Probabilistic Data Structures

Ethan Lowman

13 February 2015

1 Introduction

When dealing with large amounts of data, many of the classic data structures used to store, read, and update data become unwieldy and impractical. In certain cases, it is better to use a different class of data structures and algorithms which employ randomness to mitigate some of these problems.

2 Bloom Filters

2.1 Description

A Bloom filter is a fast, space-efficient probabilistic data structure used to check for set membership. As with all probabilistic data structures, the space-efficiency comes at a cost, though. Given a set of elements, a Bloom filter can determine whether an arbitrary element is *definitely not* a member of the set or whether the element is *probably* a member of the set, but it can not determine whether the element is definitely in the set. That is, Bloom filters are susceptible to false positives, but not false negatives. Additionally, items can not be deleted from a Bloom filter.

A Bloom filter consists of a vector of n boolean values (a bit array of n bits), initially all set to false (0), as well as k independent hash functions. Each hash function $h_0, h_1, \ldots, h_{k-1}$ maps an element to an integer in the range $[0, n-1]$.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Initial setup for $n = 10$, $k = 3$

Note that bit arrays are depicted here with bit 0 as the most significant bit to reinforce the boolean vector analogy.

For each element s_i in the set of all elements S, the bits with positions $h_1(s_i), h_2(s_i), \ldots, h_k(s_i)$ are turned on (set to 1). Any given bit may be turned on multiple times.

0	1	2	3	4	5	6	7	8	9
0	**1**	0	0	**1**	0	**1**	0	0	0

Adding s_1, where $h_1(s_1) = 1$, $h_2(s_1) = 6$, and $h_3(s_1) = 4$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	**1**	0	1	**1**	0	**1**

Adding s_2, where $h_1(s_2) = 4$, $h_2(s_2) = 7$, and $h_3(s_2) = 9$

Formally, the binary form (least significant bit is bit 0) of the bit array of a Bloom filter for a set of n elements $s_1, s_2, \ldots, e_n$ with k hash functions $h_1, h_2, \ldots, h_k$ is:

$$\begin{aligned}
&\Big((1 \ll h_1(s_1)) \mid (1 \ll h_2(s_1)) \mid \ldots \mid (1 \ll h_k(s_1))\Big) \mid \\
&\Big((1 \ll h_1(s_2)) \mid (1 \ll h_2(s_2)) \mid \ldots \mid (1 \ll h_k(s_2))\Big) \mid \ldots \mid \\
&\Big((1 \ll h_1(s_n)) \mid (1 \ll h_2(s_n)) \mid \ldots \mid (1 \ll h_k(s_n))\Big)
\end{aligned}$$

To query for an element in the set, pass it to each of the k hash functions to get k integer array positions. If any of the bits at these positions are 0, the element is definitely not in the set. If all of the bits are 1, then either the element is in the set or the bits were coincidentally set to 1 by hashes of other elements, resulting in a false positive.

Since the hashes of the elements are combined using a binary OR operation, it is not possible to recover the state of the bit array prior to adding an element. Therefore, the Bloom filter does not support removing elements. In the example above, bit 4 is turned on by the hash functions of both s_1 and s_2, so there is no way to remove s_1 from the set without explicitly storing all elements of the set, which is what the Bloom filter seeks to avoid. Permanent removal of an element from a Bloom filter can be simulated by adding the removed items to a complementary Bloom filter. Membership in the composite filter (the set defined by the relative complement of the additive set in the subtractive set) can be tested by determining if an element is in the additive filter but not in the subtractive filter. However, false positives in the subtractive filter are equivalent to false negatives in the composite filter, which may be undesirable.

2.2 Calculating the Probability of False Positives

$$m = \text{size of the bit array}$$
$$n = \text{no. of items in the set}$$
$$k = \text{no. of hash functions}$$
$$\text{Consider a particular bit } 0 \leq j \leq n-1$$
$$\text{Probability that } h_i(x) \text{ does not set bit } j : P(h_i(x) \neq j) = \left(1 - \frac{1}{m}\right)$$
$$\text{Probability that bit } j \text{ is not set by any hash: } P(Bit(j) = 0) \leq \left(1 - \frac{1}{m}\right)^{kn}$$
$$\left(1 - \frac{1}{x}\right)^x \approx \frac{1}{e} = e^{-1}$$
$$\begin{aligned}\left(1 - \frac{1}{m}\right)^{kn} &= \left[\left(1 - \frac{1}{m}\right)^n\right]^{kn/m} \\ &\approx \left(e^{-1}\right)^{kn/m} \\ &= e^{-\frac{kn}{m}}\end{aligned}$$
$$\begin{aligned}P(\text{False Positive}) &= P(\text{all } k \text{ bits of new element are already set}) \\ &= \left(1 - e^{-\frac{km}{m}}\right)^k\end{aligned}$$

This estimate for the probability of a false positive is not strictly correct, as it assumes independence for the probabilities of each bit being set. Assuming it is a close approximation, though, we can clearly see that the false positive rate decreases as the number of bits in the array increases and increases as the number of inserted elements increases. Intuitively, this makes sense because a larger bit array will result in less hash collision and inserting more elements makes the "on" bits more dense.

2.3 Optimizing the Number of Hash Functions

Given n and m, what k achieves a false positive probability p? That is, if we the cardinality of our set S and how much memory we have available for our bit array, how many hash functions should we use to achieve a false positive probability of at most p?

We previously determined that the false positive probability upper bound is

$$\left(1 - e^{-\frac{kn}{m}}\right)^k$$

The false positive probability upper bound is minimized by setting the derivative with respect to k equal to 0. To simplify the math, we can minimize the natural log of this function, which will yield

the same value of k since the natural log function is monotonically increasing.

$$\begin{aligned} 0 &= \frac{\partial}{\partial k}\left(1 - e^{-\frac{kn}{m}}\right)^k \\ 0 &= \frac{\partial}{\partial k}\left[k \cdot \ln\left(1 - e^{-\frac{kn}{m}}\right)\right] \\ &= \ln\left(1 - e^{-\frac{kn}{m}}\right) + \frac{kn}{m} \cdot \frac{e^{-\frac{kn}{m}}}{1 - e^{-\frac{kn}{m}}} \end{aligned}$$

This equation is solved by $k = (\ln 2) \cdot \frac{m}{n}$. Note that it is easy to verify this solution, but solving for it is actually pretty tricky, since it involves computing the inverse of a function of the form $f(W) = We^W$. The inverse of this function is called the Lambert W-function, or the omega function. At $k = (\ln 2) \cdot \frac{m}{n}$, the false positive probability takes on its global minimum value of

$$\left(\frac{1}{2}\right)^k \approx (0.6185)^{\frac{m}{n}}$$

We see from this that the false positive probability rate depends only on the ratio m/n.

2.4 Applications

- Google Chrome uses the Bloom filter to identify malicious URLs. Rather than check every URL against one of Google's databases and double the number of HTTP requests necessary for every page load or distribute their massive database of malicious URLs to every user, Google distributes a Bloom filter for their set of malicious URLs to every user. Only if the Bloom filter returns a positive result for a URL is a full check of the URL performed. This is a case where false positives are okay. As long as the false positive rate is low, it is acceptable to occasionally run unnecessary URL checks. Type I errors (false positives) are preferable to Type II errors (false negatives) in this case since it's better to be too cautious than not cautious enough.
- Google BigTable and Apache Cassandra (distributed databases) use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query.
- Bitcoin uses Bloom filters to speed up wallet synchronization.

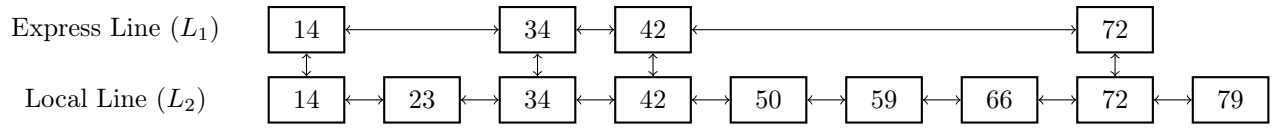
3 Skip Lists

3.1 Description

A skip list is a data structure that allows fast search within an ordered sequence of elements. When we want to store a sorted list of elements, we have several options. We could use a balanced binary search tree and search for elements in $O(\lg n)$ or use a sorted array and binary search for elements in $O(\lg n)$, among other methods. Using a sorted linked list, the worst case search time is $O(n)$, but we can augment this data structure to make the search faster.

The key optimization of skip lists is that multiple layers are used in order to allow us to skip some nodes in a linear search. The New York City 7th Avenue Line is a common real-world example of

skip lists, and it is the example we will use for now. The diagram below shows that an express line connects a few of the subway stations and a local line connects all stations. There are links (vertical arrows) between the lines at common stations.



SEARCH(x) for two linked lists

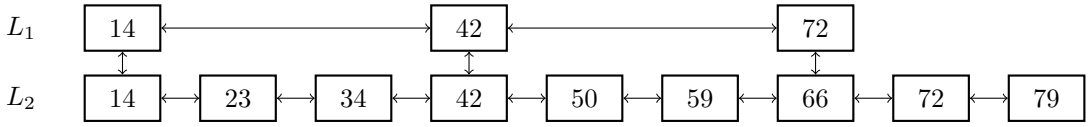
- Walk right in top linked list (L_1) until going right would go too far
- Walk down to bottom linked list (L_2)
- Walk right in L_2 until the element is found or going right would go too far

For two linked lists, the search cost is roughly $|L_1| + \dfrac{|L_2|}{|L_1|}$. We can minimize this search cost using calculus.

$$\begin{aligned}
\frac{d}{dL_1}\left[|L_1| + \frac{|L_2|}{|L_1|}\right] &= 0 \\
1 - |L_2||L_1|^{-2} &= 0 \\
1 &= |L_2||L_1|^{-2} \\
|L_1|^2 = |L_2| = n &\Rightarrow |L_1| = \sqrt{n}
\end{aligned}$$

So if $|L_1| = \sqrt{n}$ and $|L_2| = n$, the search cost is roughly

$$|L_1| + \frac{|L_2|}{|L_1|} = \sqrt{n} + \frac{n}{\sqrt{n}} = 2\sqrt{n}$$

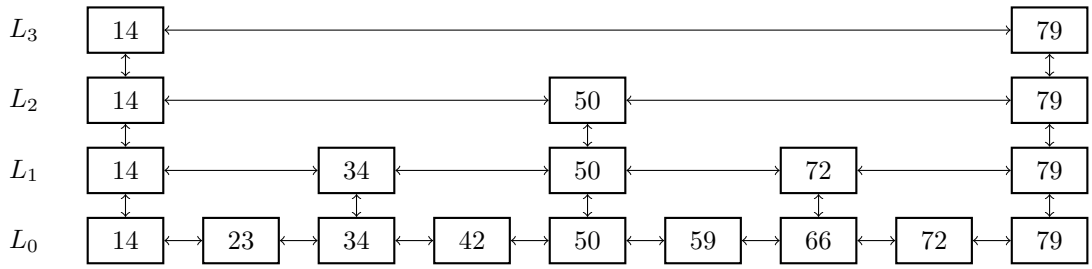


Two-level skip list with $|L_1| = \sqrt{n}$, $n = 9$.

If we repeat the same analysis for more sorted linked lists, we find the following minimized search costs:

$$\begin{aligned}
2 \text{ sorted lists} &\Rightarrow 2\sqrt{n} \\
3 \text{ sorted lists} &\Rightarrow 3\sqrt[3]{n} \\
k \text{ sorted lists} &\Rightarrow k\sqrt[k]{n} \\
\lg n \text{ sorted lists} &\Rightarrow \lg n \cdot \sqrt[\lg n]{n} \\
&= \lg n \cdot n^{\frac{1}{\lg n}} \\
&= \lg n \cdot 2^{\frac{\lg n}{\lg n}} \\
&= 2 \lg n
\end{aligned}$$

So, we are able to achieve logarithmic search time if we choose the number of lists to be $\lg n$. This $\lg n$ linked list structure is the *ideal skip list*, depicted below.



Structurally, this looks pretty similar to a binary tree because for the search process is very similar. If we are only performing searches, ideal skip lists perform very well. In practice, they may not be as good as binary search trees, but up to constant factors, they are just as good.

We need to support insertions and deletions, but we need to be careful that we maintain roughly the same structure as the ideal skip list so that search still costs logarithmic time. We could try to maintain a true ideal skip list, but then updates would cost linear time. Instead, we employ randomization to approximate an ideal skip list. This is where the probabilistic part of this probabilistic data structure comes into play.

INSERT(x)
- **SEARCH**(x) to see where x fits in the bottom list (L_0)
- Insert x into the bottom list
- Flip a fair coin. If it's heads, promote x to the next level up. Repeat until the coin is tails. If x is promoted to the top-most level, there are a number of ways to proceed, depending on the chosen implementation. One approach is to cap the number of levels. For example, if we allow at most 32 levels, then in principle the list can store 2^{32} elements. Another approach is to always promote on heads, adding levels as necessary, trusting that the number of levels will be limited by the improbability of many consecutive heads. Yet another approach is to allow the addition of at most one level per insertion. The choice of which approach to use for level addition is probably not critical.

Since the probability of promotion to the next level is $1/2$, on average $1/2$ of the elements promoted 0 levels, $1/4$ of the elements promoted 1 level, $1/8$ of the elements promoted 2 levels, and so on.

DELETE(x)
- **SEARCH**(x) to find its location in the bottom list.
- Follow the list-to-list pointers up to remove x from all lists containing it.

Two important lemmas to note are:

- With high probability, every search in an n-element skip list costs $O(\lg n)$.
- With high probability, n-element skip lists have $O(\lg n)$ levels.

3.2 Applications

One of the main advantages of skip lists over balanced binary search trees is that they are well suited for concurrent access. When accessing a balanced binary search tree concurrently, updates require mutex locks (flags that prevent two concurrent processes from accessing the same resources) on a large part of the tree. This severely hurts performance, so we'd like to minimize the number of nodes we have to lock to perform an update. Skip lists do this very effectively, since updates only modify a small part of the whole structure.