Algorithm to find top 10 search terms

I'm currently preparing for an interview, and it reminded me of a question I was once asked in a previous interview that went something like this:

"You have been asked to design some software to continuously display the top 10 search terms on Google. You are given access to a feed that provides an endless real-time stream of search terms currently being searched on Google. Describe what algorithm and data structures you would use to implement this. You are to design two variations:

(i) Display the top 10 search terms of all time (i.e. since you started reading the feed).

(ii) Display only the top 10 search terms for the past month, updated hourly.

You can use an approximation to obtain the top 10 list, but you must justify your choices."
I bombed in this interview and still have really no idea how to implement this.

The first part asks for the 10 most frequent items in a continuously growing sub-sequence of an infinite list. I looked into selection algorithms, but couldn't find any online versions to solve this problem.

The second part uses a finite list, but due to the large amount of data being processed, you can't really store the whole month of search terms in memory and calculate a histogram every hour.

The problem is made more difficult by the fact that the top 10 list is being continuously updated, so somehow you need to be calculating your top 10 over a sliding window.

Any ideas?

algorithm data-structures

edited Aug 7 '14 at 7:00 — NoobEditor — 7,558 2 9 34

asked Jul 15 '10 at 22:40 — del — 1,465 1 16 33

2 `the 10 most frequent items in an infinite list` - What is the most frequent item in a list containing one 1, two 2's, three 3's, ..., n n's, ...? What a stupid interview question – BlueRaja - Danny Pflughoeft Jul 15 '10 at 23:09

8 @BlueRaja - It's not a stupid interview question, it's a bad interpretation on the OP's part. It's not asking for the most frequent items in an infinite list, it's asking for the most frequent items of a finite subsequence of an infinite list. To continue your analogy, `what is the most frequent item in the subsequence [2; 2; 3; 3; 3; 4; 4; 4; 4; 5; 5] of your sequence?` – IVlad Jul 15 '10 at 23:15

Google does 1.21*10^8 searches per hour (searchengineland.com/...). – Unreason Jul 16 '10 at 9:40

@BlueRaja - It's certainly a difficult question, but I don't see why it's stupid - it seems representative of a fairly typical problem that companies with huge data sets are faced with. @IVlad - Fixed it as per your suggestion, bad wording on my part! – del Jul 16 '10 at 22:53

15 Answers

Well, looks like an awful lot of data, with a perhaps prohibitive cost to store all frequencies. **When the amount of data is so large** that we cannot hope to store it all, we enter the domain of **data stream algorithms**.

Useful book in this area: Muthukrishnan - "Data Streams: Algorithms and Applications"

Closely related reference to the problem at hand which I picked from the above: Manku, Motwani - "Approximate Frequency Counts over Data Streams" [pdf]

By the way, Motwani, of Stanford, (edit) was an author of the very important "Randomized Algorithms" book. ~~**The 11th chapter of this book deals with this problem**~~. **Edit:** Sorry, bad reference, that particular chapter is on a different problem. After checking, I instead recommend section 5.1.2 of *Muthukrishnan's* book, available online.

Heh, nice interview question.

edited Jul 17 '10 at 11:53

answered Jul 15 '10 at 23:35

Dimitris Andreou
5,741 1 17 28

1 +1 Very interesting stuff, there should be a way on sites to tag "to read" stuff. Thanks for sharing. – Ramadheer Singh Jul 15 '10 at 23:52

@Gollum: I have a to-read folder in my bookmarks; you could just do that. I know those links are being added to mine :) – Cam Jul 15 '10 at 23:55

+1. Streaming algorithms is exactly the topic here, and Muthu's book (the only book written so far, AFAIK) is great. – ShreevatsaR Jul 16 '10 at 0:26

+1. Related: en.wikipedia.org/wiki/Online_algorithm. btw, Motwani died recently, so perhaps *was* an author is more accurate. – Aryabhatta Jul 16 '10 at 16:34

Oh. Damn. I didn't know :(– Dimitris Andreou Jul 16 '10 at 17:31

If *exact* results were required, it would take a lot of storage, and it would be pretty slow. If approximation is allowed, the following algorithm might be part of a solution that's efficient in time and space.

Start with an empty map (red-black tree). The keys will be search terms, and the values will be a counter for the term.

1. Look at each item in the stream.
2. If the term exists in the map, increment the associated counter.
3. Otherwise, if the map has fewer candidates than you're looking for, add it to the map with a count of one.
4. However, if the map is "full", decrement the counter in each entry. If any counter reaches zero during this process, remove it from the map.

This process will identify any search term that occurs with a specified frequency. For example, if you limit the map to 99 entries, you are guaranteed to find any term that occurs more than 1/(1 + 99) (1%) of the time. This is different from finding the most frequently occurring items. For example, if the top item occurs only 0.5% of the time, you'll miss it unless you set the threshold lower. Also note that the final values of the counters don't mean much. The most common element in the stream might end up with the lowest counter.

Note that you can process an infinite amount of data with a fixed amount of storage (just the fixed-size map). The amount of storage required depends only on the threshold of interest, and the size of the stream does not matter.

Perhaps you buffer one hour of searches, and perform this process on that hour's data. If you can take a second pass over this hour's search log, you can get an exact count of occurrences of the top "candidates" identified in the first pass.

Any candidates that really do exceed the threshold of interest get stored as a summary. Keep a month's worth of these summaries, throwing away the oldest each hour, and you would have a good approximation of the most common search terms.

A Lengthy Example

Let's walk through an application of the algorithm where we are searching for any terms that occur *more* than 1% of the time. Let's say the stream contains 10^{12} elements, and just for fun, that the first $10^{10} + 1$ elements are "foo", and the remainder of the stream comprises distinct words—terms that appear only once.

The stream length doesn't matter; only our threshold. To find anything that occurs more than 1% of the time, we need track only 99 elements.

After the first $10^{10} + 1$ elements, our tracking set contains just "foo" with a count of $10^{10} + 1$.

The next 98 elements are added to the tracking set with a count of 1, because the set is not "full".

Then, the next element (the 99th non-"foo" term) has never been seen before, but the set is full, so all counters are decremented by 1. That means "foo" has a count of 10^{10}, and the other

98 elements are reduced to zero. Because they are reduced to zero, they are deleted from the tracking set.

The pattern repeats, with the next 98 terms added for tracking, only to be removed by the 99th, which also decrements the "foo" counter by one.

You can see that the counter for "foo" is decremented once every 99 elements, or a total of 10^{10} - 1 times before the end of the stream is reached. At the end of the stream, the count for "foo" is still 2, and it is retained as a candidate that exceeded the threshold.

I should also note that the tracking set also contains 98 "junk" elements with a counter of 1. The counter values are irrelevant. To ascertain how frequent these terms are, a second pass over the stream is required.

edited Jul 17 '10 at 8:19

answered Jul 15 '10 at 23:40

erickson
139k 25 217 325

I believe that this solution can act as a filter, reducing the number of search terms of interest to you. If a term makes it into the map, begin tracking it's actual statistics, even if it falls out of the map. You could then skip the second pass over the data, and produce a *sorted* top 10 from the limited statistics you gathered. – Dolph Jul 16 '10 at 0:30

I like the elegant way of pruning less-searched terms from the tree by decrementing the counters. But once the map is "full", won't that require a decrement step for every new search term that arrives? And once this starts happening, won't this result in newer search terms being quickly removed from the map before they have a chance for their counters to increment sufficiently? – del Jul 16 '10 at 23:20

1 @del - Keep in mind that this algorithm is for locating terms that exceed a specified threshold frequency, not necessarily for finding the most common terms. If the most common terms fall below the specified threshold, they generally won't be found. Your concern about removing newer terms "too quickly" might be associated with this case. One way to look at this is there are real "signals" in popularity, they will stand out noticeably from the "noise." But sometimes, there are no signals to be found, just random search static. – erickson Jul 16 '10 at 23:59

@erickson - Right - what I'm getting at is that the assumption with this algorithm is that the top 10 words are uniformly distributed across the measurement window. But so long as you keep the measurement window small enough (e.g. 1 hour), this would probably be a valid assumption. – del Jul 17 '10 at 1:15

1 @erickson, while a *uniform* distribution is not a requirement, I wonder how this would work in a more realistic distribution (power-law, Zipf). Lets assume we have N distinct words, and keep the red-black tree of capacity K, hoping it will end up with the K most frequent terms. If the cumulative frequency of the terms of (N - K) words is larger than the cumulative frequency of the K most frequent words, the tree in the end is guaranteed to contain garbage. Do you agree? – Dimitris Andreou Jul 17 '10 at 11:15

This is one of the research project that I am current going through. The requirement is almost exactly as yours, and we have developed nice algorithms to solve the problem.

The Input

The input is an endless stream of English words or phrases (we refer them as `tokens`).

The Output

1. Output top N tokens we have seen so far (from all the tokens we have seen!)
2. Output top N tokens in a historical window, say, last day or last week.

An application of this research is to find the hot topic or trends of topic in Twitter or Facebook. We have a crawler that crawls on the website, which generates a stream of words, which will feed into the system. The system then will output the words or phrases of top frequency either at overall or historically. Imagine in last couple of weeks the phrase "World Cup" would appears many times in Twitter. So does "Paul the octopus". :)

String into Integers

The system has an integer ID for each word. Though there is almost infinite possible words on the Internet, but after accumulating a large set of words, the possibility of finding new words becomes lower and lower. We have already found 4 million different words, and assigned a unique ID for each. This whole set of data can be loaded into memory as a hash table, consuming roughly 300MB memory. (We have implemented our own hash table. The Java's implementation takes huge memory overhead)

Each phrase then can be identified as an array of integers.

This is important, because sorting and comparisons on integers is **much much faster** than on

strings.

Archive Data

The system keeps archive data for every token. Basically it's pairs of `(Token, Frequency)`. However, the table that stores the data would be so huge such that we have to partition the table physically. Once partition scheme is based on ngrams of the token. If the token is a single word, it is 1gram. If the token is two-word phrase, it is 2gram. And this goes on. Roughly at 4gram we have 1 billion records, with table sized at around 60GB.

Processing Incoming Streams

The system will absorbs incoming sentences until memory becomes fully utilized (Ya, we need a MemoryManager). After taking N sentences and storing in memory, the system pauses, and starts tokenize each sentence into words and phrases. Each token (word or phrase) is counted.

For highly frequent tokens, they are always kept in memory. For less frequent tokens, they are sorted based on IDs (remember we translate the String into an array of integers), and serialized into a disk file.

(However, for your problem, since you are counting only words, then you can put all word-frequency map in memory only. A carefully designed data structure would consume only 300MB memory for 4 million different words. Some hint: use ASCII char to represent Strings), and this is much acceptable.

Meanwhile, there will be another process that is activated once it finds any disk file generated by the system, then start merging it. Since the disk file is sorted, merging would take a similar process like merge sort. Some design need to be taken care at here as well, since we want to avoid too many random disk seeks. The idea is to avoid read (merge process)/write (system output) at the same time, and let the merge process read form one disk while writing into a different disk. This is similar like to implementing a locking.

End of Day

At end of day, the system will have many frequent tokens with frequency stored in memory, and many other less frequent tokens stored in several disk files (and each file is sorted).

The system flush the in-memory map into a disk file (sort it). Now, the problem becomes merging a set of sorted disk file. Using similar process, we would get one sorted disk file at the end.

Then, the final task is to merge the sorted disk file into archive database. Depends on the size of archive database, the algorithm works like below if it is big enough:

```
for each record in sorted disk file
     update archive database by increasing frequency
     if rowcount == 0 then put the record into a list
end for

for each record in the list of having rowcount == 0
     insert into archive database
end for
```

The intuition is that after sometime, the number of inserting will become smaller and smaller. More and more operation will be on updating only. And this updating will not be penalized by index.

Hope this entire explanation would help. :)

edited Jul 16 '10 at 7:48

answered Jul 16 '10 at 7:41

SiLent SoNG
1,603 1 13 24

I don't get it. What kind of meaningful sorting or comparisons could one do in the integer IDs of the words? Aren't the numbers arbitrary? – Dimitris Andreou Jul 16 '10 at 10:27

Also, counting frequences of words is the very first example in Google's MapReduce paper (labs.google.com/papers/mapreduce.html), solving it scalably in a handful of lines. You could even move your data to google app angine and do such a MapReduce (code.google.com/p/appengine-mapreduce) – Dimitris Andreou Jul 16 '10 at 10:57

@Dimitris Andreou: Sorting on integers would be faster on strings. This is because comparing two integers is faster than comparing two strings. – SiLent SoNG Jul 16 '10 at 12:18

@Dimitris Andreou: the Google's mapreduce is a nice distributed approach on solving this problem. Ah! Thanks for providing the links. Ya, it would be good for us to sort using multiple machines. Nice approach. – SiLent SoNG Jul 16 '10 at 12:19

@Dimitris Andreou: So far I have only been considering single machine sorting approach. What a nice idea to sort in distribution. – SiLent SoNG Jul 16 '10 at 12:32

You could use a hash table combined with a binary search tree. Implement a `<search term, count>` dictionary which tells you how many times each search term has been searched for.

Obviously iterating the entire hash table every hour to get the top 10 is **very** bad. But this is google we're talking about, so you can assume that the top ten will all get, say over 10 000 hits (it's probably a much larger number though). So every time a search term's count exceeds 10 000, insert it in the BST. Then every hour, you only have to get the first 10 from the BST, which should contain relatively few entries.

This solves the problem of top-10-of-all-time.

The really tricky part is dealing with one term taking another's place in the monthly report (for example, "stack overflow" might have 50 000 hits for the past two months, but only 10 000 the past month, while "amazon" might have 40 000 for the past two months but 30 000 for the past month. You want "amazon" to come before "stack overflow" in your monthly report). To do this, I would store, for all major (above 10 000 all-time searches) search terms, a 30-day list that tells you how many times that term was searched for on each day. The list would work like a FIFO queue: you remove the first day and insert a new one each day (or each hour, but then you might need to store more information, which means more memory / space. If memory is not a problem do it, otherwise go for that "approximation" they're talking about).

This looks like a good start. You can then worry about pruning the terms that have > 10 000 hits but haven't had many in a long while and stuff like that.

answered Jul 15 '10 at 23:08

IVlad
22.3k 7 40 109

Rough thinking...

For top 10 all time

- Using a hash collection where a count for each term is stored (sanitize terms, etc.)
- An sorted array which contains the ongoing top 10, a term/count in added to this array whenever the count of a term becomes equal or greater than the smallest count in the array

For monthly top 10 updated hourly:

- Using an array indexed on number of hours elapsed since start modulo 744 (the number of hours during a month), which array entries consist of hash collection where a count for each term encountered during this hour-slot is stored. An entry is reset whenever the hour-slot counter changes
- the stats in the array indexed on hour-slot needs to be collected whenever the current hour-slot counter changes (once an hour at most), by copying and flattening the content of this array indexed on hour-slots

Errr... make sense? I didn't think this through as I would in real life

Ah yes, forgot to mention, the hourly "copying/flattening" required for the monthly stats can actually reuse the same code used for the top 10 of all time, a nice side effect.

edited Jul 15 '10 at 23:52

answered Jul 15 '10 at 23:41

R. Hill
1,993 1 9 16

case i)

Maintain a hashtable for all the searchterms, as well as a sorted top-ten list separate from the hashtable. Whenever a search occurs, increment the appropriate item in the hashtable and

check to see if that item should now be switched with the 10th item in the top-ten list.

O(1) lookup for the top-ten list, and max O(log(n)) insertion into the hashtable (assuming collisions managed by a self-balancing binary tree).

case ii) Instead of maintaining a huge hashtable and a small list, we maintain a hashtable and a sorted list of all items. Whenever a search is made, that term is incremented in the hashtable, and in the sorted list the term can be checked to see if it should switch with the term after it. A self-balancing binary tree could work well for this, as we also need to be able to query it quickly (more on this later).

In addition we also maintain a list of 'hours' in the form of a FIFO list (queue). Each 'hour' element would contain a list of all searches done within that particular hour. So for example, our list of hours might look like this:

```
Time: 0 hours
      -Search Terms:
          -free stuff: 56
          -funny pics: 321
          -stackoverflow: 1234
Time: 1 hour
      -Search Terms:
          -ebay: 12
          -funny pics: 1
          -stackoverflow: 522
          -BP sucks: 92
```

Then, every hour: If the list has at least 720 hours long (that's the number of hours in 30 days), look at the first element in the list, and for each search term, decrement that element in the hashtable by the appropriate amount. Afterwards, delete that first hour element from the list.

So let's say we're at hour 721, and we're ready to look at the first hour in our list (above). We'd decrement free stuff by 56 in the hashtable, funny pics by 321, etc., and would then remove hour 0 from the list completely since we will never need to look at it again.

The reason we maintain a sorted list of all terms that allows for fast queries is because every hour after as we go through the search terms from 720 hours ago, we need to ensure the top-ten list remains sorted. So as we decrement 'free stuff' by 56 in the hashtable for example, we'd check to see where it now belongs in the list. Because it's a self-balancing binary tree, all of that can be accomplished nicely in O(log(n)) time.

Edit: Sacrificing accuracy for space...

It might be useful to also implement a big list in the first one, as in the second one. Then we could apply the following space optimization on both cases: Run a cron job to remove all but the top *x* items in the list. This would keep the space requirement down (and as a result make queries on the list faster). Of course, it would result in an approximate result, but this is allowed. *x* could be calculated before deploying the application based on available memory, and adjusted dynamically if more memory becomes available.

edited Jul 15 '10 at 23:59

answered Jul 15 '10 at 23:41

Cam

7,117 7 33 90

Exact solution

First, a solution that guarantees correct results, but requires a lot of memory (a big map).

"All-time" variant

Maintain a hash map with queries as keys and their counts as values. Additionally, keep a list f 10 most frequent queries so far and the count of the 10th most frequent count (a threshold).

Constantly update the map as the stream of queries is read. Every time a count exceeds the current threshold, do the following: remove the 10th query from the "Top 10" list, replace it with the query you've just updated, and update the threshold as well.

"Past month" variant

Keep the same "Top 10" list and update it the same way as above. Also, keep a similar map, but this time store vectors of 30*24 = 720 count (one for each hour) as values. Every hour do

the following for every key: remove the oldest counter from the vector add a new one (initialized to 0) at the end. Remove the key from the map if the vector is all-zero. Also, every hour you have to calculate the "Top 10" list from scratch.

Note: Yes, this time we're storing 720 integers instead of one, but there are much less keys (the all-time variant has a *really* long tail).

Approximations

These approximations do not guarantee the correct solution, but are less memory-consuming.

1. Process every N-th query, skipping the rest.
2. (For all-time variant only) Keep at most M key-value pairs in the map (M should be as big as you can afford). It's a kind of an LRU cache: every time you read a query that is not in the map, remove the least recently used query with count 1 and replace it with the currently processed query.

answered Jul 16 '10 at 0:21

Bolo
6,061 2 20 48

I like the probabilistic approach in approximation 1. But using approximation 2 (LRU cache), what happens if terms that were not very popular initially become popular later? Wouldn't they be discarded every time they are added, since their count would be very low? – del Jul 16 '10 at 23:44

@del You're right, the second approximation will only work for certain streams of queries. It's less reliable, but at the same time requires less resources. Note: you can also combine both approximations. – Bolo Jul 17 '10 at 15:43

Top 10 search terms for the past month

Using memory efficient indexing/data structure, such as tightly packed tries (from wikipedia entries on tries) approximately defines some relation between memory requirements and n - number of terms.

In case that required memory is available (**assumption 1**), you can keep exact monthly statistic and aggregate it every month into all time statistic.

There is, also, an assumption here that interprets the 'last month' as fixed window. But even if the monthly window is sliding the above procedure shows the principle (sliding can be approximated with fixed windows of given size).

This reminds me of round-robin database with the exception that some stats are calculated on 'all time' (in a sense that not all data is retained; rrd consolidates time periods disregarding details by averaging, summing up or choosing max/min values, in given task the detail that is lost is information on low frequency items, which can introduce errors).

Assumption 1

If we can not hold perfect stats for the whole month, then we should be able to find a certain period P for which we should be able to hold perfect stats. For example, assuming we have perfect statistics on some time period P, which goes into month n times.
Perfect stats define function `f(search_term) -> search_term_occurance` .

If we can keep all `n` perfect stat tables in memory then sliding monthly stats can be calculated like this:

- add stats for the newest period
- remove stats for the oldest period (so we have to keep `n` perfect stat tables)

However, if we keep only top 10 on the aggregated level (monthly) then we will be able to discard a lot of data from the full stats of the fixed period. This gives already a working procedure which has fixed (assuming upper bound on perfect stat table for period P) memory requirements.

The problem with the above procedure is that if we keep info on only top 10 terms for a sliding window (similarly for all time), then the stats are going to be correct for search terms that peak in a period, but might not see the stats for search terms that trickle in constantly over time.

This can be offset by keeping info on more than top 10 terms, for example top 100 terms, hoping that top 10 will be correct.

I think that further analysis could relate the minimum number of occurrences required for an entry to become a part of the stats (which is related to maximum error).

(In deciding which entries should become part of the stats one could also monitor and track the trends; for example if a linear extrapolation of the occurrences in each period P for each term tells you that the term will become significant in a month or two you might already start tracking it. Similar principle applies for removing the search term from the tracked pool.)

Worst case for the above is when you have a lot of almost equally frequent terms and they change all the time (for example if tracking only 100 terms, then if top 150 terms occur equally frequently, but top 50 are more often in first month and lest often some time later then the statistics would not be kept correctly).

Also there could be another approach which is not fixed in memory size (well strictly speaking neither is the above), which would define minimum significance in terms of occurrences/period (day, month, year, all-time) for which to keep the stats. This could guarantee max error in each of the stats during aggregation (see round robin again).

edited Jul 16 '10 at 9:59

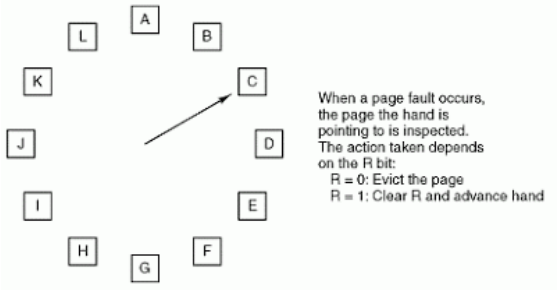
answered Jul 16 '10 at 9:23

Unreason

8,920 1 14 32

What about an adaption of the "clock page replacement algorithm" (also known as "second-chance")? I can imagine it to work very well if the search requests are distributed evenly (that means most searched terms appear regularly rather than 5mio times in a row and then never again).

Here's a visual representation of the algorithm:



answered Jul 17 '10 at 10:47

Dave O.

1,407 13 22

Store the count of search terms in a giant hash table, where each new search causes a particular element to be incremented by one. Keep track of the top 20 or so search terms; when the element in 11th place is incremented, check if it needs to swap positions with #10* (it's not necessary to keep the top 10 sorted; all you care about is drawing the distinction between 10th and 11th).

*Similar checks need to be made to see if a new search term is in 11th place, so this algorithm bubbles down to other search terms too -- so I'm simplifying a bit.

answered Jul 15 '10 at 22:54

Ether

35.3k 9 52 127

You'll want to limit the size of your hash table. What if you get a stream of unique searches? You need to be sure you don't prevent yourself from noticing a term that is searched for regularly but infrequently. Over time that could be the top search term, especially if all the other search terms are "current events", i.e. searched for a lot now, but not so much next week. Actually, considerations like these might be approximations you'd want to make. Justify them by saying, we won't catch these kinds of things because doing so makes the algorithm way more time/space expensive. – cape1232 Jul 15 '10 at 23:09

I'm pretty sure Google has a count of *everything* - some counts aren't maintained statically though, but rather calculated as needed. – Ether Jul 15 '10 at 23:18

sometimes the best answer is "I don't know".

Ill take a deeper stab. My first instinct would be to feed the results into a Q. A process would continually process items coming into the Q. The process would maintain a map of

term -> count

each time a Q item is processed, you simply look up the search term and increment the count.

At the same time, I would maintain a list of references to the top 10 entries in the map.

For the entry that was currently implemented, see if its count is greater than the count of the count of the smallest entry in the top 10.(if not in the list already). If it is, replace the smallest with the entry.

I think that would work. No operation is time intensive. You would have to find a way to manage the size of the count map. but that should good enough for an interview answer.

They are not expecting a solution, that want to see if you can think. You dont have to write the solution then and there....

answered Jul 15 '10 at 22:57

hvgotcodes
58.1k 10 93 157

12 The data structure is called a `queue` , `Q` is a letter :). – IVlad Jul 15 '10 at 23:09

2 If I were conducting the interview, "I don't know<stop>" would definitely not be the best answer. Think on your feet. If you don't know, figure it out - or at least try. – Stephen Jul 15 '10 at 23:30

in interviews, when I see someone with hibernate on their 7 page resume 5 times, and they can't tell me what ORM is, I end the interview immediately. Id rather they not put it on their resume and just say: "I don't know". Noone knows everything. @IVlad, I was pretending I was a C developer and trying to save bits...;) – hvgotcodes Jul 16 '10 at 15:50

One way is that for every search, you store that search term and its time stamp. That way, finding the top ten for any period of time is simply a matter of comparing all search terms within the given time period.

The algorithm is simple, but the drawback would be greater memory and time consumption.

answered Jul 15 '10 at 23:18

Jesse J
2,889 18 45

What about using a Splay Tree with 10 nodes? Each time you try to access a value (search term) that is not contained in the tree, throw out any leaf, insert the value instead and access it.

The idea behind this is the same as in my other answer. Under the assumption that the search terms are accessed evenly/regularly this solution should perform very well.

edit

One could also store some more search terms in the tree (the same goes for the solution I suggest in my other answer) in order to not delete a node that might be accessed very soon again. The more values one stores in it, the better the results.

answered Jul 17 '10 at 21:00

Dave O.
1,407 13 22

Dunno if I understand it right or not. My solution is using heap. Because of top 10 search items, I build a heap with size 10. Then update this heap with new search. If a new search's frequency is greater than heap(Max Heap) top, update it. Abandon the one with smallest frequency.

But, how to calculate the frequency of the specific search will be counted on something else. Maybe as everyone stated, the data stream algorithm....

answered Jul 21 '13 at 23:41

Chris
1

Use cm-sketch to store count of all searches since beginning, keep a min-heap of size 10 with it for top 10. For monthly result, keep 30 cm-sketch/hash-table and min-heap with it, each one start counting and updating from last 30, 29 .., 1 day. As a day pass, clear the last and use it as day 1. Same for hourly result, keep 60 hash-table and min-heap and start counting for last 60, 59, ...1 minute. As a minute pass, clear the last and use it as minute 1.

Montly result is accurate in range of 1 day, hourly result is accurate in range of 1 min

answered Sep 19 '14 at 4:19

Jingyi Fang
1
