Comparison of External Sorting in Java and Python

Ramya Nagarajan
Indiana University Bloomington
School of Informatics and Computing
Email: ranagara@iu.edu

Sneha Tilak
Indiana University Bloomington
School of Informatics and Computing
Email: tilaks@iu.edu

***Abstract*—External sorting is a class of sorting algorithms that can handle massive amounts of data. 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 hybrid sort-merge strategy.** [1]
There are two phases to this algorithm. During the sorting phase, data from a file is read in the form of chunks and stored in main memory. It is then sorted and written to a temporary file. In the merging phase, the individually sorted temporary files are combined into one large file.
In this paper we will introduce the External Sorting algorithm and give a brief description about our implementation. We will also compare the run-time results between Java and Python. The Java implementation is done on a Windows machine using Eclipse IDE with JDK 1.8 and the Python implementation is done on a Linux machine using Python 2.7

***Index Terms*—Hybrid Sort-Merge Strategy, Java and Python.**

I. Introduction

Sorting is the process of putting elements in particular order (be it ascending or descending). Most times, other algorithms require their elements to undergo sorting before any other computation (Example - Search algorithms). The complexity of a sorting algorithm depends on the size of the list of elements, number of swaps performed, amount of memory being used and also the type of sorting algorithm used. There are various sorting algorithms available. Some of the most commonly used algorithms are - Quick Sort, Merge Sort, Heap Sort, Selection Sort, Insertion Sort, Bubble Sort.
At times, the size of the data to be sorted exceeds the size of the memory available. One way to overcome this problem would be to use external sorting which efficiently combines two sorting algorithms to improve the performance. For example, the input array can be treated as chunks of data which will fit into memory. Each of these chunks are sorted using quick sort and written back to a temporary file. The resulting temporary files are then merged into a final file by building a min heap and deleting the smallest element in every pass.

II. Algorithm

The following are the basic steps to be followed in the External Sorting Algorithm:

1. First, we need to read a chunk of data from the input file into the main memory.
2. Once the chunk is read, we must sort the data in it (using quick sort) and write the data to a temporary file.
3. Repeat Steps 1 and 2 for every chunk of data that is read from the input file. By the end of this run, we will have one sorted temporary file for each chunk of data read into memory.
4. Next, we must build a min heap using the first record of data in every temporary file. We then remove the smallest element from the heap using the deleteMin() operation and write it to the output file.
5. We read the next record from the same temporary file for which the deleteMin() operation was performed and insert it into the heap.
6. Repeat Steps 4 and 5 till we reach the EOF for each of the temporary files and all the data is written to the output file in a sorted order.

III. Pseudocode

Psuedocode for Quick Sort [5]:

```
Partition(A, p, r)
   x <- A[p]
   i <- p - 1
   j <- r + 1
   while true do
      repeat
         j <- j + 1
      until A[j] <= x

      repeat
         i <- i + 1
      until A[i] >= x

      if i < j then
         exchange(A[i], A[j])
      else
         return j

Randomized-Partition(A, p, r)
   i <- Random(p, r)
   exchange(A[r], A[i])
   return Partition(A, p, r)

Randomized-Quicksort(A, p, r)
   if p < r then
```

```
    q <- Randomized-Partition(A, p, r)
    Randomized-Quicksort(A, p, q - 1)
    Randomized-Quicksort(A, q + 1, r)
```

Psuedocode for Min Heap [6]:

```
INSERT(key, A):
   heapsize(A) <- heapsize(A)+1
   i <- heap  size( A)
   while i>1 and A[i/2] > key
      do A[i]<-A[i/2]
         i <- i / 2
   A[i]<-key

MIN-HEAPIFY(A,i)
   if 2i <= size[A] and A[2i] < A[i]
   leftchild then
      smallest <- 2i
   else
      smallest <- i
   if 2i+1 <= size[A] and
   A[2i+1] < A[smallest] rightchild then
      smallest <- 2i +1
   if smallest != i
      then swap (A[i], A[smallest])
         MIN - HEAPIFY( A, smallest )

EXTRACT - MIN(A)
   min <- A[1]
   A[1] <- A[heap-size[A]]
   heap-size[A] <- heap-size[A]1
   MIN - HEAPIFY( A,1)
   maintain heap property
   return min
```

IV. Implementation

This section has the Java implementation of the external sorting algorithm. The basic sequence of events are as follows:

1. Takes input file: Input.txt
2. The input is a file containing 1000 randomly generated numbers separated by a space.
3. I have taken a buffer size of 1024 Bytes which can be considered as the RAM. So, at a time, the memory can fit only 1024 Bytes of data from the input file.
4. We read the data into an ArrayList using a FileInputStream. Once read, the array is passed to a function which sorts the elements of the array using quick sort.
5. The resulting sorted list is written into a temporary file. If there are x chunks, there are x sorted temporary files.
6. We then prepare a min heap with the Node structure (containing the Integer object and Scanner object) from each of the temporary files.
7. The Node containing the min element is deleted from the heap using the deleteMin() operation and written into the Output.txt file.
8. Once deleted the next element is picked from the same Scanner object.
9. Steps 7 and 8 are repeated till there are no more elements left to be picked (till EOF is reached).

V. Results

As mentioned earlier, we have implemented the external sorting algorithm using Java and Python. We ran the code for various sizes of inputs and recorded the results. We found that the implementation using Java is faster when compared to that of Python. The Python code runs an order of magnitude slower than the Java code because of the runtime.

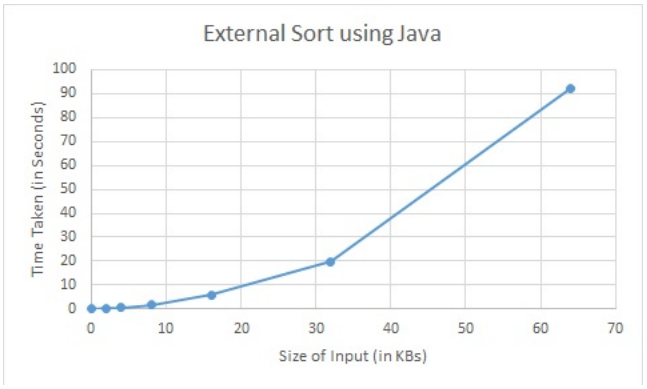
A. Table

Table I
Time taken to run the main modules in Java

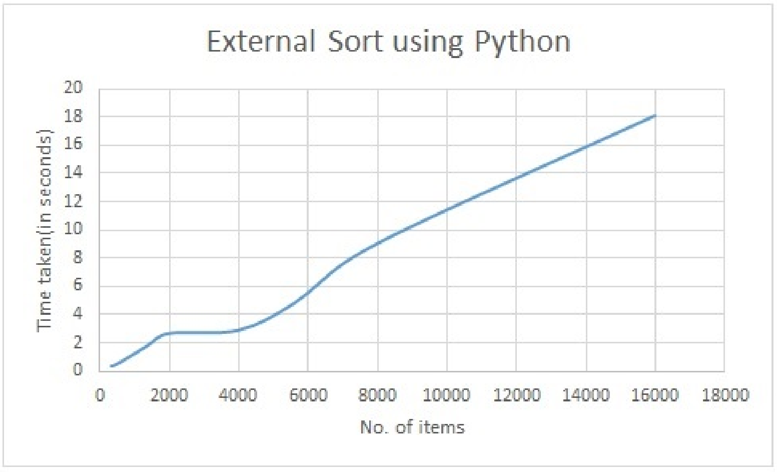
Size of Input file (in KB)	Total time (in Seconds)
2	0.3660
4	0.6170
8	1.6100
16	5.8380
32	19.6950
64	92.2080

B. Graph

External Sort using Java -



External Sort using Python -



C. Time Complexity

The length of a run is tied to the available buffer size.

Consider the number of files to be k. Every read and write is of constant time which depends on the processor used. The average complexity of quick sort is known to be $O(nlogn)$ and that of building a min heap is known to be $O(nlogn)$.

The total time taken = $O(nlogn)$.

VI. Conclusion

By combining one of the most efficient algorithms - quick sort with the help of a min heap data structure, we can sort large external files very efficiently.

Quick sort is used over merge sort as we can very easily avoid the worst case scenario (being $O(n^2)$) by picking a random pivot. Also, it requires very little additional space and exhibits a good cache locality which makes it faster than merge sort.

Next we need to insert the elements from the individually sorted files to an Output file in ascending order. Hence, we consider an array of length k which holds the minimum value from each of the k files. Since we need to pick only the smallest element from the array in each pass, we can build a Min Heap. We are using the same heap to perform deleteMin() to delete the smallest element and insertElement() to insert the next element.

Here, I have assigned a memory limit of 1024 bytes = 1 KB which can be thought of as a RAM of 1 KB. For better use of external sorting, one can run the code on a Virtual Machine with a fixed RAM and see how it performs. The speed the RAM and record the outputs can be varied.

As we can see, the implementation using Java takes lesser time when compared to the implementation using Python.

VII. Individual Contribution

Ramya Nagarajan's Contributions

- Code implementation in Python using Python 2.7 on Linux OS.
- Sorting elements in chunks done using merge sort.
- Generated graph using number of elements as $x - axis$ and time taken as $y - axis$.

Sneha Tilak's Contributions

- Code implementation in Java using JDK 1.8 on Eclipse IDE on Windows OS.
- Sorting elements in chunks done using quick sort.
- Comparison and verification of results obtained.
- Generated graph using size of input (in KBs) as $x - axis$ and time taken (in Seconds) as $y - axis$.
- Time Complexity analysis of External Sort.

We worked together to analyze and compare the results obtained from both the implementations.

VIII. Application

When deciding on a sorting technique, we must not only consider the time complexity but also the space complexity applicable. This sorting technique minimizes the number of times a record is accessed. Use of the External Sorting Algorithm are -

- Business Applications - Instances where a transaction file updates a master file.
- Databases - To sort sets of data that are too large to be loaded entirely into memory. External Sorting helps in reducing the number of disk I/Os. It is used in implementing the relational operations, projection of data (where user requests a subset of the fields in a file) where the duplicate records are removed, joining two fields and creating a new file.

IX. Future Work

We can further improve the performance of the code by implementing the following:

1. Parallelism - We can use multiple threads to speed up the process. Also, if there are multiple input files or disks, we can access them in parallel to improve the read and write speed.
2. Increase Hardware Speed - We can allot more RAM or memory in order to reduce the number of reads or passes.
3. Increase Software Speed - We can use a better sorting technique like Radix Sorting for the first phase. Compacting the files once written/read from will reduce the I/O time.

Acknowledgment

We would like to thank Professor Funda Ergun and the AIs for guiding us in completing this project.

References

[1] https://en.wikipedia.org/wiki/External_sorting - *External Sorting*

[2] http://lemire.me/blog/2010/04/01/external-memory-sorting-in-java/ - *External-Memory Sorting in Java*

[3] http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L17-ExternalSortEX2.htm - *External Sorting: Example of multiway external sorting*

[4] *Introduction to Algorithms (CLRS)*

[5] personal.denison.edu/ havill/TuesdaysF07/latex/paper.tex - *Psudocode for Quick Sort*

[6] http://www.comp.nus.edu.sg/ sma5503/recitations/03-heap.pdf - *Psudocode for Min Heap*