<http://scalejava.blogspot.in/2013/09/concurrent-merge-sorting-of-large.html>

Concurrent Merge sorting of large number of different types of files in Java

I recently worked on a customer project, part of which required sorting and merging multiple large files based on a timestamp field in the files. These files were Call data records (CDR) files which are basically CSV files containing usage records for all subscribers of 2G, 3G and GPRS for each day. The total number of files was about 2000 with varied number of 2G, 3G and GPRS files. The total size was roughly 200 Gb.

Although this can be done using BigMemory, the customer wanted a quick and dirty way of doing this.

Each of the files had a particular naming convention that would help understand what type of file it is. The 1st 2 or 3 characters enable us to understand what type of file it is. Ex. File names –

3G

'GMI02A\_13082900\_0183'

'GMI02A\_13082900\_0184'

2G

'RTPAHLR1\_13082700\_8853'

'RTPAHLR1\_13082701\_8854'

'RTPAHLR1\_13082702\_8855'

'RTPAHLR1\_13082703\_8856'

'RTPAHLR1\_13082704\_8857'

GPRS

'MOU05\_1308270005'

'MOU05\_1308270020'

'MOU05\_1308270035'

As you can see the characters before the first underscore identify what type of file it is. The characters after the underscore are timestamps. If we sort and group the files as per these timestamps it is likely that we will have the same range of timestamps in the file contents. An interesting thing to note is, there can be multiple files per timestamp, which is the character after the 2nd underscore for 3G and 2G.

Single sample line from each of the CSV was as follows (with dummy data) –

2G

5x944324461a1b18d08d7ab25d9fa1595f,'ABCDFE12\_13050112\_9998',' ','','4','43',' ',' ','','','466974301229529','886983095821','','',' ',' ','','','2013-05-01 11:10:58.000',307,' ','0','0',' ','','',' ',' ',' ','','',' ',' ','',' ',' ',' ','','1','2','1',’6899066686','','','','1','2','1','6899066686',' ',' ','','',' ',' ',' ','','',' ','08-0-9-3 ','08-0-9-3 ',' ',' ABCDFE12 ','17','0',' ',' ',' ',0,'IGMRL01','2-15','ONXDEM','7-25','0','','','778158',0,0,'','',' ',,'','','',,,'','','','','','','',' ',0,'','','','','','','',' ',' ','','',' ',' ',' ',' ','8DCA28F284000007','1','1','1 ','656935\*\*\*319','','','','','','',' ',' ',' ','2013-05-01 11:04:58.000',307,' ','',' ',181,'1','1','1 ','656935\*\*\*319',' ',' ',' ',' ',' ',' ',' ','0','2','0','1','D9110935388202','',' ','','','',' ','',' ',' ',' ',,'','',' ',' ',' ',' ',' ',' ',' ','','','','','653095821','456683539688','6544066686','0823206559','','','','','','','654066686','34222206559','',''

GPRS

5x944324461a1f18d093bfb25d9fa1595f,'MCCDSC\_1305011035','18','0','3','1339655','466974301229529','2013-05-01 10:03:37.000','355026050423934','886983095821','133.22.29.4','31.39.11.321,'11892013','INTERNET','CMSC099.M22dd66.GPRS','1','111.334.23.12','413427','239','58844','1','8 ','46697 ','0','1','2',0,0,'02','FF','00','00','00','FE','00','00','00','00','00','4A','00','02','02','93','96','83','FE','74','81','FF','FF','00','00','00','2013-05-01 09:56:37.000','2013-05-01 10:03:03.000',386,'0 ','0',0,0,'00','00','00','00','00','00','00','00','00','00','00','00',,,0,'0','','0','0','','','00','0',0,0,0,0,,,,,'5630934411','7331295359688','00','00','00','3','1','1','0','0','0','',' ','',' ','','',' ','01306800','Samsung','Galaxy','Handset','1','','','','','','0','','0','0'

3G

5x944324461a1f18d093ba1595f,'DGCCSD\_13050112\_2938',1,1,1,1,1,1,1,1,0,2013-05-01 12:55:08.000',0,'2013-05-01 12:48:04.000','11','00','413B','55','0AD8','2013-05-01 12:47:59.000','03','FF',' ','FFFF','','466974104424565','','','','','','','','FF','939644958 ','05','06','5430\*\*\*660','05','06','65535',65535,,,'65535',,'FF',65535,,,'00','03','FFFF','67822059982110','123974700000830','','0000','0000','0000','0000','0000','0000','07','886983\*\*\*686','05','05','8488',21111,466,97,'19546',886935\*\*\*416,'05',21401,466,97,'FFFFFFFFFFFFFFFF',,'FF','FF','00000000',' ','3','8','2013-05-01 12:50:49.000','2013-05-01 12:48:08.000','F5B9','8130\*\*\*660','05','06','00',' ','FFFF','FF',0,'1233804097','05','04',0,0,0,0,0,0,'886935\*\*\*374',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,' ','2013-05-01 12:47:59.000',,'',,,'00',0,0,0,0,0,0,0,0,0,'413B','55','0AD8','','FF','10','FF','00',0,0,'',' ',' ',,0,0,0,0,0,161,161.11,'01','0',0,'000000','08',,53,'1246',0,0,'FFFF','FFFFF','01','FF',436,62871450,'00','1','00','',' ',' ','FF','FF','FF','FF','FF','FF','FF',0,'2013-05-01 12:47:58.000',2,'00',,,,,,,,,'',,' ','FFFF','FF','00','FF','FF','00','FF','2013-05-01 12:47:59.000','642\*\*\*660',’1243408079','823\*\*\*686','124191389688','121\*\*\*660','066208079','','','',''

The idea was to read 3 files (of each type) at a time and insert into a shared sorted buffer. Once the reading is done, write out the sorted buffer into an output file, clear it and move onto the next batch of 3 files. Another requirement for the writer was that, all data should not be merge sorted into a single file, rather should be split into multiple output files. So the File writer thread must keep track of the number of lines written and roll over to the next output file.

The problem with this approach is that not all files will contain the same timestamp.

Ex. 2G might be till timestamp T5, 3G till timestamp T7 and GPRS till timestamp T4.

In this case we can only flush data till timestamp T4 and retain all the data after that in the buffer since the next batch of files might contain those timestamps to be sorted. This puts additional memory pressure.

The approach I used was to do a modified concurrent external merge sort.

1. Based on producer consumer pattern
2. 3 threads read from the 3 files in parallel (2G, 3G and GPRS) and insert into a shared task buffer
3. 1 thread consumes from the buffer
4. Reader threads read line by line and insert into a data structure, which internally sorts on timestamp via a custom comparator for each insert. It is basically a ConcurrentSkipListMap backed by several ConcurrentLinkedQueue. The key for the Map is the timestamp, and the value is the List of lines associated with that timestamp.
5. After the each reader thread finishes inserting into the task queue, they wait on a CyclicBarrier. The last thread to reach the barrier notifies the Consumer that the file reading and sorting is completed into the queue
6. Consumer is awakened and just spits out the sorted Map into a CSV
7. Once file is written, CyclicBarrier is reset and cycle is repeated for next batch of files

Here is a snippet of the sorted map

|  |
| --- |
| /\*\* |
| \* @author sourabh\_gh |
| \* |
| \*/ |
| public class SharedTaskQueue |
| { |
| // LinkedList might cause a race condition, if so use concurrentlinkedqueue or my concurrentlinkedlist implementation |
| private Map<DateTime, LinkedList<String[]>> map = new ConcurrentSkipListMap<DateTime, LinkedList<String[]>>(new DateComparator()); |
|  |
| public void put(DateTime timeStamp, String[] data) throws InterruptedException |
| { |
| LinkedList<String[]> value = map.get(timeStamp); |
| if(value == null){ |
| value = new LinkedList<String[]>(); |
| value.add(data); |
| map.put(timeStamp, value); |
| } |
| else{ |
| value.add(data); |
| map.put(timeStamp, value); |
| } |
| } |
|  |
| public LinkedList<String[]> get(DateTime timeStamp) throws InterruptedException |
| { |
| return map.get(timeStamp); |
| } |
|  |
| public Map<DateTime, LinkedList<String[]>> getBackingMap(){ |
| return map; |
| } |
|  |
| public void setBackingMap(Map<DateTime, LinkedList<String[]>> map){ |
| this.map = map; |
| } |
|  |
| public String toString(){ |
| return map.toString(); |
| } |
|  |
| public static class DateComparator implements Comparator<DateTime>{ |
| @Override |
| public int compare(DateTime s1, DateTime s2){ |
| return s1.compareTo(s2); |
| } |
| } |
| } |

[view raw](https://gist.github.com/sourabhghose/e5fc01b555cf343d675c/raw/3aad7af1382da4d4d410f8243361e0079bb8f964/SharedTaskQueue.java) [SharedTaskQueue.java](https://gist.github.com/sourabhghose/e5fc01b555cf343d675c#file-sharedtaskqueue-java) hosted with ❤ by [GitHub](https://github.com)

A snippet of the reader (producer) thread looks like this

|  |
| --- |
| public void run() |
| { |
| try |
| { |
| //SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS"); |
| DateTimeFormatter sdf = DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss.SSS"); |
| //CSVReader reader; |
| BufferedReader br = null; |
| String nextLine; |
| String[] splittedLine; |
| while (!twoGFileMap.isEmpty()){ |
| for(Iterator<Map.Entry<String, List<String>>> iter = twoGFileMap.entrySet().iterator(); iter.hasNext();){ |
| Map.Entry<String, List<String>> entry = iter.next(); |
| if(entry == null || entry.getValue() == null){ |
| processedTill = null; |
| iter.remove(); |
| System.out.println("2g Producer finished the file job; waiting."); |
| barrier.await(); |
| // if(count == 0){ |
| // barrier.reset(); |
| // System.out.println("2g reset"); |
| // } |
| continue; |
| } |
| for(String fileName : entry.getValue()){ |
| System.out.println("2g processing file "+fileName); |
| //reader = new CSVReader(new FileReader(FilePreprocessorRunner.dataPath+fileName)); |
| br = new BufferedReader(new FileReader(dataPath+fileName)); |
| while ((nextLine = br.readLine()) != null) { |
| //splittedLine = nextLine.split(COMMA, -1); |
| splittedLine = StringUtils.splitPreserveAllTokens(nextLine, COMMA); |
| //Date date = sdf.parse(StringUtils.replace(splittedLine[twogtimestampcolumn], APOSTROPHE, BLANK)); |
| DateTime date = DateTime.parse( StringUtils.replace(splittedLine[twogtimestampcolumn], APOSTROPHE, BLANK), sdf ) ; |
| processedTill = date; |
| broker.put(date, splittedLine); |
| } |
| br.close(); |
| //reader.close(); |
| } |
| iter.remove(); |
| System.out.println("2g Producer finished the file job; waiting. Processed till "+processedTill); |
| barrier.await(); |
| // if(count == 0){ |
| // barrier.reset(); |
| // System.out.println("2g reset"); |
| // } |
| } |
| } |
| isDone = true; |
| } |
| catch (InterruptedException ex) |
| { |
| ex.printStackTrace(); |
| } catch (BrokenBarrierException e) { |
| //TODO Auto-generated catch block |
| e.printStackTrace(); |
| } |
| catch (FileNotFoundException e) { |
| // TODO Auto-generated catch block |
| e.printStackTrace(); |
| } catch (IOException e) { |
| // TODO Auto-generated catch block |
| e.printStackTrace(); |
| } |
|  |
| } |

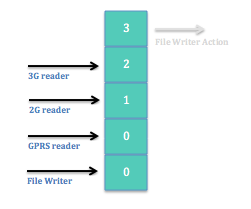
[view raw](https://gist.github.com/sourabhghose/0d2728c3895cc80edaad/raw/a1e913937d06f81280a596ad6720d20ba53acd54/FileReader.java) [FileReader.java](https://gist.github.com/sourabhghose/0d2728c3895cc80edaad#file-filereader-java) hosted with ❤ by [GitHub](https://github.com)

And a snippet of the writer (consumer) thread looks like this

|  |
| --- |
| public void run() |
| { |
| try |
| { |
| System.out.println("Consumer starting, buffer size is "+buffer.getBackingMap().size()); |
| if(outputFileNameIndex == 0) { |
| FilePreprocessorRunner.outputFileName = StringUtils.replace(FilePreprocessorRunner.outputFileName, "$", String.valueOf(outputFileNameIndex)); |
| System.out.println("Output file is.. "+FilePreprocessorRunner.outputFileName); |
| } |
| else { |
| FilePreprocessorRunner.outputFileName = StringUtils.replace(FilePreprocessorRunner.outputFileName, String.valueOf(outputFileNameIndex-1), String.valueOf(outputFileNameIndex)); |
| System.out.println("Output file is.. "+FilePreprocessorRunner.outputFileName); |
| } |
| //bw = new RollingFileWriter(new FileWriter(FilePreprocessorRunner.dataPath+FilePreprocessorRunner.outputFileName, true), this); |
| BufferedWriter bw = new BufferedWriter(new FileWriter(FilePreprocessorRunner.dataPath+FilePreprocessorRunner.outputFileName, true)); |
| DateTime processedTill = compareDates(); |
| System.out.println("Flushing data till timestamp "+processedTill); |
| Iterator<Map.Entry<DateTime, LinkedList<String[]>>> iter = buffer.getBackingMap().entrySet().iterator(); |
| DateTime currentKey; |
| while(iter.hasNext()){ |
| Map.Entry<DateTime, LinkedList<String[]>> entry = iter.next(); |
| currentKey = entry.getKey(); |
| if(currentKey.compareTo(processedTill) == 0 || currentKey.compareTo(processedTill) == -1){ |
| LinkedList<String[]> queue = entry.getValue(); |
| int queueCount = 0; |
| for(Iterator<String[]> itr = queue.iterator();itr.hasNext();){ |
| //writer.writeNext(itr.next()); |
| String[] currLine = itr.next(); |
| int length = currLine.length; |
| for(int i=0; i<length; i++){ |
| bw.write(currLine[i]); |
| if(i==(length-1)){ |
| bw.newLine(); |
| numberOfLinesWritten++; |
| if(numberOfLinesWritten >= FilePreprocessorRunner.noOfLinesToWrite){ |
| bw.close(); |
| FilePreprocessorRunner.outputFileName = StringUtils.replace(FilePreprocessorRunner.outputFileName, String.valueOf(outputFileNameIndex), String.valueOf(outputFileNameIndex+1)); |
| System.out.println(numberOfLinesWritten+" written, Rolling over to next file.. "+FilePreprocessorRunner.outputFileName); |
| bw = new BufferedWriter(new FileWriter(FilePreprocessorRunner.dataPath+FilePreprocessorRunner.outputFileName, true)); |
| outputFileNameIndex++; |
| numberOfLinesWritten = 0; |
| } |
| } |
| else |
| bw.append(COMMA); |
| } |
| queue.remove(queueCount); |
| queueCount++; |
| //itr.remove(); |
| } |
| //iter.remove(); |
| buffer.getBackingMap().remove(currentKey); |
| queue = null; |
| currentKey = null; |
| } |
| } |
| System.out.println("Number of lines written "+numberOfLinesWritten); |
| if(buffer.getBackingMap().size() < 50 || buffer.getBackingMap().size() > 500 ){ |
| buffer.getBackingMap().clear(); |
| buffer.setBackingMap(new ConcurrentSkipListMap<DateTime, LinkedList<String[]>>(new SharedTaskQueue.DateComparator())); |
| } |
| //System.out.println("Final parameters output file "+FilePreprocessorRunner.outputFileName+" outputFileNameIndex "+outputFileNameIndex); |
| //System.out.println("Consumer sleeping for 5 secs, hopefully GC will clean up :)"); |
| System.gc(); |
| //Thread.sleep(5000); |
| System.out.println("Comsumer finished its job; terminating."); |
| } |
| // catch (InterruptedException e) { |
| // // TODO Auto-generated catch block |
| // e.printStackTrace(); |
| // } |
| catch (IOException e) { |
| // TODO Auto-generated catch block |
| e.printStackTrace(); |
| } |
| } |

[view raw](https://gist.github.com/sourabhghose/8479adda22bb72ec58a0/raw/d1f45e7e9bfdb47e1c3299d70d17fedb1ad80db7/FileWriter.java) [FileWriter.java](https://gist.github.com/sourabhghose/8479adda22bb72ec58a0#file-filewriter-java) hosted with ❤ by [GitHub](https://github.com)

A sample flow of the threads with CyclicBarrier coordination is as follows –

[](http://4.bp.blogspot.com/-zmrbve249PY/UkPGYCePMZI/AAAAAAAAAtQ/smkrpmjTAvc/s1600/Screen+Shot+2013-09-26+at+1.29.50+PM.png)

On a 4 core, 35Gb machine this with 120 Gb worth of data files, this took about 2 hours to complete. This solution worked for my use case, however, optimally you would read only a predefined number of lines into the buffer rather than the entire file to avoid heap pressure.

On a side note, I tested this with the CMS and G1 GC and I found CMS to be much more performant and predictable for my use case.   
The entire code base is available here <https://github.com/sourabhghose/LargeFileMergeSort>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*