing, Line Writing, Buffered Writing and MMap Writing. Different versions of *rrmerge* for each pair of reading and writing types are tested. We also used a varying number of files and buffer size to find the optimal values. These following lines of code describe how input files are merged.



```
// Flags array
boolean[] flags = new boolean[numFiles];
for(int i=0; i<numFiles; i++) {
    flags[i] = true;
}

//Merge all files
boolean stop = false;
while(!stop) {

    for(int i=0; i<numFiles; i++) {
        if(flags[i]) {
            line = readers[i].stream_readLine();
            writer.stream_writeLine(line);
            flags[i]=!readers[i].stream_eof();
        }
    }
    stop=true;
    for(int i=0; i<numFiles; i++) {
        if(flags[i]) {
            stop = false;
        }
    }
}
```

### Expected Behavior

MMap reading is expected to continue to perform better than Buffered reading since it was the winner in the sequential reading experiment. Using memory mapping for both reading and writing is expected to be the best solution in this experiment in terms of execution time since the process of loading data into memory and writing data into disk are controlled by operating system.

## Results

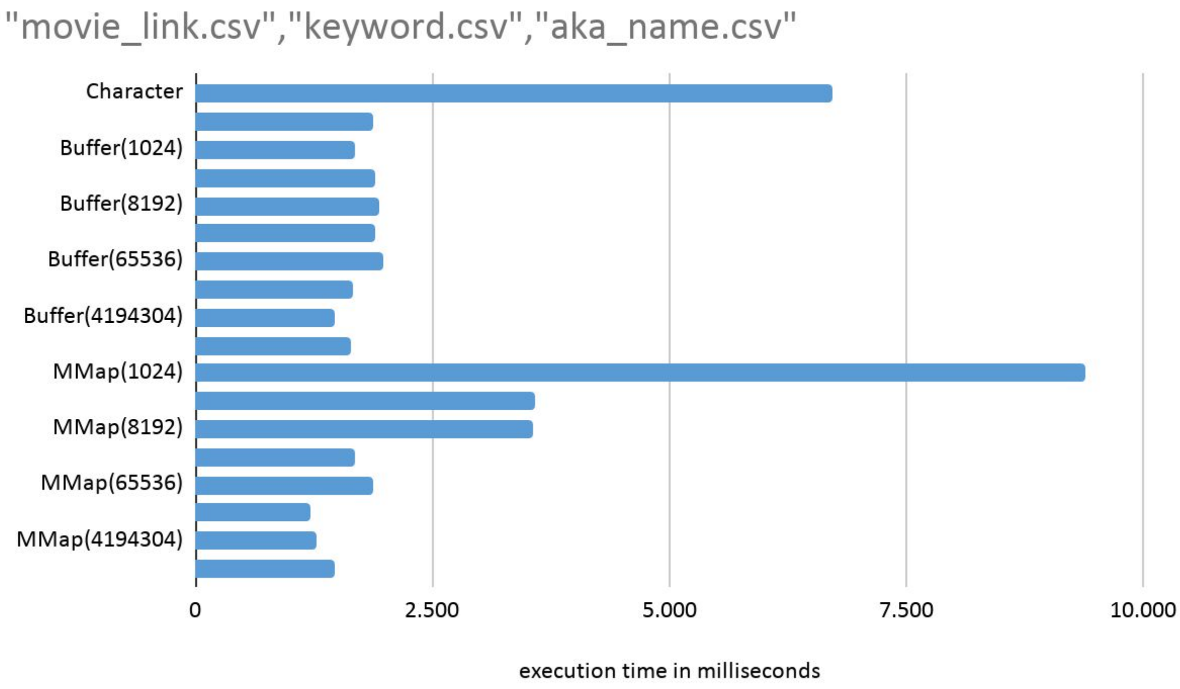
For each type of reading, these below charts show the execution time on average for merging 3 files or 4 files with different types of writing. The MMap writing appears to be the best writing implementation when it is paired with both MMap reading and Line reading.

In terms of reading, with the same MMap writing implementation, the MMap reading outperforms Line reading in merging 4 files but not in merging 3 files. This means MMap reading implementation has an advantage when dealing with a large amount of files or with large files. Line reading implementation has better performance in some cases.

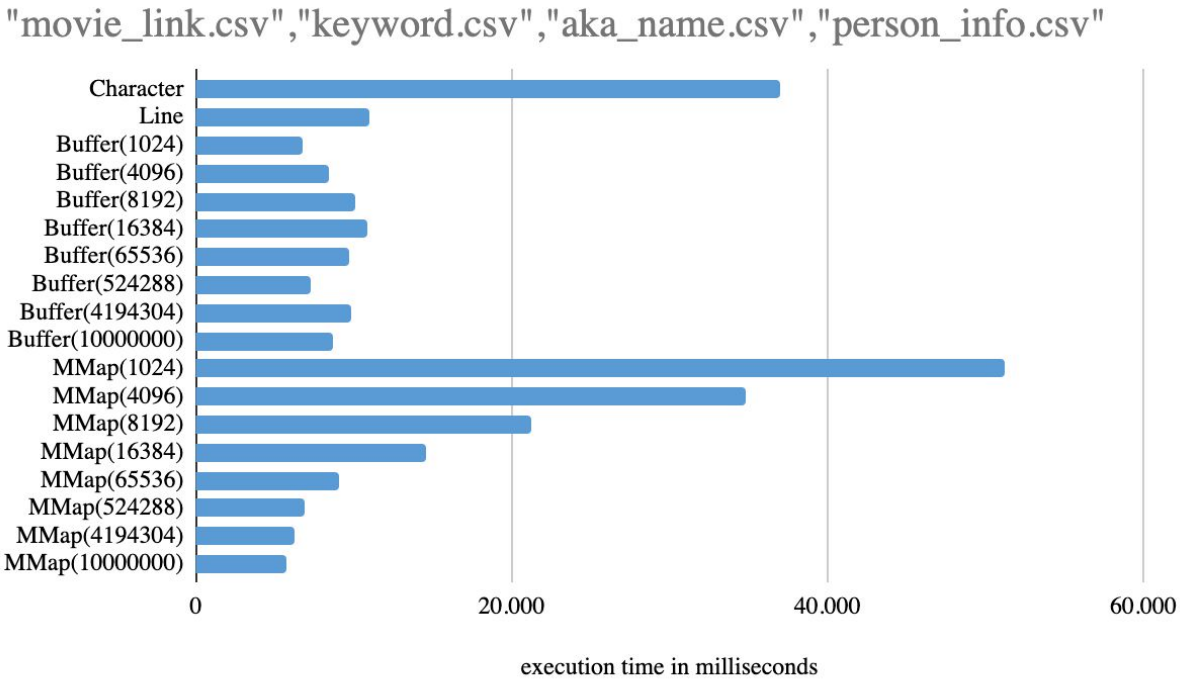
The best pair for merging 3 files is Line reading together with MMap writing. MMap reading with MMap writing is the best pair for the rest of the experiments. With a big buffer size, the number of times of creating new mmap buffer is reduced. Hence, the overhead time for system calls is decreased.

- **MMap Reading**

**3 files: movie_link.csv, keyword.csv, aka_name.csv**

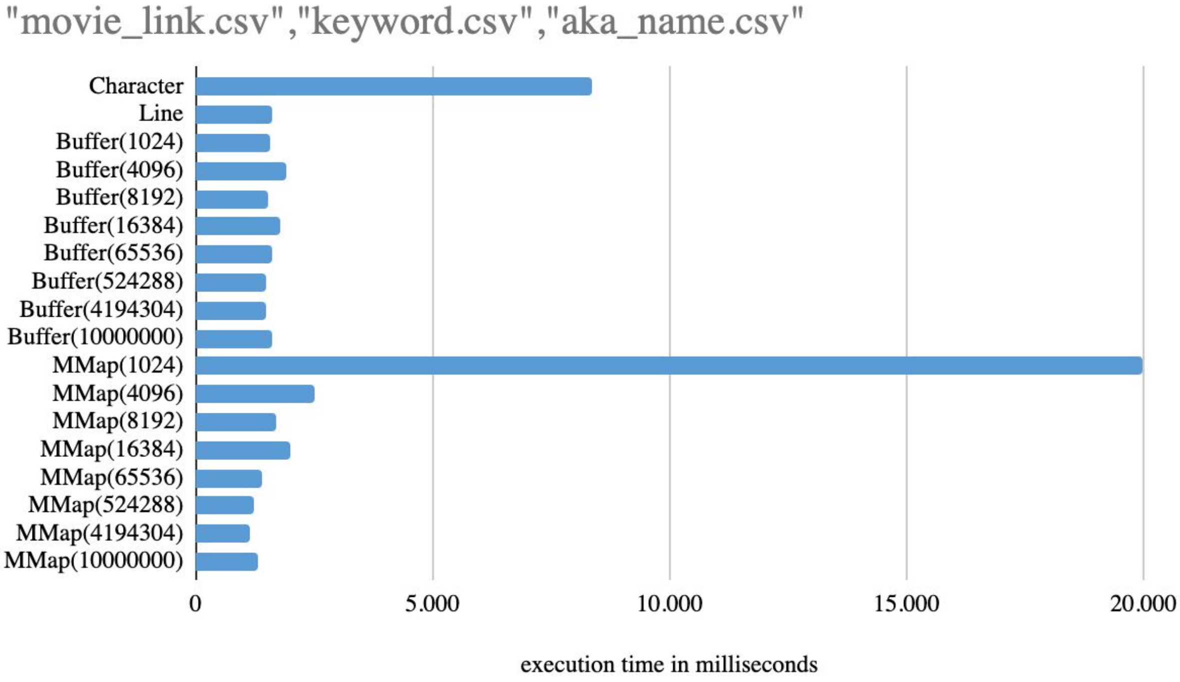


**4 files: movie_link.csv, keyword.csv, aka_name.csv, person_info.csv**

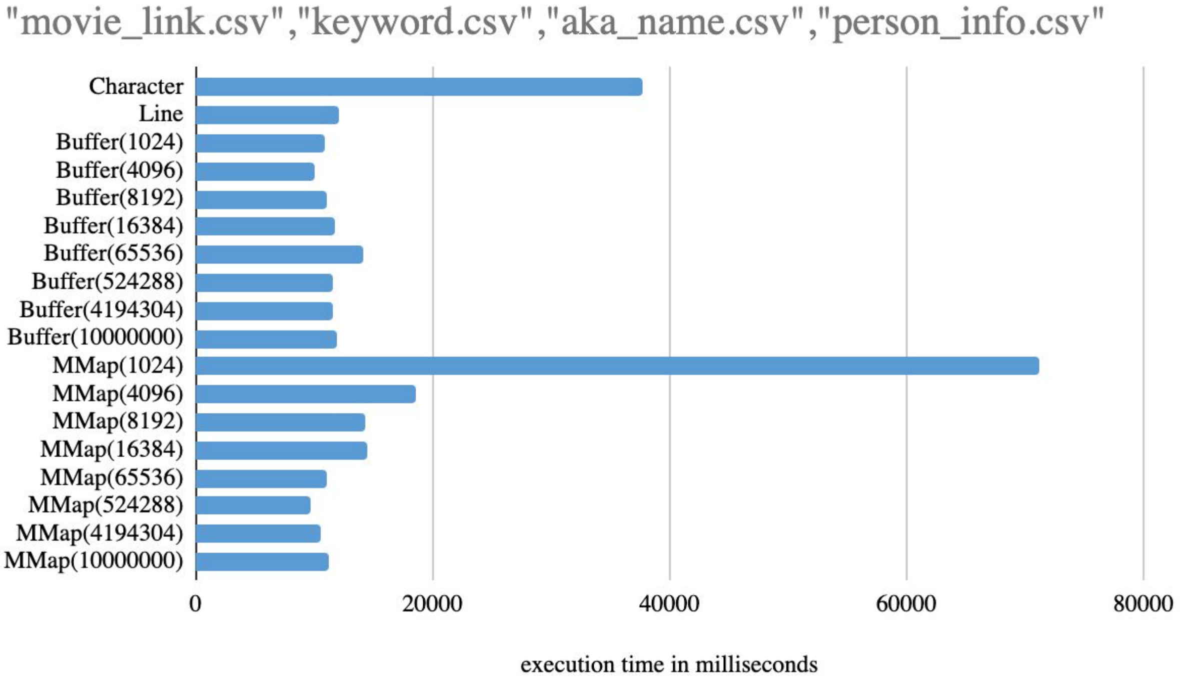


- **Line Reading**

**3 files: movie_link.csv, keyword.csv, aka_name.csv**



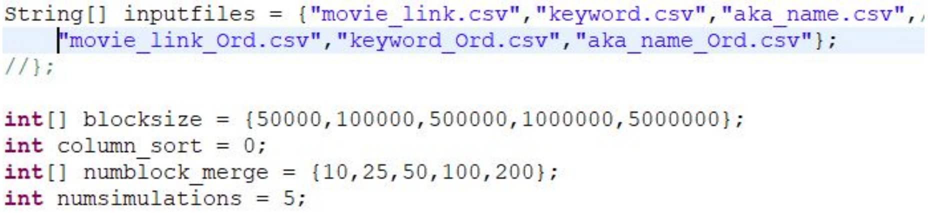
**4 files: movie_link.csv, keyword.csv, aka_name.csv, person_info.csv**



## III. Observations on multi-way merge sort

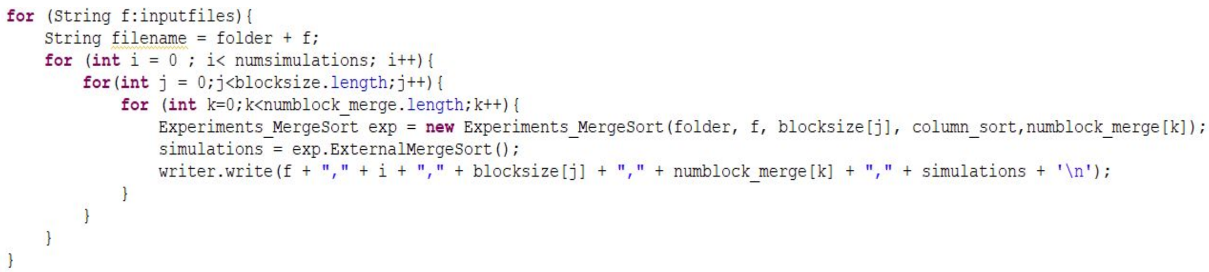
The extsort program emulates the behavior of the two phase, multi-way algorithm. The first phase consists of splitting the file into column-k sorted chunks or subfiles whose sizes are lower than M bytes. The second is in charge of merging d or less chunks at the same time and writing the lines in order until there is only one chunk left to merge. The result will be a file of the same size and ordered by the column k.
For the purpose of the experiment, the following parameters have been tested:

- Blocksize (M): size of sub filesor buffers
- Column_sort (k): number of column to order
- numblock_merge(d): number of subfiles to merge at the same time
- “_Ord”: Files already ordered



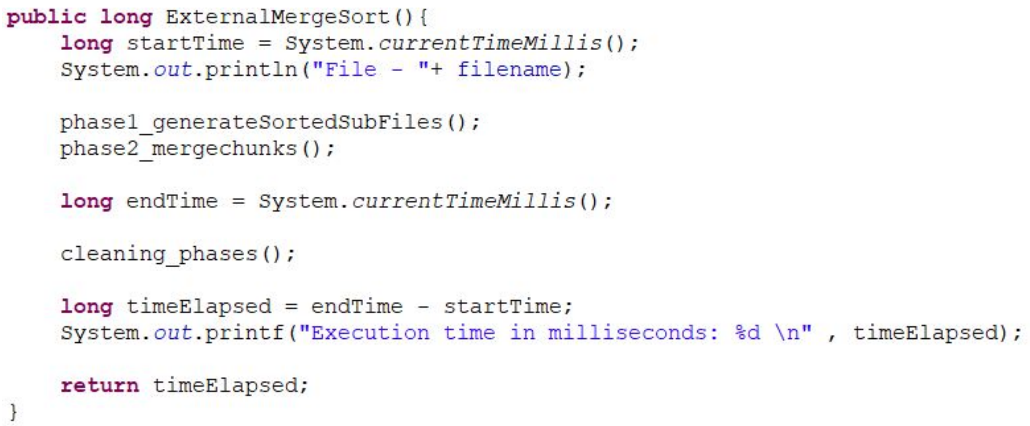
```
String[] inputfiles = {"movie_link.csv","keyword.csv","aka_name.csv",
    "movie_link_Ord.csv","keyword_Ord.csv","aka_name_Ord.csv"};
//};

int[] blocksize = {50000,100000,500000,1000000,5000000};
int column_sort = 0;
int[] numblock_merge = {10,25,50,100,200};
int numsimulations = 5;
```

The extsort program calls the Experiments_MergeSort in order to performs the two phases with the different parameters for the experiment and then stores the time for the graphical information:



```
for (String f:inputfiles){
    String filename = folder + f;
    for (int i = 0 ; i< numsimulations; i++){
        for(int j = 0;j<blocksize.length;j++){
            for (int k=0;k<numblock_merge.length;k++){
                Experiments_MergeSort exp = new Experiments_MergeSort(folder, f, blocksize[j], column_sort,numblock_merge[k]);
                simulations = exp.ExternalMergeSort();
                writer.write(f + "," + i + "," + blocksize[j] + "," + numblock_merge[k] + "," + simulations + '\n');
            }
        }
    }
}
```

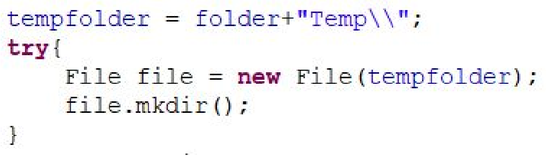
The function inside the class that executes the two phases is ExternalMergeSort(). It calls phase 1 to generate the sorted sub files and then phase 2 to merge the chunks. Finally, it executes a cleaning of the previous temporal files.



```
public long ExternalMergeSort(){
    long startTime = System.currentTimeMillis();
    System.out.println("File - "+ filename);

    phase1_generateSortedSubFiles();
    phase2_mergechunks();

    long endTime = System.currentTimeMillis();

    cleaning_phases();

    long timeElapsed = endTime - startTime;
    System.out.printf("Execution time in milliseconds: %d \n" , timeElapsed);

    return timeElapsed;
}
```

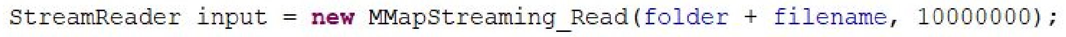
- Phase 1 GenerateSortedSubFiles()

Firstly, phase 1 creates a temporary directory to start the chunks.



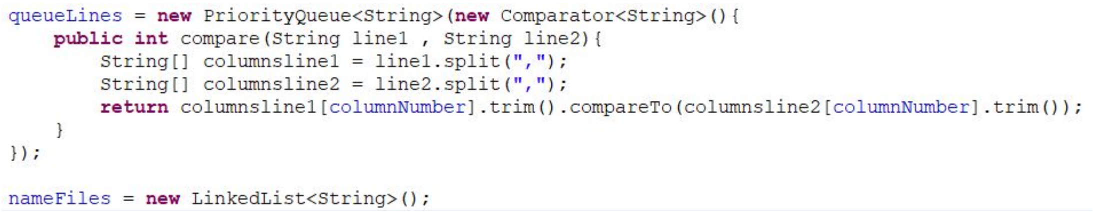
```
tempfolder = folder+"Temp\\";
try{
    File file = new File(tempfolder);
    file.mkdir();
}
```

Then, it opens the file to read with the best implementation from the previous experiment.



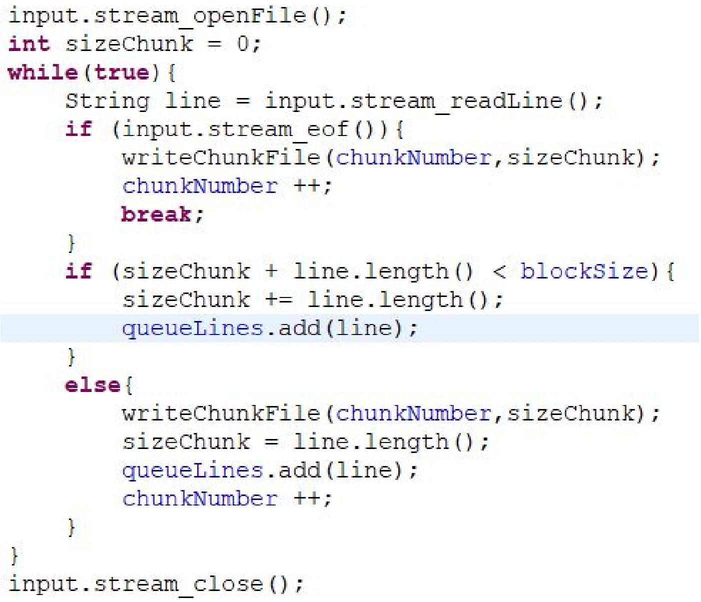
```
StreamReader input = new MMapStreaming_Read(folder + filename, 10000000);
```

It creates a queue for storing the lines in order according to column k (*queueLines*) and a queue for the name of the chunks created (*nameFiles*).



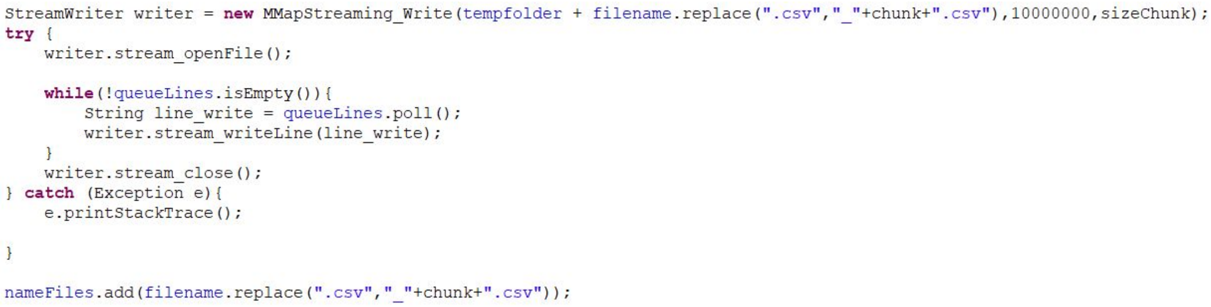
```
queueLines = new PriorityQueue<String>(new Comparator<String>(){
    public int compare(String line1 , String line2){
        String[] columnsline1 = line1.split(",");
        String[] columnsline2 = line2.split(",");
        return columnsline1[columnNumber].trim().compareTo(columnsline2[columnNumber].trim());
    }
});

nameFiles = new LinkedList<String>();
```

After getting all the variables prepared, the next step performs the core of phase 1. It opens the file, reads all the lines of the file and adds them to the *queueLines* . Every time *sizeChunk*, the current size of the buffer, exceeds the size allowed by M, it polls all the elements and writes them in a chunk file with the *writeChunkFile* function. After writing, the exceeding line is added for the next chunk (*chunkNumber*++).



```
input.stream_openFile();
int sizeChunk = 0;
while(true){
    String line = input.stream_readLine();
    if (input.stream_eof()){
        writeChunkFile(chunkNumber,sizeChunk);
        chunkNumber ++;
        break;
    }
    if (sizeChunk + line.length() < blockSize){
        sizeChunk += line.length();
        queueLines.add(line);
    }
    else{
        writeChunkFile(chunkNumber,sizeChunk);
        sizeChunk = line.length();
        queueLines.add(line);
        chunkNumber ++;
    }
}
input.stream_close();
```

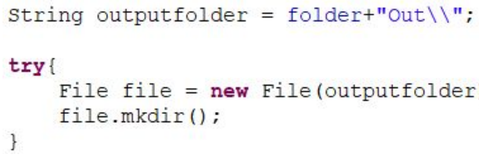
The *writeChunkFile* function creates a writer with the best implementation of the previous experiment, poll and add every line in *queueLines* until the queue is empty. Finally it adds the filename for phase2 reference to the *nameFiles* queue.



```
StreamWriter writer = new MMapStreaming_Write(tempfolder + filename.replace(".csv","_"+chunk+".csv"),10000000,sizeChunk);
try {
    writer.stream_openFile();

    while(!queueLines.isEmpty()){
        String line_write = queueLines.poll();
        writer.stream_writeLine(line_write);
    }
    writer.stream_close();
} catch (Exception e){
    e.printStackTrace();

}

nameFiles.add(filename.replace(".csv","_"+chunk+".csv"));
```

- Phase 2 Merge Chunks():

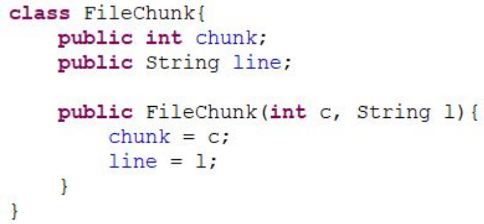
  Firstly, the folder for the output is created (final merge sort file)



```
String outputfolder = folder+"Out\\";

try{
    File file = new File(outputfolder);
    file.mkdir();
}
```

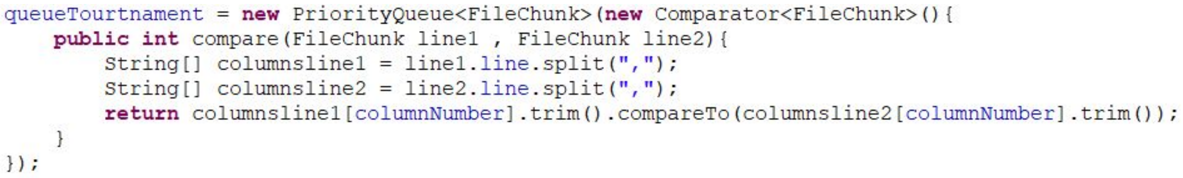
A class FileChunk, which stores the identification from 0 to d-1 and the line, and a PriorityQueue for that class were created for the tournament against chunks.

**Class**



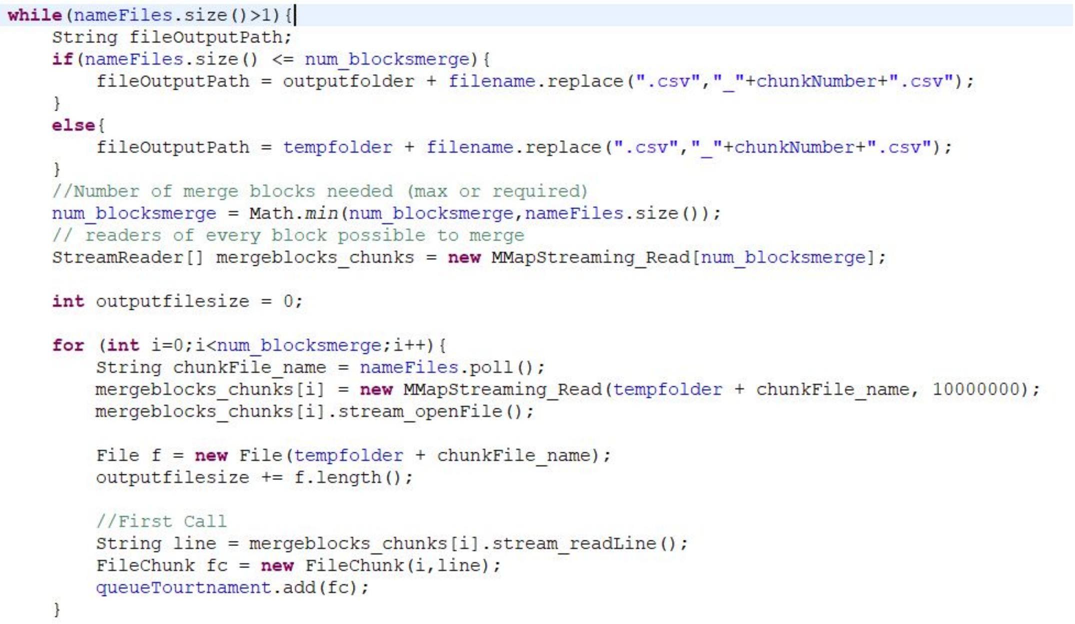
```
class FileChunk{
    public int chunk;
    public String line;

    public FileChunk(int c, String l){
        chunk = c;
        line = l;
    }
}
```

**PriorityQueue for Tournament**



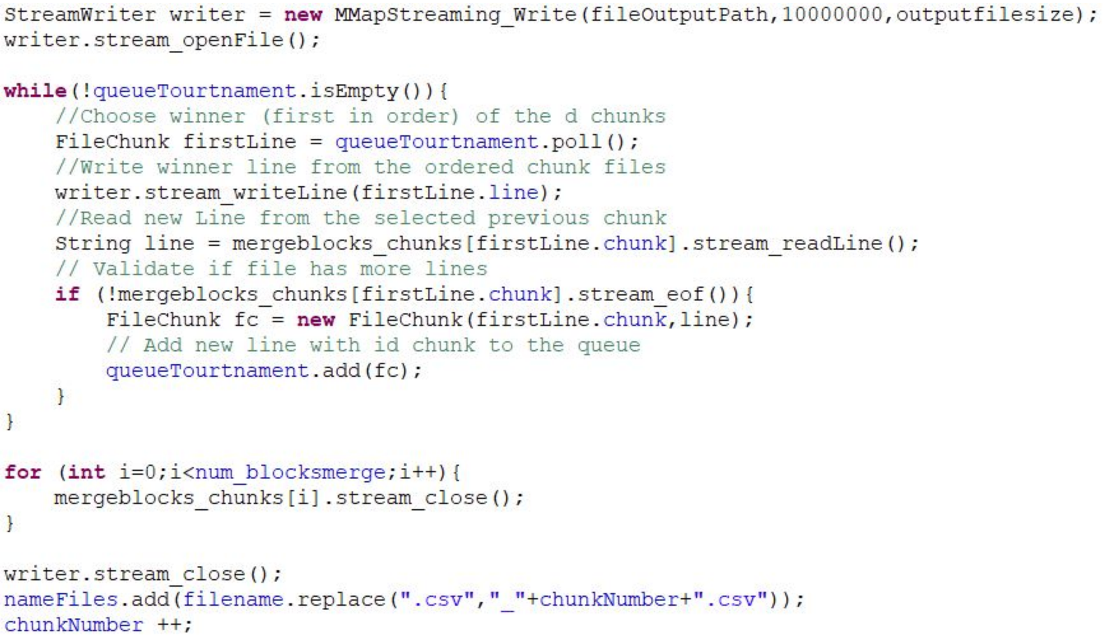
```
queueTourtnament = new PriorityQueue<FileChunk>(new Comparator<FileChunk>(){
    public int compare(FileChunk line1 , FileChunk line2){
        String[] columnsline1 = line1.line.split(",");
        String[] columnsline2 = line2.line.split(",");
        return columnsline1[columnNumber].trim().compareTo(columnsline2[columnNumber].trim());
    }
});
```

The next and main step repeats until there is only one chunk or filename in the nameFiles queue. As initialization, the *fileOutputPath* is assigned when it's the last chunk to be created. *Num_blocksmerge* will be changed to the chunks remaining only when there aren't more chunks to merge. The readers with MMAP are initialized with the respective first chunks in the queue and the first lines of every chunk in form of FileChunk class will enter into the *queueTourtnament*.



```
while(nameFiles.size()>1){
    String fileOutputPath;
    if(nameFiles.size() <= num_blocksmerge){
        fileOutputPath = outputfolder + filename.replace(".csv","_"+chunkNumber+".csv");
    }
    else{
        fileOutputPath = tempfolder + filename.replace(".csv","_"+chunkNumber+".csv");
    }
    //Number of merge blocks needed (max or required)
    num_blocksmerge = Math.min(num_blocksmerge,nameFiles.size());
    // readers of every block possible to merge
    StreamReader[] mergeblocks_chunks = new MMapStreaming_Read[num_blocksmerge];

    int outputfilesize = 0;

    for (int i=0;i<num_blocksmerge;i++){
        String chunkFile_name = nameFiles.poll();
        mergeblocks_chunks[i] = new MMapStreaming_Read(tempfolder + chunkFile_name, 10000000);
        mergeblocks_chunks[i].stream_openFile();

        File f = new File(tempfolder + chunkFile_name);
        outputfilesize += f.length();

        //First Call
        String line = mergeblocks_chunks[i].stream_readLine();
        FileChunk fc = new FileChunk(i,line);
        queueTourtnament.add(fc);
    }
```

The next step will perform the tournament until all files are empty. Everytime there is going to be a winner (the smallest according to the order function). This winner is pulled from the queue and written to a new temporal or final file depending on the *fileOutputPath*. As a result, the new files merged will also be in order and then the merged file will be added to the nameFiles queue for further merging if it's possible.



```
StreamWriter writer = new MMapStreaming_Write(fileOutputPath,10000000,outputfilesize);
writer.stream_openFile();

while(!queueTourtnament.isEmpty()){
    //Choose winner (first in order) of the d chunks
    FileChunk firstLine = queueTourtnament.poll();
    //Write winner line from the ordered chunk files
    writer.stream_writeLine(firstLine.line);
    //Read new Line from the selected previous chunk
    String line = mergeblocks_chunks[firstLine.chunk].stream_readLine();
    // Validate if file has more lines
    if (!mergeblocks_chunks[firstLine.chunk].stream_eof()){
        FileChunk fc = new FileChunk(firstLine.chunk,line);
        // Add new line with id chunk to the queue
        queueTourtnament.add(fc);
    }
}

for (int i=0;i<num_blocksmerge;i++){
    mergeblocks_chunks[i].stream_close();
}

writer.stream_close();
nameFiles.add(filename.replace(".csv","_"+chunkNumber+".csv"));
chunkNumber ++;
```

As mentioned before, this process is repeated until the queue of *nameFiles* has only one element which should be the final file ordered.

**Expected Behavior**

For the purpose of this experiment, the reading and writing implementations will be the same for every simulation, MMAP resulted the best from the previous experiments, so we can assume that this time is going to be constant (However, if the buffer size of the MMAP is less than M, it’s not accurate in real application). The expected behavior will be derived from a tradeoff among the file size (N), the buffer size (M) and the blocks to merge (d). It is expected that the cost increases when the file size increases. Otherwise, if the buffer size or the blocks to merge increases, the cost should decrease.

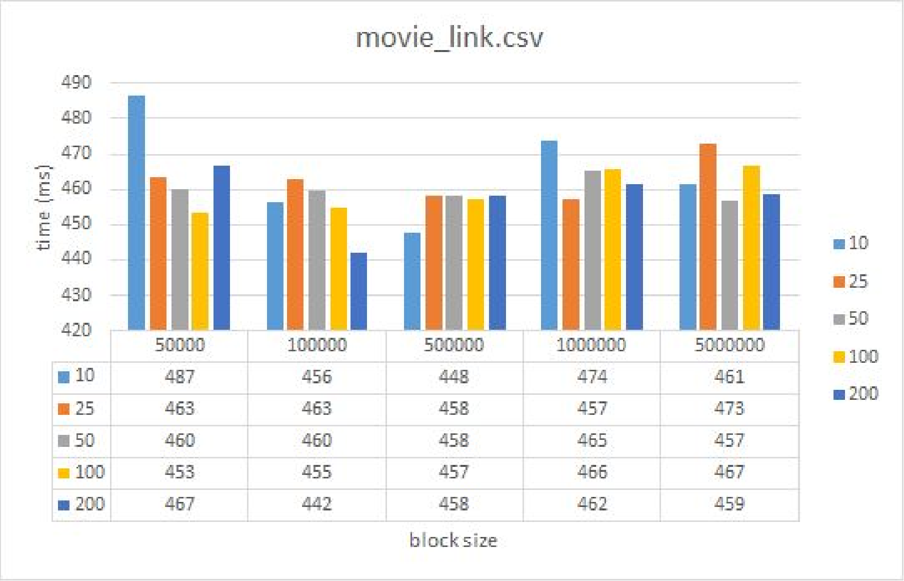
For the first phase, the cost should be 1 I/O for reading the file and [N/M] I/Os for the chunks that will be written into disk. For the second phase, the cost should be a logarithmic d function starting from the initial number of chunks in the queue [N/M] until there is only one chunk. This should be multiplied by two because of the read/write operations to disk.

$$Cost\ I/O\ = 1\ +\ \frac{N}{M}\ + 2 * [\frac{N}{M}] * [log_d[\ \frac{N}{M}]]$$

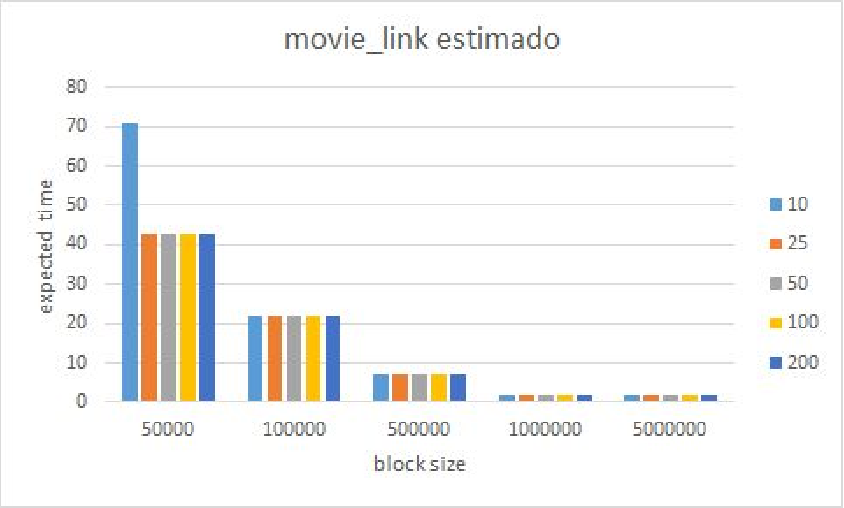
**Results**

In order to compare different parameters, each file, which represents a different size of N., has been isolated to compare the times and the trending lines of each parameter. Each file is accompanied with its expected behavior according to the formula to get a better idea of how it should perform.

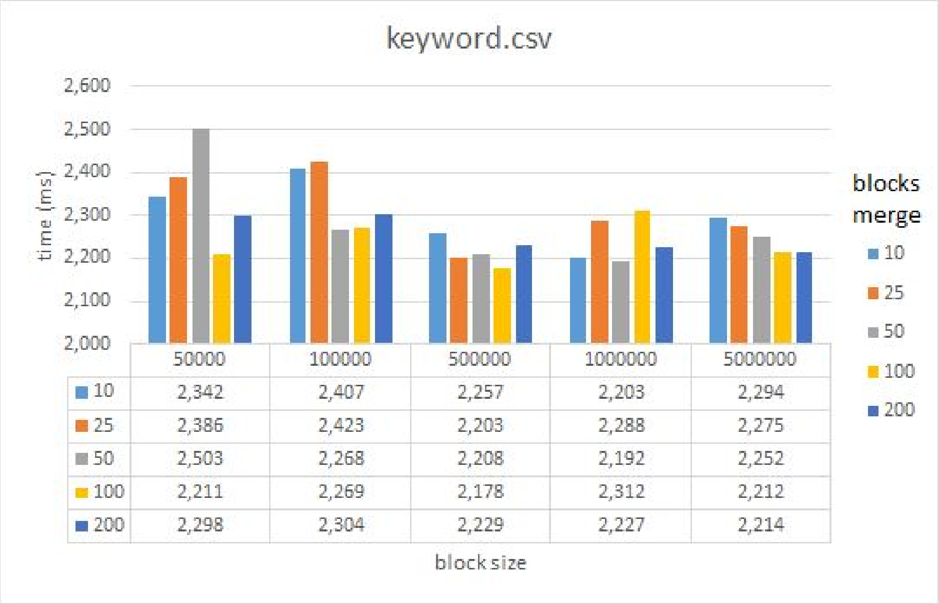
In the first file of 650,000 bytes, the lowest blocksize with the lowest number of merge blocks represents the lowest performance as expected. However, contrary to the equation, the rest of the values has a lot of deviation in the small file



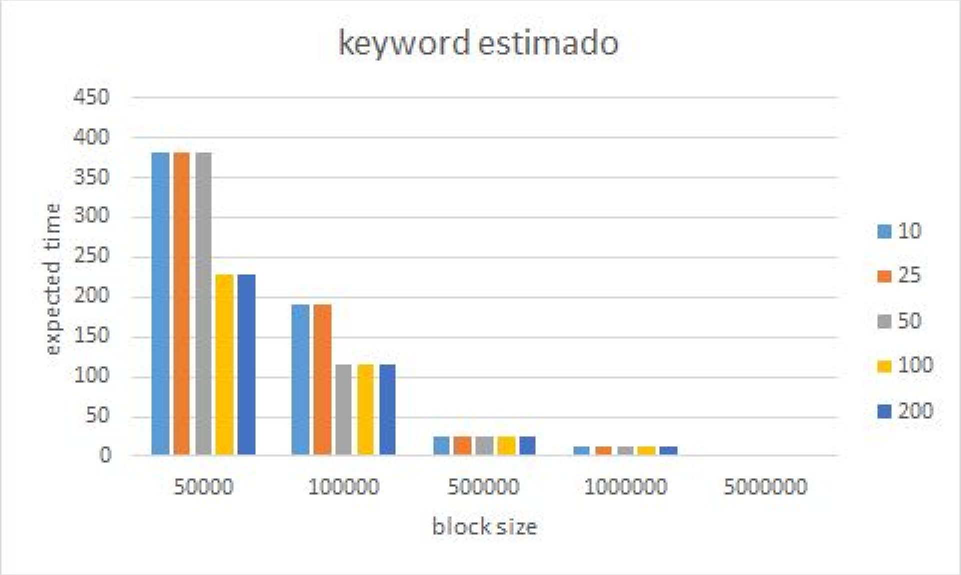
| | 50000 | 100000 | 500000 | 1000000 | 5000000 |
|---|---|---|---|---|---|
| 10 | 487 | 456 | 448 | 474 | 461 |
| 25 | 463 | 463 | 458 | 457 | 473 |
| 50 | 460 | 460 | 458 | 465 | 457 |
| 100 | 453 | 455 | 457 | 466 | 467 |
| 200 | 467 | 442 | 458 | 462 | 459 |



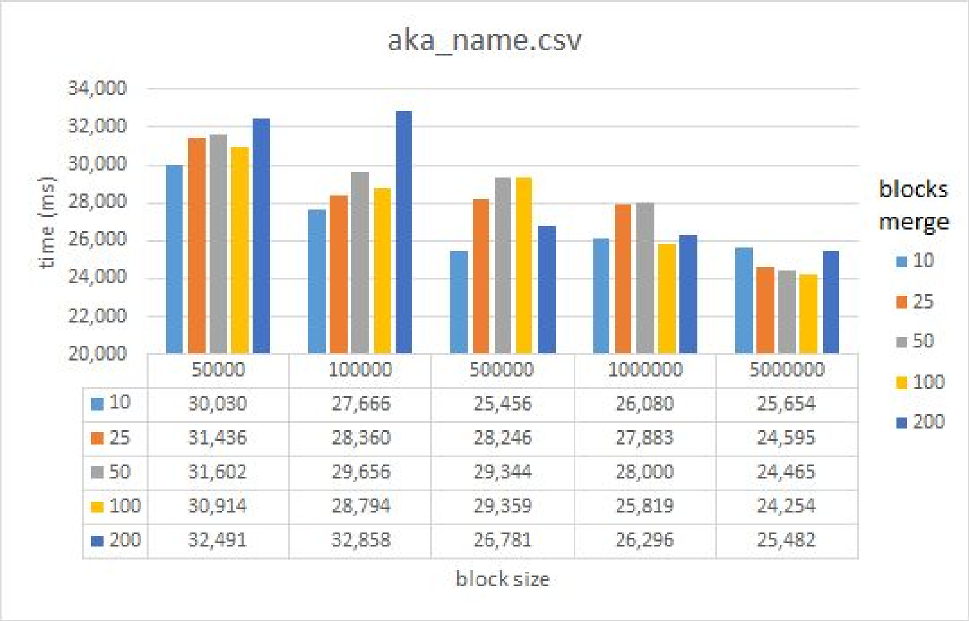
In the keyword file which contains 3,791,536 bytes, the relationship of block size is more visible and the number of block merge also influences the cost. According to the expected graph, the difference decreases steadily for block size 500,000 for the blocksize and the number of blocks to merge and the results somehow show that behavior.



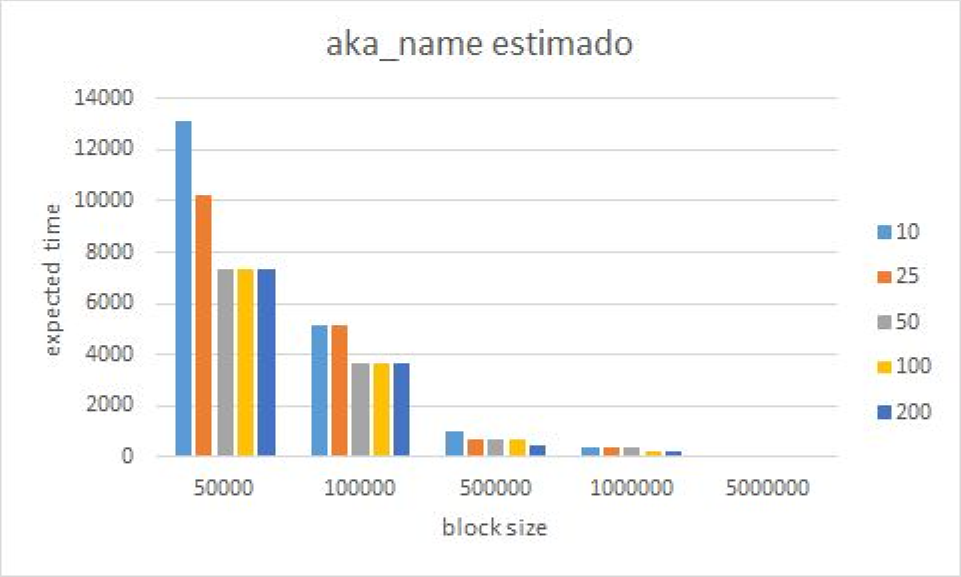
| | 50000 | 100000 | 500000 | 1000000 | 5000000 |
|---|---|---|---|---|---|
| ■ 10 | 2,342 | 2,407 | 2,257 | 2,203 | 2,294 |
| ■ 25 | 2,386 | 2,423 | 2,203 | 2,288 | 2,275 |
| ■ 50 | 2,503 | 2,268 | 2,208 | 2,192 | 2,252 |
| ■ 100 | 2,211 | 2,269 | 2,178 | 2,312 | 2,212 |
| ■ 200 | 2,298 | 2,304 | 2,229 | 2,227 | 2,214 |



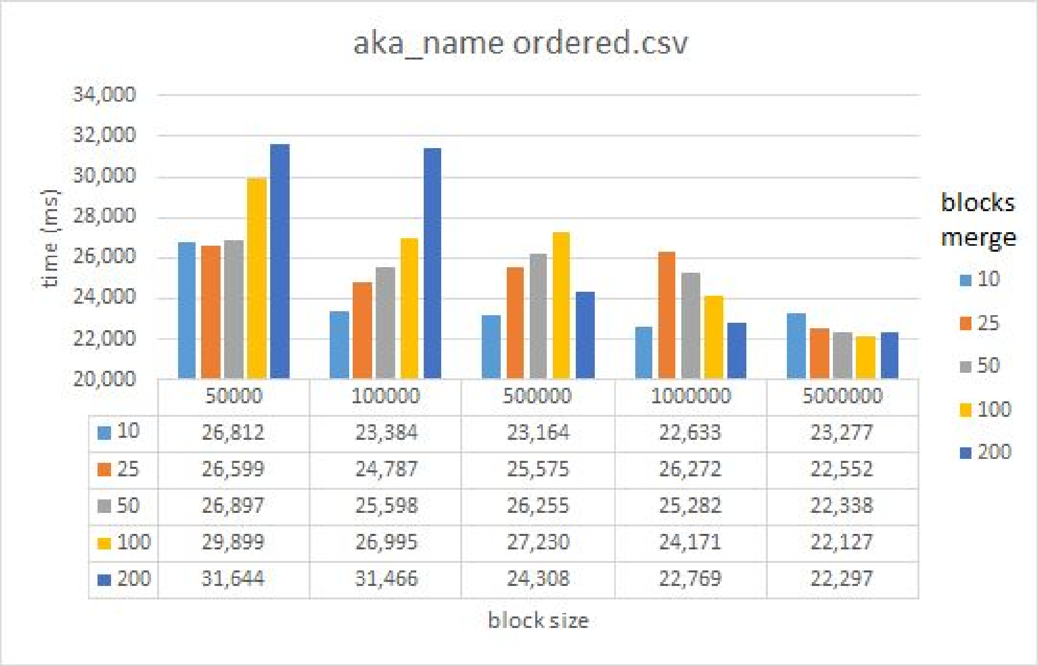
In a bigger file like aka_name (73,004,383 bytes), the difference is more noticeable by block size. However, it’s contrary to the expected when it’s analyzed by the number of blocks to merge. Like our results, the expected behavior shows below a major change by block size rather than blocks merge.



| | 50000 | 100000 | 500000 | 1000000 | 5000000 |
|---|---|---|---|---|---|
| 10 | 30,030 | 27,666 | 25,456 | 26,080 | 25,654 |
| 25 | 31,436 | 28,360 | 28,246 | 27,883 | 24,595 |
| 50 | 31,602 | 29,656 | 29,344 | 28,000 | 24,465 |
| 100 | 30,914 | 28,794 | 29,359 | 25,819 | 24,254 |
| 200 | 32,491 | 32,858 | 26,781 | 26,296 | 25,482 |



The last experiment shows the performance in the aka_name file that was previously ordered. As we can see, the performance is better when the block size is increased. However, increasing the number of blocks to merge doesn't improve the performance.



| | 50000 | 100000 | 500000 | 1000000 | 5000000 |
|---|---|---|---|---|---|
| ■ 10 | 26,812 | 23,384 | 23,164 | 22,633 | 23,277 |
| ■ 25 | 26,599 | 24,787 | 25,575 | 26,272 | 22,552 |
| ■ 50 | 26,897 | 25,598 | 26,255 | 25,282 | 22,338 |
| ■ 100 | 29,899 | 26,995 | 27,230 | 24,171 | 22,127 |
| ■ 200 | 31,644 | 31,466 | 24,308 | 22,769 | 22,297 |

**Conclusions and Discussion**

- After comparing the results from the experiments with the expected graphs, we can conclude that increasing the buffer size or block size significantly improves the performance of the external sort, especially in bigger files.
- When the number of blocks to merge increases, the cost remains the same in the majority of cases for both experiments and expected behavior.
- Applying external sorting to ordered files doesn't improve their performance

## IV. Overall Conclusion

We have learned some valuable lessons after the completion of this project, and not just about the implementation. The first learning point is about **choosing the right tool.** Initially we chose python because we are most familiar with it, however we did not consider that it might not be the most suitable language for this project. After implementing a few methods, we determine that java might be more suitable in terms of handling I/O. We could have saved some time if we properly analysed which language to use before starting this project.

Regarding the experiments, we find that the experimental observations match quite closely to the expected behavior we predicted. In terms of performance, memory mapping generally performs the best among the other algorithms. However, **there is no one best algorithm for all cases.** As explained to us by Professor Vansummeren during one of his lectures, each case may require a different algorithm that works the best. This is proven during this project, where buffered streaming (with low buffer size) performs the best for random reading.

Memory mapping appears to be a good choice for both reading/writing a large number of files or large files. The buffer size should be big enough to avoid overhead time for creating new buffer every time it is full. Buffered reading/writing can be considered as an easy way to improve the reading/writing performance since their implementation is not complicated as memory mapping. Reading/writing character by character should be avoided in most cases since it requires a huge number of I/Os.

For multi-way merge-sort, our experimental results show that **buffer size (or block size) is the main factor** in terms of performance. Contrary to our prediction, modifying the block merge does not affect performance by a large margin.

# V. References

1. J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. 2nd edition, 2009
2. https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files
3. https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html
4. https://davinci.ulb.ac.be/index.php/s/R7Jef6switwL32M
5. https://howtodoinjava.com/java7/nio/memory-mapped-files-mappedbytebuffer/