<http://www.ashishsharma.me/2011/08/external-merge-sort.html>

## Friday, August 19, 2011

### External Merge Sort

**External sorting** is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single larger file.  
  
One **example** of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. For example, for sorting 900 megabytes of data using only 100 megabytes of RAM:  
  
1) Read 100 MB of the data in main memory and sort by some conventional method, like quicksort.  
2) Write the sorted data to disk.  
3) Repeat steps 1 and 2 until all of the data is in sorted 100 MB chunks (there are 900MB / 100MB = 9 chunks), which now need to be merged into one single output file.  
4) Read the first 10 MB (= 100MB / (9 chunks + 1)) of each sorted chunk into input buffers in main memory and allocate the remaining 10 MB for an output buffer. (In practice, it might provide better performance to make the output buffer larger and the input buffers slightly smaller.)  
5) Perform a 9-way merge and store the result in the output buffer. If the output buffer is full, write it to the final sorted file, and empty it. If any of the 9 input buffers gets empty, fill it with the next 10 MB of its associated 100 MB sorted chunk until no more data from the chunk is available.  
  
Optimizing a little with additional passes -   
1) For sorting, say, 50 GB in 100 MB of RAM, using a single merge pass isn't efficient: the disk seeks required to fill the input buffers with data from each of the 500 chunks (we read 100MB / 501 ~ 200KB from each chunk at one time) take up most of the sort time. Using two merge passes solves the problem. Then the sorting process might look like this:  
2) Run the initial chunk-sorting pass as before.  
3) Run a first merge pass combining 25 chunks at a time, resulting in 20 larger sorted chunks.  
4) Run a second merge pass to merge the 20 larger sorted chunks.

**Complexity**  
Like in-memory sorts, efficient external sorts require O(n log n) time  
  
References  
http://en.wikipedia.org/wiki/External\_sorting  
http://www.daniel-lemire.com/blog/archives/2010/04/01/external-memory-sorting-in-java/

Implementation -  
Sort a 4Gb file in 10Mb memory (or a system in which many processes run and total memory of that system is x Mb)

[?](http://www.ashishsharma.me/2011/08/external-merge-sort.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184 | package main.java.algo.sorting;    import java.util.\*;  import java.io.\*;      // Goal: offer a generic external-memory sorting program in Java.  //  // It must be :  //  - hackable (easy to adapt)  //  - scalable to large files  //  - sensibly efficient.    // This software is in the public domain.    public class ExternalSort {          // we divide the file into small blocks. If the blocks      // are too small, we shall create too many temporary files.      // If they are too big, we shall be using too much memory.      public static long estimateBestSizeOfBlocks(File filetobesorted) {          long sizeoffile = filetobesorted.length();          // we don't want to open up much more than 1024 temporary files, better run          // out of memory first. (Even 1024 is stretching it.)          final int MAXTEMPFILES = 1024;          long blocksize = sizeoffile / MAXTEMPFILES ;          // on the other hand, we don't want to create many temporary files          // for naught. If blocksize is smaller than half the free memory, grow it.          long freemem = Runtime.getRuntime().freeMemory();          if( blocksize < freemem/2)              blocksize = freemem/2;          else {              if(blocksize >= freemem)                System.err.println("We expect to run out of memory. ");          }          return blocksize;      }         // This will simply load the file by blocks of x rows, then       // sort them in-memory, and write the result to a bunch of       // temporary files that have to be merged later.       //       // @param file some flat  file       // @return a list of temporary flat files        public static List<File> sortInBatch(File file, Comparator<String> cmp) throws IOException {          List<File> files = new ArrayList<File>();          BufferedReader fbr = new BufferedReader(new FileReader(file));          long blocksize = estimateBestSizeOfBlocks(file);// in bytes          try{              List<String> tmplist =  new ArrayList<String>();              String line = "";              try {                  while(line != null) {                      long currentblocksize = 0;// in bytes                      while((currentblocksize < blocksize)                      &&(   (line = fbr.readLine()) != null) ){ // as long as you have 2MB                          tmplist.add(line);                          currentblocksize += line.length() // 2 + 40; // java uses 16 bits per character + 40 bytes of overhead (estimated)                      }                      files.add(sortAndSave(tmplist,cmp));                      tmplist.clear();                  }              } catch(EOFException oef) {                  if(tmplist.size()>0) {                      files.add(sortAndSave(tmplist,cmp));                      tmplist.clear();                  }              }          } finally {              fbr.close();          }          return files;      }          public static File sortAndSave(List<String> tmplist, Comparator<String> cmp) throws IOException  {          Collections.sort(tmplist,cmp);  //          File newtmpfile = File.createTempFile("sortInBatch", "flatfile");          newtmpfile.deleteOnExit();          BufferedWriter fbw = new BufferedWriter(new FileWriter(newtmpfile));          try {              for(String r : tmplist) {                  fbw.write(r);                  fbw.newLine();              }          } finally {              fbw.close();          }          return newtmpfile;      }         // This merges a bunch of temporary flat files       // @param files       // @param output file       // @return The number of lines sorted. (P. Beaudoin)        public static int mergeSortedFiles(List<File> files, File outputfile, final Comparator<String> cmp) throws IOException {          PriorityQueue<BinaryFileBuffer> pq = new PriorityQueue<BinaryFileBuffer>(11,              new Comparator<BinaryFileBuffer>() {                public int compare(BinaryFileBuffer i, BinaryFileBuffer j) {                  return cmp.compare(i.peek(), j.peek());                }              }          );          for (File f : files) {              BinaryFileBuffer bfb = new BinaryFileBuffer(f);              pq.add(bfb);          }          BufferedWriter fbw = new BufferedWriter(new FileWriter(outputfile));          int rowcounter = 0;          try {              while(pq.size()>0) {                  BinaryFileBuffer bfb = pq.poll();                  String r = bfb.pop();                  fbw.write(r);                  fbw.newLine();                  ++rowcounter;                  if(bfb.empty()) {                      bfb.fbr.close();                      bfb.originalfile.delete();// we don't need you anymore                  } else {                      pq.add(bfb); // add it back                  }              }          } finally {              fbw.close();              for(BinaryFileBuffer bfb : pq ) bfb.close();          }          return rowcounter;      }        public static void main(String[] args) throws IOException {          if(args.length<2) {              System.out.println("please provide input and output file names");              return;          }          String inputfile = args[0];          String outputfile = args[1];          Comparator<String> comparator = new Comparator<String>() {              public int compare(String r1, String r2){                  return r1.compareTo(r2);}};          List<File> l = sortInBatch(new File(inputfile), comparator) ;          mergeSortedFiles(l, new File(outputfile), comparator);      }  }      class BinaryFileBuffer  {      public static int BUFFERSIZE = 2048;      public BufferedReader fbr;      public File originalfile;      private String cache;      private boolean empty;        public BinaryFileBuffer(File f) throws IOException {          originalfile = f;          fbr = new BufferedReader(new FileReader(f), BUFFERSIZE);          reload();      }        public boolean empty() {          return empty;      }        private void reload() throws IOException {          try {            if((this.cache = fbr.readLine()) == null){              empty = true;              cache = null;            }            else{              empty = false;            }        } catch(EOFException oef) {          empty = true;          cache = null;        }      }        public void close() throws IOException {          fbr.close();      }          public String peek() {          if(empty()) return null;          return cache.toString();      }      public String pop() throws IOException {        String answer = peek();          reload();        return answer;      }        } |

<http://en.wikipedia.org/wiki/External_sorting>

External sorting

From Wikipedia, the free encyclopedia

**External sorting** is a term for a class of [sorting](http://en.wikipedia.org/wiki/Sorting) [algorithms](http://en.wikipedia.org/wiki/Algorithm) that can handle massive amounts of [data](http://en.wikipedia.org/wiki/Data). External sorting is required when the data being sorted do not fit into the [main memory](http://en.wikipedia.org/wiki/Main_memory) of a computing device (usually [RAM](http://en.wikipedia.org/wiki/RAM)) and instead they must reside in the slower [external memory](http://en.wikipedia.org/wiki/External_memory) (usually a [hard drive](http://en.wikipedia.org/wiki/Hard_drive)). External sorting typically uses a sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single larger file.

|  |
| --- |
| **Contents**    [[hide](http://en.wikipedia.org/wiki/External_sorting)]   * [1 External merge sort](http://en.wikipedia.org/wiki/External_sorting#External_merge_sort)   + [1.1 Additional passes](http://en.wikipedia.org/wiki/External_sorting#Additional_passes)   + [1.2 Tuning performance](http://en.wikipedia.org/wiki/External_sorting#Tuning_performance) * [2 Other algorithms](http://en.wikipedia.org/wiki/External_sorting#Other_algorithms) * [3 References](http://en.wikipedia.org/wiki/External_sorting#References) * [4 External links](http://en.wikipedia.org/wiki/External_sorting#External_links) |

[[edit](http://en.wikipedia.org/w/index.php?title=External_sorting&action=edit&section=1)]External merge sort

One example of external sorting is the external [merge sort](http://en.wikipedia.org/wiki/Merge_sort) algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together.[[1]](http://en.wikipedia.org/wiki/External_sorting#cite_note-1)[[2]](http://en.wikipedia.org/wiki/External_sorting#cite_note-2) For example, for sorting 900 [megabytes](http://en.wikipedia.org/wiki/Megabyte) of data using only 100 megabytes of RAM:

1. Read 100 MB of the data in main memory and sort by some conventional method, like [quicksort](http://en.wikipedia.org/wiki/Quicksort).
2. Write the sorted data to disk.
3. Repeat steps 1 and 2 until all of the data is in sorted 100 MB chunks (there are 900MB / 100MB = 9 chunks), which now need to be merged into one single output file.
4. Read the first 10 MB (= 100MB / (9 chunks + 1)) of each sorted chunk into input buffers in main memory and allocate the remaining 10 MB for an output buffer. (In practice, it might provide better performance to make the output buffer larger and the input buffers slightly smaller.)
5. Perform a 9-way [merge](http://en.wikipedia.org/wiki/Merge_algorithm) and store the result in the output buffer. If the output buffer is full, write it to the final sorted file, and empty it. If any of the 9 input buffers gets empty, fill it with the next 10 MB of its associated 100 MB sorted chunk until no more data from the chunk is available. This is the key step that makes external merge sort work externally -- because the merge algorithm only makes one pass sequentially through each of the chunks, each chunk does not have to be loaded completely; rather, sequential parts of the chunk can be loaded as needed.

[[edit](http://en.wikipedia.org/w/index.php?title=External_sorting&action=edit&section=2)]**Additional passes**

That example shows a two-pass sort: a sort pass followed by a merge pass. Note that we had one merge pass that merged all the chunks at once, rather than in regular merge sort, where we merge two chunks at each step, and take \log n merge passes total. The reason for this is that every merge pass requires reading and writing *every value* in the array from and to disk once. Disk access is usually slow, and so reads and writes should be avoided as much as possible.

However, there is a trade-off with using fewer merge passes. As the number of chunks increases, the amount of data we can read from each chunk at a time during the merge process decreases. For sorting, say, 50 GB in 100 MB of RAM, using a single merge pass isn't efficient: the disk seeks required to fill the input buffers with data from each of the 500 chunks (we read 100MB / 501 ~ 200KB from each chunk at a time) take up most of the sort time. Using two merge passes solves the problem. Then the sorting process might look like this:

1. Run the initial chunk-sorting pass as before.
2. Run a first merge pass combining 25 chunks at a time, resulting in 20 larger sorted chunks.
3. Run a second merge pass to merge the 20 larger sorted chunks.

Like in-memory sorts, efficient external sorts require [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*) time: exponential increases in data size require linear increases in the number of passes. If one makes liberal use of the gigabytes of RAM provided by modern computers, the logarithmic factor grows very slowly: under reasonable assumptions, one could sort at least 500 GB of data using 1 GB of main memory before a third pass became advantageous, and could sort many times that before a fourth pass became useful.[[3]](http://en.wikipedia.org/wiki/External_sorting#cite_note-3)

[[edit](http://en.wikipedia.org/w/index.php?title=External_sorting&action=edit&section=3)]**Tuning performance**

The [Sort Benchmark](http://sortbenchmark.org/), created by computer scientist [Jim Gray](http://en.wikipedia.org/wiki/Jim_Gray_(computer_scientist)), compares external sorting algorithms implemented using finely tuned hardware and software. Winning implementations use several techniques:

* **Using parallelism**
  + Multiple disk drives can be used in parallel in order to improve sequential read and write speed. This can be a very cost-efficient improvement: a recent Sort Benchmark winner in the cost-centric Penny Sort category uses six hard drives in an otherwise midrange machine.[[4]](http://en.wikipedia.org/wiki/External_sorting#cite_note-4)
  + Sorting software can use [multiple threads](http://en.wikipedia.org/wiki/Thread_(computer_science)), to speed up the process on modern multicore computers.
  + Software can use [asynchronous I/O](http://en.wikipedia.org/wiki/Asynchronous_I/O) so that one run of data can be sorted or merged while other runs are being read from or written to disk.
  + Multiple machines connected by fast network links can each sort part of a huge dataset in parallel.[[5]](http://en.wikipedia.org/wiki/External_sorting#cite_note-5)
* **Increasing hardware speed**
  + Using more RAM for sorting can reduce the number of disk seeks and avoid the need for more passes.
  + Fast external memory, like 15K RPM disks or [solid-state drives](http://en.wikipedia.org/wiki/Solid-state_drives), can speed sorts (but adds substantial costs proportional to the data size).
  + *Many* different factors can affect hardware's maximum sorting speed: CPU speed and number of cores, RAM access latency, input/output bandwidth, disk read/write speed, disk seek time, and others.
  + Cost-efficiency as well as absolute speed can be critical, especially in cluster environments where lower node costs allow purchasing more nodes.
* **Increasing software speed**
  + Some Sort Benchmark entrants use a variation on [radix sort](http://en.wikipedia.org/wiki/Radix_sort) for the first phase of sorting: they separate data into one of many "bins" based on the beginning of its value. Sort Benchmark data is random and especially well-suited to this optimization.
  + Compacting the input, intermediate files, and output can reduce time spent on I/O, but is not allowed in the Sort Benchmark.
  + Because the Sort Benchmark sorts long (100-byte) records using short (10-byte) keys, sorting software sometimes rearranges the keys separately from the values to reduce memory I/O volume.

[[edit](http://en.wikipedia.org/w/index.php?title=External_sorting&action=edit&section=4)]Other algorithms

External merge sort is not the only external sorting algorithm; there are also *distribution sorts*, which work by partitioning the unsorted values into smaller "buckets" that can be sorted in main memory. Like merge sort, external distribution sort also has a main-memory sibling; see [bucket sort](http://en.wikipedia.org/wiki/Bucket_sort). There is a [duality](http://en.wikipedia.org/wiki/Duality_(mathematics)), or fundamental similarity, between merge- and distribution-based algorithms that can aid in thinking about sorting and other external memory algorithms.[[6]](http://en.wikipedia.org/wiki/External_sorting#cite_note-6) There are [in-place algorithms](http://en.wikipedia.org/wiki/In-place_algorithm) for external sort, which require no more disk space than the original data.