# CSE462/562: Database Systems (Spring 23)

## Lecture 6: Hashing Techniques

## 2/16/2023

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

# In this lecture

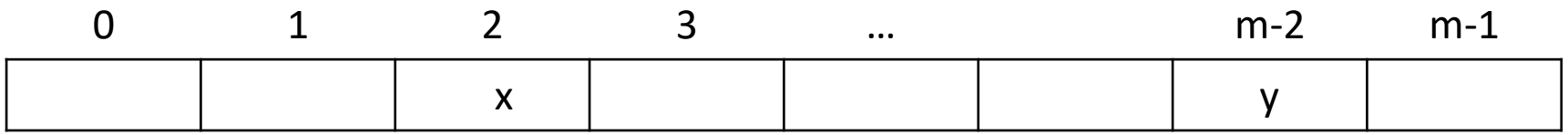
- Hashing basics
  - In-memory hash tables
- How to design a good hash function?

# Hashing basics

- Hash function $h: U \rightarrow [m]$
  - $U$: key domain, $[m] = \{0,1,2, \ldots m-1\}$
  - *Deterministic*
  - Examples:
    - Multiplicative hashing for integers: $h(x) = \lfloor m \cdot frac(x * a) \rfloor$
      - $a$: a real number with a good mixture of 0s and 1s
      - $frac(y)$: the fractional part of a real number
      - can be efficiently implemented as $h(x) = \left(\frac{ax}{2^q}\right) \ mod \ m$ for appropriately chosen integers a, q, m
    - String hashing: SHA-1, MD5
      - often available off the shelf
      - can combine a salt to create different hash functions
        - e.g., SHA-1(concat(a, s)) for some randomly chosen string $a$
      - not that secure, but works well due to its efficiency

# Hashing basics

- (In-memory) hash table
  - With a hash function $h: U \rightarrow [m]$

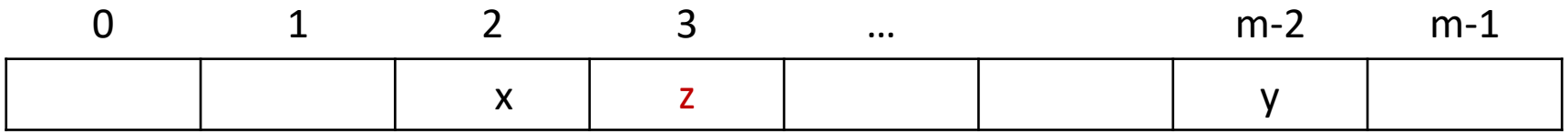



h(x) = 2
h(y) = m-2

# Hashing basics

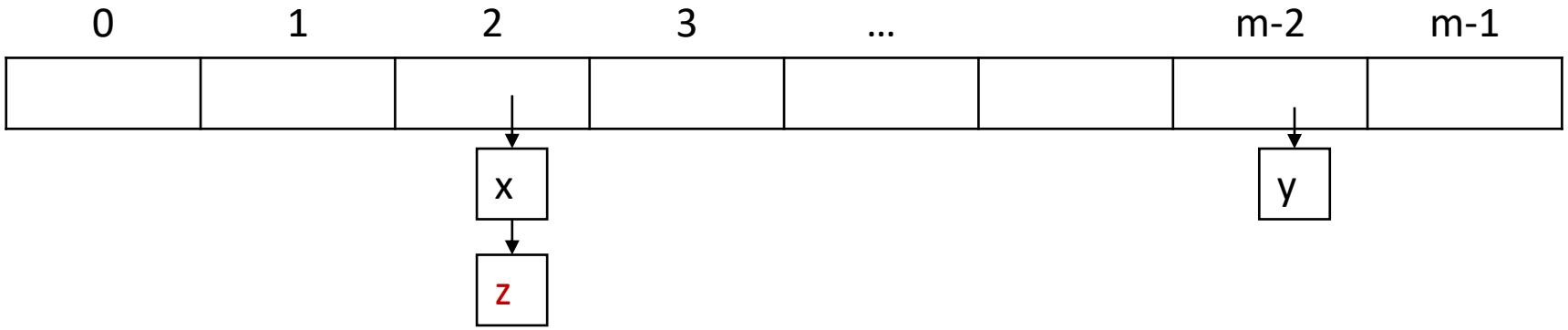
- (In-memory) hash table
  - With a hash function $h: U \to [m]$
  - How to handle collision?
    - Closed hashing vs open hashing
    - Sometimes also called open addressing vs closed addressing

closed hashing with linear probing




h(x) = 2
h(y) = m-2
h(z) = 2

open hashing with linked list



# What might go wrong with hashing?

- Too many items with the same hash value
  - Any hash table design will fail in this case
- Why can that happen?
  1. Too many entries with the same key?
     - Not much that we can do, but we can try to incorporate other fields to make the keys distinct if it's possible for the user to provide the entire key during lookups
     - Alternatively, consider using other types of index
  2. Hash collision
     - Some hash functions are prone to too many hash collisions
       - For instance, you're hashing pointers of int64_t,
         - using modular hashing $h(x) = x \; mod \; m$ with $m = 2^d$ for some d is going to leave many buckets completely empty

# Designing Good Hash Functions

- Formal set up: let U=[N] denote the numbers {0, 1, 2, . . . , N − 1}. For any set S ⊆ U, where |S|=n, we want to support these ***efficiently***:
  - add(x): add the key x to S
  - query(x): is the key q ∈ S?
  - delete(x): remove the key x from S

*We consider a specific set S. Note that even though S is fixed, we don't know S ahead of time. Imagine it's chosen by an adversary from* $\binom{N}{n}$ *possible choices.*

***Our hash function needs to work well for any such a (fixed) set S.***

# Static vs Dynamic

- Static: Given a set S of items, we want to store them so that we can do lookups quickly. (e.g., a fixed dictionary).

- Dynamic: here we have a sequence of insert, lookup, and perhaps delete requests. We want to do all these efficiently.

# Hash Function Basics

- We will perform inserts and lookups by
  an array A of M buckets, and a hash function h : U → {0,... ,M − 1} (i.e., h : U → [M]).
  Given an element x, the idea of hashing is we want to store it in A[h(x)].
  - If N=|U| is small, this problem is trivial. But in practice, N is often big.

- Collision happens when $x \neq y \wedge h(x) = h(y)$
  - Open hashing with linked list/overflow pages
  - Extendible/linear hashing can be used to alleviate the problem
    but can't handle it well if there is skewness in hash values

# Desirable Properties

- Small probability of distinct keys colliding
  - If $x \neq y$, then the probability of $h(x) = h(y)$ is "small".
  - We will see a formal definition of this shortly.
- Small range: we want M to be small.
  - At odds with first desired property
  - ideally $M=O(n)$ but it takes too much space.
- Small number of bits to store a hash function h.
- h should be easy to compute
- Given this, the time to lookup an item x is O(length of list A[h(x)])

# Bad News

- One way to spread elements out nicely is to spread them randomly.
  - Unfortunately, we can't just use a random number generator to decide where the next element goes
    - won't be able to find it again.
  - `h` should be something "pseudorandom".

- (Bad news) For any deterministic hash function h (i.e., $|H|=1$), if $|U| \geq (n-1)M + 1$, there exists a set S of n elements that all hash to the same location.
  - simple argument via pigeon hole principle
    - exercise in homework assignment

# Randomness to Rescue

- Introduce a family of hash functions, H with |H|>1, that h will be randomly chosen from for each key (but use the same choice for the same key).

- Universal Hashing: if $x \neq y \in S$ then $\Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/M$.

- Property of universal hashing:
  - for any set $S \subseteq U$ of size n,
    - if we choose h at random in a universal hash family H
      - for any $x \in U$ (e.g., that we might want to lookup, x may not come from S)
        - the expected number of collisions between x and other elements in S is at most n/M.

# Property of Universal Hashing

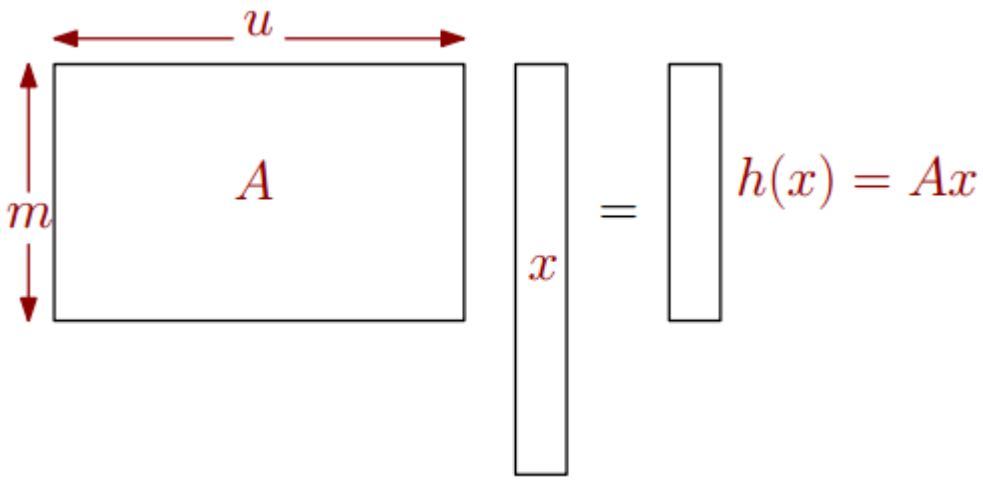
- Proof:
  - Each y ∈ S (y ≠ x) has at most a 1/M chance of colliding with x by the definition of “universal”. So
  - Let $C_{xy}$= 1 if x and y collide and 0 otherwise.
  - Let $C_x$ denote the total number of collisions for x. So, $C_x = \sum_{y \in S \wedge y \neq x} C_{xy}$.
  - We know E[$C_{xy}$] = Pr(x and y collide) ≤ 1/M.
  - So, by linearity of expectation, E[$C_x$] = $\sum_{y \in S \wedge y \neq x} E\left[C_{xy}\right] \leq$ n/M.

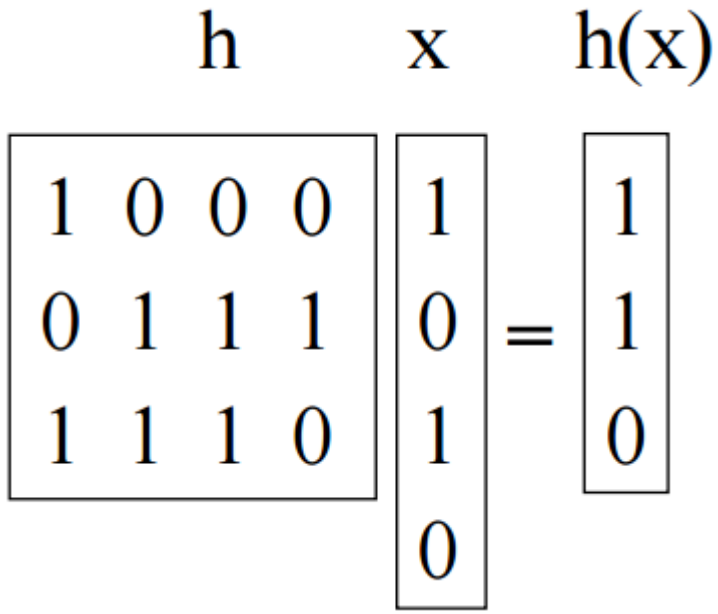
# How to Construct Universal Hashing?

- Consider the case where $|U| = 2^u$ and $M = 2^m$

- Take an u × m matrix A and fill it with random bits.
- For $x \in U$, view x as a u-bit vector in $\{0, 1\}^u$
  - h(x) := Ax where the calculations are done modulo 2.

- There are $2^{um}$ hash functions in this family H

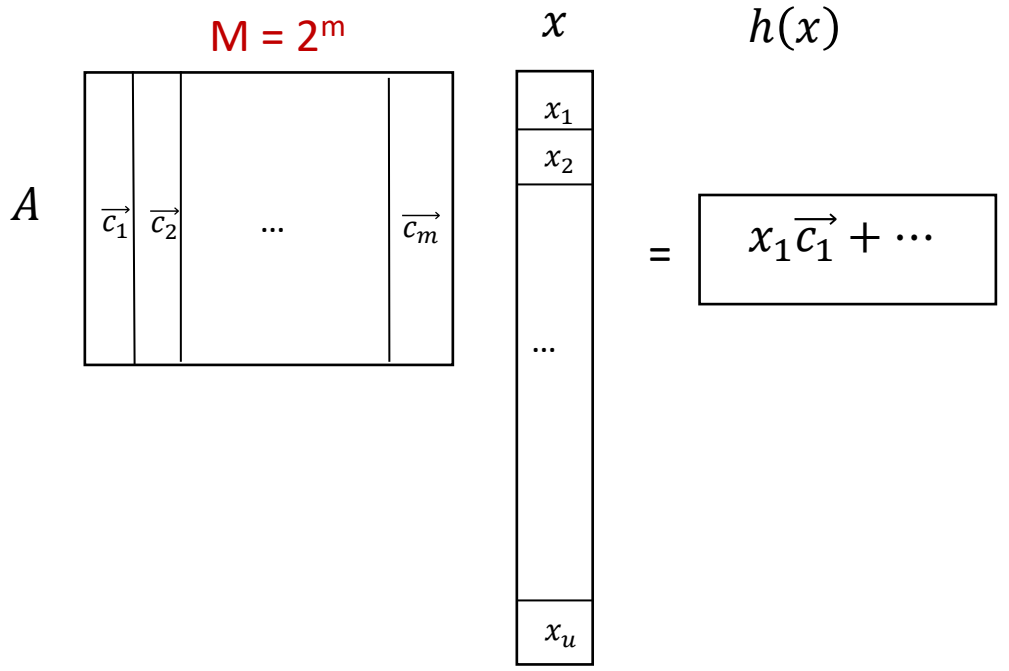



Note that $h(\vec{0}) = 0$, so picking a random function from H does not map each key to a random place

# Why it is a universal hash family?

- Proof:
- Let $A = (\overrightarrow{c_1}, \overrightarrow{c_2}, \dots, \overrightarrow{c_m})$, where $\overrightarrow{r_i}$ is the $i^{th}$ row of the matrix $A$.
- Let's view $x \in U$ as a vector of {0,1}, e.g., x = (0, 0, 1, 0, …, 1).
  - Then $\mathrm{h}(x) = x_1\overrightarrow{c_1} + x_2\overrightarrow{c_2} + \cdots + x_m\overrightarrow{c_m}$.




- Suppose we have $x^{(1)}, x^{(2)} \in U$, s.t., $x^{(1)} \neq x^{(2)}$.
  - They will differ in at least one bit.
  - Without loss of generality, let's assume it's bit 1, i.e., $x_1^{(1)} = 0, x_1^{(2)} = 1$.
  - Denote $h'(x^{(j)}) \coloneqq h(x^{(j)}) - x_1^{(j)}\overrightarrow{c_1} = x_2^{(j)}\overrightarrow{c_2} + \cdots + x_m^{(j)}\overrightarrow{c_m}$

- For any $\overrightarrow{c_2}, \dots \overrightarrow{c_m} \in \{0,1\}^u$, let's fix those vectors.
  - No matter how $\overrightarrow{c_1}$ changes, $h(x^{(1)}) = h'(x^{(1)}) + x_1^{(1)}\overrightarrow{c_1} = h'(x^{(1)})$ remain the same.
  - On the other hand,
    - $h(x^{(2)}) = x_1^{(2)}\overrightarrow{c_1} + h'(x^{(2)}) = \overrightarrow{c_1} + h'(x^{(2)})$ are all different
    - (because each $\overrightarrow{c_1}$ will be different from any other vectors by at least one bit)
    - We have only 1 out of $2^m$ different $\overrightarrow{c_1}$ so that $h(x^{(1)}) = h(x^{(2)})$.
  - Therefore, $\Pr(h(x^{(1)}) = h(x^{(2)}) | \overrightarrow{c_2}, \dots \overrightarrow{c_m}) = \frac{1}{2^m}$

- $\Pr\left(h(x^{(1)}) = h(x^{(2)})\right) = \sum_{\overrightarrow{c_2}, \dots \overrightarrow{c_m}} \Pr(h(x^{(1)}) = h(x^{(2)}) | \overrightarrow{c_2}, \dots \overrightarrow{c_m}) \Pr(\overrightarrow{c_2}, \dots \overrightarrow{c_m}) = \frac{1}{2^m} = \frac{1}{M}$ ■

This is not the only way to construct universal hash family though.

# Perfect Hashing (for static case)

- We say a hash function is perfect for S if all lookups involve O(1) work.
- Naïve method: an $O(n^2)$ space solution
- Let H be universal and $M = n^2$. Then just pick a random h from H and try it out!

- Claim: If H is universal and $M = n^2$, then $\Pr_{h \sim H}(no\ collisions\ in\ S) \geq 1/2$

# Naïve method: $O(n^2)$ space

- Proof:
  - Randomly pick a pair of different x and y from S
  - How many pairs (x,y) in S are there?
    - $n(n-1)/2$
  - For each pair, the chance they collide is ≤ 1/M by definition of "universal"
  - So, $\Pr_{h \leftarrow H}(exists\ a\ collision) \leq n(n-1)/2M = n(n-1)/2n^2 < 1/2.$ (by union bound)

# An O(n) space solution (for static S)

- First, hash all values in S into a table of size n using universal hashing.
  - This will produce some collisions (unless we are extraordinarily lucky)
- Then, for each bin
  - rehash using the naïve method until having zero collisions.

Formally:

- a first-level hash function h and first-level table A,
- n second-level hash functions h1,... ,hn and n second-level tables A1,... ,An
- To lookup an element x, we first compute i = h(x) and then find the element in $A_i$ [$h_i(x)$].
- We omit the analysis of this method.

# Dynamic S?

- Cuckoo hashing
  - Linear space
  - Constant lookup time

  - Pagh, Rasmus; Rodler, Flemming Friche (2001). "Cuckoo Hashing". *Algorithms — ESA 2001*

# Summary

- Today's lecture
  - Hashing basics
  - How to construct a good hash function

- Project 3 released today (2/16/2023)
  - Due on Tuesday, 2/28/2023, 1:00 AM EST
  - Write-up due on Thursday, 3/2/2023, 1:00 AM EST
  - *Solution code for project 2 will be released on Piazza by the end of today*

- Next lecture
  - Hash indexes