Answer more versus appreciated more

Zhaoen Su

suzhaoen@gmail.com

Abstract

I report the solutions to finding top-10 users who have most posts and top-10 users who have most accepted posts from data of posts (47.8 GB) and users (1.81 GB). The project starts with analyzing the properties of the data such that optimal framework and strategy are chosen. The implementation applies information compressing, faster parsing and data pre-processing. Various data structures, primitive list, hashtable, red-black tree and trie, are explored to store tens of millions of items in the memory. To obtain the top-K items, three algorithms, using heap, partial sorting and partition, are explored and their time complexities and memory costs are compared. With the pre-processed data, optimal data structures and algorithms I explored, the Java program finds the top-10 users who have most posts with 32 MB memory in 17.2 seconds, and the top-10 users who have most accepted posts with 65 MB memory in 31.2 seconds.

I. Introduction

Background

The "Top-K" problem is nature in many scenarios. It is desirable for department stores to know the top-K sale products among millions of products; it is valuable to know the top-K popular websites visited; it is useful for researcher to extract some most informatic records among many. In our scenario, we want to extract the top-K records from posts of web forums. Specifically, people asks questions in Stack Overflow via a post. Zero or more users may answer the question via posts. The askers usually accepted of them as the best answer. Two interesting questions are asked.

- The top-10 users who have answered most posts (Top-K problem)
- The top-10 users who have most answered most posts and accepted (Top-K-Accepted problem)

The relation in the data set

In data Posts.xml, a log records the PostId, Type, AcceptedPostId or ParentPostId, OwnerId. Other fields are irrelevant. Type = 1 means a question post and it usually has an acceptedPostId denoting the post accepted by the asker. Type = 2 means an answer post and it has a ParentPostId denoting the question post.

To solve the Top-K problem, we can simply iterate through the posts of Type-2 in one pass, count the frequencies of OwnerIds and pick the top-K OwnerIds.

To solve the Top-K-Accepted problem, there are two approaches.

- To iterate through the posts of Type-2, use the ParentPostId of the current post to look up the question post. Update the frequency of the OwnerId only when the answer post is accepted in the question post log.
- To iterate through the posts of Type-1, use the AcceptedPostId of the current post to track the answer post, and update the frequency of the OwnerId of the answer post.

Since a question post is usually followed by multiple answer posts, the later approaches is faster as its tracking always leads to an OwnerId to update. Thereby, I will adapt the later approach and keep only the following fields

- Type-1 : PostId, Type, AcceptedPostId
- Type-2 : PostId, Type, OwnerId

In data Users.xml, a log records the information of an OwnerId. Because the mapping between OwnerId and DisplayName is one-to-one, it is redundant to keep both OwnerId and DisplayName in the data processing. Therefore, I will not include the DisplayNames before to display the final result. Besides, I will only find the mapping of those top-K users.

Data set and computational setup

Posts.xml is 47.8 GB

Users.xml is 1.81 GB.

Total number of posts is 34 million;

Total number of answered posts is 10 million;

Total number of users is 6.4 million;

Computer: CPU (i5-2430, 2.4, 2.4 GHz), RAM (8 GB)

Language: Java 8

Probabilistic versus deterministic approaches

Before I outline the rest of the report, I will compare two approaches. Facing a great of amount of samples, it is reasonable to think of probabilistic approach by sampling. Indeed there are algorithms developed to solve top-k problems in some circumstance with decently low error rates[1]. Probabilistic top-K algorithms trade space and speed for accuracy. Is the trade safe? My answer is that it depends. Sampling is safe when the size characterized by the average count of each item is large, as the uncertainty is proportional to $1/\sqrt{\text{size}}$. In ref.1 where the top-k words in books are persuaded, the size is large as all of the not-super-rare English words fills only a couple pages while the whole text data is about several books. In our problem, however, the average count of each user is only 4 – 5 and probabilistic approach is thus not suitable.

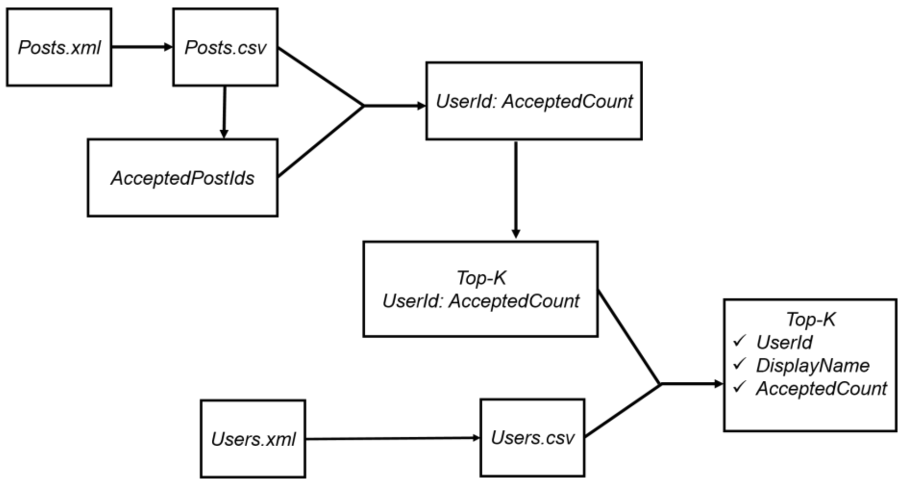

Figure.1 The process chart of the implementation

Outline of the report

Both top-k problems are solved, however, to simplify the discussion, we will focus on the Top-K-Accepted problem as the Top-K problem is a trivial case of it. The body the report follows the processes illustrated in Fig.1. In Section II, I will discuss the necessities to compress the data and pre-process the data. Sections III describes various data structures to store the posts that are accepted. In Section IV, we build the mapping between UserId and the count of accepted posts. Section V discusses and compares some algorithms to find the top-K users. Finally, we pre-process the Users.xml file to find the DisplayNames for the top-K users.

II. Data pre-processing

Parse: pattern match versus string comparison

Posts.xml is loaded and parsed line after line. The parse process can be optimized. A commonly seen approach is to pattern match a single field in a pass. It reads and parses 0.1 million lines in 3.45 s in average; using direct string comparison to parse all fields in a single pass is found to be ***10-20 %*** faster. It reads and parses 0.1 million lines in 3.43 s in average. Here I have subtracted the time to read the file which is 3.31 seconds per 0.1 million lines. The test code can be found in CompressingTest.java.

Information compressing

Can we parse the Post.xml and save the whole data of relevant fields in memory? Yes, if we compress the information. The naïve version saves each log as a Post object that has four String objects. It processes 6 million posts and throws an “OutOfMemory” error. Two places to compress. 1). Frist, the fields are number-like and should not be saved as strings. A 7-digit (10 million) PostId (or AcceptedPostId/Ownerld) that takes 14 bytes can be converted to an 32-bit integer. TypeIds can represented by 1-bit boolean values. Further can be done by bitmap. 2). Furthermore, the difference is more significant as every Java objects require at least 16 bytes for the object header and reference [2]. When the fields of Post objects are Java Strings (also Java objects), at least 90 bytes are used for useless information. Considering both, my compression adapts primitive data structures and uses a list of int-type to save the fields. With such improvements, the whole data can be loaded to memory! The associated code is in CompressingTest.java.

Shall we carry on with the data in the memory? Probably not. Keeping such amount of data is heavy for the memory. The Java program claims up to 1.7 GB RAM near the end of the loading. Besides the approach can fail if larger data is given. However, the idea of compressing is applied to the following processes.

Pre-processing

Instead of holding such much data in memory, I save it to a file in comma-separated values (csv) format. The reading is convenient as the file is a 2D table. The pre-processed file is much better than the original Posts.xml. As shown in Table.1, the size is greatly reduced. More importantly, reading all lines (without any other operations) only takes 14 seconds, which is acceptable. The code to accomplish the conversion is Xml2Csv.java.

Table. 1 The size and read time of original and pre-processed Posts files

	Size	Time to read all lines
Posts.xml	47.8 GB	29 minutes
Posts.csv	560 MB	14 seconds

Two improvements are implemented during the data pre-processing:

- To save memory, every 0.5 million posts, I dump the data to the csv file and clear the buffer for the next 0.5 million posts such that the buffer is never too large.

- Data cleaning is performed at the same time by filtering the non-log lines, the question posts with no answers and answer posts with no owerIds (yes, very strange, but there are).

Later we will need to read the data from csv file. Naturally, one would think of using database because data access in file is fast there. Here we will stick to non-database approaches.

III. Store AcceptedPostIds (cleverly)

We need to know what posts are accepted in advanced, as later it will serve as a look-up table when iterating the posts in Posts.csv. To know these accepted posts, we iterate the Type-1 posts in Posts.csv and record the AcceptedPostId. In the end, we record something that might look like 7, 23, 14,…, 31239482,…,33023751. Besides, we need to be able look up these AcceptedPostIds fast. Constant time is good, O(lgn) might be fine. There tens of millions of such AcceptedPostsIds, how to store them in memory? I have explored the following data structures.

Store the Ids as list indices

If we store these number directly, we need at least 100 MB because there are big numbers and we have to use int-type to store them (32 bit × 34 million). Here, cleverly, we take advantage of the properties of the Ids: they are integers between zero and 34 million. We claim a continue memory space and build a boolean-type list of 34 million long. The i-th list element is true only if the i-th post is accepted. Theoretically, we only need 1 bit× 34 million! In Java the boolean datatype costs more and the full list costs 35 MB. The look-up time is also perfect, O(1)! This simple data works well and is adapted in the final implementation.

This approach is scalable. Supposed the maximum allowed size of list is L and the maximum value of the Ids is I (I > L). We can save a Ids into the (I // L)-th list at the position of (I % L). Low memory cost and fast look-up remain. Notice that process can be done in parallel, such that mapreduce applies.

Hashtable

Hash map and Hash set are explored as proper Hashtables also offer O(1) look-up. To mitigate the effect of frequent collisions and re-locations, I initialize the Java HashSet with various filling factors and sufficient capacities (allowed). However, all of my attempts fail and the best could only complete 5 million Ids. Although the program throws “OutOfMemory” errors finally, memory shortage might not be the really cause because the memory used before the crash is less than 20 MB. I suspect that the hash processing slows down due to frequent collisions. Next, let’s try data structures with open access to the memory: linked data structures.

Trie

We can treat the AcceptedIds as words. A trie node has at most 10 children and a boolean value to denote if an Id ends here. The look-up time is still O(1) as the the trie depth is at most 8 (34 million Ids). Unlike hashtable, a trie adds nodes via references and should be able to add nodes as many as possible before the memory is too full. It turns out that the building processes also fail before completing 30 million nodes. First, I used the paritial trie of the apache library[3]. I suspect that because the tree build process is complex as paritial trie is implemented to take full advantage of nodes. For this reason, I built the most common tries by myself whose building process is simple and straightforward. One uses lists for the children nodes, the other uses hashmaps for the children nodes. Unfortunately, neither managed to build 30 million nodes.

Red-black tree

Another linked data structure I have tried is red-black tree (O(lgn look-up time)), i.e., TreeSet in Java. It stops making new nodes after 5.4 millions.

Overall, all of advanced data structure fail to store 10 million items. None of them shows memory usage above 20 MB when the program crashes. I have explored some but not comprehensive. It would be very interesting to know the cause and solution. Alternatively, dividing the AcceptedIds into multiple blocks and build a smaller data structure for each blocks can be a direction for now. Code of all test are in DataStructureTest.java.

IV. Build UserId : AcceptedCount relation

Now with the AcceptedPostId look-up, we can iterate the type-2 posts in Posts.csv and update the counts of UserIds if the posts are accepted, in one pass. In the end, we record UserId: AcceptedCount pairs. The same question is back: what data structure to store them? The data structure that my final implementation adapted is list of int-type where the i-th list element records the count of accepted posts by UserId of i. The other data structures mentioned in Section III fail for the same reason.

Similarly, this approach is scalable and can be combined with parallel methods. However, one should remember that ultimately all sub-lists must be combined into a single list (or other data structure). Reducing information in individual sub-lists is not allowed (unless a UserIds has zero count. To see that, let’s picture a mistake that we trash a tiny count, say 2, of of User-L in a block in order to reduce one node. Finally, its rank is K+1. The rank-K user, however, has all counts kept and now it exceeds User-L by only 1 count. In other word, User-L can be a top-K user if we did not trash its small contributions in individual blocks.

V. Find top-k

With all the UserId : AcceptedCount relations in the memory, it is straightforward to find the top-K users. The original code iterates all relations K times to pick out the top-K users, giving raise to (K×n) time complexity and O(k) space cost. If the relation is between the list indices and list items, like in my final implementation, we still need to build an extra O(n) long sequence where each element stores UserId and AcceptedCount. The following alternatives algorithms provide better performances and the first is adapted to the final implementation.

Heap

Starting with a simple scenario, we build a maxHeap (with AcceptedCount comparator) to add all the UserId : AcceptedCount relations. It costs O(n) in time and space. Then we extract heap heads K times and the heads are the Top-K users in order. It takes K×O(lgn) in time and O(K) in space. Given that K << n, the time complexity is much better than (K×n). In practice, we should not and can not load all relations into the heap. It is memory expensive and may even give raise to “OutOfMemory” errors. I divide the relations into M blocks. Each block is added to a heap and only the top-K users of the heap is kept. The rest is trashed. Here differs from Section-IV where such process in individual is not allowed. To sum up, the time cost is O(n) + $\mathrm{KM}(\lg\frac{n}{M})$ and space cost is O(n/M) (if not operated parallel).

Partition

This algorithm builds a list of Users whose fields are UserId and AcceptedCount. It takes O(n) to find the users whose AcceptedCount is K-th largest then we use the user to partition the list in O(n). This approach is also done in multiple blocks. As a result, the time cost is O(2n) and space cost is O(n/M) (if not operated parallel).

Partial sorting

The algorithm maintains a K-long descending list (with AcceptedCount comparator). While iterating the UserId : AcceptedAccount relations, if an AcceptedAcount is larger than that of last list element, insert the new user and pop the last list element. The time complexity is at worst O(K×n) and space cost is O(k). Dividing the work in block is not necessary, unless parallel computing is applied.

Pruning and parallelism

For all approaches, while iterating the UserId : AcceptedCount relations, we shall not add the users whose AcceptedCounts are zero into the heap or list. The pruning speeds up the process and can save memory.

All approaches can be combined with parallel methods such as mapreduced or parallelism for further performance improvement.

VI. Look up the DisplayNames of the top-K users

Finally, we have top-K users (User.Id, User.AcceptedCount). What is missing is their DisplayNames. They can be found in the Users.xml as there is one-to-one mapping between User.Ids and DisplayNames. To keep the tradition of data pre-process, I parsed and converted the Users.xml to Users.csv where each line is a valid user record that saves UserId and DisplayName only. The performance difference is shown in Table.2. We first sort the top-K users according to their UserIds, then we iterate the users in the Users.csv file in one pass to find out all of the User.DisplayNames.

Table. 2 The size of read time of original and pre-processed Users files

	Size	Time to read all lines
Users.xml	1.81 GB	75 seconds
Users.csv	115 MB	3 seconds

VII. Performance of the final implantation

Top-K problem

With the pre-processed csv files, the final complete implementation takes 17.2 seconds in average. Approximately 14 second is on reading Posts.csv and 3 seconds is on Users.csv. The maximum memory used is 32 MB in average, mostly used to store the UserId : Count relation in the list. The finally result is printed in Table.3.

Table.3 Top-10 Stack Overflow users who have most answer posts.

Id	DisplayName	Answer Counts
22656	Jon Skeet	33,475
29407	Darin Dimitrov	21,216
115145	CommonsWare	18,207
157882	BalusC	16,963
100297	Martijn Pieters	16,619
6309	VonC	16,114
17034	Hans Passant	15,819
19068	Quentin	15,697
34397	SLaks	14,519
23354	Marc Gravell	13,619

Top-K-Accepted problem

With the pre-processed csv files, the final implementation takes 31.2 seconds in average to get the final results. Approximately 28 seconds is on reading Posts.csv twice and 3 seconds is on Users.csv. The maximum memory used is 65 MB in average, mostly used to the boolean IsAccepted list and the UserId : AcceptedCount relation in the list. The finally result is printed in Table.4.

Table.4 Top-10 Stack Overflow users who have most accepted answer posts.

Id	DisplayName	Accepted Counts
22656	Jon Skeet	20,580
29407	Darin Dimitrov	12,514
100297	Martijn Pieters	11,952
157882	BalusC	11,497
115145	CommonsWare	10,926
17034	Hans Passant	9,478
6309	VonC	8,088
157247	T.J. Crowder	7,457
23354	Marc Gravell	7,196
34397	SLaks	6,804

VIII. Summary

With pre-processed data, the Java program manages to solve the top-K problem of both types from 34 million posts and 6.4 million users with low memory costs and decent times. The report displays how data pre-processing, information compression and improved string match greatly reduce the memory and loading time costs. Various data structures are explored to store tens of millions of items in memory. The advantages and disadvantages in terms of time complexity and memory costs of some algorithms to obtain the top-K items are explored.

Regarding feature improvements, the time cost can be reduced further by exploring alternative file format (such as json?) or combing database; For memory aspect, looking for solutions to storing tens of millions of nodes and hashtable items in memory is important, and can be necessary in some circumstances.

Reference

[1] http://highscalability.com/blog/2012/4/5/big-data-counting-how-to-count-a-billion-distinct-objects-us.html

[2] http://java-performance.info/overview-of-memory-saving-techniques-java/

[3] https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/trie/PatriciaTrie.html