Databases (67506)

Exercise 3: External Merge Sort And Selection
Due date: Monday, 3.12.2018, 23:55

Exercise Goal: In this exercise, you will experience, hands-on, the problem of manipulating data that is too large to fit into the main memory. In the Part A of the exercise, you will implement the external merge sort algorithm, as taught in class. In Parts B-D, you will consider the problem of both selecting and sorting data. In your programming and analysis, you will show how “pushing” the selection operator, i.e., performing selection early on when sorting, is an important optimization technique.

General Information: Note that you will be provided with code (appearing in Appendix B of this document, and available for download on the course website), and your goal will be to implement three functions appearing in this code.

Throughout this exercise, we will assume that the input table to your program is provided as a text file. Each line of the input file contains a series of lines (tuples) each of which contains several space-delimited columns (attributes). A line may also have only one column. All lines have the same number of columns, and each column is a string of length 20.

To generate such inputs, you can download the utility fileGenerator.jar from the course website. To generate a file, use the command:

```
java -jar path_of_utility output_file number_of_columns number_of_lines
```

where path_of_utility is the path for the utility you downloaded from Moodle, output_file is the pathname of the file that is being generated, and number_of_columns and number_of_lines are positive integers. You'll then have a file with *number_of_lines* lines, where every line is a sequence of *number_of_columns* space-delimited columns, and each column is a string of 20 random [a-z] characters (there will be no digits or capital letters, for example).

For example, using number_of_columns 3 and number_of_lines 3, you may derive the file:

```
abcdefghijklmnopqrst aabbccddeeffgghhiijj tsrqponmlkjihgfedcba
aaaaaaaaaaaaaaaaaaaa zzzzzzzzzzwwwwwwwwww iifgmnpewrjlmvnsdper
jasdnasiasjkoekdjooo asldjwocmlaooppwkknn aswuunsyhkowhtysrrrr
```

Important: You will test your program on large inputs (up to size 1GB). If you develop your program on the lab computers, you probably will not have enough space in your account to store such input files (or the output file generated, or the temporary files created while running external sort). To overcome this problem, you should use the directory: **/tmp**

Note that this is an absolute path, and files in this directory are erased once in a while. Therefore, you can store inputs / outputs in this directory, but you **cannot** store your code in this directory.

Part A: External Merge Sort (50 points)

In Part A, you will implement external merge sort in Java. Your program will receive, as input, a text file, of the above described format, to sort. Your program will also receive a column number that determines which column should be used for sorting. The program you write should correctly sort this file by some column so that the output contains the same lines as in the input file, but in ascending lexicographic order (of the sorted column).

The method's signature you will implement is:

sort(String in, String out, int colNumSort, String tmpPath)

in – the pathname of the file to read from (pathname includes the file name).
out – the pathname of the file to write to (pathname includes the file name).
colNumSort – sort input file by this column number in lexicographic order.
tmpPath – the path for saving temporary files.

For example, if your input file is:

```
abcdefghijklmnopqrst aabbccddeeffgghhiijj tsrqponmlkjihgfedcba
aaaaaaaaaaaaaaaaaaaa zzzzzzzzzzwwwwwwwwww iifgmnpewrjlmvnsdper
jasdnasiasjkoekdjooo asldjwocmlaooppwkknn aswuunsyhkowhtysrrrr
```

Calling the method sort on this file, with colNumSort 3 should result in the output file:

```
jasdnasiasjkoekdjooo asldjwocmlaooppwkknn aswuunsyhkowhtysrrrr
aaaaaaaaaaaaaaaaaaaa zzzzzzzzzzwwwwwwwwww iifgmnpewrjlmvnsdper
abcdefghijklmnopqrst aabbccddeeffgghhiijj tsrqponmlkjihgfedcba
```

Your program should be able to sort 1GB input files in less than 5 minutes (Note that an efficient program can also accomplish this in less than a minute).
If you use your own development environment, and not the one at the computers in the lab, make sure before submitting that your program meets this requirement at the computers in the lab, since we will test it there.

When running your program, we will limit the size of the RAM that can be utilized by your program, usually to 50MB. This is accomplished by running your program with the following command (from the command line):

```
java -jar -Xmx50m -Xms50m ExternalMemory.jar A InputToSort.txt
SortedOutput.txt ColumnNumber TmpFolder
```

In class, when presenting the external sort algorithm, we assumed that we knew the block size, as well as the number of free blocks available in internal memory. When developing your program, it will be hard for you to determine these values precisely. For example, the block size depends on both the file system and the operating system.

Instead of determining these values precisely, we recommend that you start with some assumptions on the block size and the number of available blocks, and code your program using these predetermined values. You can then adjust them to see if this increases or decreases the runtime, until achieving results within the required time bound.

For example, you can start by assuming that each line needs X bytes in memory (where X is determined by taking the length of the line, which you can check at the beginning of the program, multiplied by the number of bytes that a character occupies, which is 2 bytes). Then assume that the size of a block is Y bytes (thereby determining how many rows fit into a block) and that you have M blocks in the main memory. For example, you can start with M = 1000, Y = 20000, and increase/decrease these values according to the time it took your program to run.

With these initial values, you do not use the entire 50MB that your program can use. It is actually essential to leave enough memory space (for the proper functioning of the JVM, for example). How much memory you should keep aside, it is something that cannot be determined apriori. Note that these example values certainly don’t have to be the values you’ll use, but this can be a starting point from which you can adjust the values until your program will meet the above-mentioned time constraint. This is mainly a matter of trial and error.

Notes:

1. We number the columns from one rather than zero. For instance, in the example above, we sorted column three and not column two. The reason for that is that you will check your code’s correctness using shell commands that also start numbering columns from one.
2. If two (or more) rows have the same value on the sorted column, you can order them in your output file as you want. We will accept any legal result. This situation should be rare when generating an input file using our utility (as strings are generated randomly), but if it happens, you may wonder why it seems that your result is wrong even though it may be absolutely correct (see below for how you can test your code's correctness). In this case, we recommend you generate a different input file.
3. There will be no empty line at the end of the input file, and you should not create an empty line in your output file.

Testing your code (part A):

To check that your code works correctly, you can use the shell command “sort” to sort your file. Then, you can check if your output is the same as that of the sort command using the shell command “diff.”

For example:

```
sort inputFile -k columnNumber
```

will sort lines that are in inputFile by its <columnNumber>th column.
Then, you can use diff:

diff sortedFile mySortedFile

for checking differences between the files (correctly implemented code should cause the same result. Thus, there should be no differences).
Notice that *sort* does not use zero-based indexing (it refers to the first column as 1 and not as 0, as we do in this exercise).

Part B: External Merge Sort With Selection (10 points)

In this part, you will implement a functionality of sort with a selection condition in a naive manner. To achieve this, in this section you will **only** implement the selection operator. Sorting with selection will then be performed by first sorting, and then selecting.

Your program will receive as input a text file (as before), and your program should correctly sort this file by some column, in lexicographic order, but this time you should output only lines that fulfill a specified condition. The lines selection is based on two values the method gets: a column number *i* (some positive integer) and a sequence of characters. For every line (in the input file) you should check whether its *i*-th column contains the given sequence of characters. If so, your output file should include this line; otherwise, it should not.

We have provided an implementation of a method called sortAndSelect that uses two methods: the first one is 'sort' that you've implemented in part A, and the second is 'select' that you will implement in this part of the exercise.

The method's signature you will implement is:
select(String in, String out, int colNumSelect, String substrSelect, String tmpPath)

in – the pathname of the file to read from (pathname includes the file name).
out – the pathname of the file to write to (pathname includes the file name).
tmpPath – the path for saving temporary files.
colNumSelect – the column number for the selection operation.
substrSelect – a string to check whether it's a substring of the given column in a line.

For example, if your input file is:

```
abcdefghijklmnopqrst aabbccddeeffgghhiijj tsrqponmlkjihgfedcba
aaaaaaaaaaaaaaaaaaaa zzzzzzzzzzwwwwwwwwww iifgmnpewrjlmvnsdper
jasdnasijsjkoekdjooo asldjwocmlaooppwkknn aswuunsyhkowhtysrrrr
```

and we would like to select lines that contain the string "ij" in their first column (colNumSelect = 1, substrSelect = "ij"), the output of select would be the file

```
abcdefghijklmnopqrst aabbccddeeffgghhiijj tsrqponmlkjihgfedcba
jasdnasijsjkoekdjooo asldjwocmlaooppwkknn aswuunsyhkowhtysrrrr
```

Therefore, calling our sortAndSelect method with colNumSelect = 1, substrSelect = “ij” and colNumSort = 3, your output should be:

```
jasdnasijsjkoekdjooo asldjwocmlaooppwkknn aswuunsyhkowhtysrrrr
abcdefghijklmnopqrst aabbccddeeffgghhiijj tsrqponmlkjihgfedcba
```

As we did before, when running your program, we will limit the size of the RAM that can be utilized by your program to 50MB, with the following command:

```
java -jar -Xmx50m -Xms50m ExternalMemory.jar B Input.txt Output.txt
ColumnNumberSort TmpFolder ColumnNumberSelect SubstringSelect
```

Part C: Efficient External Merge Sort With Selection (25 points)

In this part, you will implement an efficient external merge sort with a selection. Your program will receive the same input as you got in part B, and your output file should be the same result as your result in part B, but this time we’re expecting you to implement the selection and the sorting efficiently, by filtering out the lines that will not be in the final result during the first stage of sorting (i.e., when creating the sorted sequences of stage 1 of the algorithm).

The method’s signature you will implement is:

sortAndSelectEfficiently(String in, String out, int colNumSort, String tmpPath, int colNumSelect, String substrSelect)

in – the pathname of the file to read from (pathname includes the file name).
out – the pathname of the file to write to (pathname includes the file name).
colNumSort – sort input file by this column number in lexicographic order.
tmpPath – the path for saving temporary files.
colNumSelect – the column number for the selection operation.
substrSelect – a string to check whether it’s a substring of the given column in a line.

Again as before, when running your program, we will limit the size of the RAM that can be utilized by your program to 50MB, with the following command:

```
java -jar -Xmx50m -Xms50m ExternalMemory.jar C Input.txt Output.txt
ColumnNumberSort  TmpFolder ColumnNumberSelect SubstringSelect
```

Testing your code (parts B and C):

As before (in part A), you will use shell commands *sort* and *diff*. This time you may want to use *awk* as well.

For example, if you want to filter lines in "input.txt" that their first column contains the string "xyz" and then sort them by their third column, you should use:

```
awk '$1 ~ /xyz/' input.txt | sort -k 3
```

You can then, as before, save this result and use *diff* to compare it with your result. Notice that *awk* does not use zero-based indexing.

Part D: Comparing Performances (15 points)

The goal of this part is to compare the performance of your naive sort and select (Part B) with pushing the selection into the sort (Part C). For this comparison, you should create and submit three different graphs. Each graph should display the time it took your program to sort and select (y-axis) plotted against the size of the input file (x-axis). The length of the string the selection operation is based on, varies from one graph to the other:

- In graph 1, the selection is made with a condition of length 1 (a single letter).
- In graph 2, the selection is made with a condition of length 4.
- In graph 3, the selection is made with a condition of length 7.

(You can choose the contents of the strings for the conditions according to your own preference.)

Each graph should have two lines: one for the execution time of sortAndSelect (part B) and the second for the execution time of sortAndSelectEfficiently (part C).

The graphs should contain the results of the time to compute for the input sizes 250MB, 500MB, 750MB and 1GB (approximately).

Note that you are using the same condition for both parts when checking with a particular input file (that is, if you check part B execution time with a specific input file, column number and substring, use the same arguments when checking part C execution time).

Make sure the graph is clear: axes have units, lines are distinguishable, and each can be associated with one of the exercise parts (clear legend, etc.)

For this part of the exercise, you will submit a PDF file. The file should contain the following information:

1. At the top of the file, one or two lines (depending on whenever you are a pair or a single student) that contains your IDs and usernames (as in README, see below).
2. The three graphs you created as explained above.
3. Below each graph, specify the following details:
    a. for each input file - the file's size, and the number of lines and number of columns it has.
    b. for each input file - the column number that the input file was sorted by, and the condition you used (column number and content of the string).

For example, one may write below its first graph:

File size of 250MB, 1000 lines, 5 columns
ColumnNumberSort=1, ColumnNumberSelect=1, SubstringSelect='a'

File size of 500MB, 2000 lines, 5 columns
ColumnNumberSort=1, ColumnNumberSelect=1, SubstringSelect='b'

File size of 750MB, 3000 lines, 5 columns
ColumnNumberSort=1, ColumnNumberSelect=1, SubstringSelect='c'

File size of 1GB, 4000 lines, 5 columns
ColumnNumberSort=1, ColumnNumberSelect=1, SubstringSelect='d'

if you use the same ColumnNumberSort, ColumnNumberSelect, and SubstringSelect for testing all files, there is no need to write them more than once.

Exercise Submission

You are provided with a template to use (appendix B). It is available on the course website. You will **only submit** the file ExternalMemoryImpl.java, so this is the only file you should change. Note that you will not submit jar file. Rather, we will compile the jar from your Java source. For your tests, you can either compile a JAR and run one of the commands above, run Java directly on the compiled class files or configure your IDE. Instructions for IntelliJ can be found here.

Your submission should be a zip file containing three files: README, ExternalMemoryImpl.java and a pdf file.

The README file will have only one or two lines, depending on whenever you are a pair or a single student. The format of the README file should be

```
ID Username
ID Username
```

For example:

```
491395749 ike5894
823481094 mike
```

Do not add any whitespace before the ID number. When doing this exercise in pairs, only one member of the pair should submit the exercise.

General Notes, Tips and Requirements::

- A character in Java occupies 2 bytes.
- Java objects incur memory overhead. The precise amount of memory is JVM dependent.
- Take a look at the BufferedReader and BufferedWriter classes. These can be very helpful in your implementation.

- Make sure your code is as modular as possible, since you may want to use parts of your code in different parts of the exercise.
- As we wrote above, you do not have enough disk space in your personal directory (at the computers in the lab) for huge files, but there is a directory with read and write permissions you can use instead: /tmp
  Notice that this is an absolute path, and files in this directory are erased once in a while.
- Your program **must delete** all temporary files you've created. Take a look at the File class documentation (createTempFile and deleteOnExit may be helpful).
- Your program should support both absolute and relative paths (for the arguments: input, output, and temp). For example, you should be able to pass either "/tmp" or "./tmp" to your program for the tempFolder argument (if both paths exist, of course). If you're not familiar with these concepts, you may find appendix A helpful.

In your code you are not allowed to do any of the following:

- Use memory mapped files.
- Call shell commands or any other utilities from your code.
- Use external libraries that are not provided with Java.
- Use any code that was not written by yourself. In particular, you cannot use code from the Internet.

Good luck!

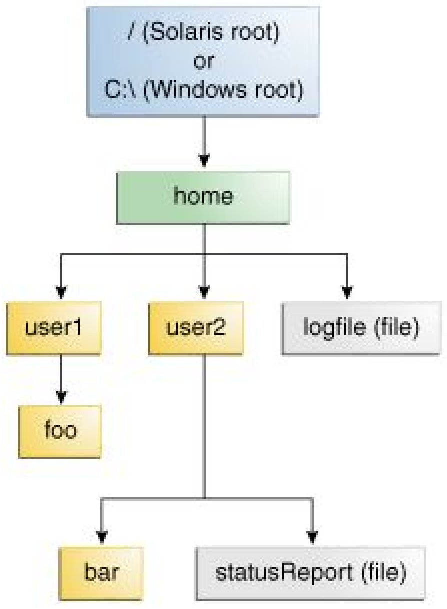
Appendix A:

A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved. Most file systems in use today store the files in a tree (or *hierarchical*) structure. At the top of the tree is one (or more) root nodes. Under the root node, there are files and directories (*folders* in Microsoft Windows). Each directory can contain files and subdirectories, which in turn can contain files and subdirectories, and so on, potentially to an almost limitless depth.

What Is a Path?

The following figure shows a sample directory tree containing a single root node. Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as C:\ or D:\. Linux supports a single root node, which is denoted by the slash character /

In the next figure, Solaris OS is mentioned. This is because this appendix was taken from a source that talks about Solaris, but in the context that we'll talk about here (paths, hierarchical structure, etc.), this is also true for Linux that you're probably more familiar with.



A file is identified by its path through the file system, beginning from the root node. For example, the statusReport file in the previous figure is described by the following notation in Linux:
/home/sally/statusReport

In Microsoft Windows, statusReport is described by the following notation:

C:\home\sally\statusReport

The character used to separate the directory names (also called the *delimiter*) is specific to the file system: The Linux OS uses the forward slash (/), and Microsoft Windows uses the backslash slash (\).

Relative or Absolute?

A path is either *relative* or *absolute*. An absolute path always contains the root element and the complete directory list required to locate the file. For example, /home/sally/statusReport is an absolute path. All of the information needed to locate the file is contained in the path string.

A relative path needs to be combined with another path in order to access a file. For example, joe/foo is a relative path. Without more information, a program cannot reliably locate the joe/foo directory in the file system.

Access a file using File API

In order to access a file using File API, developers normally use the following constructor: File(String pathname)

This constructor accepts a file path as an argument, either relative or absolute.

For example, in Microsoft Windows:

File absoluteFile = new File("C:\\home\\sally\\statusReport");

File relativeFile = new File("/home/sally/statusReport");

Both objects refer to the same file, *absoluteFile* uses an absolute path while *relativeFile* uses a relative path assuming that our application exists on the C drive.

In Linux, we'll use it this way:

File absoluteFile = new File("/home/sally/statusReport");

File relativeFile = new File("home/sally/statusReport");

Again, both objects refer to the same file, assuming that our application was initialized from root (/).

Tip: When you work with a relative path and problems arise, print the returned value of: System.getProperty("user.dir")). This will print a complete absolute path from where your application was initialized and will help you debug your code.

Taken from:

https://docs.oracle.com/javase/tutorial/essential/io/path.html

https://www.programmergate.com/java-io-difference-absoluterelative-canonical-path/

Appendix B

main.java

```java
package edu.hebrew.db.external;

public class Main {

        public static void main(String[] args) {

                if (args.length != 5 && args.length != 7) {
                        System.out.println("Wrong number of arguments");
                        System.out.println("For part A: exercise_part "
                                        + "input_file output_file column_to_sort temp_path");
                        System.out.println("For parts B and C: exercise_part "
                                        + "input_file output_file column_to_sort"
                                        + " temp_path column_to_select substring_to_select");
                        System.exit(1);
                }

                String exercisePart = args[0];
                String in = args[1];
                String out = args[2];
                int columnToSort = Integer.valueOf(args[3]);
                String tmpPath = args[4];

                IExternalMemory e = new ExternalMemoryImpl();

                if (exercisePart.equals("A") || exercisePart.equals("a")) {
                        e.sort(in, out, columnToSort, tmpPath);

                } else {

                        int columnToSelect = Integer.valueOf(args[5]);
                        String substrToSelect = args[6];

                        if (exercisePart.equals("B") || exercisePart.equals("b"))
                                e.sortAndSelect(in, out, columnToSort, tmpPath, columnToSelect,
                                                substrToSelect);

                        else if (exercisePart.equals("C") || exercisePart.equals("c"))
                                e.sortAndSelectEfficiently(in, out, columnToSort, tmpPath,
                                                columnToSelect, substrToSelect);

                        else
                                System.out.println("Wrong usage: first argument"
                                                + "should be a, b, or c only!");
                }
        }
}
```

IExternalMemory.java

```java
package edu.hebrew.db.external;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public interface IExternalMemory {

        public void sort(String in, String out, int colNumSort, String tmpPath);

        public void select(String in, String out, int colNumSelect, String substrSelect,
                        String tmpPath);

        public void sortAndSelectEfficiently(String in, String out, int colNumSort,
                        String tmpPath, int colNumSelect, String substrSelect);

        /**
         * Do not modify this method!
         *
         * The method sorts the input file in a lexicographic order of a column, selects
         * rows according to the selection condition (colNumSelect and substrSelect), and
         * saves the result in an output file.
         *
         * @param in
         * @param out
         * @param colNumSort
         * @param tmpPath
         * @param colNumSelect
         * @param substrSelect
         */

        default public void sortAndSelect(String in, String out, int colNumSort,
                        String tmpPath, int colNumSelect, String substrSelect) {
                String outFileName = new File(out).getName();
                String tmpFileName = outFileName.substring(0, outFileName.lastIndexOf('.'))
                                + "_intermed" + outFileName.substring(outFileName.lastIndexOf('.'));
                String tmpOut = Paths.get(tmpPath, tmpFileName).toString();

                this.sort(in, tmpOut, colNumSort, tmpPath);
                this.select(tmpOut, out, colNumSelect, substrSelect, tmpPath);

                try {
                        Files.deleteIfExists(Paths.get(tmpPath, tmpFileName));
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

}
```

ExternalMemoryImpl.java (The only file you should change)

```
package edu.hebrew.db.external;

public class ExternalMemoryImpl implements IExternalMemory {

        @Override
        public void sort(String in, String out, int colNum, String tmpPath) {
                // TODO: Implement
        }

        @Override
        public void select(String in, String out, int colNumSelect,
                        String substrSelect, String tmpPath) {
                // TODO: Implement
        }

        @Override
        public void sortAndSelectEfficiently(String in, String out, int colNumSort,
                        String tmpPath, int colNumSelect, String substrSelect) {
                // TODO: Implement
        }

}
```