

# Database Systems Architectures Report

INFO H 417

Prof. Vansummeren Stijn

Project Title : Algorithms in Secondary Memory

6th January

Gjini Jurgen,

Hullebroeck Nathan,

Sefu Kevin

## Problem Statement and Environment

In this project we were asked to implement an external-memory merge-sort algorithm and to make a study of its performance under various parameters. The work has been divided into two parts firstly the exploration of several different ways to read data from, and write data to secondary memory and secondly the implementation of the External multi-way merge-sort algorithm. The main goal of this project was to give us real-world experience with the performance of external-memory algorithms and to help us to have our personal vision on this subject.

Our experiments were done on HP Notebook, processor intel(R) core(TM) i3-5005U CPU @ 2.00GHz ( OS : Windows) and Details Ordi Nathan ici -> +++ details cpu etc ..

We decided to use Java.

+++ external librarie ????

++++ how data was generated and size

## Background reading and writing: streams

Our merge sort algorithm was supposed to be able to sort disk files containing 32 bit integers. To achieve this goal our implementation needed to read data from, and write data to disk. The first step was the development of stream classes that could sequentially read and write a le consisting of 32-bit integers. Below the java classes we used for reading/writing and a brief definition of them.

**FileInput/OutputStream :** The FileInputStream class obtains input bytes from a file in a file system.

This class is meant for reading streams of raw bytes.

The FileOutputStream is a class meant for writing streams of raw bytes.

**DataInput/OutputStream :** The DataInput/OutputStream class allows to read/write primitive Java data types from/to an input stream.

**BufferedInput/OutputStream :** The BufferedInputStream class adds functionality to another input stream-namely, the ability to buffer the input by the creation of an internal buffer.

**FileChannel :** The FileChannel class provide a channel that is connected to a file. Using a file channel allow the reading/writing of data from/to a file.

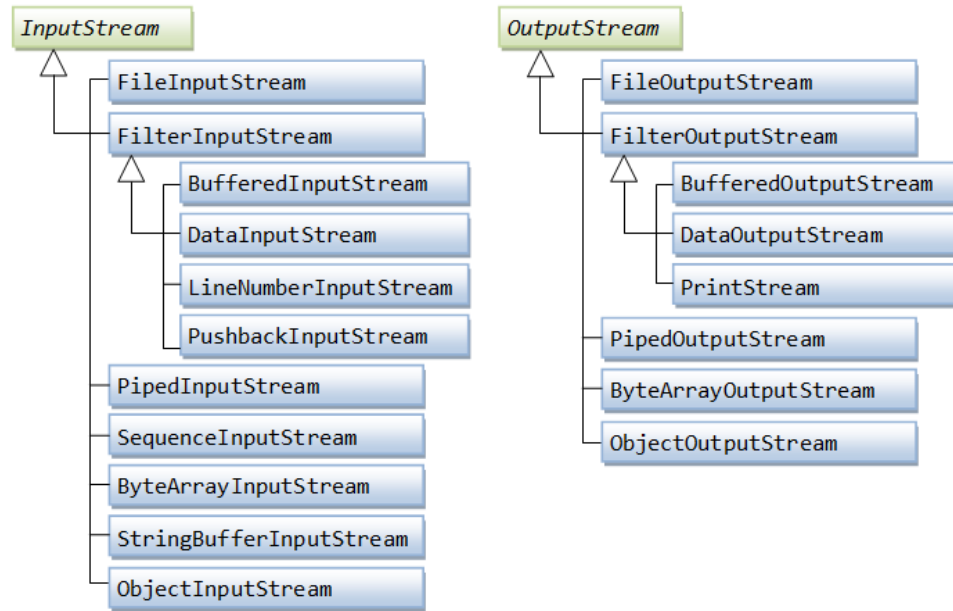


Figure 1: Input and Output Stream hierarchy

## Observations on stream

### 0.0.1 Expected behavior

**Mechanism 1** The first I/O mechanism use two classes of Java.io named FileInput/OutputStream and DataInput/OutputStream. This implementation use a DataInput/OutputStream that is wrapped around a FileInput/OutputStream.

Before running any experiment and regarding to the definitions seen in the previous section we thought that this mechanism will simply read/write streams of raw bytes and write/read primitive data types (int, char..) to/from a file.

+++ formule

**Mechanism 2** The implementation of the second mechanism is almost similar to the first one to the difference that in this second mechanism the java.io.DataInputStream object has been wrapped around a java.io.BufferedInputStream object that itself has been wrapped around a java.io.FileInputStream object. For this second mechanism we expected higher performances than the first one. In the sense that the 2nd mechanism used a BufferedInputStream object in order to improve performances by manipulating a restricted dataset.

+++ formule

**Mechanism 3** For the thrid mechanism we were asked to design an I/O mechanism and to equip it with our own buffer. For this aim we decided to use the java.nio.IntBuffer class in order to implement the buffer and we reused the same reading/writing model as in the mechanism 1.

At first we didn't see a big difference with the previous mechanism except the fact that, instead of using the internal buffer provided by the BufferedInputStream class we used our own buffer. Therefore we were expecting almost the same behavior than the 2nd mechanism.

+++ formule

**Mechanism 4** The last mechanism perform the read and the write by using memory mapping. In order to write a B element portion of the input file to internal memory we used the *map* method from `java.nio.channels.FileChannel` class.

As said in the previous lines, we used memory mapping. This concept consist in a direct assignation of a segment of virtual memory to some portion of a file or file-like resource. This correlation is a byte-for-byte relation and once it has been established, it allow applications to treat the mapped portion as if it were primary memory.

For this last mechanism, we didn't have particular expectations because we had never used the `FileChannel` class and memory mapping, this implementation was something new for us.

+++ formule

## 0.0.2 Observations after experimentations and researches

**Common observations on mechanism 1, 2, 3 :** We observed that the first three mechanisms used classes from `Java.io` while the fourth mechanism uses a class from `Java.nio`. Because of their attachment to `Java.io` the first three mechanisms have in common the fact that they are streams oriented which means that one or more bytes are read at a time, from a stream. Furthermore, we observed also that we can't move forth and back in the data in a stream to move forth and back in the data read from a stream, wa have to cache data in a buffer (mechanism 3).

We also discover that, `Java.io` streams are blocking meaning that when a thread invokes a `read()` or `write()` method, that thread is blocked until there is some data to read, or the data is fully written.

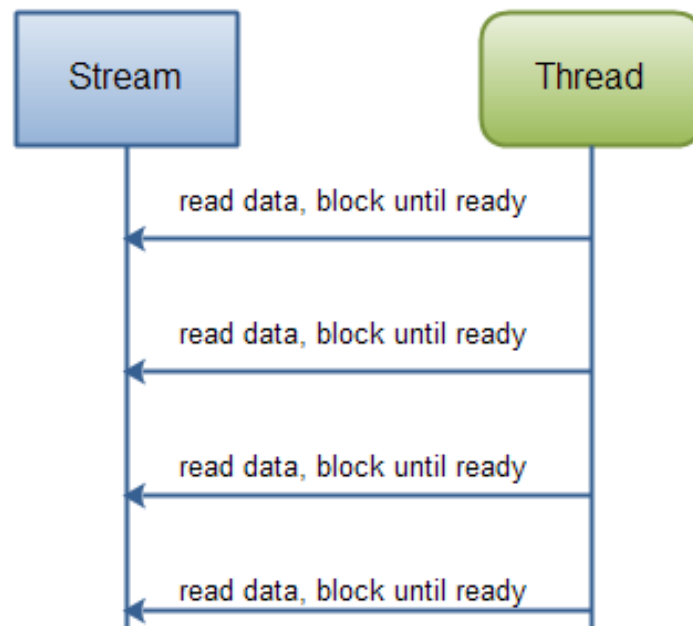


Figure 2: Java IO reading of data

**Mechanism 1** For this mechanism after experimentations and researchs we discovered that our first intuition wasn't very far from truth in the sense that a `DataInput/OutputStream` wrapped around a `FileInput/OutputStream` read/write bytes and read/write primitive java data type :

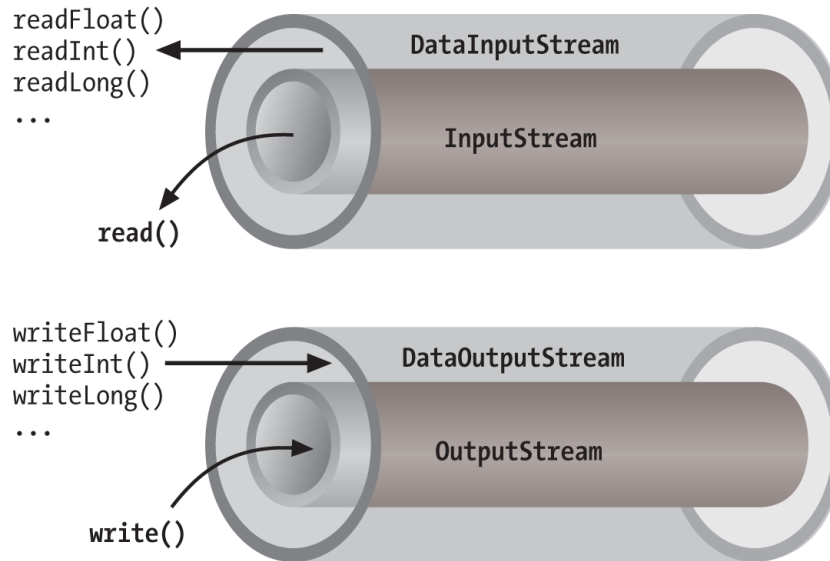


Figure 3: Mechanism 1

**Advantages and disadvantages** The `DataInputStream` wrapped around a `InputStream` is handy if the data we need to read/write consists of Java primitives larger than one byte each, like `int`, `float` etc.

But a big disadvantage of this method is the fact that we are reading one byte at a time from a file, this can be very slow because of the huge number of reading/writing instructions.

**Mechanism 2** Experimentation and researches we made on this mechanism though us that the `BufferedInputStream` class is a kind of `InputStream` that reads data from a stream and uses a buffer to optimize speed access to data. Data is basically read ahead of time and this reduces disk access. We also observed by experimentation that, this mechanism is faster than the previous one. In this mechanism, input byte are split into blocks of bytes in order to optimize as said before the access to data and to reduce the number of low-level I/O operations performed.

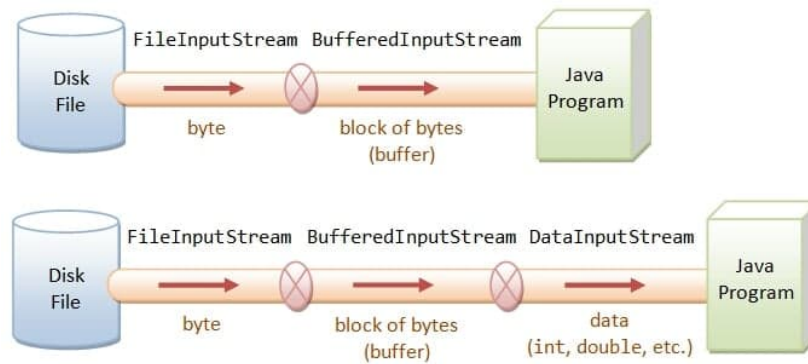


Figure 4: Mechanism 2

**Advantages and disadvantages** Regarding advantages and disadvantages we can emphasize the fact that the bufferization allow us to solve the big problem of slowness that can be caused by the reading/writing of one byte at a time. Rather than read/writing one byte at a time from the disk, the `BufferedInputStream`

reads a larger block at a time into an internal buffer. We can also add to mechanism two's advantages the fact that :

1. The storage of data into blocks of byte, speed up the processing of input/output by reducing the number of reading and writing instructions
2. It more efficient than the mechanism 1 for large files (over 100MB)

A disadvantage that could be retained concerning this method, is the fact that its may require a huge number of experiments with different buffer sizes to find out which buffer size seems to produce the best performance on a specific hardware. The optimal buffer size may depend on the concrete hardware in the computer. For instance if the hard disk is anyways reading a minimum of 4KB at a time, it's foolish to use less than a 4KB buffer.

**Mechanism 3** By using an `java.nio.IntBuffer` this mechanism create a buffer in/from which data can be write/read. In java, buffers has two mode, the reading mode and the writing mode. The transition from the reading mode to the writing mode ( and write-to-read ) is done by using the `flip ()` method. Whenever the buer becomes empty/full a portion of buffer size is read/written from/to the le.

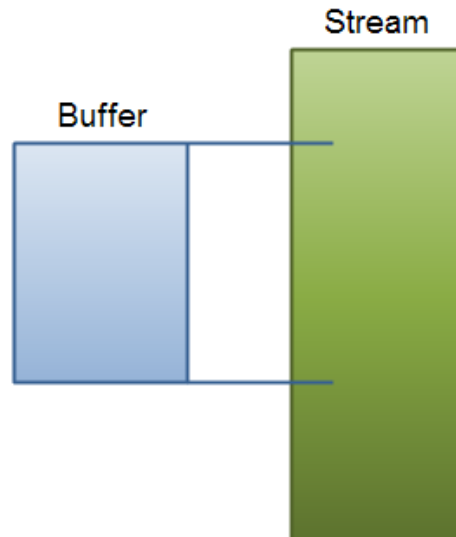


Figure 5: Mechanism 3

The following image explain what we discover from our experimentation with `IntBuffer`. The **capacity** label determine the size of the buffer. We can only write capacity bytes, longs, chars etc. into the Buffer. Once the Buffer is full, you need to empty it before writing into it again. The label **position** determine the position from which that will be read or write and finally the **limit** label determine the limit of reading/writing. In write mode the limit is equal to the capacity of the Buffer while in the reading mode the limit means the limit of how much data you can read from the buffer. Therefore the limit label is set to write position of the write mode. In other words we could only read as many bytes as were written.

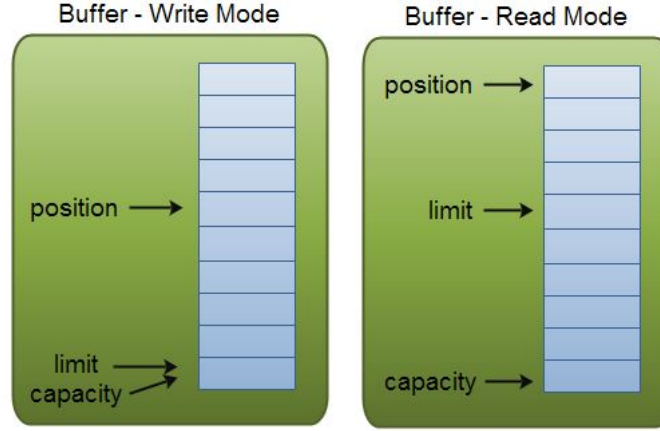


Figure 6: Mechanism 3

**Mechanism 4** We discover that this mechanism use the principle of memory mapping. This mapping between a file and memory space enables applications, including multiple processes, to modify the file by reading and writing directly to the memory. Memory mapping increase I/O performances because accessing memory mapped files is faster than using direct read and write operations, a system call is orders of magnitude slower than a simple change to a program's local memory.

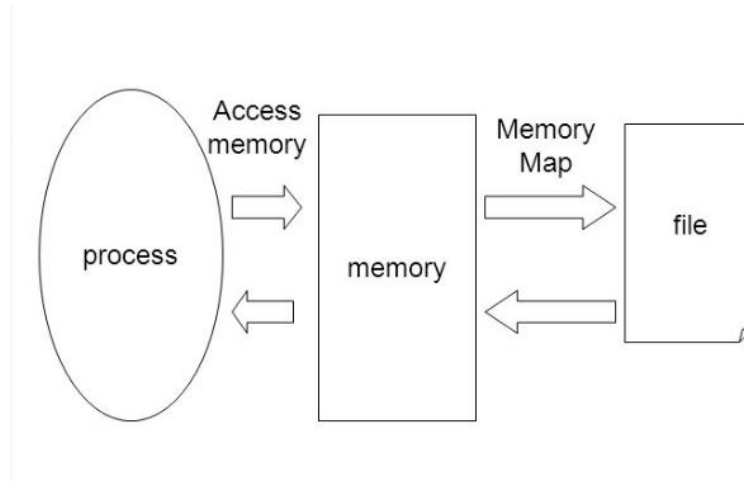


Figure 7: Mechanism 4

**Advantages and disadvantages** As said before, this method increase I/O performances by using the principle explained in the lines above but one disadvantage that could be retained about this method is the fact that a file section larger than the addressable space can have only portions mapped at a time, this may complicate the reading process.

### 0.0.3 Discussion of expected behavior vs experimental observations

We have found that in a general, experimentations and implementations of these mechanisms helped us to learn many new things (Mechanism 4). The knowledge we had about these different mechanisms was mostly

incomplete (Mechanism 4) or partial (Mechanism 2, 3). The implementation and research that we have done on these modes of writing and reading have allowed us to improve our understanding of the internal behavior of each mechanism.

## **Optimum Solution and Process of this discovery**

### **Sources**

1. Database Systems: The Complete Book (second, international edition)" by H. Garcia-Molina, J. D. Ullman, and J. Widom (ISBN-13: 978-0131354289)
2. Java™ Platform, Standard Edition 7 API Specification
3. Figure 1 : Input and Out Stream hierarchy
4. Figure 2: Java Io reading of data
5. Figure 3: Mechanism 1
6. Figure 4: Mechanism 2
7. Figure 5 : Mechanism 3
8. Figure 6 : Mechanism 3
9. Figure 4 : Mechanism 4
10. Jenkov Aps