Bloom Filters

One of the fundamental operations on a data set is **membership testing**: given a value \( x \), is \( x \) in the set? So far we have focused on data structures that provide exact answers to this question. But sometimes an answer that is probably correct is good enough.

Consider this situation: suppose you are running a customized web search service. When a user enters a query (for example, the user might enter “stegosaurus artichoke uranium”), you locate and display the top 10 pages that match the query. Unfortunately each web search takes a long time. Fortunately you have the facility to cache local copies of the search results – so if the query is repeated you can access the local copies of the results without repeating the web search.

But experience has shown that the vast majority of queries are never repeated. Real world data suggests that out of the set of all search queries about 75% are never repeated – which means that the other 25% are repeated gazillions of times. This partially explains why Google is so frighteningly good at correctly guessing what a query is, given just the first word or two – we’re all searching for the same things. It’s both comforting and depressing to think that humanity is so predictably uniform. Long before the Internet, Isaac Asimov used the idea of predictable social group behaviour as the basis for his brilliant “Foundation” trilogy (yes, I know there were other books added to the series, but they don’t hold up to the standard of the first three) ... in which the invented science of “psychohistory” is used to map out the future. But I digress.

So you don’t want to cache the results of every query – most of that information will never be accessed again. The strategy adopted by Akamai Technologies (for a similar problem) is to cache the search results after the *second* occurrence of the query.

But how can we detect that a query has been made before? We could maintain a sorted array of all the queries – searching this would take \( O(\log n) \) time, but adding a new query would take \( O(n) \) time. A Red-Black tree would give us \( O(\log n) \) time for searching and inserting. If \( n \) is very large, even \( O(\log n) \) time may be too slow – ideally we would like constant search time ... and that suggests a hash-table.

So we could take the search query, run it through a hash function, and then store the words of the search query in the appropriate element of a hash table (using whatever collision resolution technique we choose). Then for each query we check to see if it is in the hash table. If so, this is a repeat query – we check the cache to see if the results are there. If they are not (which happens when we are experiencing the first repetition of the query) we perform the web search and cache local copies. If the search results are already in the local cache, we win!

From this point on, I’m going to refer to search queries as “keys” since that is the role they
play in our analysis. In the domain of search queries, the keys are just the words that make up the query. In a practical application we might apply some sort of normalization of the words such as converting them all to lower-case, and putting them in alphabetical order - but these details don’t concern us here.

The hash table is going to need to be pretty big. The number of keys to store in it is going to be large, and storing the keys (the words of each query) is going to take a lot of memory, so this table is going to be a space-hog (not to be confused with “Pigs In Space”, the classic sci-fi TV series featured on The Muppet Show).

So here’s a slightly different approach. What if we are willing to accept a small number of incorrect results from the “is this key in the set?” test? There are two types of errors:

- false negative – this key is in the set, but our test says it isn’t
- false positive – this key is not in the set, but our test says it is

In the application we are discussing, a false negative results in performing the web search when we probably didn’t need to. Conversely, a false positive results in checking the cache, not finding the results, and then performing the web search. Since our goal is to \textit{never} do a web search if we don’t have to, we want to make sure that false negatives never happen. But a false positive only costs us the time to check the cache – which should not be too costly since the cache is in local memory.

We can achieve the goals of preventing false negatives and reducing the memory requirements of the structure by replacing the space-hog hash table with a bit-vector of some size. Now we can hash the key into this table and simply set the bit at that location to 1.

Certainly this makes false negative results impossible – when a key is added, the appropriate bit will be set to 1 and never set back to 0, so if we ask “Is that key in the set?” we are guaranteed a positive response. But false positives are possible – if two keys hash to the same location, both will test positive even if only one of them has actually been added to the set.
We can predict the probability of a false positive result on a membership test for an arbitrary key. Suppose the bit-vector has length $m$, and we have “stored” $n$ different keys in it. Each time we “store” a key, we choose a bit and set it to 1 (even if it was already 1). For each bit, the probability that it will be chosen is $\frac{1}{m}$ so the probability that it will not be chosen is $1 - \frac{1}{m}$. Since all the stored keys are different, the probability that a particular bit will be unchosen after all the keys are added is $\left(1 - \frac{1}{m}\right)^n$. Thus the probability that a particular bit has been set to 1 after the $n$ keys have been added is $1 - \left(1 - \frac{1}{m}\right)^n$. A membership test for a key that is not in the set will choose one bit in the bit-vector. As we have just seen, the probability that the chosen bit is already set to 1 is $1 - \left(1 - \frac{1}{m}\right)^n$. Thus the probability of a false positive for a key that is not in the set is $1 - \left(1 - \frac{1}{m}\right)^n$.

For example, if we want the probability of a false positive to be $< 0.01$ (ie 1%) we need $\left(1 - \frac{1}{m}\right)^n$ to be $> 0.99$ .... which means that if $n = 5$, $m$ needs to be about 500.

That doesn’t seem very efficient! We are using 500 bits to store 5 keys, and we still have a 1% chance of false positive errors.

But we can do much better ... paradoxically, we do this by using more bits of the bit-vector for each key we insert. A Bloom Filter is based on this idea: it uses more than one hash function for each key. For example, we might choose to use two hash functions. To “store” a key in the bit-vector, we compute both hash functions and set both resulting bits to 1.

Now for a key to result in a false positive, it has to hit a 1-bit on both hash functions. Working out the probabilities in the same manner as we did above (but adapting the formulas because we set two bits for each key and test two bits for the new key), we find that the probability of a false positive is $\left(1 - \left(1 - \frac{1}{m}\right)^{2n}\right)^2$.

And ... this is where it gets interesting ... to achieve 1% probability of a false positive when $n = 5$, we only need about 320 bits ... compared to the 500 bits we needed when we only used one hash function. (Note: all these values of $m$ are rounded to the nearest multiple of 10)
To me, this is totally counter-intuitive! By setting more bits per key, we need fewer bits over-all to meet the 1% error rate.

Let’s keep going. If we use three hash functions, we only need 70 bits to store 5 keys with 1% probability of false positives.

Time for a table:

<table>
<thead>
<tr>
<th>Number of hash functions</th>
<th>Bit-vector length required to achieve 1% probability of false positive after 5 keys have been added</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>500</td>
</tr>
<tr>
<td>2</td>
<td>320</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
</tr>
<tr>
<td>9</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>11</td>
<td>60</td>
</tr>
<tr>
<td>12</td>
<td>60</td>
</tr>
</tbody>
</table>

First thing to observe: with five hash functions, we only need 50 bits to hit the 1% target! That’s just 10 bits to “store” each key, even though the keys themselves may have been very long strings.

Second thing: the required bit-vector length initially goes down as we increase the number of hash functions, but then it starts to go up again ... why?

It starts to increase because eventually we are setting so many bits to 1 for each key that for small bit-vectors, most of the bits are set to 1 – which makes false positives quite probable. To counteract this and bring the false positive probability down to where we want it, we have to make the bit-vector longer.
The interplay between \( n \), \( m \) and the number of hash functions is complex. In general, if we have stored \( n \) keys in a bit-vector of length \( m \), and we are using \( h \) hash functions, the probability of a false positive is

\[
\left( 1 - \left( 1 - \frac{1}{m} \right)^{h \times n} \right)^h
\]

The typical application of this formula is to choose a value for \( n \) and a desired false positive probability, and then find a satisfactory combination of \( m \) and \( h \) to make it work. Usually the criterion for “satisfactory combination” is to keep \( m \) small.

In the sample code that I have posted, the keys being “stored” are all 8-character strings. Storing \( n \) such strings explicitly would take \( 64 \times n \) bits. Thus a plausible goal might be to achieve 1% probability of false positives, with \( m \) as small as possible. It turns out that with 15 hash functions, we can use a bit-vector of length about \( 13 \times n \) … which is approximately \( \frac{1}{5} \) of the \( 64 \times n \) bits needed to store the strings in their natural form.

An 80% reduction in memory use, at a cost of 1% probability of false positives … that’s not bad!

Note that there is nothing magical about the 1% false positive probability target – we can aim for 13% or 0.000007% or whatever.

Bloom filters have some interesting properties that are worth mentioning:

The membership test takes \( O(1) \) time, since we have predetermined how many hash functions we will use.

Since the keys are not actually stored in the structure, security is high. However if an attacker gains access to the bit-vector and the hash functions, they can attempt to engineer a key value that will give a positive result.

It is not possible to delete a value from a Bloom filter without potentially deleting other values as well. If we set all the bits that correspond to a particular key to 0, some of those bits probably also correspond to other keys … which will now give negative results when we test them for membership.

There is no upper limit to the number of keys that can be added to a Bloom filter of size \( m \) … but the more keys we add, the greater the probability of false positives. Eventually
we may have all bits are set to 1, at which point the false positive probability equals 100%

Here are two more applications of Bloom Filters. I suspect that you will be able to find/discover many others.

**Hyphenation:** When formatting English text, sometimes we have to split a long word when it occurs at the end of a line. Most words in English follow simple and standard hyphenation rules for splitting (split between two syllables, etc.) but there are quite a few exceptions (yay English!). An amusing example is “hyphenation” itself. A simple application of the split-between-syllables rule might break it into “hy-” “phenation” which is not generally considered correct.

We could create a Red-Black tree to store all the exceptions and their rules. Then whenever we need to split a word, we could search the tree to discover if our word is an exception – if it isn’t, we split it according to the standard rules, and if it is an exception then we use whatever special rules apply to this word. But the vast majority of words we will be checking will not be exceptions, so searching the tree for them is a waste of time.

If we create a Bloom Filter for the exceptions, then we can check a word to see if it is an exception in O(1) time. For the infrequent times when we get a positive result, we can search the Red-Black tree (or whatever structure we have chosen to actually give us the proper hyphenation of the exceptions). Some of the positive results will be false positives but we accept that as an acceptable trade-off for the O(1) time to get negative results for most words.

**User Name Selection:** When you register for HeresMyPrivateInfo or ExploitMePlease or whatever your favourite social media platform is, one of the first things you have to do is choose a username. You confidently enter the concatenation of your first and last names, only to be told that username is already in use and you have to choose a different one – sometimes it even suggests one for you.

But we know social media platforms have quintillions of users – how is the system able to instantly determine that your proposed username is already in use? How is it able to suggest usernames that are not already taken?

If I were designing such an abominable waste of human and natural resources, I would use a Bloom Filter. All existing usernames would be hashed into the bit-vector – so checking a proposed username against the set of all existing usernames would take O(1) time. A positive result (even if it is a false positive) would reject the proposed user name and request another. To recommend an acceptable username I would take the proposed username and append a random number to it, repeating this until I found one that tests negative on the Bloom Filter.
Regarding the demo programs:

The most challenging part of creating a Bloom Filter is coming up with a lot of hash functions that are mutually independent. In one of the examples above I mentioned using 15 hash functions to achieve the accuracy target.

For the demo, since all the keys are 8-character strings I decided to use tabulation hashing. In this method (which I described in the posted notes on hashing), each hash function is defined by an array filled with randomly generated bitstrings. We use the characters of the key to pick bitstrings out of the array, and then we return the “exclusive or” of the chosen bitstrings.

The advantage of this is that if we want to create \( h \) different hash functions, we just have to create \( h \) different arrays of bitstrings. We can evaluate all of the hash functions for a given key by calling a general method that takes a key and a bitstring array as parameters and returns the hash value..

If we accept that the random number generator in our chosen programming language is unbiased, then the \( h \) arrays of bitstrings will all be statistically independent of each other, which is exactly what we want.

Due to the nature of these hash functions, all computed table addresses will be in the range \([0 \ldots 2^r - 1]\) where \( r \) is the length of the bitstrings stored in the arrays that define the hash functions. Thus my bit-vectors are always \( 2^r \) bits long.

The first demo consists of creating a Bloom Filter and adding a set of keys, then testing the false positive rate with a collection of keys that are not in the original set.

In the demo I use the formulas shown above to predict the number of bits that will be set to 1 after all the keys were added, and to predict the false positive rate. Then I conduct the experiment and report the number of bits actually set, and the actual false positive rate. The experimental results show strong agreement with the predictions. From this I conclude that the hash functions are indeed independent, and that the prediction formulas are valid.

The second demo uses the prediction formula for the false positive rate to explore the relationship between the length of the bit-vector (as determined by the length of the bitstrings in the hash function tables), the number of hash functions, and the false positive rate. This information can be used to make an informed decision about how large a bit-vector to use, how many hash functions to use, etc.
Ad Libertatem Per Structura