SAD2 exam report

Radoslaw Niemczyk
radn@itu.dk

Sigurt Bladt Dinesen
sidi@itu.dk

December 13, 2015

Introduction

Ranking has become an increasingly important problem, with ever-growing datasets both in the industry and in academia. In the 21. century gathering large data sets is no longer considered novel. More and more people are getting access to the internet and more content like films, music is created and evaluated by them. 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.

In this project, we analyze different approaches to keeping a ranked data set, while receiving live updates to the data. We choose algorithms to analyze based on factors like: time and space requirements, and the quality of the solutions in case of approximation algorithms.

Ordering movie ratings in a dynamic setting

To motivate our approach, we present our project through the following concrete problem: An `IMDB` like service maintains a list of movies, and lets users rank movies on some scale (say $[1, 10] \cap \mathbb{N}$). We want to provide the set of movies, sorted by user-supplied ratings. It is likely that ratings of individual movies will span a large portion of the valid range, with some movies getting a few maximum ratings, but a low overall score. It is therefore necessary to use an aggregate for the ranking, such as the average user-rating for each movie. For simplicity, we assume that ratings cannot be changed or deleted, only added. This makes it possible to maintain running averages, as opposed to the full set of ratings, without integrity loss

If ratings are added frequently, we can consider the input — the set of $(user, rating, movie)$ triplets — to be *dynamic*. The input is dynamic in the sense that the *true* global ranking may change over time, with every added user-supplied rating. In a static setting, where the global rank does not evolve over time, there are simple algorithms that solve our problem. E.g. sorting the full data set would do. In a dynamic setting, where 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.

We will start by analyzing the most direct approach of maintaining a binary tree of the movies, updating it with each rating in the input. We will then explore different algorithms in light of the 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. We note that in our `IMDB` like scenario, the running time of getting a total ordering is irrelevant. The running time per rating is all that matters, though they will often be closely related.

General approach

One of the basic and also efficient concepts is using a data structure which will provide mechanism for extracting interesting data efficiently.

Binary Search Tree

Creation of BST is an insertion of each element in $O(logn)$ time. As a result we get sorted structure of our inputs - accessing all of them takes $O(n)$. Using this approach requires to store all input elements - which consumes $O(n)$ space. And if we want to find Low k elements of set we still need to travers all set.

Heap and Min-Max Heap

Ordering items using Heap takes $O(n \log n)$. Where creation of a heap takes $O(n)$. Then we are traversing through heap n times using fuction which takes $O(logn)$. Which results in $O(n + n \log n)$, which leads to our prevoiusly stated complexity.

In scenario when we are only interested in Top and Low k elements of a set we might takes different heap based approach called Min-Max Heap.

Provides properties of regular heap like a:

- $O(n)$ built time,
- $O(logn)$ for insertion and deletion

But even more important are properties carried by Min-Max variation. Which are guaranteeing us a constant time for Min and Max operations. The Top/Low(k) finding operation can be performed in $O(k \log k)$ time.

Using this approach requires to store all input elements. It takes $O(n)$ for large sets.

Based on : `M. D. Atkinson, J.R. Sack, N. Santoro, and T. Strothotte, Communications of the ACM, October, 1986, Min-Max Heaps and Generalized Priority Queues`

General approach extension

Parallelization

Utilizing power of multicore architecture in this scope is very tempting. But our current approach is not supportive. By using data structure like Binary Tree or Heap in a single thread scenario we are receiving multiple benefits like logarithmic time for insertion for each element or limited space usage. For each new element we have to only perform insertion with logarithmic cost. For scenario when we are interested in intermidiate results it is a very big adventage against sorting algorithms like QuickSort or MergeSort.

But these algorithms have adventage over data structure based sorting - they are much more prone to Parallelization. Depending on the architecture or use case even in light of need processing partial input it might be profitiable to resort our data multiple time using benefits of multicore, multithreaded computing if time is our main constraint.

Creation of BST is an insertion of each element in $O(logn)$ time. As a result we get sorted structure of our inputs - accessing all of them takes $O(n)$. Using this approach requires to store all input elements - which consumes $O(n)$ space. And if we want to find Low k elements of set we still need to travers all set.

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. Parallization in algorithms is very natural for sorting/selection. But it is still very overlooked. It might be a challening to see the real impact of this - because we are affecting the overall time consmuption by using it. Moreover usally we are adding more complexity and overhead to our solution - by creating and mantaining parallel task and jobs. This mean on of our point of emphasis will analysis of the pros and cons carried by this approach.

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 exactness 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 $S = \alpha_1, \alpha_2, \ldots, \alpha_{|S|}$. 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. We let $|S|$ denote the cardinality of the stream.

The simplest algorithm to solve our problem is then simply calculating the normalized frequency vector for the stream, and sort it when queried. However, this is not very satisfactory. The working memory is sublinear in $|S|$, but linear in the universe size $|U|$. 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 exact. To be precise, this algorithm would require $O(m)$ working memory, and constant time for each stream item, m being the number of distinct movies in the stream, which we assume to be $|U|$. A query would then require $O(m \log(m))$ time.

The rest of this section discusses techniques that alleviate these problems with different trade-offs.

Order Maintenance

In the simple algorithm, results were not *current*, because every query required a sort of the frequency vector — which is long. If we allow ourselves to use more than constant processing time per stream element, this problem can be solved by maintaining an *always sorted* data structure with pointers into the frequency vector, such as a search tree. This algorithm is equivalent to maintaining an ordered set of running averages, and is thus the same as the online-sorting approach described previously.

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 should improve the $log(n)$ insertion time in binary search trees.

In summary, we get $O(log(n))$ processing time for each stream element, but queries can now be performed in $O(n)$. This a very natural change from the simple algorithm, that really only moves the required work from the time of querying, to the time of input.

Approximation based on sampling

In addition to the lack of currency, the simple algorithm requires a lot of memory. Not surprisingly, streaming algorithms lets us buy a lower memory requirement, at the cost of exactness.

There seem to be two obvious approaches; normal reservoir sampling over the stream, or sampling over the movies, deliberately making sure that all movies are represented in the sample. The later obviously fails to improve memory consumption, and is only suggested because taking a random sample over the stream seems dangerous, as it might well discard movies from the stream, by not picking any of their ratings for the sample. As it turns out, this is not a big problem.

Although, a uniformly random sample of ratings is not an answer to our original problem, it does have some nice properties: As stated in `Ikonomovska--Zelke` (p. 243); all $\binom{|S|}{k}$ possible samples, where k is the sample size, are equally likely to be our result. It follows directly from this that popular ratings for a movie are more likely to occur than unpopular ones. However, the likelihood of a movie occurring in the sample similarly correlates to how many ratings it has, not – as we would want – how high it's average rating is. In other words, reservoir sampling gives us a random set of ratings, with no guarantee that the sample contains good movies.

The common reservoir sampling algorithm described in `Ikonomovska-Zelke` remembers k samples. After the first k, each stream element $\alpha_i, k < i \leq |S|$, replaces one of the k samples with probability k/i, choosing the sample to be replaced at random.

The sampling approach solves our memory issues by parametrizing the memory consumption. The algorithm uses $O(k)$ memory, independent of both $|U|$ and $|S|$.

We can modify the reservoir sampling algorithm to keep running averages instead of samples, modifying the sample when seeing a stream element that refers to the same movie, and only replacing a sample when seeing an element that is not already in the set of remembered movie samples. We then no

longer get sampled ratings, but estimates of the movie averages — which is what we wanted. Maintaining running averages can be thought of as keeping a frequency vector, and changing the stream so that each element (j, r) becomes (j, r'), where r' is the change r imposes on the kept average for movie j. Despite the possibility of r' being negative, we are still in the strict turnstile model, as the average will never be negative, regardless of what subset if S we look at.
The problem of missing movies persists however. If wish achieve the $O(k)$ memory bound, we can not hope to find the exact solution using sampling in this way. However, if we limit our problem to find the top-l movies, we can.

> DRAFT NOTE: This would be a good place to analyze the quality of the solution — in terms of k

If we alter our running-average sampling to replace the *smallest* sample, instead picking one at random, the probability of our k samples containing the top $l < k$ movies increases. `Metwally et al` present an algorithm that does just that, albeit for frequencies rather than averages.

> DRAFT NOTE: Now actually go into `Metwally's solution, adapt it to the running averages, and analyse it.` Also mention it's relation to sketching.

Approximation based on Sketching

> DRAFT NOTE: Consider moving this to before the sampling section. And maybe extract the general stuff to the (or a new) super section.

Sketching lets us sacrifice exactness for lower memory consumption, without having to worry about loosing entire movies from the solution.