# Exploring Big Data Sorting

Alex Sverdlov
alex@theparticle.com

## 1 Introduction

Many data operations have sorting at their core. For example, joining of two tables is often accomplished by sorting both tables on the join key followed by a merge operation to do the actual join. Group-by aggregates can be performed by sorting on the group-by key, and doing a single scan of the data to calculate aggregates.

A lot of the sorting methods that are taught and analyzed in algorithms classes are not suitable for large datasets. For example, bubble-sort, insertion-sort, heap-sort, etc., require that the dataset fit in memory, and do not scale well for cases when disk or multiple machines are involved (a typical case with Big Data).

In these class notes, we are going to take a look at merge-sort and quick-sort, and how they generalize to big data. Both are divide-and-conquer and both have bits that can be done in parallel when sorting on a cluster.

## 2 Mergesort

A typical merge sort starts by cutting the array in half, recursively sorting each half, and then performing a merge. The merge operating takes two sorted arrays, initializes both read pointers to the beginning of each array. We compare the two elements under the read pointers to each other, and output the smallest one. Advance read pointer for the element we consumed, and repeat until both input pointers have gone through the whole array.

Generalizing merge sort to bigger datasets seems trivial: instead of cutting the array (or dataset) into two, cut it into $N$ buckets (often such that each bucket is small enough to fit in memory). In fact, this often is exactly how big data is stored already (slices of the entire dataset are often stored in multiple files on multiple machines). This generalized merge sort just needs to sort each data bucket, and perform the merge operation: and this is where we run into trouble.

Doing a merge on two files is trivial, merging potentially thousands of files is not. One approach we could try is merging pair-by-pair and iterating. After first iteration we end up with half as many files, and after another iteration with half as many files again—eventually we end up with just 1 file, which has everything sorted. Parts of this pair-by-pair merge

can happen in parallel. The problem is that with each iteration, we are using less and less parallelism—and we are rewriting the entire dataset (often to disk).

One solution is to maintain a heap queue (priority queue) that will give us the read pointer with the smallest element. That way our process can consume $N$ buckets at the same time, with minimal data overwriting. The performance of heap-queue is $O(\log N)$, where $N$ is the number of buckets.

The problem with the above approach is that this heap-queue business cannot happen in parallel, and we are forced to pipe the entire dataset through this single node for the merge operation.

# 3 Quicksort

A typical quicksort starts by selecting a pivot element. Often it is the first element, or middle element. Quicksort then scans (often from both ends) the array and relocates all elements less than the pivot to the left half, and all elements greater than pivot to the right half. Then recursively sort the left and right half—with the resulting array being fully sorted.

There are a few obvious improvements that we can make: if we could choose a pivot that is an exact median of the data, the split would happen exactly in the middle, and we would get the best-case $O(n \log n)$ performance. Most problems with quicksort are related to choosing the pivot that is an outlier, resulting in a skewed split and degrading performance. This rarely happens—to get worst performance possible one would have to select pivots multiple times and end up with the worst pivot every-time. Note: if we pick the first element as pivot, and the array is already sorted, then this unlikely-scenario will happen every time: so pick a random location for the pivot.

Another way of finding a better pivot is by sampling. One approach is to pick three elements, and use the median of those three as the pivot. This mostly ensures that the worst-case does not happen (or has a much lower chance of happening).

Generalizing to bigger datasets is not as simple as merge sort: for one, we want more than one pivot—in fact, we want $N$ pivots, such that each resulting partition can fit in memory. One way of doing this is to sample the data—with enough samples to get us a hint of what the distribution looks like: perhaps $100 * N$ samples. Out of these samples, we estimate the $N$ pivots—these are the boundaries.

There is a function called `lower_bound` that can be used to find the bucket number of every element given the bucket boundaries. It is $O(\log N)$, so relatively quick to run it on a large dataset.

Once each record has a bucket number, we shuffle the entire dataset based on that number—this number represents which machine/file the results go to. Each record will only land on one machine, and boundaries ensure that there are no overlaps. Now records in each bucket can be sorted (independency, and in parallel), and the resulting dataset is sorted (no merging necessary).

Note that unlike merge-sort, this generalized quicksort has no single-point bottlenecks: sampling can be done in parallel, figuring out the boundaries is operating on a relatively

small dataset (and this may have to be done in a single machine). Applying `lower_bound` on each record and shuffling is easily parallelized. Sorting each bucket can also be done in parallel. So unlike merge-sort, quicksort generalizes to big data much better.

# 4 Conclusion

For big data: $Quicksort > Mergesort$