How to sort 100GB worth of strings

Given a harddrive with 120GB, 100 of which are filled with the strings of length 256 and 2 GB Ram how do I sort those strings in Java most efficiently? How long will it take?

java sorting

edited Sep 15 '12 at 2:47
Bill the Lizard ♦
161k 107 390 671

asked Apr 2 '10 at 11:49
Christian
6,530 12 61 114

1 You would almost definitely need an *in-place* sorting algorithm. – stakx Apr 2 '10 at 11:50

1 How are the strings delimited? As in: is it one sequence with null characters between them or are they a bunch of buffers with some set length and filled with characters. My basic question is: How easy is it to find and move the strings around? – Travis Gockel Apr 2 '10 at 11:52

10 This was a Google interview question. I know, because I got the question when I interviewed there. – Paul Tomblin Apr 2 '10 at 11:53

1 It is just as important to have the right tool for the job as well as the right strategy. A 160 Gb drive costs £28, assuming there is no space available somewhere on some server or PC. In a company like google, I would find that hard to believe. Having the right hardware would simplify and speed up the task enormously, saving the company money and getting the task finished earlier. Disk space is reusable, your time isn't. – Peter Lawrey Apr 2 '10 at 17:38

15 @Peter: sure, but at Google they probably want to know that you can code against the limits of available hardware, *as well as* having the ability to call for more hardware when needed. I very much doubt that if you said, "why can't I just have a machine with 128GB RAM, I know you have some", they'd say "yes, correct answer, next question". They might say, "yes, good answer, now would you like to assume you can't have one, or would you like us to increase the numbers and start again?" ;-) – Steve Jessop Apr 4 '10 at 1:23

show **1** more comment

7 Answers

I am basically repeating Krystian's answer, but elaborating:

Yes you need to do this more-or-less in place, since you have little RAM available. But naive in-place sorts would be a disaster here just due to the cost of moving strings around.

Rather than actually move strings around, just keep track of which strings should swap with which others and actually move them, once, at the end, to their final spot. That is, if you had 1000 strings, make an array of 1000 ints. array[i] is the location where string i should end up. If array[17] == 133 at the end, it means string 17 should end up in the spot for string 133. array[i] == i for all i to start. Swapping strings, then, is just a matter of swapping two ints.

Then, any in-place algorithm like quicksort works pretty well.

The running time is surely dominated by the final move of the strings. Assuming each one moves, you're moving around about 100GB of data in reasonably-sized writes. I might assume the drive / controller / OS can move about 100MB/sec for you. So, 1000 seconds or so? 20 minutes?

But does it fit in memory? You have 100GB of strings, each of which is 256 bytes. How many strings? 100 * 2^30 / 2^8, or about 419M strings. You need 419M ints, each is 4 bytes, or about 1.7GB. Voila, fits in your 2GB.

edited Apr 2 '10 at 14:31

answered Apr 2 '10 at 12:15


Thorbjørn Ravn Anderser
41.1k 7 88 204


Sean Owen
39.5k 8 68 112

3 Good point, but I would be a little bit worried worried about seek times. This method sounds like requiring a lot of seeks, so sustained throughput of 100MB/sec may not be the best measure. We have to move around 100*2^30/2^8 ~ 100*2^22 strings. If we are not careful, we might need say one seek per 100 writes. If each seek is 4ms ~ 2^-8 sec, it would take something like 2^14 sec ~ 4.5 h. – Krystian Apr 2 '10 at 12:57

1 @Kristian - I think that estimate of 1 seek per 100 records written is highly optimistic ... – Stephen C Apr 2 '10 at 14:28

4 "The running time is surely dominated ..." I challenge you to prove this. In particular, I am interested in how quicksort compares the strings, seeing you don't have enough RAM to store them all. (You are not proposing to read from the disk for every comparison, are you? If you are, you might wish to read up about seek times.) – meriton Apr 5 '10 at 22:46

1 Very rough? More like wild-ass-guess. Assuming you do one disk seek for each comparision done by quick-sort, where quick sort selects pivots optimally and each disk seek takes 0.01 seconds, the time spent seeking is 419000000 * log(419000000) * 0.01 ~= 4 years. Granted, you will have some cache hits so it would not be quite as bad. Still, this solution is at least two orders of magnitude slower than the approach described by Stephen C. – meriton Apr 6 '10 at 22:29

4 Sorry if this answer got under your bonnet, this is just StackOverflow fun-time discussion. I like your thinking, those seeks really must dominate even with optimistic assumptions. Go petition the OP to change the accepted answer so we can rest easy! – Sean Owen Apr 6 '10 at 23:14

show **5** more comments

A1. You probably want to implement some form of **merge-sort**.

A2: Longer than it would if you had 256GB RAM on your machine.

Edit: stung by criticism, I quote from Wikipedia's article on merge sort:

*Merge sort is so inherently sequential that it is practical to run it using slow tape drives as input and output devices. It requires very little memory, and the memory required does not depend on the number of data elements.

For the same reason it is also useful for sorting data on disk that is too large to fit entirely into primary memory. On tape drives that can run both backwards and forwards, merge passes can be run in both directions, avoiding rewind time.*

edited Apr 2 '10 at 12:02

answered Apr 2 '10 at 11:52


High Performance Mark
52.6k 3 46 87

Merge sort does not necessarily sort in-place, which would mean that it is impossible to do. – Travis Gockel Apr 2 '10 at 11:54

1 Not impossible at all ! – High Performance Mark Apr 2 '10 at 12:03

Care to elaborate, @High? You haven't addressed merge-sort's space requirements. – Marcelo Cantos Apr 2 '10 at 12:05

3 No, I don't care to elaborate. Merge sort is well known and well documented; for example the Wikipedia article gives a much better explanation than I could write. As for space requirements, my understanding is that one can write merge sort to use whatever memory is available. Years ago I used to do merge sorts on tapes with 10GB data in 64K RAM. – High Performance Mark Apr 2 '10 at 12:13

3 We should clarify what memory we are talking about. High Performance Mark is certainly correct in that mergesort can operate with O(1) RAM. However, what about disk space? With only 20% more memory than your dataset, during the final merge, you can not hold both an input and the output list fully on disk, because that would already require 150 GB. How do you intend to implement merging to free that memory early? – meriton Apr 5 '10 at 22:23

show **1** more comment

Here is how I'd do it:

Phase 1 is to split the 100Gb into 50 partitions of 2Gb, read each of the 50 partitions into memory, sort using quicksort, and write out. You want the sorted partitions at the top end of the disc.

Phase 2 is to then merge the 50 sorted partitions. This is the tricky bit because you don't have enough space

on the disc to store the partitions AND the final sorted output. So ...

1. Do a 50-way merge to fill the first 20Gb at the bottom end of disc.
2. Slide the remaining data in the 50 partitions to the top to make another 20Gb of free space contiguous with the end of the first 20Gb.
3. Repeat steps 1. and 2. until completed.

This does a lot of disc IO, but you can make use of your 2Gb of memory for buffering in the copying and merging steps to get data throughput by minimizing the number of disc seeks, and do large data transfers.

EDIT - @meriton has proposed a clever way to reduce copying. Instead of sliding, he suggests that the partitions be sorted into reverse order and read backwards in the merge phase. This would allows the algorithm to release disc space used by partitions (phase 2, step 2) by simply truncating the partition files.

The potential downsides of this are increased disk fragmentation, and loss of performance due reading the partitions backwards. (On the latter point, reading a file backwards on Linux / UNIX requires more syscalls, and the FS implementation may not be able to do "read-ahead" in the reverse direction.)

Finally, I'd like to point out that any theoretically predictions of the time taken by this algorithm (and others) are largely guesswork. The behaviour of these algorithms on a real JVM + real OS + real discs are just too complex for "back for the envelope" calculations to give reliable answers. A proper treatment would require actual implementation, tuning and benchmarking.

edited Nov 20 '12 at 23:58

answered Apr 2 '10 at 14:19

Stephen C

283k 21 237 510

Time estimate based on how much data is written (assuming that the computation can be done in parallel and hence is free): 100GB (first phase) + 100GB (final output) + 80GB (slide 1) + 60GB (slide 2) + 40GB (slide 3) + 20GB (slide 4) = 400GB written. About four hours, assuming a conservative 30MB/s sustained write. Faster on decent hardware, but what decent hardware only has 2GB of RAM? ;-) – Steve Jessop Apr 3 '10 at 13:12

... but add some time for the fact that reading/sorting/writing in phase 1 can't be parallel. Also a possible quibble over what "given 2GB of RAM" means. You've assumed the availability of 2GB contiguous address space all backed by RAM not swap, which I think is fair enough given that it's a hypothetical question. But if the *machine* has 2GB RAM and 32-bit addressing, your chunks in the first phase will have to be smaller, resulting in a more-than 50 way sort later. Eventually, a too-many-way merge will get slow. – Steve Jessop Apr 3 '10 at 13:15

I think that an N-way merge can be done with logN comparisons per record written. – Stephen C Apr 3 '10 at 14:10

Sounds about right, since then you can sort N inputs using an N way merge in O(N log N) comparisons, as expected. These things have a habit of ending up zero-sum. If the block size doesn't matter much, then I'd guess that in practice you can speed up the first phase, by having a block reading, a block writing, and a block sorting, simultaneously. Worth an experiment, at least, and if you're writing to the same place you're reading from (not where that data came from originally) that may or may not be ideal. I've never timed the effects of disk seeks. – Steve Jessop Apr 4 '10 at 1:16

Average seek time is 8 to 10ms for typical HDDs is ... according to pcguide.com/ref/hdd/perf/perf/spec/posSeek-c.html – Stephen C Apr 4 '10 at 1:36

show **5** more comments

Sounds like a task that calls for External sorting method. Volume 3 of "The Art of Computer Programming" contains a section with extensive discussion of external sorting methods.

answered Apr 2 '10 at 11:58

Krystian

1,127 5 10

@Krystian, do you know of an external sort that doesn't require 2n space? – Marcelo Cantos Apr 2 '10 at 12:07

add a comment

I think you should use BogoSort. You might have to modify the algorithm a bit to allow for inplace sorting, but that shouldn't be too hard. :)

answered Jan 20 '12 at 22:44

Alderath

1,177 1 8 22

+1 - for pure audacity :-) – Stephen C Oct 6 '12 at 7:41

add a comment

You should use a trie (aka: a prefix tree): to build a tree-like structure that allows you to easily walk through your strings in an ordered manner by comparing their prefixes. In fact, you don't need to store it in memory. You can build the trie as a tree of directories on your file system (obviously, not the one which the data is coming from).

answered Apr 2 '10 at 12:01

Itay Maman
16.3k 4 40 80

add a comment

AFAIK, merge-sort requires as much free space as you have data. This may be a requirement for any external sort that avoids random access, though I'm not sure of this.

answered Apr 2 '10 at 12:04

Marcelo Cantos
96.4k 13 184 246

See my comment to your comment below. – High Performance Mark Apr 2 '10 at 12:13

add a comment

Not the answer you're looking for? Browse other questions tagged java sorting or ask your own question.