CS5112: Algorithms and Data Structures for Applications

Streaming algorithms and hashing

Ramin Zabih

Some content from: Wikipedia;
J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

Cornell University

CORNELL TECH

Administrivia

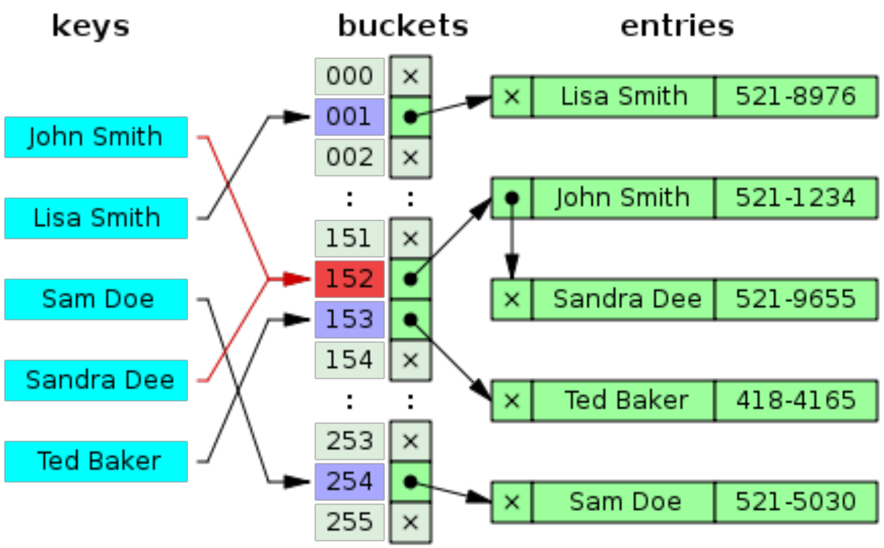
- Prelim regrades will close tonight
  - You can still appeal a problem where you submitted a regrade
  - There are some problems the course staff just finished discussing
- Prelim was probably a bit too long
  - Though shorter than last year's final exam
  - We will try to strike the right balance with corrections/errata
- Exponential sliding window for popular items
  - One stream per item sold, time advances on each sale

Lecture Outline

- Hashing and collisions
- Hashing-based streaming algorithms
  1. Flajolet-Martin: how many distinct elements?
  2. Bloom filters: did this element (potentially) appear before?
- Perfect minimal universal hashing
- Another fun application of special hash functions

Handling collisions

- More common than you think!
  - Birthday paradox
  - Example: 1M buckets and 2,450 keys uniformly distributed
  - 95% chance of a collision
- Easiest solution is chaining
  - E.g. with linked lists



CORNELL TECH

Examples of good and bad hash functions

- What is the best and worst hash function you can think of?
- Nice example from MMDS book:
  - Assume you want to hash an integer
  - Obvious choice: $h(x) = x \ mod \ B$
  - What is a good choice of B?
- To hash even integers, consider $B = 10$
  - What about $B = 11$
  - In practice we pick prime B but this does not avoid collisions
    - Prime sized hash tables

CORNELL TECH

Can you determine if there are collisions?

- Given a hashing function h from bit strings to bit strings
  - No limits on the length of input or output
- Digression: cryptographic hash functions shouldn't have collisions
  - Two inputs with same output: $h(s) = h(s')$
- Can we tell this by inspecting the hash function h?

Different excuses for failure

"I can't find an efficient algorithm, but neither can all these famous people."

Garey & Johnson, *Computers and Intractability*

Uncomputable vs intractable

- Uncomputable: proven to be impossible
  - Determine if h has any collisions
  - Almost any question about a program
  - Some very subtle problems where the input size is unbounded
- Intractable: proven at least as hard as famous open problems
- Tractable/efficient: polynomial time
  - Note this doesn't always imply practical
  - The exceptions are famous (such as Simplex)

Another difficult program

- Suppose you want to write a program like Zip that takes a file and shrinks it (file compression)
  - Without loss of information, i.e. exactly invertible
- Modern compression is pretty good!

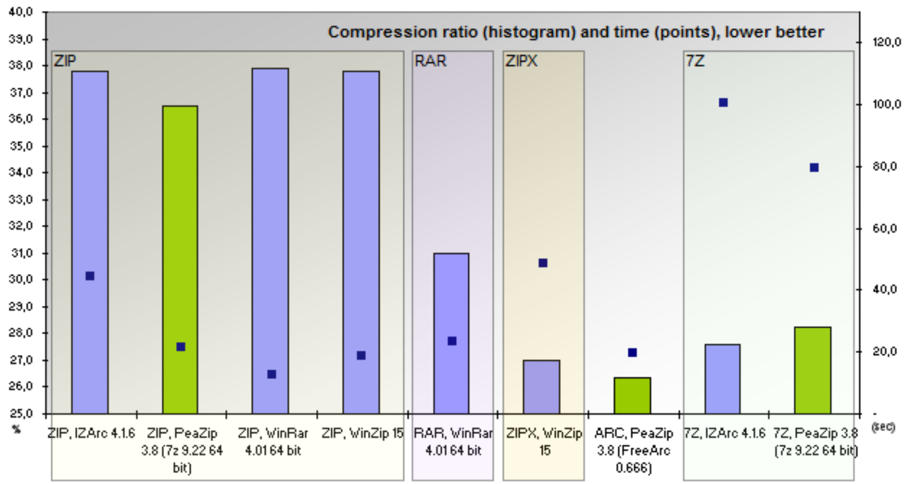


Figure source

1. Flajolet-Martin algorithm

- Basic idea: the more different elements we see, the more different hash values we will see
  - We will pick a hash function that spreads out the input elements
  - Typically uses universal hashing (later this lecture!)

Flajolet-Martin algorithm

- Pick a hash function h that maps each of the n elements to at least $\log_2 n$ bits
- For input a, let $r(a)$ be the number of trailing 0s in $h(a)$
  - $r(a)$ = position of first 1 counting from the right
  - E.g., say $h(a) = 12$, then 12 is 1100 in binary, so $r(a) = 2$
- Record R = the maximum $r(a)$ seen
- Estimated number of distinct elements = 2^R
  - Anyone see the problem here?

Why It Works: Intuition

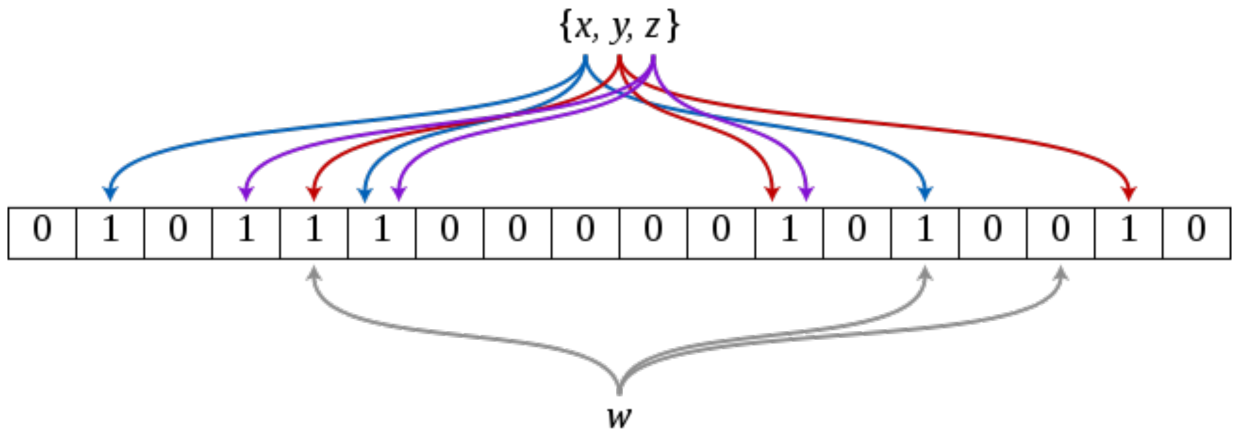
- Very rough intuition why Flajolet-Martin works:
  - $h(a)$ hashes a with equal probability to any of n values
  - Sequence of $\log_2 n$ bits where 2^{-r} fraction of $a's$ have tail of r zeros
    - About 50% hash to ***0
    - About 25% hash to **00
    - So, if we saw the longest tail of r=2 (i.e., item hash ending *100) then we have probably seen about 4 distinct items so far
  - Hash about 2^r items before we see one with zero-suffix of length r

2. Bloom filters

- Suppose you are processing items, most of them are cheap but a few of them are very expensive to process
  - Can we quickly figure out if an item **is** expensive?
  - Could store the expensive items in an associative array
  - Or use a binary valued hash table?
    - Efficient way to find out if an item **might be** expensive
- We will query set membership but allow *false positives*
  - I.e. the answer to $s \in S$ is either ‘possibly’ or ‘definitely not’
- Use a few hash functions h_i and bit array A
  - To insert s we set $A[h_i(s)] = 1 \; \forall \; i$

Bloom filter example

- Example has 3 hash functions and 18 bit array
- $\{x, y, z\}$ are in the set, w is not



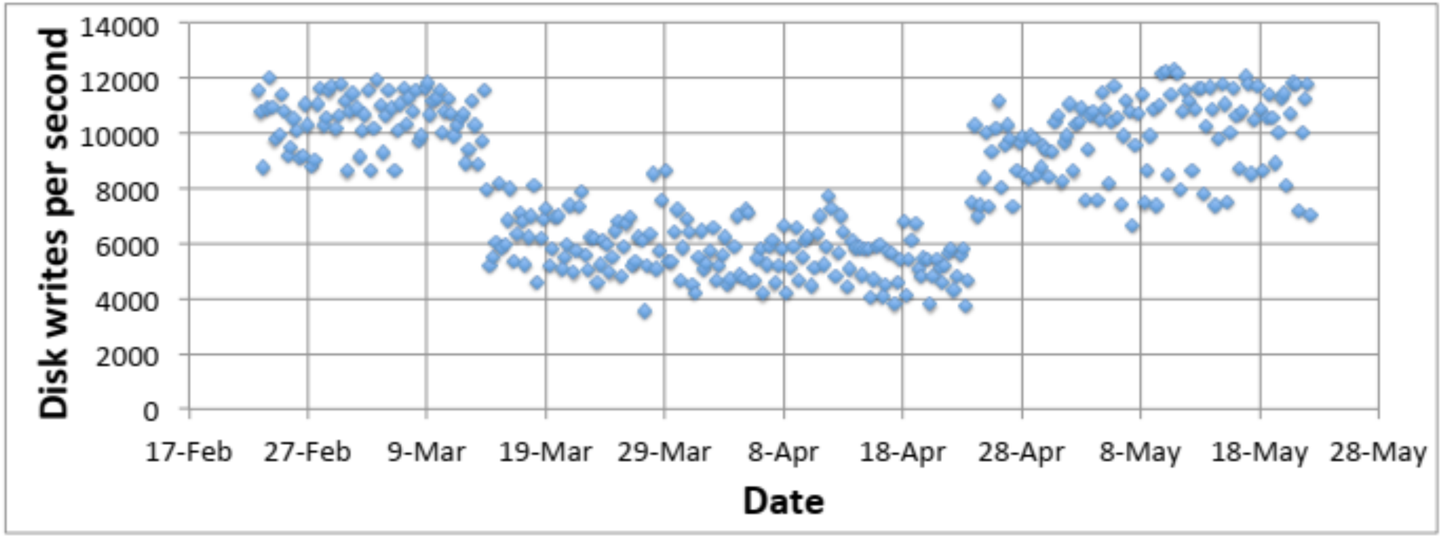
Source

Application: web caching

- CDN's, like Akamai, make the web work (~70% of traffic)
- About 75% of URL's are 'one hit wonders'
  - Never looked at again by anyone
  - Let's not do the work to put these in the disk cache!
    - Cache on second hit
- Use a Bloom filter to record URL's that have been accessed
- A one hit wonder will not be in the Bloom filter
- See: Maggs, Bruce M.; Sitaraman, Ramesh K. (July 2015), "Algorithmic nuggets in content delivery" (PDF), *SIGCOMM Computer Communication Review*, New York, NY, USA,**45** (3): 52–66

CORNELL TECH

Bloom filters really work!



- Figures from: Maggs, Bruce M.; Sitaraman, Ramesh K. (July 2015), "Algorithmic nuggets in content delivery" (PDF), *SIGCOMM Computer Communication Review*, New York, NY, USA,**45** (3): 52–66

CORNELL TECH

Cool facts about Bloom filters

- You don’t need to build different hash functions, you can use a single one and divide its output into fields (usually)
- Can calculate probability of false positives and keep it low
- Time to add an element to the filter, or check if an element is in the filter, is independent of the size of the element (!)
- You can estimate the size of the union of two sets from the bitwise OR of their Bloom filters

Perfect & minimal hashing

- Choice of hash functions is data-dependent!
- Let's try to hash 4 English words into the buckets 0,1,2,3
  - E.g., to efficiently compress a sentence
- Words: {"banana", "glib", "epic", "food"}
  - Can efficiently say sentence like "epic glib banana food" = 3,2,1,0
- Can you construct a minimal perfect hash function that maps each of these to a different bucket?
  - Needs to be efficient, not (e.g.) a list of cases

Perfect hashing example

- For this particular example, it is easy

ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(	0010 1000
N	0100 1110	l	0110 1100	)	0010 1001
				space	0010 0000

Universal hashing

- We can randomly generate a hash function h
  - This is NOT the same as the hash function being random
  - Hash function is deterministic!
  - Can re-do this if it turns out to have lots of collisions
- Assume input keys of fixed size (e.g., 32 bit numbers)
- Ideally h will spread out the keys uniformly

$$P[h(x) = h(y) |\, x \neq y] \leq \frac{1}{2^{32}}$$

  - Think of this as fixing $x, y | x \neq y$ and then picking h randomly
- If we had such an h, the expected number of collisions when we hash N numbers is $\frac{N}{2^{32}}$

CORNELL TECH

Universal hashing by matrix multiplication

- This would be of merely theoretical interest if we could not generate such an h
- There's a simple technique, not efficient enough to be practical
  – More practical versions follow the same idea
- Now assume the inputs/outputs are 4 bit numbers/3 bit numbers respectively, i.e. inputs: 0-15, outputs: 0-7
- We will randomly generate a 3x4 array of bits, and hash by 'multiplying' the input by this array

CORNELL TECH

Universal hashing example

- We multiply using AND, and we add using parity
  - Technically this is mod 2 arithmetic

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Balls into bins

- There is an important underlying idea here
  - Shows up surprisingly often
- Suppose we throw m balls into n bins
  - Where for each ball we pick a bin at random
  - How big should n be so that with probability > ½ there are no collisions?
  - This is the opposite of the birthday paradox
- Answer: need $n \approx m^2$
- So to avoid collisions with probability ½ we need our hash table to be about the square of the number of elements

Perfect hashing from universal hashing

- We can use this to create a perfect hash function
- Generate a random hash function h
  - Technically, from a universal family (like binary matrices)
- Use a "big enough" hash table, from before
  - I.e., size is square of the number of elements
- Then the chance of a collision is < ½
- In expectation we do this twice to get a perfect hash function

Rabin-Karp string search

- Find one string (“pattern”) in another
  - Naively we repeatedly shift the pattern
  - Example: To find “greg” in “richardandgreg” we compare greg against “rich”, “icha”, “char”, etc. (‘shingles’ at the word level)
- Instead let’s use a hash function h
- We first compare h(“greg”) with h(“rich”), then h(“icha”), etc.
- Only if the hash values are equal do we look at the string
  - Because $x = y \Rightarrow h(x) = h(y)$ (but not $\Leftarrow$ of course!)

Rolling hash functions

- To make this computationally efficient we need a special kind of hash function h
- As we go through “richardandgreg” looking for “greg” we will be computing h on consecutive strings of the same length
- There are clever ways to do this, but to get the flavor of them here is a naïve way that mostly works
  – Take the ASCII values of all the characters and multiply them
  – Reduce this modulo something reasonable