COMP 480/580 — Probabilistic Algorithms and Data Structure Sep 16, 2025

Lecture 6

Lecturer: Anshumali Shrivastava

Scribe By: Peyton Elebash (pe12), Lazeen Rafik Manasia (lm152), Michael Mitchell (mkm10)

1 Motivation & Setup

Problem (Malicious URL check): We want a fast, inexpensive way to test whether a URL is malicious or not.

Exact storage is too big. If we have a list of all malicious URLs and plan to check that list against the user's URL every time, things become expensive.
Example sanity check: 100 strings $\times$ 50 chars $\times$ 8 bits $\approx$ **40,000** bits. Given a universal hash of $h : strings \Rightarrow [0 - 999]$, we would need to compress 40,000 bits to 1000, which is infeasible.

Relaxed objective (probabilistic membership): Let's relax this problem so it becomes feasible!

- If an item *is not in our malicious list*, return **NO** with certainty.
- If we get a small probability that an item *is in our malicious list*, do an **additional, slower check**.

2 Universal Hashing Refresher

Perfect hashing (zero collisions) is generally infeasible.
With a perfect hash function, your range must be bigger than the number of queries.
Instead, use a *universal* family with small collision probability.
In this universal family of imperfect hash functions, you can work in any range.

The trade-off is False Positives!

False positives occur when we hash a query and get that we have seen it (1) when we have not (should be 0).

Given N strings, probability for a false positive is $< 1 - (1 - \frac{1}{R})^N$

- $1 - \frac{1}{R}$ is the chance for collision for one of the strings.
- $(1 - \frac{1}{R})^N$ is the chance for collision across all N strings.

3 From a Bit-Map to Bloom Filters

Simple Approach: Use one hash function and a bit-map. To add an item, hash it and set the bit at that index to 1. If we have encountered the query before, when we hash it, the bit at the hashed index should be 1. If we have not, it should be 0.
The Flaw: This works, but collisions (imperfect hashing) lead to false positives. A new, unseen item might happen to hash to an index that was already set to 1 by a different item.

- **Pros:** Simple.
- **Cons:** Collisions cause *false positives.*
- **No false negatives:** Once a bit is set for inserted x, the query for x will always see a 1; A collision can't cayse a 1 to become 0.

The probability for a false positive is $< 1 - (1 - \frac{1}{R})^N$, *Can we do better?*
The simple solution would be to increase the size of our bit-map (size is R), but then our memory usage goes up.
The trade-off is once again more memory for less collisions

4 The power of K choices

Some intuition: Say we have a dice. We get penalized if we roll a 1. Therefore, the chance that we are penalized is $\frac{1}{6}$. However, what if we could create two universes, where we are only penalized if in BOTH universes, we roll a 1. Now our chance of being penalized becomes $\frac{1}{6^2}$ or $\frac{1}{36}$. We made our chance of penalization *exponentially* smaller!
K choices is the idea of creating "K" universes where we only count an event as happening if it happens across all K universes, decreasing the probability of an event happening exponentially.

5 Bloom Filters

Bloom Filters use K-choices: Instead of one hash function, we use K independent hash functions.

- To add an item, we hash it K times and set the bits at all K resulting indices to 1
- To check an item, we hash it K times and check all K indices. If all of them are 1, we say the item has been "seen". If even one is 0, it's definitely not in the set.

Definition 1 (Bloom Filter)**.** *A Bloom filter consists of a bit array of size R and K independent hash functions, $h_1, h_2, \ldots, h_K$.*

- ***To insert an element** s: For each hash function h_i where $i \in [1, K]$, compute the index $h_i(s)$ and set the bit at that array position to 1.*
- ***To query an element** q: For each hash function h_i, check the bit at index $h_i(q)$. If **all** K bits are 1, the element is considered to be in the set. If **any** bit is 0, the element is definitively not in the set.*

The Math: By using K hash functions, the chances of a false positive decrease *exponentially.* We are "gaining linearly but paying exponentially."

6 Example with Small Filter

Suppose $R = 5$ and $k = 2$. Define hash functions:

$$h_1(x) = x \bmod 5, \quad h_2(x) = (2x + 3) \bmod 5.$$

- Insert 9: $h_1(9) = 4$, $h_2(9) = 1$. Bits at positions $1, 4$ set.

$$[0, 1, 0, 0, 1]$$

- Query 15: $h_1(15) = 0$, $h_2(15) = 3$. Bits $(0, 3)$. At least one zero $\Rightarrow$ "Not seen".
- Query 16: $h_1(16) = 1$, $h_2(16) = 0$. Bits $(1, 0)$ are both set $\Rightarrow$ "Seen" (false positive).

7 Analysis and Efficiency

Why it's so good: A false positive occurs when all K bits for a query element are 1. The probability that a bit in the array is *not* set to 1 after inserting N strings is $(1-1/R)^{KN}$. Thus, the probability that a given bit *is* 1 is $1 - (1 - 1/R)^{KN}$. The false positive rate (FPR) is the probability that all K bits for a new element are 1:

$$FPR = \left(1 - \left(1 - \frac{1}{R}\right)^{KN}\right)^K \approx \left(1 - e^{-KN/R}\right)^K$$

This approximation holds for a large array size R. The false positive probability is minimized when optimal number of hash functions, K, is chosen:

$$k_{opt} = \frac{R}{N} \ln 2.$$

At this optimum k, the false positive probability is approximately:

$$P_{fp} \approx \left(\frac{1}{2}\right)^k \approx (0.6185)^{R/N}.$$

Example: If we use 10 bits per object ($R = 10N$), then the optimal $k \approx 6.9$ (we choose $k = 7$). This yields an FPR below 1%.

Memory Savings: This approach offers a massive **40X** reduction in memory compared to a standard hash table.

8 Real World Applications

Ubiquity: There is hardly any large-scale tech that doesn't use Bloom filters.
Caching Rule: A core principle of caching is "if you see something more than once, cache it."

- **Caching**: Content Delivery Networks (CDNs) like **Akamai** use Bloom filters to avoid caching "one-hit-wonders." An object is only cached upon its second request, which is detected by the Bloom filter.
- **Databases**: Systems like **Google Bigtable**, **Apache Cassandra**, and **Postgresql** use Bloom filters to quickly rule out the existence of non-existent rows or columns, avoiding costly disk lookups.
- **Networking and Security**: Used in web browsers for malicious URL detection.

9 Deletions

Standard Bloom: Cannot safely delete by resetting bits from 1 to 0, because doing so might cause a false negative for another item that happened to share one of those bits.

Workarounds:

(a) **Two filters:** One records inserts; a second records deletions.

(b) **Counting Bloom filter:** Replace each bit with a small counter; increment on insert, decrement on delete.

10 Set Operations & Distributed Use

If two Bloom filters use the same hash family, their **union** is just the bitwise OR of their bit arrays. This is powerful for distributed systems (e.g., multiple regions/servers) because they can merge knowledge without exchanging raw logs.