Analysis of Algorithms, I

CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

Thursday, March 26, 2015

Outline

Today

1. Recap
2. Saving space: hashing-based fingerprints
3. 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 the following operations **efficiently**.

- `Insert`(x): add x to S if $x \notin S$
- `Delete`(x): delete x from S, if $x \in S$
- `Lookup`(x): determine if $x \in S$

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

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 $|U|$!

2. Store S in a **linked list**
   - Space: proportional to $|S|$
   - 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)?*

Chain hashing

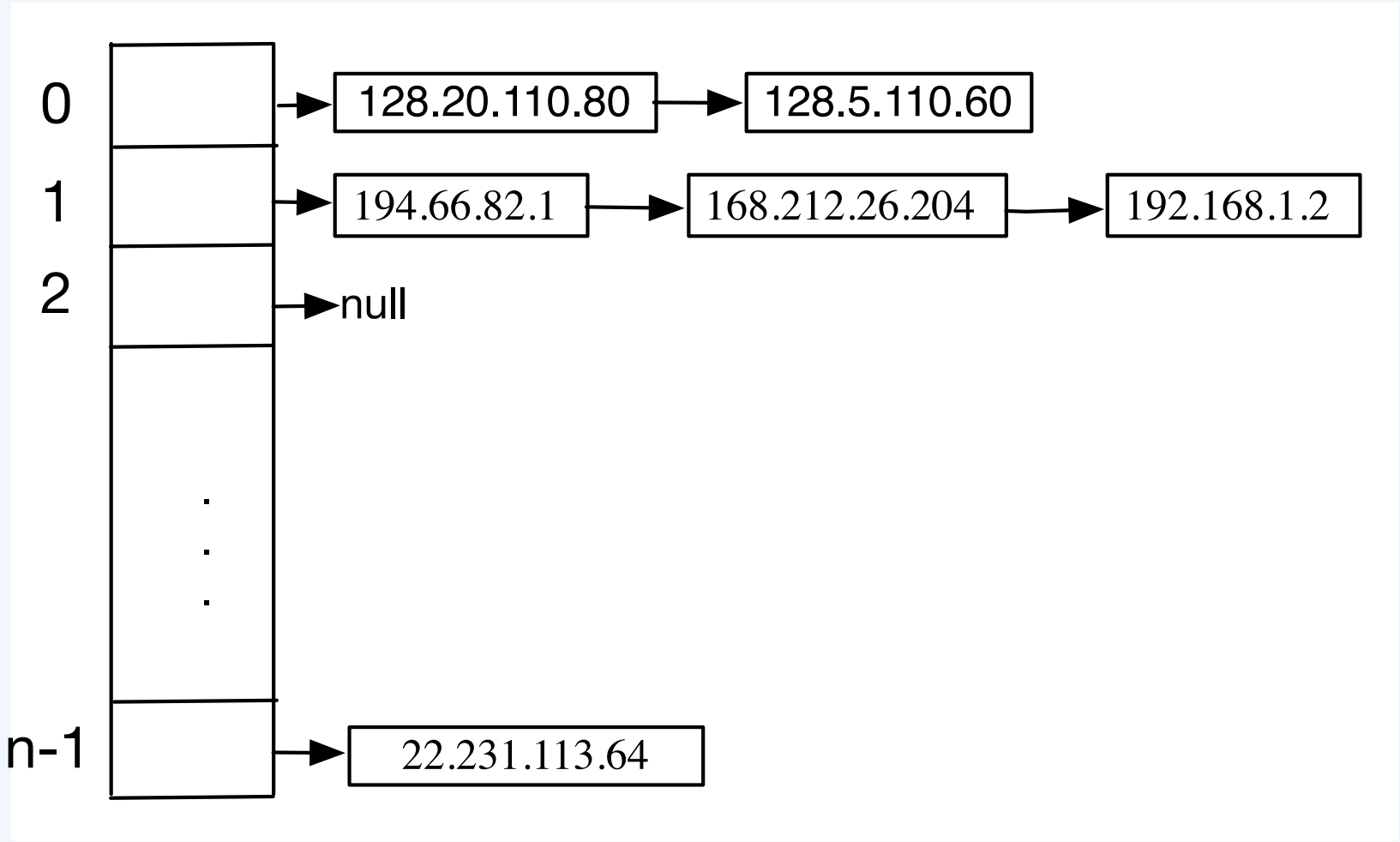
Idea: work with array of size $|S|$ rather than one of size $|U|$ by assigning a *short nickname* to each element in U

Hashing

- Hash function $h : U \to \{0, \ldots, n-1\}$
  - Typically, $n \ll |U|$ and is close to $|S|$
- An array H of size n
- Store element $x \in U$ at entry $h(x)$ of H
- Every entry in the hash table is a **linked list** of elements x such that $h(x) = j$; hence we can think of entry j as a **bin** storing all elements x such that $h(x) = j$

Chain hashing

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



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
   - this is 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
- allows for efficient dictionary operations with high probability (w.h.p.)

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

Universal class of hash functions

Idea: choose h at random from a carefully selected class H of functions into $[0, n-1]$ with the following properties:

1. h behaves almost like a completely random hash function; specifically, for any $x, y \in U$,

$$\Pr_h [h(x) = h(y)] \leq \frac{1}{n}$$

2. Can select a random h efficiently.
3. Given h, can compute $h(x)$ efficiently.

Such a family of hash functions is called universal; universal hash functions **exist**; their design relies on number theory.

Completely random hash functions

To simplify the analysis, we shall henceforth assume that we are given a *completely random hash function* $h : U \to [0, n-1]$. That is,

- for all $x \in U$, $j \in [0, n-1]$,

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

- the values of $h(x)$ for every x are independent of each other

Remark 1.

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

Notation

- U is the universe of all elements
- S is the set of elements we are maintaining
- m is the size of S
- n is the size of the hash table
- h is a completely random hash function such that for all $i \in U$, for all $j \in [0, n-1]$

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

Time/space efficiency of hash table when $n = m$

- *What is the expected load of a bin?* $O(1)$
  Corresponds to expected time for `Lookup`(x),
  assuming we can compute $h(x)$ in $O(1)$.

- *What is the load of the fullest bin?* $\Theta(\ln n / \ln \ln n)$ w.h.p.
  Corresponds to worst-case time for `Lookup`(x) w.h.p.

- *What is the expected number of empty bins?*
  Corresponds to expected wasted space in the hash table.

Expected *wasted space (#empty slots in the hash table)*

Equivalently, we want the #empty bins when m balls are thrown uniformly and independently at random into n bins.

- Fix a bin j.
  - Pr[ball i does not fall in bin j] $= 1 - 1/n$
  - Pr[**no** ball falls in bin j] $= (1 - 1/n)^m$
  - Define indicator r.v. $Y_j = 1$ if and only if bin j is empty.
- The #empty bins is given by $Y = \sum_{j=1}^{n} Y_j$.

$$E[Y] = n(1 - 1/n)^m \approx ne^{-m/n}$$

For $n = m$, we expect more than $1/3$ of the slots to be empty.

Summary on hash table when $n = m$

- Expected `Lookup` time: $O(1)$
- Worst-case `Lookup` time: $O(\ln n / \ln \ln n)$ w.h.p.
- Expected wasted space: more than $n/3$ empty slots

Today

1 Recap

2 Saving space: hashing-based fingerprints

3 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 $h : \{0, \ldots, 63\} \to \{0, \ldots, 31\}$ 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

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

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

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

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

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

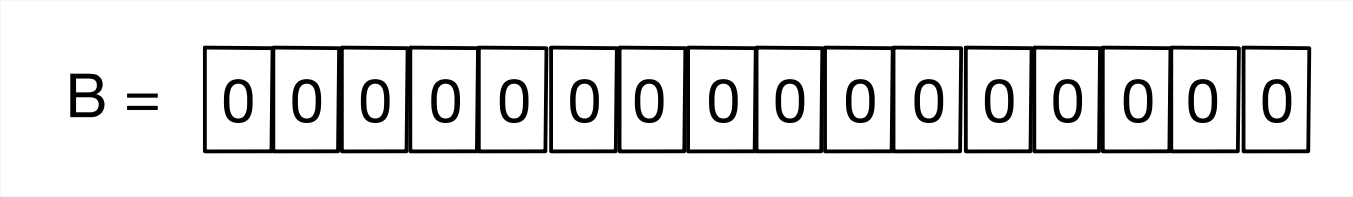
- False positives: answer **yes** when $x \notin S$
- False negatives: answer **no** when $x \in S$

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.



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

A basic Bloom filter supports

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

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

`SetupBloomFilter`$(S, h_1, \ldots, h_k)$

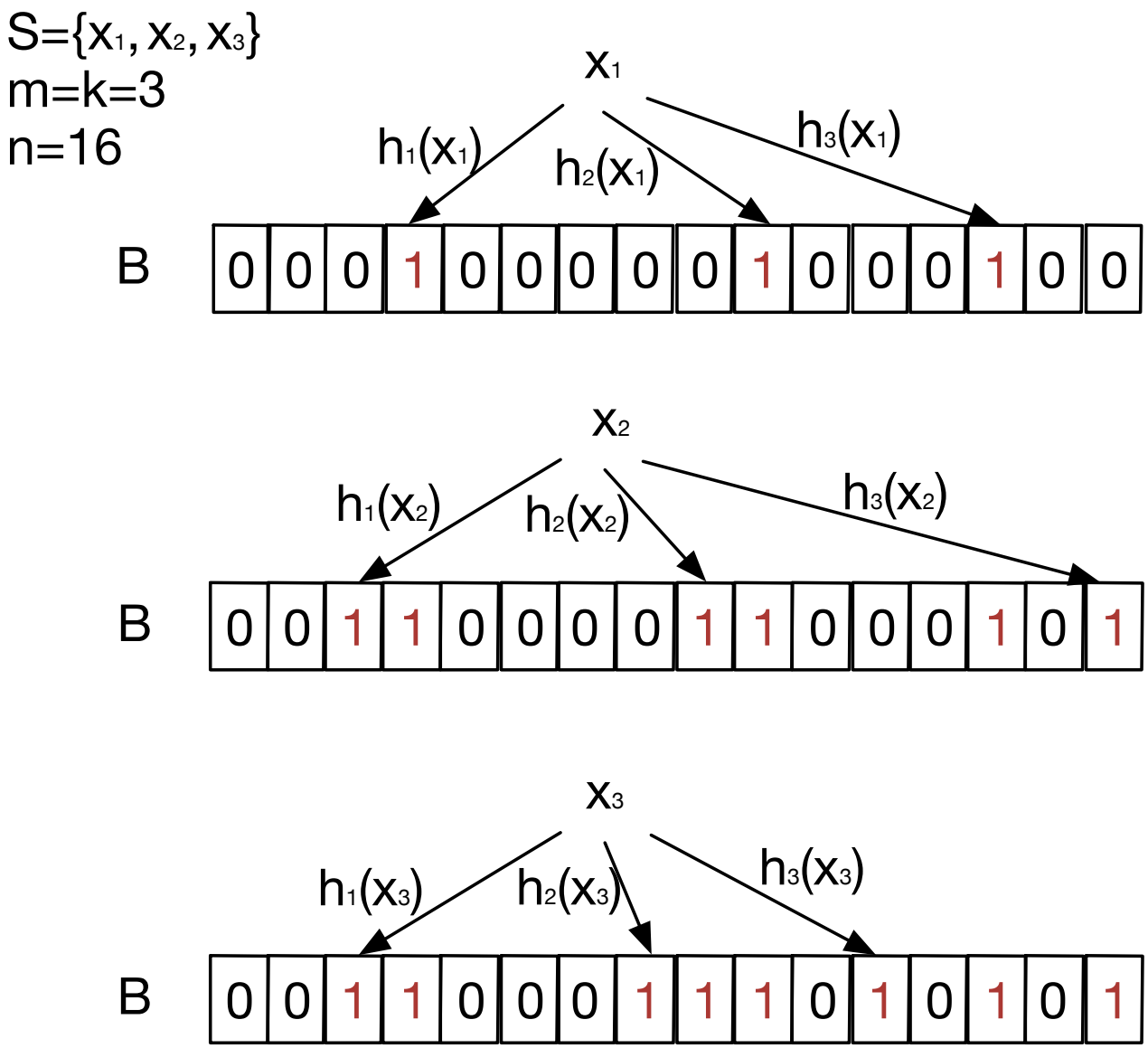
Initialize array B of size n to all zeros
for $i = 1$ to m **do**
    Insert(x_i)
end for

`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:

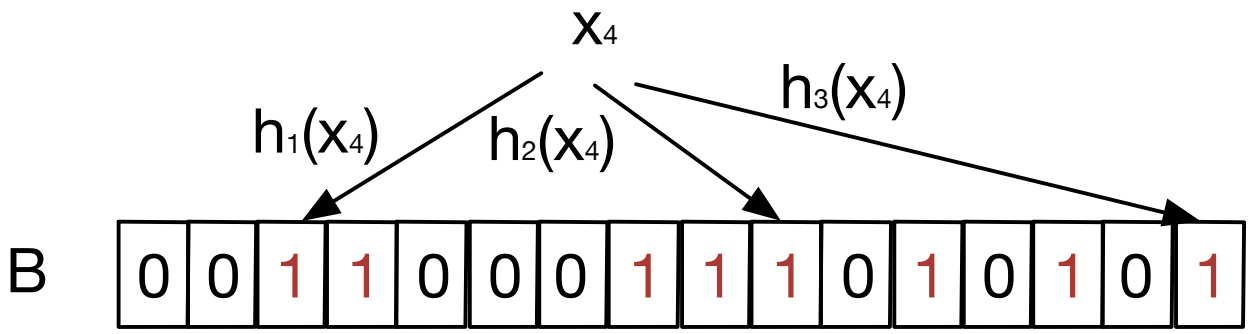
```
Lookup(x)
    for i = 1 to k do
        compute h_i(x)
        if B[h_i(x)] == 0 then
            return no
        end if
    end for
    return yes
```

Remark 2.

- *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): $h_1(x_4)=h_2(x_4)=h_3(x_4)=1$

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