SAD2 exam report

Radoslaw Niemczyk
radn@itu.dk

Sigurt Bladt Dinesen
sidi@itu.dk

December 7, 2015

Ordering movie ratings in a dynamic setting

Ranking has become an increasingly important problem, with ever-growing datasets both in the industry and in academia.

Ranking is used commercially to provide customers with proposed subsets of products, to make it easier to choose a product of interest. Ranking provides what might be the simplest type of recommendation system: recommend the items that score best on a global total ranking, independently of the recipient of the recommendation. For this it might be valuable to have the entire set of products ordered by rank, but it can be sufficient to have a *top-k* list, of the k highest (and/or lowest) ranked items.

In a static setting, where the global rank does not evolve over time, there are several simple algorithms: sorting the full data set, or probing for the k largest elements. In a dynamic setting however, were the data evolves over time, those algorithms would need to be re-run to maintain current solutions (as the underlying input changes, so does the global ranking, and hence the correct solution). Hence, there may be better algorithms that maintain and evolve the solution according to the input.

To motivate our approach, we present it through the following concrete problem: A service maintains a list of movies, and lets users rank movies on some scale (say $[1, 10] \cap \mathbb{N}$). Since ratings for individual movies will likely be spread out in the valid range, rather than clustered on a single agreed-upon value, we will consider the average rating of each movie. We want to provide an ordered set of the best $m \leq n$ movies (In particular, m might be equal to n). The data is dynamic in the sense that the *true* global ranking may change with every new user-supplied rating. We assume new ratings will be frequent enough that re-sorting the average ranks is infeasible.

In this scenario, we want to explore trade-offs between memory consumption, correctness, and currency. In particular it should be possible to achieve a memory bound lower than m by sacrificing correctness, achieve both by sacrificing currency, i.e. allowing the output to become somewhat outdated as the underlying data set evolves.

Following is a list of techniques that we deemed promising at the project outset, and hence wished explore the effectiveness of with respect to our problem:

Parallelization The use of parallelized sorting algorithms, such as parallelized merge- or radix- sort.

Approximation We would like to explore the possibility of performing an approximate sort. User–provided rankings are often inconsistent. For instance, if a user ranks a move m lower than a movie m', does that mean he liked m less, or that he likes that genre less? It is unclear weather or not the scale is linear, and the same user may have different experiences on different days. This means that an *exact* ordering might not be necessary. If it provides a speedup, it may be well worth it to perform a partial sorting of the rankings — such that a top–ten might really be a top–eight, plus 12 and 14.

Online sorting Sorting is an inherently offline problem — you can not sort a set without having all the values. However, for problems where the

full data set is not immediately available, onlineness can be achieved, or approximated, by sorting partial data sets, and then in the end sorting the whole. Exploiting the efficiency of certain sorting algorithms when dealing with partially ordered data, to lessen the time spend waiting for the data.

Online sorting

From `Algorithms and Theory of Computation Handbook`:

> An algorithm that must process each input in turn, without detailed knowledge of future inputs.

Not every online algorithm has an offline counterpart. And often also online algorithms cannot match the performance of offline algorithms. But in our scenario we want to create competive, suitable solution - the online algorithm seems to be great fit.

- Storing in right order each input batch on heap.
- Taking adventage of algorithm like Insertion Sort
- Partial sotring,Odds algorithm - which can be valueable for defining optimal stopping
- By combining techniques listed above.

With n inputs we are creating a heap - which takes n log n operations. This gives us a required data. Then if the next input arives the algorithm inserts each of new input with log n. If the item is already on the heap then we are changing it average. So each input is processed in input size log input size. Which is very good score but we have to story each unique movie.

Insertion sorts allows us to eaisly distribiute results into multiple locations which could be adventage for very large sets.

Odds algorithm, Partial sorting - approach is tempting but it is compromising the quality of output. Also if we managed to utilize a algorithm based on partitioning we might be able to parallize it. Which can be more important factor for effiency than algorithm cost.

A Stream Based Approach

Streaming algorithms provide excellent solutions to many problems where data sets are large enough that we wish (or need) to sacrifice correctness for low memory usage and time consumption. From `Ikonomovska-Zelke`:

> Streaming algorithms drop the demand of random access to the input. Rather, the input is assumed to arrive in arbitrary order as an input stream. Moreover, streaming algorithms are designed to settle for a working memory that is much smaller than the size of the input.

To be precise, they require that the size of the working memory is sublinear in both the cardinality of the stream and the universe. Due to this nature of streaming algorithms they are not commonly used for problems that require analysing parts of non-constant, non-parameterized size of the data set, for each given input. Hence an approach to solve our problem — sorting – based on streaming algorithms will provide some interesting trade-offs.

We begin with a definition of our stream, and a simple algorithm for our problem. Our input is a turnstile stream. The universe U is the set of movies in our database, and each stream item is a pair $(j, r) \in U \times ([1, 10] \cap \mathbb{N})$. With our definition of movie ranks we get a strict turnstile stream — in fact the delta r for every stream element is positive.

The simplest algorithm to solve our problem is then simply calculating the frequency vector for the stream, and sort it when our algorithm is queried. However, this algorithm is not very satisfactory. The working memory is sublinear in the cardinality of the stream, but not in the universe size. It does not provide a current solution either, as we have to sort the frequency vector when queried. On the positive side, the solution provided by the algorithm is correct. To be precise, this algorithm would require $O(m)$ working memory, and time for each stream item, m being the number of distinct movies in the stream, or the universe size $|U|$. A query would then require $O(n \log(m))$ time, which is unacceptable.

1. Maintaining ordering with each item
2. Approximation based on sampling
3. sketching approach to approximation

For 1 — maintaining ordering — we diverge from the classical definition of streaming algorithms. Both by using linear memory, and by using non-constant time for each stream item. Although the latter is common, it is unusual to use time non-constant in the stream size. We note that this solution will be equivalent to maintaining an ordered set of running averages, and is thus the same as the online-sorting approach. Knowing that for most stream items (j, r), the movie represented by j will already be in the ordered set, it might be possible to achieve insertion time linear in the number of inversions needed to reorder the set, though it is not clear that this improves on the $log(n)$ insertion time in binary search trees..

For 2 — sampling the stream — there seem to be two obvious approaches; Sampling as normal (reservoir sampling) over the stream, or sampling over the movies, deliberately making sure that all movies are represented in the sample, or that the number of samples... Neither seem to be much good

Finally 3 — sketching — looks good. We can use `Count-Min sketch` to get approximate point-queries (although we would query all points, or at least max-k points). It may even be possible to combine this solution with 1, to avoid sorting the samples when queried, though this requires further analysis.