Algorithms for Data Science

CSOR W4246

Eleni Drinea
Computer Science Department

Columbia University

Hashing, bloom filters

Outline

Today

1. Hashing
2. Analyzing hash tables using balls and bins
3. Saving space: hashing-based fingerprints
4. Bloom filters

The problem

A data structure maintaining a dynamic subset S of a huge universe U.

- Typically, $|S| \ll |U|$

The data structure should support

- efficient **insertion**
- efficient **deletion**
- efficient **search**

We will call such a data structure a **dictionary**.

Dictionary data structure

A dictionary maintains a subset S of a universe U so that inserting, deleting and searching is efficient.

Operations supported by a dictionary

1. `Create`(): initialize a dictionary with $S = \emptyset$
2. `Insert`(x): add x to S, if $x \notin S$
   - additional information about x might be stored in the dictionary as part of a record for x
3. `Delete`(x): delete x from S, if $x \in S$
4. `Lookup`(x): determine if $x \in S$

A concrete example

We want to maintain a dynamic list of 250 IP addresses

- e.g., these correspond to addresses of currently active customers of a Web service
- each IP address consists of 32 bits, e.g. 128.32.168.80

The challenge: U is enormous, that is, $|U| \gg |S|$

1. Maintain **array** S of size $|U|$ such that $S[i] = 1$ if and only if $i \in S$
   - `Insert`, `Delete`, `Lookup` require $O(1)$ time

   Can't store an array of size anywhere close to $|U|$!

   - S should have $|U| = 2^{32} \approx 4$ billion entries
   - S would be mostly empty (huge waste of space)

2. Store S in a **linked list**
   - Space: proportional to $|S| = 250$
   - Time for `Lookup`: proportional to $|S|$; too slow

Can we support fast `Insert`, `Delete`, `Lookup` *(as in array implementation) but only use space proportional to $|S|$ (linked list implementation)?*

Work with array of size $|S|$ rather than one of size $|U|$

Idea: assign a short *nickname* to each element in U

- Each of the 2^{32} IP addresses is assigned a number between 1 and $|S| = 250$
  - range will be slightly adjusted
- Total amount of storage: approximately $|S|$, independent of $|U|$
- If not too many IP addresses per nickname, then `Lookup` is efficient (*details coming up*)

How can we assign a short name?

By hashing: use a hash function $h : U \to \{0, \ldots, n-1\}$

- Typically, $n \ll |U|$ and is close to $|S|$

For example,

- $h : \{0, \ldots, 2^{32} - 1\} \to \{0, \ldots, 249\}$
- IP address x gets name $h(x)$
- Hash table H of size 250: store address x at entry $h(x)$

So `Insert`(x) takes constant time. *What if we try to insert $y \neq x$, with $h(x) = h(y)$?*

Collisions

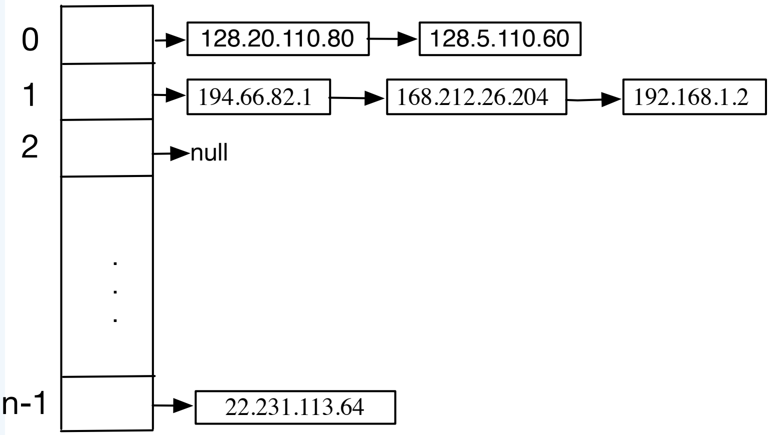
Collision: elements $x \neq y$ such that $h(x) = h(y)$

Easiest way to deal with collisions: chain hashing

- Entry i in the hash table is a **linked list** of elements x such that $h(x) = i$
- Alternatively, can think of every entry in the hash table as a **bin** containing the elements that hash to the same location

Chain hashing

Maintain a linked list at $H[i]$ for all x such that $h(x) = i$.



Chain hashing: running time for Lookup(x)

Time for `Lookup`(x):

1. time to compute $h(x)$; typically, constant
2. time to scan the linked list at position $h(x)$ in hash table
   - proportional to the *length* of the linked list at $h(x)$, which is proportional to the # elements that collide with x

Goal: find a hash function that "spreads out" the elements well

Simple hash functions might not work

Consider the following two simple hash functions that hash an IP address x from $\{0, \ldots, 2^{32} - 1\}$ to $\{0, \ldots, 255\}$:

- assign the last 8 bits of x as its name
- assign the first 8 bits of x as its name

Remark 1.

Nothing is inherently wrong with these hash functions: the problem is that our 250 IP addresses might not be drawn uniformly at random from among all 2^{32} *possibilities.*

No single hash function can work well on *all* data sets

- **Fix** the hash function h.
- h distributes $|U|$ elements into n names.

$\Rightarrow$ exists data set of at least $\frac{|U|}{n}$ elements that all map to the same name

$\Rightarrow$ if our customers come from this data set, lots of collisions

Fact: for any **fixed** (**deterministic**) $h : U \to \{0, 1, \ldots, n-1\}$ where $|U| \geq n^2$, there exists some set S of n elements that all map to the same position.

Randomization can help

- **Extreme example:** for every $0 \leq j \leq n-1$, assign name j to element x with probability $\frac{1}{n}$.
  - Fix $x, y \in U$. Then $\Pr[h(x) = h(y)] = \frac{1}{n}$.
  - This doesn't quite work. (Think `Lookup`(x): *where is x?*)

- However, intuitively, hash functions that spread things around in a *random* way can effectively reduce collisions.

$\Rightarrow$ Trade-off in hash function design: h must be "random" to scatter things around for all inputs but still be a function

Goal: design h that allows for efficient dictionary operations with high probability

A careful use of randomization

- Randomize over the **choice** of the hash function from a suitable class of functions into $[0, n-1]$ (*details coming up*)
- h must have a compact representation

Universal hash function

Idea: choose h at random from a carefully selected class of functions H with the following properties:

1. h behaves almost like a completely random hash function.
   - For $x, y \in U$. The probability that a randomly chosen $h \in H$ satisfies $h(x) = h(y)$ is at most $1/n$.
2. Can select a random h efficiently.
3. Given h, can compute $h(x)$ efficiently.

Such hash functions are called universal; their design relies on number theoretic facts.

Example of universal hash function

- Pick a prime p close to $|S| = 250$; set $n = p$
  - E.g., pick $p = 257$; set the size n of the hash table to 257
- Look at IP address x as (x_1, x_2, x_3, x_4), where x_1, x_2, x_3, x_4 are integers $\mod n$.
- Define $h : U \to \{0, 1, \ldots, n-1\}$ as follows:
  - Choose a_1, a_2, a_3, a_4 randomly from $\{0, 1, \ldots, n-1\}$
    - E.g., $a_1 = 80, a_2 = 35, a_3 = 168, a_4 = 220$
  - Map IP address x to $h(x) = \left(\sum_{i=1}^{4} a_i x_i\right) \mod n$
    - E.g., $x = 128.32.168.80$,
      $h(x) = (80 \cdot 128 + 35 \cdot 32 + 168 \cdot 168 + 220 \cdot 80) \mod 257$

h is a universal hash function

Claim 1.

Consider any pair $x = (x_1, x_2, x_3, x_4)$*,* $y = (y_1, y_2, y_3, y_4)$*. If* $a_1, \ldots, a_4$ *are chosen uniformly at random from* $\{0, \ldots, n-1\}$*, then*

$$\Pr[h_a(x_1, \ldots, x_4) = h_a(y_1, \ldots, y_4)] = \frac{1}{n}$$

The proof relies on elementary number theory.

Corollary 1.

Fix $x \in U$*. The expected #elements colliding with* x *is less than 1. Hence the expected lookup time is constant.*

Ideal hash functions

From now on, assume a *completely random hash function* exists.

△ *Does not exist! But can provide a good rough idea of how hashing schemes perform in practice.*

- Let $h : U \to \{0, 1, \ldots, n-1\}$ be a completely random (ideal) hash function. For all $x \in U$, $0 \leq j \leq n-1$

$$Pr[h(x) = j] = \frac{1}{n}$$

Remark 2.

$h(x)$ is **fixed** *for every x: it just takes* **one** *of the n possible values with equal probability.*

Today

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

This is just an occupancy problem!

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

Occupancy problems, revisited: find the distribution of balls into bins when m balls are thrown independently and uniformly at random into n bins.

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

Occupancy problems, revisited: find the distribution of balls into bins when m balls are thrown independently and uniformly at random into n bins.

Hashing as an occupancy problem:

- balls correspond to elements from U
- bins are slots in the hash table
- each ball falls into one of the n bins independently and with probability $1/n$

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

Hashing as an occupancy problem:

- balls correspond to elements from U
- bins are slots in the hash table
- each ball falls into one of the n bins independently and with probability $1/n$

Q1 (rephrased): How many balls can we throw before it is more likely than not that some bin contains at least two balls?

Answer: $\Omega(\sqrt{n})$ (see the birthday paradox)

Towards analyzing time/space efficiency of hash table

- *What is the expected time for* `Lookup`(x)*?*
- *What is the expected wasted space in the hash table?*
- *What is the worst-case time for* `Lookup`(x)*?*

Towards analyzing time/space efficiency of hash table

- *What is the expected time for* `Lookup`(x)*?*
  Corresponds to expected load of a bin.
- *What is the expected wasted space in the hash table?*
  Corresponds to expected number of empty bins.
- *What is the worst-case time for* `Lookup`(x)*?*
  Corresponds to load of the fullest bin.

Towards analyzing time/space efficiency of hash table

For $n = m$

- *What is the expected time for* `Lookup`(x)*?*
  $O(1)$.
- *What is the expected wasted space in the hash table?*
  At least a third of the slots are empty.
- *What is the worst-case time for* `Lookup`(x)*, with high probability?*
  $\Theta(\ln n / \ln \ln n)$, with high probability.

Max load in any bin, with high probability (case $m = n$)

Proposition 1.

When throwing n balls into n bins uniformly and independently at random, the maximum load in any bin is $\Theta(\ln n / \ln \ln n)$ with probability close to 1 as n grows large.

Two-sentence sketch of the proof.

1. Upper bound the probability that **any** bin contains more than k balls by a union bound: $\sum_{j=1}^{n} \sum_{\ell=k}^{n} \binom{n}{\ell} \left(\frac{1}{n}\right)^{\ell} \left(1 - \frac{1}{n}\right)^{n-\ell}$.
2. Compute the smallest possible k^* such that the probability above is less than $1/n$ (which becomes negligible as n grows large).

$\square$

Today

1. Hashing
2. Analyzing hash tables using balls and bins
3. Saving space: hashing-based fingerprints
4. Bloom filters

A password checker

- We want to maintain a dictionary for a set S of 2^{16} **bad** passwords so that, when a user tries to set up a password, we can check as quickly as possible if it belongs to S and reject it.

- We assume that each password consists of 8 ASCII characters
  - hence each password requires 8 bytes (64 bits) to represent

A dictionary data structure that uses less space

Let S be the set of **bad** passwords.

Input: a 64-bit password x, and a query of the form
"Is x a **bad** *password?"*

Output: a dictionary data structure for S that answers queries as above and

- is **small**: uses **less space** than explicitly storing all bad passwords
- allows for erroneous **yes** answers occasionally
  - that is, we occasionally answer "$x \in S$" even though $x \notin S$

Approximate set membership

The password checker belongs to a broad class of problems, called *approximate set membership* problems.

Input: a large set $S = \{s_1, \ldots, s_m\}$, and queries of the form *"Is $x \in S$?"*

We want a dictionary for S that is **small** (smaller than the explicit representation provided by a hash table).
To achieve this, we allow for some probability of error

- False positives: answer **yes** when $x \notin S$
- False negatives: answer **no** when $x \in S$

Output: small probability of false positives, no false negatives

Fingerprints: hashing for saving space

- Use a hash function $h : \{0, \ldots, 2^{64} - 1\} \to \{0, \ldots, 2^{32} - 1\}$ to map each password into a 32 bit string.
- This string will serve as a short *fingerprint* of the password.
- Keep the *fingerprints* in a sorted list.
- To check if a proposed password is **bad**:
  1. calculate its *fingerprint*
  2. binary search for the *fingerprint* in the list of fingerprints; if found, declare the password **bad** and ask the user to enter a new one.

Setting the length b of the fingerprint

Why did we map passwords to 32-bit fingerprints?

Motivation: make fingerprints long enough so that the false positive probability is acceptable

Let b be the number of bits used by our hash function to map the m bad passwords into fingerprints, thus

$$h : \{0, 1, \ldots, 2^{64} - 1\} \to \{0, \ldots, 2^b - 1\}$$

We will choose b so that the probability of a false positive is acceptable, e.g., at most $1/m$.

Determining the false positive probability

There are 2^b possible strings of length b.
Let x be a **good** password.
Fix a $y \in S$ (recall that all m passwords in S are **bad**).

- $\Pr[x \text{ has the same fingerprint as } y] = 1/2^b$
- $\Pr[x \text{ does } \textit{not} \text{ have the same fingerprint as } y] = 1 - 1/2^b$
- let $p = 1 - 1/2^b$
- $\Pr[x \text{ does } \textit{not} \text{ have the same fingerprint as } \textit{any } w \in S] = p^m$
- $\Pr[x \text{ has the same fingerprint as some } w \in S] = 1 - p^m$

Hence the false positive probability is

$$1 - p^m = 1 - (1 - 1/2^b)^m \approx 1 - e^{-m/2^b}$$

Constant false positive probability and bound for b

To make the probability of a false positive less than, say, a constant c, we require

$$1 - e^{-m/2^b} \leq c \Rightarrow b \geq \log_2 \frac{m}{\ln\left(1/(1-c)\right)}.$$

So $b = \Omega(\log_2 \frac{m}{\ln(1/(1-c))})$ bits.

Improved false positive probability and bound for b

Now suppose we use $b = 2\log_2 m$.

Plugging back into the original formula for the probability of false positive, which is $1 - (1 - 1/2^b)^m$, we get

$$1 - \left(1 - \frac{1}{m^2}\right)^m \leq 1 - \left(1 - \frac{1}{m}\right) = \frac{1}{m}$$

Thus if our dictionary has $|S| = m = 2^{16}$ bad passwords, using a hash function that maps each of the m passwords to 32 bits yields a false positive probability of about $1/2^{16}$.

Today

1. Hashing
2. Analyzing hash tables using balls and bins
3. Saving space: hashing-based fingerprints
4. Bloom filters

Fast approximate set membership

Input: a *large* set S, and queries of the form *"Is $x \in S$?"*

Fast approximate set membership

Input: a *large* set S, and queries of the form *"Is $x \in S$?"*

We want a **data structure** that answers the queries

- **fast** (faster than searching in S)
- is **small** (smaller than the explicit representation provided by hash table)

Fast approximate set membership

Input: a *large* set S, and queries of the form *"Is $x \in S$?"*

We want a **data structure** that answers the queries

- **fast** (faster than searching in S)
- is **small** (smaller than the explicit representation provided by hash table)

To achieve the above, allow for some probability of error

- False positives: answer **yes** when $x \notin S$
- False negatives: answer **no** when $x \in S$

Fast approximate set membership

Input: a *large* set S, and queries of the form *"Is $x \in S$?"*

We want a **data structure** that answers the queries

- **fast** (faster than searching in S)
- is **small** (smaller than the explicit representation provided by hash table)

To achieve the above, allow for some probability of error

- False positives: answer **yes** when $x \notin S$
- False negatives: answer **no** when $x \in S$

Output: small probability of false positives, no false negatives

Bloom filter

A Bloom filter consists of:

1. an array B of n **bits**, initially all set to 0.

B =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2. k independent random hash functions $h_1, \ldots, h_k$ with range $\{0, 1, \ldots, n-1\}$.

A basic Bloom filter supports

- `Insert`(x)
- `Lookup`(x)

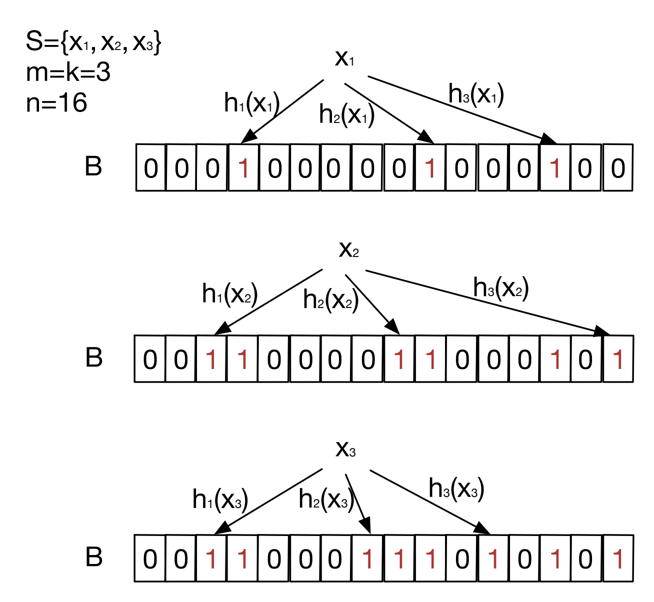
Representing a set $S = \{x_1, \ldots, x_m\}$ using a Bloom filter

`SetupBloomFilter`$(S, h_1, \ldots, h_k)$
  Initialize array B of size n to all zeros
  for $i = 1$ to m **do**
    Insert(x_i)
  end for

`Insert`(x)
  for $i = 1$ to k **do**
    compute $h_i(x)$
    set $B[h_i(x)] = 1$
  end for

Remark: an entry of B may be set multiple times; only the first change has an effect.

Setting up the Bloom filter



Bloom filter: Lookup

To check membership of an element x in S do:

`Lookup`(x)

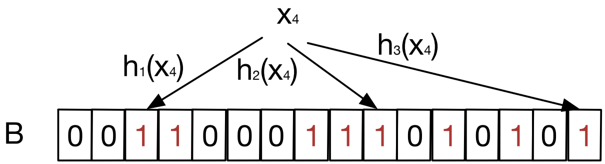
    for $i = 1$ to k **do**
        compute $h_i(x)$
        if $B[h_i(x)] == 0$ **then**
            `return` **no**
        end if
    end for
    `return` **yes**

Remark 3.

- *If $B[h_i(x)] \neq 1$ for some i, then clearly $x \notin S$.*
- *Otherwise, answer "$x \in S$" —might be a false positive!*

False positive example

Query: “is $x_4 \in S$?”



Lookup(x_4): $h_1(x_4)=h_1(x_4)=h_1(x_4)=1$

Answer: “yes”

Probability of false positive

- After all elements from S have been hashed into the Bloom filter, the probability that a specific bit is still 0 is

$$\left(1 - \frac{1}{n}\right)^{km} \approx e^{-km/n} = p.$$

- To simplify the analysis, *assume* that the fraction of bits that are still 0 is **exactly** p.
  - The fraction of bits is a random variable; we *assume* that it takes a value equal to its expectation.
- The probability of a false positive is the probability that all k hashes evaluate to 1:

$$f = (1 - p)^k$$

Optimal number of hash functions

$$f = (1-p)^k = (1-e^{-km/n})^k$$

- Trade-off between k and p: using more hash functions
  - gives us more chances to find a 0 when $x \notin S$;
  - but reduces the number of 0s in the array!
- Compute optimal number k^* of hash functions by minimizing f as a function of k:

$$k^* = (n/m) \cdot \ln 2$$

- Then the **false positive probability** is given by

$$f = (1/2)^{k^*} \approx (0.6185)^{n/m}$$

Big savings in space

- **Space** required by Bloom filter *per element of S*: n/m bits.

- For example, set $n = 8m$. Then $k^* = 6$ and $f \approx 0.02$.

$\Rightarrow$ Small constant false positive probability by using only 8 bits (1 byte) per element of S, **independently** of the size of S!

Summary on Bloom filters

Bloom filter can answer approximate set membership in

- **"constant"** time (time to hash)
- **constant** space to represent an element from S
- **constant** false positive probability f.

Application 1 (historical): spell checker

- Spelling list of $210KB$, $25K$ words.
- Use 1 byte per word.
- Maintain $25KB$ Bloom filter.
- False positive = accept a misspelled word.

Application 2: implementing joins in database

- **Join:** Combine two tables with a common domain into a single table.
- **Semi-join:** A join in distributed DBs in which only the joining attribute from one site is transmitted to the other site and used for selection. The selected records are sent back.
- **Bloom-join:** A semi-join where we send only a BF of the joining attribute.

Example

Empl	Sal	Add	City
Bale	90K	...	New York
Jones	45K	...	New York
Fletcher	45K	...	Pittsburg
Rodriguez	80K	...	Chicago
Shaw	45K	...	Chicago

City	Cost Of Living
New York	60K
Chicago	55K
Pittsburg	40K

Create a table of all employees that make $< 50K$ and live in city where Cost Of Living = COL $> 50K$.

Empl	Sal	Add	City	COL

- **Join:** send (City, COL) for COL > 50.
- **Semi-join:** send just (City) for COL > 50.
- **Bloom-join:** send a Bloom filter for all cities with COL > 50