Exercise 3: External Merge Sort and Sort Merge Join

Due date: Monday, 16.12.2019, 23:55

For this exercise, submission is allowed in pairs!

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 Part A of the exercise, you will implement the `sort` method that performs a two-phase external merge sort, as taught in class. You will receive as input a file with the content R(A,B,C), and you will output the table sorted by the first column.
- In Part B, you will implement the `join` method that performs a sort-merge join. You will receive as input two files with the content of two **sorted** tables R(A,B,C) and S(A,D,E), and you will output the joined table RS(A,B,C,D,E).
- In Part C, you will consider the problem of selecting data. You will implement the `select` method that receives as input table RS(A,B,C,D,E) and a condition and outputs only tuples that satisfies the condition.
- In Part D, you will consider the problem of selecting and joining data efficiently. You will implement the `joinAndSelectEfficiently` method that receives tables R(A,B,C) and S(A,D,E) sorts them, and joins them, but this time, the selection condition is checked in the first reading of the files and not after the join.

The exercise will be tested by running the main program from a jar file `ExternalMemory.jar`. We provide the code for the main method of the jar file `ExternalMemory (Main.java)` as well as the abstract class `IExternalMemory.java`. In your solution you will be implementing four methods in the `ExternalMemoryImpl.java` file.

The different parts will be tested by running `ExternalMemory` with the part letter (A, B, C or D) as parameters, as will be detailed in each part.

Finally,

- In Part E you will perform runtime comparison. In your analysis, you will generate graphs that show how “pushing” the selection operator, i.e., performing selection early on when joining, is an important optimization technique. You will generate three graphs with different select conditions and show how runtime changes over different input sizes.

NOTE: This exercise ***must*** be implemented with the algorithms taught in class. If you have not yet reviewed and understood these algorithms, do so before beginning to solve this exercise.

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

Throughout this exercise, we will assume that the input tables to your program are provided as text files. For part A, a single file with the content of relation R(A,B,C) or S(A,D,E), and for part B-C, two files of relations R(A,B,C) and S(A,D,E) with a key-foreign_key relationship. Each input file contains a series of lines (tuples) each of which contains three space-delimited columns (attributes). All lines have three columns. The first column is an id attribute of the form "idx", where x is an integer, but the attribute is written as a string of length 10 (2 chars for id, 8 chars for x, padded with 0 from the left if needed), and each of the other columns is a string of length 20.

To generate such inputs, you can download the utility `FileGenerator.jar` from the course website. To generate files, use the command:

```
java -jar path_of_utility out1.txt num_lines1 out2.txt
num_lines2
```

where `path_of_utility` is the path for the utility you downloaded from Moodle, `out1`, `out2` are the pathnames of the first and second files that are being generated, and `num_lines1`, `num_lines2` are positive integers. The first column of `out1` is a key i.e., no idx value will appear more than once in this file. The first column of `out2` is a foreign-key of the first column in `out1`.

In both files, every line is a sequence of three space-delimited columns, the first column is an id of the form "idx", and each of the other columns is a string of 20 random [a-z] characters (there will be no digits or capital letters, for example).

For example, using the the following command will generate 2 files:

```
  java -jar FileGenerator.jar  file1.txt 4 file2.txt 10
```

file1.txt R(A,B,C):

```
id00000002 vqejvuuurqibgdnemdqp unogibxnauesnydlgwfa

id00000000 rklnnrtckohyjgqftite rqsutpiqwrmmesfgyakk

id00000001 kniwduhgdrrpbyyaclwn ajjapsstiqbkaqmldhmy

id00000003 asdbhasjkdbnkashdian ermdsfikewlxcvkdcvbf
```

file2.txt S(A,D,E) (continued on next page):

```
id00000002 frlsqprbmgommskretbb qrgfjmbpytoehgvudpyn

id00000000 fsjaejoxfnxeppotfsie xkkaqzueulmeaqmbohhf

id00000000 vktfofiasboyduoggkqf xltojrbtmjcfcuvnskkk

id00000000 fzcqbepaljfusgkbkmma bfpoejmukpfgliyrzxtx

id00000002 okfcywtilqqmzvqyvddd iwhszalobihvasgrpucp

id00000000 abweksgdcjtauejihnpu zerokcuoxnckhcacrgoe

id00000002 hftfvzacnmvjchtugscj tcddokonslqhixnfzzkw
```

```
id00000002 apeyvspbnrpkmncohxlt fbosmcscregsddzhjzro

id00000001 vkihtvfyklugpsvideht khnrismapslaartcunrm

id00000001 lwtbvigifhrvbkukoxrd zirxxnydqcdxvwbevytq
```

Note: The files are generated by a random process. Therefore, repeated applications of the same command may yield different results.

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: Two Phase External Merge Sort (35 points)

In Part A, you will implement external merge sort in Java according to the algorithm taught in class. In this exercise, we assume that the conditions for a two phase-external sort hold on the inputs.

Your program will receive, as input, a text file, with the content of a relation R(A,B,C) or S(A,D,E) of the above described format, to sort. The program you write should correctly sort this file by the **first** column so that the output contains the same lines as in the input file, but in ascending lexicographic order (of the sorted column).

If two (or more) rows have the same value on the sorted column, you should order them by the next column. This way, the result of sort is deterministic, and it will be easier for you to check that your program works correctly.

The method's signature you will implement is:

```
        sort(String in, String out, 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.

For example, if your input file is:

```
id00000002 vqejvuuurqibgdnemdqp unogibxnauesnydlgwfa
id00000000 rklnnrtckohyjgqftite rqsutpiqwrmmesfgyakk
id00000001 kniwduhgdrrpbyyaclwn ajjapsstiqbkaqmldhmy
id00000003 asdbhasjkdbnkashdian ermdsfikewlxcvkdcvbf
```

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

```
id00000000 rklnnrtckohyjgqftite rqsutpiqwrmmesfgyakk
id00000001 kniwduhgdrrpbyyaclwn ajjapsstiqbkaqmldhmy
id00000002 vqejvuuurqibgdnemdqp unogibxnauesnydlgwfa
id00000003 asdbhasjkdbnkashdian ermdsfikewlxcvkdcvbf
```

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 to 50MB. This is accomplished by running your program with the following command (from the command line):

```
java -Xmx50m -Xms50m ExternalMemory.jar A InputToSort.txt
                 SortedOutput.txt 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. For the purpose of this exercise, we will assume that the block size is 4KB, as standard in many systems. Note that you can assume that $\left\lceil \frac{B(R)}{M} \right\rceil < M$.

Notice that it is actually essential to leave enough memory space for the proper functioning of the JVM, for example, so do not use all of the 50MB as buffer. How much memory you should keep aside, it is something that cannot be determined apriori. This is mainly a matter of trial and error.

Note: 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 -k1 -k2 -k3 > sortedFile
```

will sort lines that are in inputFile by its first column, then by second and by third.

Then, you can use `diff`:

```
diff sortedFile mySortedFile
```

for checking differences between the files (correctly implemented code should create the same result. Thus, there should be no differences).

Part B: Sort-Merge Join (35 points)

In this part, you will implement the sort-merge join algorithm, using the two-phase external sort you implemented in part A.

Your program will basically perform the operation $R \bowtie S$.

Your join method will receive as input two text file, containing two **sorted** tables R(A,B,C) and S(A,D,E) with a key-foreign_key relationship, and your program should correctly join the tables on the first attribute, resulting with a table with the schema RS(A,B,C,D,E).

You will implement a join method. The method's signature you will implement is:

```
 join(String in1, String in2, String out, String tmpPath)
```

`in1` – the pathname of the first sorted file to read from containing R (pathname includes the file name).

`in2` – the pathname of the second sorted file to read from containing S (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.

For example, if your sorted input files are:

file1   R(A,B,C)

```
id00000000 rklnnrtckohyjgqftite rqsutpiqwrmmesfgyakk

id00000001 kniwduhgdrrpbyyaclwn ajjapsstiqbkaqmldhmy

id00000002 vqejvuuurqibgdnemdqp unogibxnauesnydlgwfa

id00000003 asdbhasjkdbnkashdian ermdsfikewlxcvkdcvbf
```

file2   S(A,D,E)

```
id00000000 abweksgdcjtauejihnpu zerokcuoxnckhcacrgoe

id00000000 fsjaejoxfnxeppotfsie xkkaqzueulmeaqmbohhf

id00000000 fzcqbepaljfusgkbkmma bfpoejmukpfgliyrzxtx

id00000000 vktfofiasboyduoggkqf xltojrbtmjcfcuvnskkk

id00000001 lwtbvigifhrvbkukoxrd zirxxnydqcdxvwbevytq

id00000001 vkihtvfyklugpsvideht khnrismapslaartcunrm

id00000002 apeyvspbnrpkmncohxlt fbosmcscregsddzhjzro

id00000002 frlsqprbmgommskretbb qrgfjmbpytoehgvudpyn

id00000002 hftfvzacnmvjchtugscj tcddokonslqhixnfzzkw

id00000002 okfcywtilqqmzvqyvddd iwhszalobihvasgrpucp
```

and we would like to join the files, the output of join would be the file

```
id00000000 kheucdhfiyqpobdujedw aqyqprwmktufflmmrxyf abweksgdcjtauejihnpu zerokcuoxnckhcacrgoe
id00000000 kheucdhfiyqpobdujedw aqyqprwmktufflmmrxyf fsjaejoxfnxeppotfsie xkkaqzueulmeaqmbohhf
id00000000 kheucdhfiyqpobdujedw aqyqprwmktufflmmrxyf fzcqbepaljfusgkbkmma bfpoejmukpfgliyrzxtx
id00000000 kheucdhfiyqpobdujedw aqyqprwmktufflmmrxyf vktfofiasboyduoggkqf xltojrbtmjcfcuvnskkk
id00000001 ajkwadbmrfosxqdfujpu vyamwsxicqqduptgiiqk lwtbvigifhrvbkukoxrd zirxxnydqcdxvwbevytq
id00000001 ajkwadbmrfosxqdfujpu vyamwsxicqqduptgiiqk vkihtvfyklugpsvideht khnrismapslaartcunrm
id00000002 gwsnbgekwfwhabsfckve lvtlnegsbgzrkzmbwzwn apeyvspbnrpkmncohxlt fbosmcscregsddzhjzro
id00000002 gwsnbgekwfwhabsfckve lvtlnegsbgzrkzmbwzwn frlsqprbmgommskretbb qrgfjmbpytoehgvudpyn
id00000002 gwsnbgekwfwhabsfckve lvtlnegsbgzrkzmbwzwn hftfvzacnmvjchtugscj tcddokonslqhixnfzzkw
id00000002 gwsnbgekwfwhabsfckve lvtlnegsbgzrkzmbwzwn okfcywtilqqmzvqyvddd iwhszalobihvasgrpucp
```

In order to join files that are not yet sorted, we have provided an implementation of a method called `sortAndJoin` that uses two methods: the first one is `sort` from Part A, and the second is `join` that you will implement in this part of the exercise.

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 -Xmx50m -Xms50m ExternalMemory.jar B Input1.txt
Input2.txt Output.txt TmpFolder
```

Testing your code part B:

As before (in part A), you can use a shell command: `join` that joins two sorted files to test your solution. For example, if you want to join the sorted files `file1` and `file2` on the first column of each, you should use:

```
join -1 1 -2 1 file1 file2
```

You can then, as before, save this result and use `diff` to compare it with your result.

Part C: Sort Merge Join With Selection (10 points)

In this part, you will implement a functionality of join with a selection condition in a naive manner. To achieve this, in this section you will **only** implement the `select` method. Joining with selection will then be performed by the provided `joinAndSelect` that calls `sortAndJoin` and `select`.

The `joinAndSelect` method receives as input two text files (as before), and will correctly join these files by the first column, but this time the output will only contain lines that fulfill a specified condition. The lines selection is based on one value the method gets: a sequence of characters. For every line (in the input file) you should check whether its *first* 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 `joinAndSelect` that uses two methods: the first one is `sortAndJoin` from Part B, 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, String str, 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).
`str` – a string to check whether it's a substring of the first column in a line.
`tmpPath` – the path for saving temporary files.

For example, if your input file is:

```
id00000002 frlsqprbmgommskretbb qrgfjmbpytoehgvudpyn

id00000000 fsjaejoxfnxeppotfsie xkkaqzueulmeaqmbohhf

id00000000 vktfofiasboyduoggkqf xltojrbtmjcfcuvnskkk

id00000000 fzcqbepaljfusgkbkmma bfpoejmukpfgliyrzxtx

id00000002 okfcywtilqqmzvqyvddd iwhszalobihvasgrpucp

id00000000 abweksgdcjtauejihnpu zerokcuoxnckhcacrgoe

id00000002 hftfvzacnmvjchtugscj tcddokonslqhixnfzzkw

id00000002 apeyvspbnrpkmncohxlt fbosmcscregsddzhjzro

id00000001 vkihtvfyklugpsvideht khnrismapslaartcunrm

id00000001 lwtbvigifhrvbkukoxrd zirxxnydqcdxvwbevytq
```

and we run `select("file1", "file1s", "2", "/tmp")`
the output of select would be the file

```
id00000002 frlsqprbmgommskretbb qrgfjmbpytoehgvudpyn

id00000002 okfcywtilqqmzvqyvddd iwhszalobihvasgrpucp

id00000002 hftfvzacnmvjchtugscj tcddokonslqhixnfzzkw

id00000002 apeyvspbnrpkmncohxlt fbosmcscregsddzhjzro
```

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 -Xmx50m -Xms50m ExternalMemory.jar C Input1.txt
    Input2.txt Output.txt substrSelect TmpFolder
```

Part D: Efficient Sort Merge Join With Selection (10 points)

In this part, you will implement an efficient sort-merge join with a selection. You will implement the method `joinAndSelectEfficiently`, that will receive the same input as you got in part C for `joinAndSelect`, and your output file should be the same result as your result in part C, but this time we're expecting you to implement the selection and the join efficiently, by selecting while reading the files for the first time.

The method's signature you will implement is:

```
joinAndSelectEfficiently(String in1, String in2, String
        out, String substrSelect, String tmpPath)
```

`in1` – the pathname of the first file to read R from (pathname includes the file name).

`in2` – the pathname of the second file to read S from (pathname includes the file name).

`out` – the pathname of the file to write to (pathname includes the file name).

`substrSelect` – a string to check whether it's a substring of the first column in a line.

`tmpPath` – the path for saving temporary files.

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 -Xmx50m -Xms50m ExternalMemory.jar D Input1.txt
    Input2.txt Output.txt substrSelect TmpFolder
```

Testing your code (parts C and D):

As before (in parts A-B), you will use shell commands `sort`, `join` 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 first column, you should use:

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

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 (i.e., the first column is column number 1).

Part E: Comparing Performance (10 points)

The goal of this part is to compare the performance of your naive join and select (Part C) with pushing the selection into the join (Part D). For this comparison, you should create and submit three different graphs. Each graph should display the time it took your program to join and select (y-axis) plotted against the size of the input files (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 digit).
- In graph 2, the selection is made with a condition of length 3.
- In graph 3, the selection is made with a condition of length 5.

(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 `joinAndSelect` (part C) and the second for the execution time of `joinAndSelectEfficiently` (part D).

The graphs should contain the results of the time to compute for the input sizes

1. file1 with 50,000 rows, file 2 with 5,000,000 rows (approximately 250MB).
2. file1 with 50,000 rows, file 2 with 10,000,000 rows (approximately 500MB).
3. file1 with 50,000 rows, file 2 with 15,000,000 rows (approximately 750MB).
4. file1 with 50,000 rows, file 2 with 20,000,000 rows (approximately 1GB).

Note that you are using the same condition for both parts when checking with a particular input file (that is, if you check part C execution time with a specific input file and range, use the same arguments when checking part D 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 need to submit three image files. Name them: plot1, plot3, plot5 (choose any common image extension you'd like: jpeg, png, etc.)

For example, you may submit:

plot1.jpeg

plot3.jpeg

plot5.jpeg

Exercise Submission

You are provided with a template to use (Appendix A). 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 a 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 five files:
`README`, `ExternalMemoryImpl.java` and three image files.

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

ID1 Username1

ID2 Username2

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, and add the other member as a partner via moodle.

<u>General Notes, Tips and Requirements:</u>

- 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).

In your code you are <u>not</u> 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:

Main.java

```java
package Join;

public class Main {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		if (args.length != 4 && args.length != 5 && args.length != 6) {
			System.out.println("Wrong number of arguments");
			System.out.println("For part A: exercise_part "
					+ "input_file output_file temp_path");
			System.out.println("For part B: exercise_part "
					+ "input_file1 input_file2 output_file"
					+ " temp_path");
			System.out.println("For part C: exercise_part "
					+ "input_file1 input_file2 output_file"
					+" substrSelect temp_path");
			System.out.println("For part D: exercise_part "
					+ "input_file1 input_file2 output_file"
					+" substrSelect temp_path");
			System.exit(1);
		}
		String exercisePart = args[0];
		IExternalMemory e = new ExternalMemoryImpl();

		if (exercisePart.equals("A") || exercisePart.equals("a")) {
			String in = args[1];
			String out = args[2];
			String tmpPath = args[3];
			e.sort(in, out, tmpPath);
		} else if (exercisePart.equals("B") || exercisePart.equals("b")){
			String in1 = args[1];
			String in2 = args[2];
			String out = args[3];
			String tmpPath = args[4];
			e.sortAndJoin(in1,in2, out, tmpPath);
		}
		else if (exercisePart.equals("C") || exercisePart.equals("c")){
			String in1 = args[1];
			String in2 = args[2];
			String out = args[3];
			String substrSelect=args[4];
			String tmpPath = args[5];
			e.joinAndSelect(in1, in2, out, substrSelect, tmpPath);
		}
		else if (exercisePart.equals("D") || exercisePart.equals("d")){
			String in1 = args[1];
			String in2 = args[2];
			String out = args[3];
			String substrSelect=args[4];
			String tmpPath = args[5];
			e.joinAndSelectEfficiently(in1, in2, out, substrSelect,tmpPath);
		}
		else
			System.out.println("Wrong usage: first argument"
					+ "should be a, b, c OR d only!");
	}
}
```

IExternalMemory.java

```java
package Join;

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

public abstract class IExternalMemory {

	public abstract void sort(String in, String out, String tmpPath);

	protected abstract void join(String in1, String in2, String out, String tmpPath);

	protected abstract void select(String in, String out, String substrSelect, String tmpPath);


	public abstract void joinAndSelectEfficiently(String in1, String in2, String out, String
										substrSelect, String tmpPath);

	/**
	 * Do not modify this method!
	 *
	 * The method sorts the input files in a lexicographic order of first column, joins the files, and
	 * saves the result in an output file.
	 *
	 * @param in1
	 * @param in2
	 * @param out
	 * @param tmpPath
	 */

	public void sortAndJoin(String in1, String in2, String out,
			String tmpPath) {
		String outFileName = new File(out).getName();
		String tmpFileName1 = outFileName.substring(0, outFileName.lastIndexOf('.'))
				+ "_intermed1" + outFileName.substring(outFileName.lastIndexOf('.'));
		String tmpOut1 = Paths.get(tmpPath, tmpFileName1).toString();
		String tmpFileName2 = outFileName.substring(0, outFileName.lastIndexOf('.'))
				+ "_intermed2" + outFileName.substring(outFileName.lastIndexOf('.'));
		String tmpOut2 = Paths.get(tmpPath, tmpFileName2).toString();

		this.sort(in1, tmpOut1,tmpPath);
		this.sort(in2, tmpOut2,tmpPath);
		this.join(tmpOut1,tmpOut2,out,tmpPath);

		try {
			Files.deleteIfExists(Paths.get(tmpPath, tmpFileName1));
			Files.deleteIfExists(Paths.get(tmpPath, tmpFileName2));
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	/**
	 * Do not modify this method!
	 *
	 * The method joins the files, then selects the lines that contain substrSelect, and
	 * saves the result in an output file.
	 *
	 * @param in1
	 * @param in2
	 * @param out
	 * @param substrSelect
	 * @param tmpPath
	 */
	public void joinAndSelect(String in1, String in2, String out, String substrSelect, String tmpPath)
{
		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();
		sortAndJoin(in1,in2,tmpOut,tmpPath);
		this.select(tmpOut,out,substrSelect,tmpPath);
		try {
			Files.deleteIfExists(Paths.get(tmpPath, tmpFileName));
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }
}
```

IExternalMemoryImpl.java

```
package Join;

public class ExternalMemoryImpl extends IExternalMemory {

        @Override
        public void sort(String in, String out, String tmpPath) {

                // TODO: Implement
        }

        @Override
        protected void join(String in1, String in2, String out, String tmpPath) {

                // TODO Auto-generated method stub

        }

        @Override
        protected void select(String in, String out, String substrSelect, String tmpPath) {

                // TODO Auto-generated method stub

        }

        @Override
        public void joinAndSelectEfficiently(String in1, String in2, String out,
                        String substrSelect, String tmpPath) {

                // TODO Auto-generated method stub

        }


}
```