COMP 480/COMP 580 - Probabilistic Algorithms and Data Structures
September 6, 2022
Lecturer: Anshumali Shrivastava
Scribe by: Ze Yuan Zhang, Phuong Tran, Angela Cao

Lecture 5: Compressed Cache and Bloom Filters

The Need for a Compressed Cache

Imagine we want to create a local cache of 1 million malicious URLs at an average length of 50 characters. We would need about 50MB to store the Strings as they are, but only have 2MB to work with. We cannot store the strings in any form, as they cannot be compressed to 25x smaller (lossless compression algorithms normally achieve an approximate 2x reduction in size). With the power of hash functions, however, we can achieve the desired reduction while sacrificing only a small amount of certainty.

Theoretical Compressed Cache: Perfect Hashing with Bitmap

Since we need such a drastic degree of compression, we can't simply store the strings as is. In this case, the cache only needs to return whether a given string is in it. So, storing each string as a single bit is perfect for this application.

Assuming we have access to a perfect hashing function and a bitmap, we can implement the cache as follows:

1. Start with the perfect hash function h and a bitmap of 0s
2. To insert query q into the cache, set h(q) = 1
3. If h(q) = 1, it means q is in the cache. If h(q) = 0, it means q is not in the cache.

This allows us to store each string as a bit in the cache. Assuming an average string length of 50 characters, **this allows for compression of 50 characters/string * 8 bits/character / 10 slots/string = 40x.** Since perfect hashing means no collisions, our cache will always be completely accurate.

Naive Compressed Cache: Universal Hashing with Bitmap

Unfortunately, no such perfect hash function exists. If we try the above approach with universal hashing, a practical alternative, we may experience false positive cache hits due to collisions.

The chance of getting a false positive cache hit given a bitmap of size R and n queries are:

$$\textbf{p(false positive)} = 1 - (1 - 1/R)^n$$

This is because:

1. The chance of collision is 1/R on average
2. The chance of no collision is 1 - 1/R on average
3. The chance of n independent queries not colliding is $(1 - 1/R)^n$

4. The chance of n independent queries having at least 1 collision is $1 - (1 - 1/R)^n$

Using the in-class example, where R = 10n and n = 10^6, the chances of getting at least 1 false-positive is 0.095 (9.5%). This can definitely be improved upon.

Bloom Filters: Naive Compressed Cache with Power of K-choices

We can significantly improve the performance of our compressed cache by leveraging the power of k-choices with our previous approach. This hugely popular and powerful combination is known as the Bloom filter.

A Bloom filter is a probabilistic data structure that essentially uses K independent hash functions to mark bits on the same hash table. To insert q, we set each h(q) to 1, for $h_1, h_2, \ldots, h_k$. To query q, we take the bitwise AND of $h_1, h_2, \ldots, h_k$.
This approach keeps the no false negative guarantee, while exponentially decreasing the chance of false positives.

Here given K worlds and R and n as defined previously,

$$\textbf{p(false positive)} < (1 - (1 - 1/R)^{Kn})^K \approx (1-e^{(Kn/R)})^K$$

(The approximation by exponential function comes from the fact that $\lim_{R\to\infty} (1-1/R)^R = e^{-1}$.) This is because:

1. Given n insertions, p(a bit is not set) = $1 - (1/R)^{Kn}$
2. p(1 hash function output for q is falsely set) < $1 - (1 - 1/R)^{Kn}$
3. p(K hash function output for q is falsely set) < $(1 - (1 - 1/R)^{Kn})^K$

p(false positive) is minimized at K = ln(2) * R/n ≈ 6.9 for R = 10n. **At this minimum value, we've decreased p(false positive) to at most 0.008** - a whopping 10x reduction compared to the previous approach!

Applications

Bloom filters are used to solve the Generic Set Compression problem because they allow us to compress each object to around 10 bits, while still answering membership queries efficiently and accurately.

Bloom filters are commonly applied in web content caching, detecting malicious URLs, preventing the caching of URLs that users only visit once (one-hit wonders), preventing recommendations for already-visited pages, etc.

- Web content caching/one-hit wonders: use a Bloom filter to hold URLs the first time they are accessed; only cache if the URL is already in said Bloomfilter (small amount of false positives is acceptable since the cache will only cache a little more than needed)
- Detecting malicious URLs: all malicious URLs need to be detected (no false negatives), but it is fine if a non-malicious URL is occasionally falsely flagged (small chance of false positives) as malicious since a server check can be done in that case

Bloom Filter Deletions

We cannot delete entries from Bloom filters without risking false negatives due to collision.

- Clearing the K-bits for the deleted entries will make it so that any entry with any bit that collides with any of the K-bits will be marked absent from the cache.

By using 2 Bloom filters (one tracking deleted entries, and one tracking entries), we can reduce but not eliminate false negatives:

- We can get a correct positive in the entries Bloom filter, then a false positive in the deletions Bloom filters, leading to a false negative.

However, we also reduce false positives:

- We can get a false positive in the entries Bloom filter and another false positive in the deletions Bloom filters, leading to a correct result.

By using 1 Bloom filter where each table entry is a counter instead of a boolean indicator that is incremented/decremented upon insertion/deletion, we can also accomplish Bloom filter deletions. This option is more expensive storage-wise and also has risks of false positives and false negatives.

Bloom Filter Unions

We can take the union of two Bloom filters if they use the same hash functions by taking their bitwise-OR. This allows Bloom filters to be distributed across different storage systems and synchronized cheaply. We cannot easily intersect two Bloom filters.

Bloom Filter Size Manipulation

We can halve the size of a Bloom filter by bitwise-OR-ing the first half of the bitset by the second half. This has the same effect as applying (modulo the new size of the bitset) to the hash functions.

Doubling the size of a Bloom filter is impossible because we do not save any information about the string inputs to the hash functions.

Bloom Filter Weaknesses

A few weaknesses of Bloom filters we should mention are:

- They need fully independent hash functions, which is hard to achieve
- They take up 1.44x as much space as the theoretically best solution
- Dynamically growing Bloom filters is hard and depends on the false positive rate and number of insertions
- Deletion support is difficult to achieve

Recap

In the malicious URL cache example, we saw the need for compressing the size of the cache by 25x, which is drastic enough to require a probabilistic approach rather than optimizing a deterministic approach. We saw that if we had a perfect hash function, we could simply use a bitmap to compress the

cache. We saw that we can use universal hashing as an approximate to perfect hashing (approach 2). Finally, we saw that we can use Bloom filters to improve upon the universal hashing and bitmap approach thanks to the power of k-choices (approach 3).

The downside of both approaches 2 and 3 is that it's possible to get a false positive result from the cache due to collisions in the universal hash functions.

$p(\text{false positive} \mid \text{approach 2}) < 1 - (1 - 1/R)^n$,

where R is the size of the bitmap, and n is the number of queries.

$p(\text{false positive} \mid \text{approach 3}) < (1 - (1 - 1/R)^{kn})^n \approx (1-e^{(Kn/R)})^K$,

where R is the size of the bitmap, n is the number of queries, and k is the number of hash functions used.