CS124

Lecture on Hashing: Intro to Hashing

03-21-2022

Lecturer: Adam *Scribe: Eli*

Hashing is about finding a function H that maps an element $x \in$ our universe $\mathcal{U}$ to some hash value in $0...m-1 = [m]$. We are interested in the collisions when two elements in our universe map to the same hashed values.

- In the first example, $\mathcal{U}$ is a set of people, and H maps a person to a birthday.
- In the second example, $\mathcal{U}$ is a set of numbered balls, and H maps a ball to a specific bin.
- In the later examples, $\mathcal{U}$ is a set of possible passwords, and H maps a password to an index into an array A.

For all these examples we assume uniform iids whenever we pick from a distribution.

1 Birthday Paradox

How many people need to be in a room before it is likely that two people share a birthday?

Answer: Let E_{shared} be the event that there is at least one collision: two people share the same birthday. Then we have:

$$P[E_{shared}] = 1 - P[E_{notshared}]$$

We can find $P[E_{notshared}]$. The first person will not have a collision. The second person will have at least $365-1$ remaining days that will not result in a collision. The third will have $365-2$. The nth will have $365-(n-1)$. To get the probabilies, we divide these values by 365, the total number of days.

$$P[E_{shared}] = 1 - P[E_{notshared}]$$

$$P[E_{shared}] = 1 - (1 \cdot (1-1/365) \cdot (1-2/365) \cdot ... \cdot (1-(n-1)/365))$$

So, if we want $P[E_{shared}] = 0.5$, then the number of people is ≈ 23.

2 Balls to Bins

We throw n balls into m bins.

2.1 P[Bin 0 is empty]

Answer:

$$P[\text{Bin 0 is empty}] = P[\text{Ball 0 misses}] \cdot P[\text{Ball 1 misses}] \cdot ... \cdot P[\text{Ball n-1 misses}]$$

$$= \left(1 - \frac{1}{m}\right)^n$$

$$= \left(\left(1 - \frac{1}{m}\right)^m\right)^{n/m}$$

$$\approx e^{-n/m}$$

We will come back to this later. The remaining subsections **2.2** and **2.3** are not as important for later examples.

2.2 Expected Number of Empty Bins

Intuitively, this is equal to the number of bins (m) times the expectation of an indicator of a single bin being empty. An expectation of an indicator is equivalently to a probability[1], as calculated in **2.1**.

$$= m \cdot \left(1 - \frac{1}{m}\right)^n$$

$$\approx me^{-n/m}$$

2.3 Expected Number of Bins with Exactly 1 Ball

Same idea:

$$= m \cdot P[\text{Bin 0 has 1 ball}]$$

But any of the n balls is equally likely to be the ball in Bin 0.

$$= m \cdot n \cdot P[\text{Bin 0 has Ball 0}]$$

Ball 0 hits, the other $n - 1$ miss:

$$= m \cdot n \cdot \frac{1}{m} \cdot (1 - 1/m)^{n-1}$$

$$= n \cdot (1 - 1/m)^{n-1}$$

Key idea: All bins are equal, all balls are equal, so we can calculate for a specific bin 0 and ball 0 and multiply by their respective numbers m and n.

[1] For example, the expectation of an indicator that a coin is heads is the probability that the coin is heads.

3 Hash Functions

Properties:

- Deterministic (same element universe will hash to same value every time)
- Produces seemingly random hash values for similar inputs (like uniform iids)

3.1 Passwords

- Suppose there is a locked door that requires a password.[2]
- Authorized people each have their own password to open the door.
- The door could store a complete list of valid passwords, and only let the correct passwords in, but this is insecure in case of a data breach, and may use a lot of memory.
- The door can keep track of all the valid passwords by hashing each valid password, and remembering all the associated valid hashed values instead.
- The door can remember valid hashed values in an array A, where $A[i] = 1$ iff hashed value i is valid. So if x is a valid password, then $H(x)$ is a valid index, and so we set $A[H(x)] = 1$.
- The problem is that an invalid password could hash to the same value of a valid one: $H(x_{invalid}) = H(x_{valid}) = i_{valid}$, with $A[i_{valid}] = 1$. This means $x_{invalid}$ is a false positive.

3.2 Probability of False Positive

- First we throw n balls into m bins (the door remembering our n passwords by setting some of the m indexes in A to 1). Each index of A is a bin here.
- Then, we have a new password that is false. We want the probability that this the ball goes into a populated bin. If the bin is not empty, then that bin (or index in A) was set to 1 by a another ball (or valid password). So, the H will be fooled and think the invalid password is valid, leading to a false positive.
- Suppose the invalid password gets hashed to bin b. The false positive rate is that b is not empty:

$$1 - P[\text{Bin b is empty}]$$

We already calculated this in **2.1**:

$$\approx 1 - e^{-n/m}$$

[2] I am changing the example to be a physical door to support intuition. Passwords are associated with usernames, but we don't care about who is logging in. We only care about 'universal access' if that makes sense. Of course, the example from lecture is exactly the same.

4 Bloom Filters

The lead engineer comes back and says, 'there are too many false positives! Make the door back to how it was!' But you've already gotten this far, and you suppose you can solve this with more hashing, and not less. So, instead, you decide to *ahem dig yourself deeper ahem* and have the door keep not just 1 (as in **3.1**), but k hashing functions!

4.1 Differences

- When we hash a password, we hash it k times, fill all (at most k) indexes in A to be 1.
- When we check a password, we check it k times, only accepting it if all of indexes are valid. That is, if $A[H_i(x)] = 1$ for all hash functions iterated by i.
- A is shared by all hash functions (not one array per hash function).

4.2 Key Intuition for Choosing K

- If k is too large, then we risk setting too many values to 1, increasing the false positive rate (consider the extreme scenario where the array A is completely 1, then any string will be 'valid'). Intuitively, this increase in false positive rate is due to over-hashing.
- If k is too small, then our 'certainty' is low. Each additional hash can be thought of as a 'double check' to make sure that the password was indeed valid (it satisfied all hash functions), and not just some fluke. Intuitively, this increase in false positive rate is due to under-checking.
- Unfortunately, with large k, we over-check but we also over-hash. With small k we under-hash, but we also under-check. So, we need find a good k somewhere in between.

4.3 Choosing K to minimize P[False Positive]

Suppose our invalid password maps to b.

$$P[\text{False Positive}]$$

A positive means that all k hash functions were fooled:

$$= P[A[H_1(b)] == 1] \cdot P[A[H_2(b)] == 1] \cdot ... \cdot P[A[H_k(b)] == 1]$$

$$= P[A[H_1(b)] == 1]^k$$

To calculate this, we note that there are m bins, and nk balls. This is because for each of the n valid passwords, we hash it k times, so we are filling our bins with nk balls, not n balls as before. Then, the answer is the same as **3.2**

$$\approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

Intuitively, the outer k is the gain associated from double checking k times, and inner k is the loss from filling out kn values of 1 into our array, increasing the probability of a false positive rate by saturating the array with 1s.

To optimize k, calculus can happen, and we get $k \approx \frac{m}{n} \ln 2$ and the false positive rate is $\approx .62^{m/n}$.