CPSC 320 Sample Solution: Bloom Filters, or Trading Off Memory for. . . Wrong Answers

December 6, 2016

You work for the well-known web infrastructure company Wawaiki. Your company's business is caching the files associated with requested URLs so browsers can load pages faster. Recent analysis of your caching logs show that more than 3/4 of requested URLs are never requested again.[1]

You're trying to figure out a way to avoid caching the files associated with URLs until the **second** time you see the URL.

1 Data Structures Question? The Answer Is Probably Hash Tables

One approach would be to keep a hash set of URLs (a hash table where the keys are URLs and there are no values, since you just care about presence/absence).

Determine how much memory we will use to store 1 billion URLs in our hash set.

Assume:

- the hash table's load factor is 1
- the hash table uses chaining to resolve collisions
- a pointer on your system takes 8 bytes (64 bits)
- the average URL is a 78 bytes long (including a nil-terminator)[2]

SOLUTION: Because the load factor (number of keys—URLs, in this case—in table divided by number of entries in table) is 1 and there are 1 billion URLs in the table, there must also be 1 billion entries in the table. Each entry is a linked list "bucket", which presumably means each entry is a head pointer for a linked list that may point to a node in the list or may point to `NULL`. That, then, is 1 billion pointers, i.e., 8 billion bytes.

Each key will be stored in a single node in one of these linked lists. A node then needs a key (URL) and a pointer to the next node. It's tempting to say that these 1 billion nodes will therefore take 78 bytes (URL, on average) plus 8 bytes (pointer) each.

There **is** a strategy we could use to represent the nodes this way, but it would require some clever (nasty?) tricks. Instead, we typically cannot store the URL directly in the node because we don't know how long one URL is. (78 bytes is an average. The longest URL may be 1000s of bytes long!)

Therefore, the more likely strategy is to have each node contain a pointer to the URL (and thus allocate the right amount of space for **that** URL) and a pointer to the next node. That's 78 + 8 + 8 bytes per node for 1 billion nodes.

That totals 78 + 8 + 8 + 8 times 1 billion = 102 billion bytes.

[1] "One-hit wonders" and the solution we use here are real. See Algorithmic Nuggets in Content Delivery by Maggs and Sitaraman: https://people.cs.umass.edu/ramesh/Site/HOME_files/CCRpaper_1.pdf.

[2] See http://www.supermind.org/blog/740/average-length-of-a-url-part-2. Note that URLs are ASCII; so, 1 byte per character.

2 A Bit Smaller

Let's design a hash set except that each entry in the table is a single bit, where a 1 indicates that the key is present in the set and a 0 indicates that it is not.

1. Determine how much memory we will **now** use to store 1 billion URLs. Assume we make the table have 10 times as many entries as there are elements in the table.

   SOLUTION: There are 10 billion entries, and each entry is **just** a bit. (There's no possible way with a single bit to store either the key—i.e., the URL—or a pointer to chain in more keys!) That's 10 billion bits, or 1.25 billion bytes. About 100 times smaller than the previous solution!

2. Why might the load factor (the number of non-zero entries divided by the total number of entries) be less than 0.1?

   SOLUTION: Wow. My use of load factor was super-confusing, wasn't it? :)

   Anyway, if two keys hash to the same "bucket" in the table, it now means simply that that "bucket" will be set to 1 twice. The odds that this will happen at least once are astronomically huge. (Actually, they're bigger. They're computerologically huge.) In fact, it will likely happen quite a bit more than once. Thus, we'll set somewhat fewer than 10% of the table entries to 1.

3. If I ask whether a URL is in this strange hash set, the answer can be wrong. Under what circumstances might the answer be wrong? Under what circumstances will it **never** be wrong?

   SOLUTION: There's a couple ways to think of this.

   Version 1: If we ask whether a URL is in the hash set and get the answer "no", then it **cannot** be in the table. However, if we get the answer "yes", that answer may be coincidence (because the URL really isn't in the table, but we happened to hash to an entry that was previously used).

   Version 2: If we ask about a URL that we **have** seen before, we're guaranteed to get the answer "yes" back. However, if we ask about a URL that we have **not** seen before, we may or may not get a "yes".

4. With the load factor indicated above, if you ask the hash set a query that can be wrong, what is the probability that the answer **will** be wrong?

   Do your best to find a formula here, and then move on!

   SOLUTION: The queries that **can** be wrong are queries for a key that is not in the table. (If the key is in the table, the result will be "yes", guaranteed.) In that case, the query is wrong if it maps to the same entry as any previous key.

   So, what is the probability that, for every previous key, the current key maps to a different entry? We'll assume these are all independent questions so that we can multiply each probability together. In that case, we want the product of the probability of missing each individual key.

   We'll also assume our hash function is very good, which means that each key essentially maps to an entry uniformly at random. So, the probability that two keys map to the same entry is $\frac{1}{m}$, where m is the number of entries. (No matter where the first one goes, the second has a $\frac{1}{m}$ chance of landing in that spot.) The probability that that does **not** happen is $1 - \frac{1}{m}$.

   Overall, then, the probability that we avoid hitting any existing key is $(1 - \frac{1}{m})^n$, where n is the number of keys presently in the table.

   It turns out we can approximate this by the simpler formula $e^{-n/m}$.

5. What should we **do** if the answer is wrong? (Can we "fall back" to a better solution?)

   SOLUTION: There are many things we could do. Some solutions are independent of the domain we're applying Bloom Filters to, others are not. Let's consider two possibilities, one of each type.

Independent of domain, we could have a backup set that **is** sound and complete. In cases where we get a "yes" answer, we could double-check it against this set. There are some huge disadvantages to this. For example, we'll definitely use a lot of memory keeping this backup set. (If it's implemented as a hash table using chaining with a load factor of 1, we'll use the memory we computed above. Of course, we could implement it in many other ways, but almost any will need to store the keys themselves, which is a huge memory overhead.)

However, this won't happen on every query. If our false positive rate is fairly low, we know that in **this** domain about $\frac{3}{4}$ of the time we'll get a "no" answer and skip the extra lookup. So, maybe that's fine? Maybe these occasional lookups also won't swap our main Bloom Filter out of cache; so, we'll also get improved performance on average?

Another solution that's much more domain dependent is just to say that we'll treat false positives as true positives. In that case, we don't need a backup data structure to check. In this domain, that would mean that anytime we get a "yes" back, we cache the URL. Some of the time, we'll be caching that URL when we "shouldn't" because we've only seen it once. However, the goal was to greatly reduce the unnecessary caching we did, and as long as our false positive rate is low, we'll still accomplish that. So, maybe this is a good solution. (In other domains, it might be unacceptable to treat false positives the same as true positives.)

3 If Hash Is Good, More Hash Is Better

Imagine we have two totally different, excellent hash functions.

Originally, to insert a key into the table, we: hash the key, mod that by the table size, and set the entry at the resulting index to 1.

Now, we simply do that twice, once for the first hash function and once for the second.

1. Determine how much memory we will **now** use to store 1 billion URLs. Assume the load factor is still 0.1, and that "load factor" still means "number of keys inserted into the table divided by table size".

   SOLUTION: Since the number of keys is 1 billion and the load factor is 0.1, with **this** definition of load factor, we can see we'll use 10 billion entries, each of size 1 bit, which is 10 billion bits or 1.25 billion bytes.

2. If I ask whether a URL is in this strange hash set, the answer can be wrong. Under what circumstances might the answer be wrong? Under what circumstances will it **never** be wrong?

   SOLUTION: Just as before, if we ask for a key that **is** in the table, the answer will be "yes". However, just because the answer is "yes" does not mean the key is really in the table.

   Unlike before, both places we search for a key that gives us a false positive must individually say "yes" for the result to be "yes". (If either says "no", we know the key really wasn't in the table after all.) Maybe that means we'll have a lower false positive rate?

3. With the load factor indicated above, if you ask the hash set a query that can be wrong, what is the probability that the answer **will** be wrong? (Assume that each key (independently) has equal probability of being mapped to each entry in the table by each hash function (independently).)

   This is hard to answer precisely. To derive a relatively simple bound on the answer, try imagining that there were **no** collisions among any of the indices chosen by either hash function for any of the already-inserted elements.

   SOLUTION: Remember that before we ended up with $(1 - \frac{1}{m})^n$. That was because $\frac{1}{m}$ was the odds of colliding with something already in the table, and n was the number of "somethings" in the table.

Well, now there are kn "somethings" in the table, where k is the number of hash functions we use, i.e., $k = 2$ here. So, the odds of not colliding with anything on **one** hash are roughly $(1 - \frac{1}{m})^{kn}$.

That's not the end of the story, however. Remember that to get a false positive now we need to collide on **both** hashes of a new query. The odds that we **do** collide with one hash are one minus the odds that we don't: $[1 - (1 - \frac{1}{m})^{kn}]$. To get the false positive, we need to collide on every hash. If we consider each independent, that's k independent collisions: $[1 - (1 - \frac{1}{m})^{kn}]^k$. So, that's our probability of a false positive for a query of a key that's not in the table with a table size of m, n keys in the table, and using k hash functions per key.

4 Bloom Filters

A Bloom Filter is a generalized version of what we've just done. It is a bit vector V with m entries and k (ideally-independent) hash functions. To insert a new element i, we set $V[h_j(i)] = 1$ for each of the k hash functions h_j. To check whether an element i is in the set, we AND together each value $V[h_j(i)]$ for all k hash functions.

The optimal number of hash functions is $k = \frac{m}{n} \ln 2$. For that number of hash functions, and given a choice of false positive rate, the ratio $\frac{m}{n} = -\frac{\ln p}{(\ln 2)^2}$, where n is the number of keys in the table and p is the false positive rate.

1. Compute the approximate number of bits per entry and the number of hash functions needed for a false positive rate of 1%.

   SOLUTION: For $p = 0.01$ (i.e., 1%), we need $\frac{m}{n} = -\frac{\ln 0.01}{(\ln 2)^2} \approx 9.6$. So, we need about 9.6 times as many entries as there are keys, which is 9.6 bits per key or about 1.2 billion bytes, just under our previous table sizes. That's pretty cool. We get a very low false positive rate at about the same table size we'd already been considering!

   How many hash functions will we use? $k = \frac{m}{n} \ln 2 \approx 9.6 \ln 2 \approx 6.6$. Of course, we cannot use 6.6 hash functions. There are a few ways we could handle this, but a natural one is just to round up to 7 hash functions.

   (Universal hash functions give us one way to easily pick these 7 hash functions. In some applications and with some hash functions, it will also suffice to compute one hash function and separate the bits of the result into 7 parts, each of which we treat as its own hash function.)

2. What happens to the false positive rate as users request more and more URLs? How should we resolve this problem?

   SOLUTION: As users request more URLs, the table gets fuller, i.e., n goes up. That increases our false positive rate.

   We can resolve this by occasionally resizing the table, but that brings up a messy question. **HOW** do we resize the table? Normally, we make a new table twice as large and rehash everything from the old table into the new table, but we **do not know** what was in the old table! All we know is which entries are 1s! We could take each entry i that has a 1 and set the entries i and $i + m$ to 1 in the new table (since the mod operation on a hash could map to either of those two indices in a table twice as large), but that makes our resize useless! We double the number of entries **and** the number of filled entries at the same time.

   So, we have choices. We could start over with a new, empty table, maybe a bigger table. We could use a backup data structure (as suggested above) and rebuild a new, larger table from that accurate backup. In this domain, we could just insert the URLs we have cached. (That will overlook URLs we believe we've seen only once, but might be better than starting all over.)

   All of these choices impose tradeoffs!

5 Big Picture/Challenges

We've just looked at three big ideas in data structures and algorithms:

1. Randomized algorithms can give you enormous power and flexibility. Check out the awesome and heavily-used Miller-Rabin primality test to learn more about one randomized algorithm. (Testing primality deterministically takes polynomial time, but the randomized test will almost certainly serve your needs better if you want to find a big prime!)

2. Universal hash functions make it easy to use multiple different hash functions on the same piece of data. Once we can do that, lots of nifty ideas become possible. Check out Cuckoo Hashing for one cool example! (There is also such a think as a Cuckoo Filter.)

3. If we abandon a restriction to correctness (i.e., soundness, completeness, or both), we can often make progress on problems that are otherwise intractable. For example, the Daikon invariant detector exploits unsoundness to document invariants in software, a problem that is uncomputable in general.