**Rationale for the “Very Large File Sorting Solution”**

by Darcio Pacifico

**How to Check and Execute the Solution:**

* Decompress the solution zip file.
* Call the sorting-tool.jar file as follow:

java -Dfile=<txt file path> -jar sorting-tool.jar

* Project structured as a Scala SBT default project.

**Solution Design:**

The solution uses Scala 2.11, in a simplified solution abstraction and design, focusing mainly on the sorting algorithms and the technical characteristics that allow the implementation of the external sorting of huge files.

The solution was divided into two parts: the partitioning of original file and the merging of data into a final sorted file.

The program divide the whole file into small enough parts to be sorted in the given memory efficiently, and store these parts into separated files. The amount of lines to be sorted at each of these steps dynamically varies accordingly to JVM memory setup (-Xms -Xmx).

The merge operation starts by opening each file partition using the Java BufferedReader class. Each of these Readers use, as a buffer size, the amount of available memory divided by the quantity of partitions plus one additional part for BufferedWriter, so total memory / partitions + 1.

A PriorityQueue (PQ) is loaded with a tuple of the first line of each reader and the reader itself ((String, BufferedReader)). The comparator of the PQ considers only the line for comparisons.

At each remove operation of the PQ, the line is written on the PrintWriter, and the next line of the bufferReader is read. The BufferReader and the next line are inserted back in the PQ, for the recursive next call, until PQ becomes empty.

@tailrec  
**def** consumePQ(pq: PriorityQueue[(String, BufferedReader)],destFile: PrintWriter): Unit = {

**if** (!pq.isEmpty) {  
 // consumes until PQ becomes empty  
 **val** (line, reader) = pq.remove()  
  
 //append this line to the dest file  
 destFile.println(line)  
  
 //update the PQ for next recursive call  
 **val** newLine = reader.readLine()  
 **if** (newLine != **null**) {  
 pq.add((newLine, reader))  
 }  
  
 //recursive call  
 consumePQ(pq, destFile)  
 }  
}

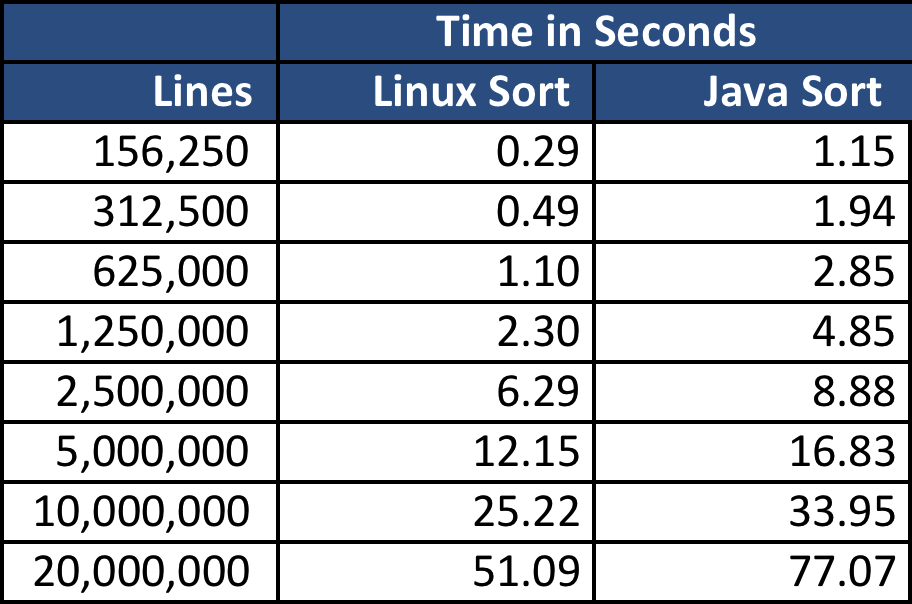
The PQ operations cost is proportional to N Lg Partitions line comparisons, which is quite scalable, even for thousands of file partitions (please, see the Limitations section at the end).

**Performance and Results:**

Was used a comparison based algorithm and external partitioning to accomplish the requirements. The achieved performance is proportional to N log N comparisons, scaling in a “linearithmic” progression.

As a performance benchmark, was used the Linux sort command. Disregarding the overhead of the JVM initialization, when comparing sorting of files with more than 5 million lines, the time to sort the files in the java solution was about 40% greater than Linux sort.

Table: Performance Results Comparison:



\* Each result equity checked using file checksum (md5sum).

**Quality:**

Again, considering the Linux sort as a quality reference for sorting results, I have considered that all sorting results should be exactly the same as Linux sort, using checksum (md5sum) for a precise comparison. All results cited above matches the checksum exactly.

**Limitations**

The solution was designed using a single-pass merging operation strategy, which means that the merge step will read all the partitions at same time, no matter the quantity of partitions, dividing the memory for buffer by the readers of these files. This strategy assures that all file information will be read and write only two times in disc for the whole data (1x R/W for partitioning + 1x R/W for merging).

For a more constrained condition, as a real small amount of RAM, a huge file with dozens of GBs and a HDD device for storage (not SSD), maybe the amount of partition files makes the solution to become inefficient, due to the overhead of handling and buffering too many files at same time, being even worse than multiple-pass merging, even considering more R/W operation for the whole data.

This weakness could be solved by a recursive merge strategy, that executes the merge operation in a limited quantity of files at a time, considering an optimal number of partitions, until achieve the final file.