Description

Implementation of two join algorithms, Sort Merge Join (SMJ) and Nested Loops Join (NLJ). Written in Java. The program take as input 2 comma-separated files and performs the user specified equi-join algorithm in order to join them on selected attributes.
The program must accept the following command line arguments:

-f1 <file1 path>: full path to file1
-a1 <file1_join_attribute>: the column to use as join attribute from file1 (counting from 0)
-f2 <file2 path>: same as above for file2
-a2 <file2_join_attribute>: same as above for file2
-j <join_algorithm_to_use>: SMJ or NLJ
-m <available_memory_size>: we use as memory metric the number of records*
-t <temporary_dir_path>: a directory to use for reading/writing temporary files
-o <output_file_path>: the file to store the result of the join

For example, in order to join two relations stored in files “R.csv” and “S.csv” on the 1st column of R and the 2nd column of S, using Sort Merge Join, having available memory = 200 records and saving the result to file “results.csv” one should execute the following command:

java –jar joinalgs.jar –f1 R.csv –a1 0 –f2 S.csv –a2 1 –j SMJ –m 200 –t tmp –o results.csv

or

java –jar joinalgsUsingThreads.jar –f1 R.csv –a1 0 –f2 S.csv –a2 1 –j SMJ –m 200 –t tmp –o results.csv

or

java –jar joinalgsAlternativeMerge.jar –f1 R.csv –a1 0 –f2 S.csv –a2 1 –j SMJ –m 200 –t tmp –o results.csv

or

java –jar joinalgsAlternativeMergeUsingThreads.jar –f1 R.csv –a1 0 –f2 S.csv –a2 1 –j SMJ –m 200 –t tmp –o results.csv

These are the classes of the project:

Main: It parses the arguments of the user. Executes SMJ or NLJ accordingly.

Tuple: This is the class for records in a relation. Its fields are the number of its attributes, the list of its attributes and the relation's name, in which it belongs.

Attribute: This is the class for columns in a record. Its field are the value and the name of the column.

TupleComparator: It is used to sort the sublists of tuple objects, based on a given attribute, during SMJ.

SortMergeJoin: It contains the implementation of the non-efficient SMJ. The sorting phase of the 2-phase sort does not exceed m buffers, in any case. In order to return to previous tuples of a relation (in case many tuples to be joined match on the specific attribute library), the "Mark-Reset" functionality of the Buffered Reader is used. Each sublist is sorted using the "sort" method of the "Collections" library. The merge phase of the external sorting algorithm has been implemented. There has also been included an alternative slower approach for the merge phase (its use is inside comments).

NestedLoopJoin: It contains the implementation of the NLJ. The blocked NLJ algorithm is used. The smaller relation is chosen to be iterated in the outer while loop. Super-naive NLJ implementation is also included as an alternative, but it is not used.

Utilities: It contains all the methods needed to write and read data to and from "*.csv*" files. Also, contains the methods *splitCSVToSortedSublists* and *splitCSVToDuplicateSortedSublists* which are used for the sorting phase of the SMJ algorithm. The latter method is used in case we want to join a relation with itself. It saves I/Os, by creating duplicate sublists for a relation, rather than reading the same relation again.

For the SMJ algorithm, there is also included a thread implementation. During the merge phase, we can use a different thread to merge each relation into one sorted result. We need two threads, since we are joining two relations. With this implementation we can achieve faster execution times, since we are merging two relations simultaneously. The use of threads is inside comments.

The following classes extend the Thread class:

MergeThread: It runs the merge phase of the external sorting algorithm. Given a number of sublists, the algorithm splits the sublists to 2 teams. In each iteration, it merges 2 sublists from each team, thus dividing the total number of sublists by 2, up until only one sublist exists.

MergeAlternativeThread: It reads one tuple at a time from each sorted sublist of the relation and writes the minimum in the sorted relation file. The algorithm proceeds to compare the next tuple from sublist that the last one was fetched from. The sorting finishes when all tuples have been written in the result file.

The console output of each run can be found in these folders:
"runs", ***"runsUsingThreads"***, ***"runsAlternativeMerge"***, ***"runsAlternativeMergeUsingThreads"***
Please note that the ***".csv"*** relations to be joined, should be copied in the ***"testdata"*** folder.

Execution times

In all the executions done, memory does not exceed the limit of 200 buffers. In the following table, the execution time of some equi-joins are shown:

“joinalgs.jar” results

#	Equi-Join	Execution Time	Tuples
1	f1: D, a1: 3, f2: C, a2: 0, m: 200, J: NLJ	14.159 sec	6029
2	f1: D, a1: 3, f2: B, a2: 0, m: 200, J: NLJ	4.305sec	1783
3	f1: A, a1: 3, f2: E, a2: 0, m: 200, J: NLJ	0.512 sec	148
4	f1: B, a1: 1, f2: B, a2: 2, m: 200, J: NLJ	1.292 sec	357
5	f1: D, a1: 3, f2: C, a2: 0, m: 200, J: SMJ	2.367 sec	6029
6	f1: D, a1: 3, f2: B, a2: 0, m: 200, J: SMJ	1.766 sec	1783
7	f1: A, a1: 3, f2: E, a2: 0, m: 200, J: SMJ	4.911 sec	148
8	f1: B, a1: 1, f2: B, a2: 2, m: 200, J: SMJ	0.64 sec	357

The following table contains the results of the implementation that uses threads. Threads are only used by the SMJ algorithm:

“joinalgsUsingThreads.jar” results

#	Equi-Join	Execution Time	Tuples
5	f1: D, a1: 3, f2: C, a2: 0, m: 200, J: SMJ	2.447 sec	6029
6	f1: D, a1: 3, f2: B, a2: 0, m: 200, J: SMJ	1.69 sec	1783
7	f1: A, a1: 3, f2: E, a2: 0, m: 200, J: SMJ	4.919 sec	148
8	f1: B, a1: 1, f2: B, a2: 2, m: 200, J: SMJ	0.658 sec	357

These are the results produced while running the equi-join algorithms by using an alternative slower merge method:

“JoinalgsAlternativeMerge.jar” results

#	Equi-Join	Execution Time	Tuples
5	f1: D, a1: 3, f2: C, a2: 0, m: 200, J: SMJ	11.952 sec	6029
6	f1: D, a1: 3, f2: B, a2: 0, m: 200, J: SMJ	8.782 sec	1783
7	f1: A, a1: 3, f2: E, a2: 0, m: 200, J: SMJ	96.054 sec (1.6009 min)	148
8	f1: B, a1: 1, f2: B, a2: 2, m: 200, J: SMJ	1.223 sec	357

“JoinalgsAlternativeMergeUsingThreads.jar” results

#	Equi-Join	Execution Time	Tuples
5	f1: D, a1: 3, f2: C, a2: 0, m: 200, J: SMJ	9.339 sec	6029
6	f1: D, a1: 3, f2: B, a2: 0, m: 200, J: SMJ	8.605 sec	1783
7	f1: A, a1: 3, f2: E, a2: 0, m: 200, J: SMJ	90.644 sec (1.5107 min)	148
8	f1: B, a1: 1, f2: B, a2: 2, m: 200, J: SMJ	1.022 sec	357

Observations:

- As expected, the execution time using threads is less. To achieve this, it is a requirement to run the program on a hardware with at least 2 CPU cores. If a relation is very small, the thread implementation is not beneficial. For instance we won't benefit by running the merge phase on a separate thread for relation A, in equi-join #7, because it contains only 150 tuples.
- As we have can observe, the results of the implementation that uses the alternative merge algorithm are worst than the implementation that uses the external sorting merge algorithm. Also, the results of the alternative implementation that uses threads are better than the results of the alternative implementation that does not use threads.