Algorithms for Data Science: Lecture 5

Barna Saha

1 Hashing

The balls-and-bins exercise that we did in Homework 1 is also useful for modeling Hashing. A hash function h from a universe $U = [0, 1, .., m-1]$ into a range $[0, ..., n-1]$ can be thought of as a way of placing items from the universe into n bins. The collection of bins is called a *hash table*. We can model the distribution of items in bins with the same distribution as m balls placed randomly in n bins. We are making a rather strong assumption here by presuming that our hash function maps words into bins in a fashion that appears random, so that the location of each item is independent and identically distributed. There is a great deal of theory on developing hash functions that appear random. We will not delve into that theory here, and will rather assume that our hash functions are indeed random. In other words, we assume the following holds

- for each $x \in U$, $\Pr(h(x) = j) = \frac{1}{n}$ for all $j \in [0, 1, .., n-1]$
- for any k items $x_1, x_2, ..., x_k \in U$ with $x_i \neq x_j$, for $i, j \in [1, k], i \neq j$,

$$\Pr(h(x_1) = y_1 \wedge h(x_2) = y_2 \wedge h(x_3) = y_3 \wedge .... \wedge h(x_k) = y_k) = \frac{1}{n^k}$$

Notice that this does not mean every evaluation of $h(x)$ yields a different random answer! The value of $f(x)$ is fixed for all time; it is just equaly likely to take on any value in the range.

1.1 Families of Universal Hash Functions

The assumption of a completely random hash function simplifies the analysis for a theoretical study of hashing. In practice, however, completely random hash functions are too expensive to compute and store, so the model does not fully reflect reality.

Two approaches are commonly used to implement practical hash functions. In many cases, heuristics or ad hoc functions designed to appear random are used. Although these functions may work suitably for some applications, they generally do not have any associated provable guarantees, making their use potentially risky. Another approach is to use hash functions for which there are some provable guarantees. We trade away the strong statements one can make about completely random hash functions for weaker statements with hash functions that are efficient to store and compute.

Here we describe one such computationally simplest classes of hash functions that are widely used in practice: universal families of hash functions.

Definition. Let U be a universe with $U = \{0, 1, 2, ..., m-1\}$ and let $V = \{0, 1, 2, ..., n-1\}$. A family of hash functions $\mathcal{H}$ is said to be strongly k-universal if, for any elements $x_1, x_2, ..., x_k \in U$, any values $y_1, y_2, ..., y_k \in V$, and a hash function h chosen uniformly at random from $\mathcal{H}$, we have

$$\Pr(h(x_1) = y_1 \wedge h(x_2) = y_2 \wedge h(x_3) = y_3 \wedge .... \wedge h(x_k) = y_k) = \frac{1}{n^k}$$

We will often be interested in 2-strongly universal family of hash functions. When we choose a hash function h from that family uniformly at random, the values $h(x_1)$ and $h(x_2)$ behave as pair-wise independent, since the probability that they take on any specific pair of values is $\frac{1}{n^2}$. Because, of this, hash functions chosen from a family of strongly 2-universal hash functions are called pair-wise independent. Similarly, hash functions chosen from a family of strongly k-universal hash functions are called k-wise independent.

Example 1 (A 2-strongly Universal Family of Hash Functions)**.** *Consider the family of hash functions obtained by choosing a prime number $p \geq m$, letting*

$$h_{a,b}(x) = ((ax + b) \ mod \ p) \ mod \ n$$

and then taking the family

$$\mathcal{H} = \{h_{a,b} \mid 1 \leq a \leq p-1, 0 \leq b \leq p\}$$

Note that it is important that a cannot be 0

Example 2 (A k-strongly Universal Family of Hash Functions)**.** *Consider the family of hash functions obtained by choosing a prime number $p \geq m$, letting*

$$h_{a_1,a_2,..,a_k}(x) = ((a_1x^{k-1} + a_2x^{k-2} + ... + a_1) \ mod \ p) \ mod \ n$$

and then taking the family

$$\mathcal{H} = \{h_{a,b} \mid 1 \leq a_1 \leq p-1, 0 \leq a_i \leq p, i \in [2,k],\}$$

Note that it is important that a_1 cannot be 0

2 Bloom Filter

Application: Password Checker. A password checker prevents people from using common, easily cracked passwords by keeping a dictionary of unacceptable passwords. When a user tries to set up a password, the application would like to check if the requested password is part of the unacceptable set. One possible approach for a password checker would be to store the unacceptable passwords alphabetically and do a binary search on the dictionary to check if a proposed password is unacceptable. A binary search would require $\Theta(\log m)$ time for m words.

Application: Spam Detector. A spam detector prevents sending spam emails to the inbox by keeping a dictionary of acceptable email ids. When an email arrives, the spam detector checks if the email id belongs to the acceptable email id dictionary, and forwards the email to the inbox if the search result is positive. One possible approach for a spam detector would be to store the acceptable email ids alphabetically and do a binary search on the dictionary to check if the email id is acceptable. A binary search would require $\Theta(\log m)$ time for m words.

How can we reduce the search time?

We can reduce the search time by maintaining the words in a hash table. However, if we use $n = O(m)$, then by the balls-in-bin exercise, the maximum load on a bin, and hence the worst case search time, can be about $\Theta(\frac{\log m}{\log \log m})$. Though the expected search time is $O(1)$.

Can we have $O(1)$-worst case search time?

We will use a data structure called Bloom Filter to achieve the above.

A Bloom filter consists of an array of n bits, $A[0]$ to $A[n-1]$, initially all set to 0. A Bloom filter uses k independent random hash functions $h_1, h_2, .., h_k$ with range $[0, 1, .., n-1]$. We make the usual assumption for analysis that these hash functions map each element in the universe to a random number uniformly over the range $\{0, 1, , ..., n-1\}$

Suppose, we use a Bloom filter to represent a set $S = \{s_1, s_2, .., s_m\}$ of m elements from a large universe U. For example, S can be the set of unacceptable passwords. It could also be the set of valid email ids. For each element $s \in S$, the bits $A[h_i(s)]$ are set to 1 for $1 \leq i \leq k$. A bit location can be set to 1 multiple times, but only the first change has an effect. To check if an element x is in S, we check whether all locations $A[h_i(x)]$ for $1 \leq i \leq k$ would be set to 1 by construction. If all $A[h_i(x)]$ are set to 1, we assume that x is in S, and no otherwise.

We would be wrong if $x \notin S$ but all of the positions $A[h_i(x)]$ were set to 1 by elements of S. However, if $x \in S$, then we will always correctly detect x is in S. Hence, the data structure can have report a *false positive*, but never a *false negative*.

Example 3. *In the context of password checker, it will always detect if a password is unacceptable, but it is over conservative, and may declare some valid password as unacceptable.*

Example 4. *In the context of spam detector, it will always detect if an email id is non spam, but it has some slack, and may declare some spam emails as non-spam.*

What is the false positive probability?

Suppose $x \notin S$.

Let us consider the ith bin.

$$\Pr(i\text{th bin has a 0 bit}) = \left(1 - \frac{1}{n}\right)^{km} = p$$

The above holds since using k hash functions, we are hashing a total of km items (balls) in n bins. Thus a given item does not fall in bin i with probability $(1 - \frac{1}{n})$. Since, the hash functions are perfectly random and chosen independently, bin i remains empty even after throwing km items is $\left(1 - \frac{1}{n}\right)^{km}$.

To simplify the analysis, let us temporarily assume that fraction p of entries are 0 in the hash table after all the elements is S have been hashed by the k hash functions.

The probability of false positive is then

$$\Pr(\text{fasle positive}) = (1-p)^k = \left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^k \approx \left(1 - e^{-\frac{km}{n}}\right)^k$$

Bloom filter is highly effective even if $n = cm$ for a small constant c, such as $c = 8$. In this case, when $k = 5$ or $k = 6$, the false positive probability is just over 0.02.

The optimum number of hash functions for minimum false positive rate.

Suppose that we are given m and n, and would like to optimize the number of hash functions to minimize the false positive rate. There are two competing forces in action: having more hash functions gives use more chances to find a 0 bit for an element; on the other hand having fewer hash functions implies more bins with 0 bits. By taking derivative of the false positive probability and equating to 0, we get when $k = \frac{n}{m}(\ln 2)$, the false positive probability is minimized.

More Applications. Bloom filters have found innumerable applications in networking, in web technology,

Akamai Content Distribution Network.

"*Akamai's web servers use Bloom filters to prevent* **one-hit-wonders** *from being stored in its disk caches. One-hit-wonders are web objects requested by users just once, something that Akamai found applied to nearly three-quarters of their caching infrastructure. Using a Bloom filter to detect the second request for a web object and caching that object only on its second request prevents one-hit wonders from entering the disk cache, significantly reducing disk workload and increasing disk cache hit rates.*"

Accessing objects from cache is must faster. Bloom filter allows the detection of objects that are requested for the second time, rather than wasting cache space for one-hit wonders.[1]

Google " Google BigTable, Apache HBase and Apache Cassandra use Bloom filters to reduce the disk lookups for non-existent rows or columns in SSTables. Avoiding costly disk lookups considerably increases the performance of a database query operation." [2]

"The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result). " [3]

Learn more about Bloom Filter

Video link: `https://www.youtube.com/watch?v=947gWqwkhu0`

[1] Reference: "Bruce M. Maggs and Ramesh K. Sitaraman, Algorithmic nuggets in content delivery, ACM SIGCOMM Computer Communication Review (CCR), July 2015." (PDF).

"BloomFilterDisk" by Ramesh K. Sitaraman - https://people.cs.umass.edu/ ramesh/Site/PUBLICATIONS.html. Licensed under CC BY-SA 4.0 via Commons - https://commons.wikimedia.org/wiki/File:BloomFilterDisk.png

[2] Reference: Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson; Wallach, Deborah; Burrows, Mike; Chandra, Tushar; Fikes, Andrew; Gruber, Robert (2006), "Bigtable: A Distributed Storage System for Structured Data", OSDI.

[3]Reference: Wikipedia.