COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Cameron Musco

University of Massachusetts Amherst. Fall 2019.

Lecture 4

LAST TIME

Last Class We Covered:

- **Union bound:** $\Pr(A \cup B) \leq \Pr(A) + \Pr(B)$ for *any* events A, B. Application to bounding the maximum server load when using randomized routing.
- **Exponential tail bounds:** Bernstein and Chernoff bounds and the Central Limit Theorem.

This Time:

- **Bloom Filters:** Random hashing to maintain a large sets in very small space.
- **Distinct Elements:** Estimating the number of unique items in a data stream via hashing. Prelude to audio fingerprinting, document search, etc.

A GENERAL COMMENT

We have covered a lot in the first three classes.

The proofs in class are meant to illustrate techniques that can be used to tackle many algorithmic and data related problems. You do not need to have these proofs or their conclusions memorized.

- Know and be comfortable applying: independence, linearity of expectation, linearity of variance, union bound, Markov's inequality, Chebyshev's inequality, basic probability calculations.
- Able to use techniques like: breaking random variables into sums of indicator variables and analyzing expected collisions (used in CATCHA analysis, two-level hashing analysis, *and* variance calculations for randomized load balancing.)
- Know definitions of 2-universal and pairwise independent hash functions and why they are useful.
- Able to apply exponential tail bounds (not have them memorized.)
- Understand law of large numbers and central limit theorem at a high level.

APPROXIMATELY MAINTAINING A SET

Want to store a set S of items from a massive universe of possible items (e.g., images, text documents, IP addresses).

Goal: support *insert*(x) to add x to the set and *query*(x) to check if x is in the set. Both in $O(1)$ time.

- Allow small probability $\delta > 0$ of false positives. I.e., for any x,

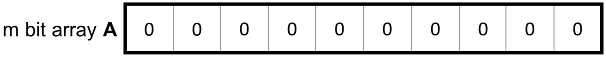
$$\Pr(\textit{query}(x) = 1 \text{ and } x \notin S) \leq \delta.$$

Solution: Bloom filters (repeated random hashing).

BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

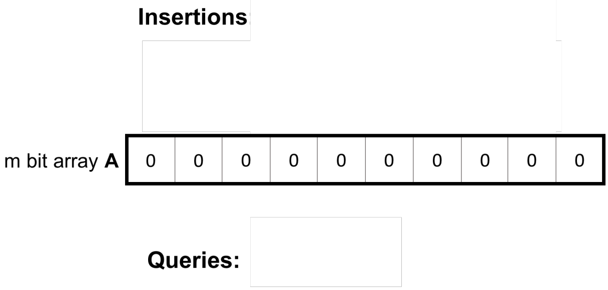
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

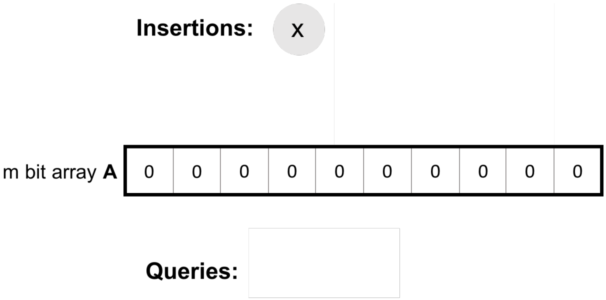
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

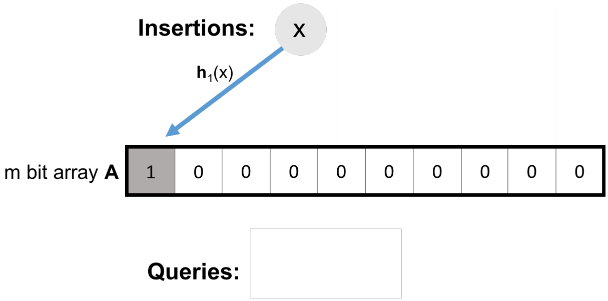
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

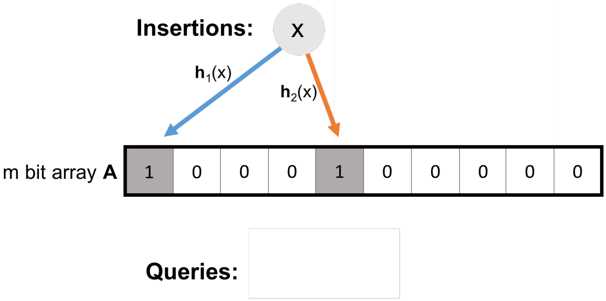
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

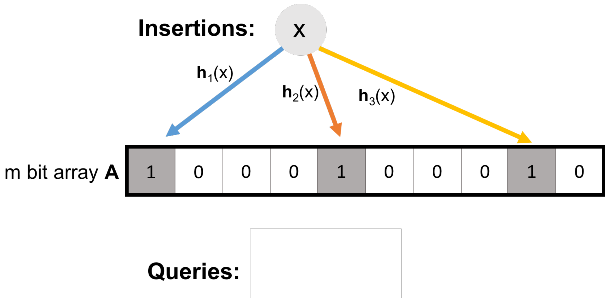
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

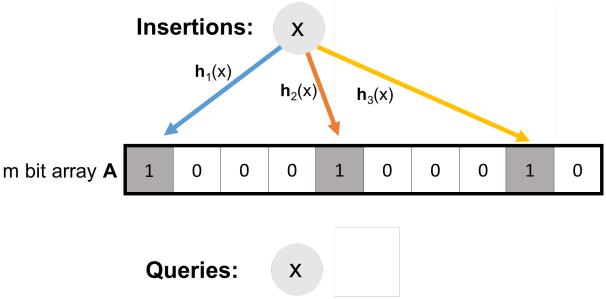
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

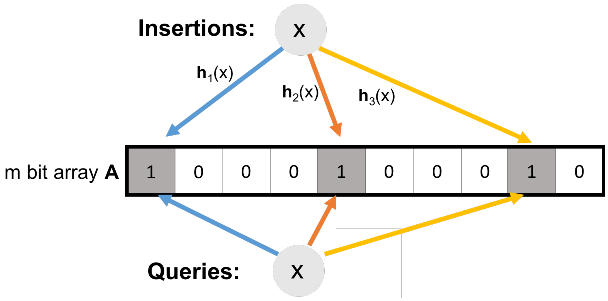
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

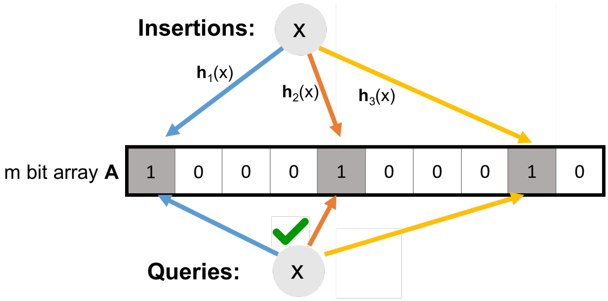
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

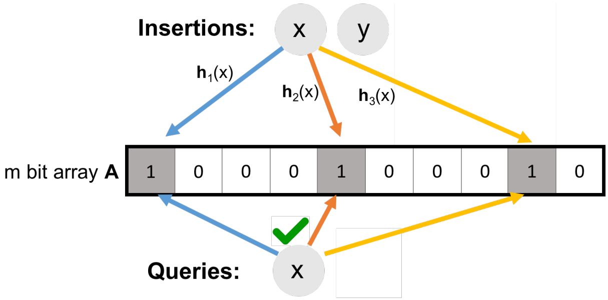
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

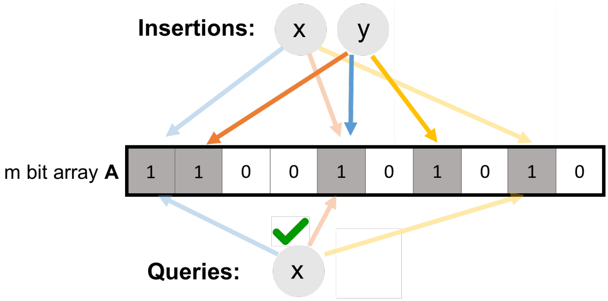
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

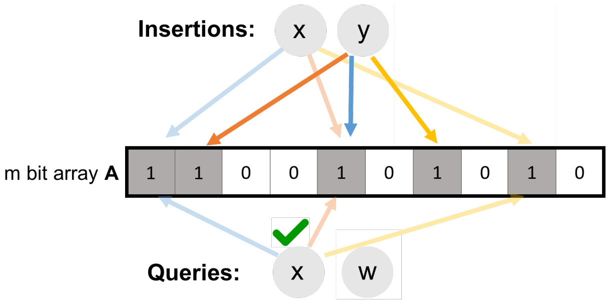
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

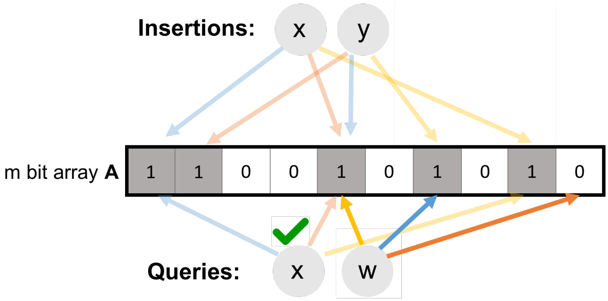
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

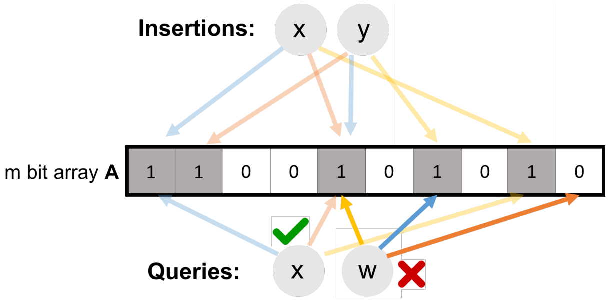
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



BLOOM FILTERS

Chose k random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

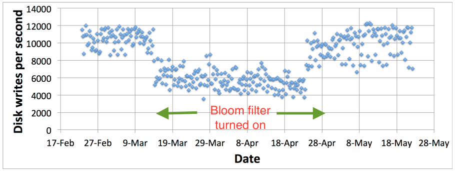
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*(x): return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



No false negatives. False positives more likely with more insertions.

APPLICATIONS: CACHING

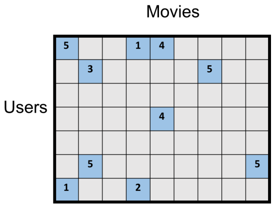
Akamai (Boston-based company serving $15-30\%$ of all web traffic) applies bloom filters to prevent caching of 'one-hit-wonders' – pages only visited once full over 75% of cache.



- When url x comes in, if $query(x) = 1$, cache the page at x. If not, run $insert(x)$ so that if it comes in again, it will be cached.
- **False positive:** A new url (possible one-hit-wonder) is cached. If the bloom filter has a false positive rate of $\delta = .05$, the number of cached one-hit-wonders will be reduced by at least 95%.

APPLICATIONS: DATABASES

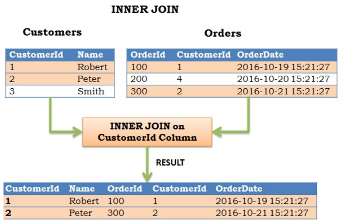
Distributed database systems, including Google Bigtable, Apache HBase, Apache Cassandra, and PostgreSQL use bloom filters to prevent expensive lookups of non-existent data.



- When a new rating is inserted for $(user_x, movie_y)$, add $(user_x, movie_y)$ to a bloom filter.
- Before reading $(user_x, movie_y)$ (possibly requiring an out of memory access), check the bloom filter, which is stored in memory.
- **False positive:** A read is made to a possibly empty cell. A $\delta = .05$ false positive rate gives a 95% reduction in these empty reads.

APPLICATIONS: DATABASES

Bloom filters are used by Oracle and other database companies to speed up database *joins*.



CustomerId	Name
1	Robert
2	Peter
3	Smith

OrderId	CustomerId	OrderDate
100	1	2016-10-19 15:21:27
200	4	2016-10-20 15:21:27
300	2	2016-10-21 15:21:27

CustomerId	Name	OrderId	CustomerId	OrderDate
1	Robert	100	1	2016-10-19 15:21:27
2	Peter	300	2	2016-10-21 15:21:27

- Matches up a key in column **A** of one table to a key in column **B** of another, and merges corresponding information.
- A bloom filter can be used to quickly eliminate entries that appear in **A** but not in **B**.
- A false positive rate of δ means that a $1 - \delta$ fraction of these entries can be eliminated in the initial bloom filter check.

MORE APPLICATIONS

- Recommendation systems (Netflix, Youtube, Tinder, etc.) use bloom filters to prevent showing users the same recommendations twice.
- Spam/Fraud Detection:
    - Bit.ly and Google Chrome use bloom filters to quickly check if a url maps to a flagged site and prevent a user from following it.
    - Can be used to detect repeat clicks on the same ad from a single IP-address, which may be the result of fraud.
- Digital Currency: Some Bitcoin clients use bloom filters to quickly pare down the full transaction log to transactions involving bitcoin addresses that are relevant to them (SPV: simplified payment verification).

ANALYSIS

For a bloom filter with m bits and k hash functions, the insertion and query time is $O(k)$. How does the false positive rate δ depend on m, k, and the number of items inserted?

Step 1: What is the probability that after inserting n elements, the i^{th} bit of the array A is still 0? $n \times k$ total hashes must not hit bit i.

$$\begin{aligned}\Pr(A[i]=0) &= \Pr\big(\mathbf{h}_1(x_1) \neq i \cap \ldots \cap \mathbf{h}_k(x_k) \neq i \\ &\qquad \cap\, \mathbf{h}_1(x_2) \neq i \ldots \cap \mathbf{h}_k(x_2) \neq i \cap \ldots\big) \\ &= \underbrace{\Pr\left(\mathbf{h}_1(x_1) \neq i\right) \times \ldots \times \Pr\left(\mathbf{h}_k(x_1) \neq i\right) \times \Pr\left(\mathbf{h}_1(x_2) \neq i\right) \ldots}_{k\cdot n \text{ events each occuring with probability } 1-1/m} \\ &= \left(1 - \frac{1}{m}\right)^{kn}\end{aligned}$$

ANALYSIS

How does the false positive rate δ depend on m, k, and the number of items inserted?

Step 1: What is the probability that after inserting n elements, the i^{th} bit of the array A is still 0?

$$\Pr(A[i] = 0) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$$

Step 2: What is the probability that querying a new item w gives a false positive?

$$\begin{aligned}\Pr\left(A[\mathbf{h}_1(w)] = A[\mathbf{h}_2(w)] = \ldots = A[\mathbf{h}_k(w)] = 1\right) \\ &= \Pr(A[\mathbf{h}_1(w)] = 1) \times \ldots \times \Pr(A[\mathbf{h}_k(w)] = 1) \\ &= \left(1 - e^{-\frac{kn}{m}}\right)^k\end{aligned}$$

Actually Incorrect! Dependent events.

n: total number items in filter, *m*: number of bits in filter, *k*: number of random hash functions, $\mathbf{h}_1, \ldots \mathbf{h}_k$: hash functions, *A*: bit array, δ: false positive rate.

CORRECT ANALYSIS SKETCH

Step 1: To avoid dependence issues, condition on the event that the A has t zeros in it after n insertions, for some $t \leq m$. For a non-inserted element w, after conditioning on this event we correctly have:

$$\begin{aligned}\Pr(A[\mathbf{h}_1(w)] = \ldots = A[\mathbf{h}_k(w)] = 1) \\ &= \Pr(A[\mathbf{h}_1(w)] = 1) \times \ldots \times \Pr(A[\mathbf{h}_k(w)] = 1).\end{aligned}$$

I.e., the events $A[\mathbf{h}_1(w)] = 1$,..., $A[\mathbf{h}_k(w)] = 1$ are independent conditioned on the number of bits set in A.

- Conditioned on this event, for any j, since $\mathbf{h}_j$ is a fully random hash function, $\Pr(A[\mathbf{h}_j(w)] = 1) = \frac{t}{m}$.
- Thus conditioned on this event, the false positive rate is $\left(1 - \frac{t}{m}\right)^k$
- It remains to show that $\frac{t}{m} = O\left(\left(1 - \frac{1}{m}\right)^{kn}\right)$ with high probability. We already have that $\mathbb{E}[\frac{t}{m}] = \frac{1}{m}\sum_{i=1}^{m} \Pr(A[i] = 0) = \left(1 - \frac{1}{m}\right)^{kn}$.

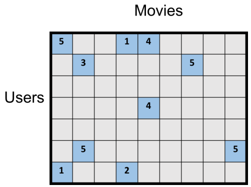
CORRECT ANALYSIS SKETCH

Need to show that the number of zeros t in A after n insertions is bounded by $O\left(\left(1-\frac{1}{m}\right)^{kn}\right)$ with high probability.

Can apply Theorem 2 of: `http://cglab.ca/~morin/publications/ds/bloom-submitted.pdf`

FALSE POSITIVE RATE

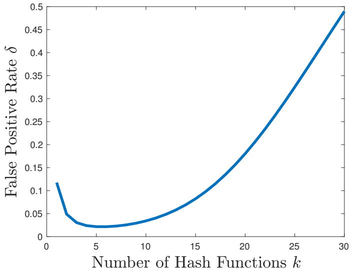
False Positive Rate: with m bits of storage, k hash functions, and n items inserted $\delta \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$ (proof can be fixed).



- We have 100 million users and $10,000$ movies. On average each user has rated only 10 movies so of these 10^{12} possible (user,movie) pairs, only $10 * 100,000,000 = 10^9 = n$ (user,movie) pairs have non-empty entries in our table.
- We allocate $m = 8n = 8 \times 10^9$ bits for a Bloom filter (1 GB). How should we set k to minimize the number of false positives?

FALSE POSITIVE RATE

False Positive Rate: with m bits of storage, k hash functions, and n items inserted $\delta \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$ (proof can be fixed).



- Can differentiate to show optimal number of hashes is $k = \ln 2 \cdot \frac{m}{n}$.
- Balances between filling up the array with too many hashes and having enough hashes so that even when the array is pretty full, a new item is unlikely to have all its bits set (yield a false positive)

SOMETHING TO THINK ABOUT

What if we wanted to maintain a set with possible false negatives but no false positives?

Turns out that this is extremely difficult.

Questions on Bloom Filters?

STREAMING ALGORITHMS

Stream Processing: Have a massive dataset X with n items $x_1, x_2, \ldots, x_n$ that arrive in a continuous stream. Not nearly enough space to store all the items (in a single location).

- Still want to analyze and learn from this data.
- Typically must compress the data on the fly, storing a data structure from which you can still learn useful information.
- Often the compression is randomized. E.g., bloom filters.
- Compared to traditional algorithmic design, which focuses on minimizing runtime, the big question here is how much space is needed to answer queries of interest.

SOME EXAMPLES

- **Sensor data:** images from telescopes (15 terabytes per night from the Large Synoptic Survey Telescope), readings from seismometer arrays monitoring and predicting earthquake activity, traffic cameras and travel time sensors (Smart Cities), electrical grid monitoring.

SOME EXAMPLES

- **Sensor data:** images from telescopes (15 terabytes per night from the Large Synoptic Survey Telescope), readings from seismometer arrays monitoring and predicting earthquake activity, traffic cameras and travel time sensors (Smart Cities), electrical grid monitoring.
- **Internet Traffic**: 500 million Tweets per day, 5.6 billion Google searches, billions of ad-clicks and other logs from instrumented webpages, IPs routed by network switches, ...
- **Datasets in Machine Learning:** When training e.g. a neural network on a large dataset (ImageNet with 14 million images), the data is typically processed in a stream due to storage limitations.

STORM Flink Spark Streaming samza

DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements in the stream. E.g.,

$$1, 5, 7, 5, 2, 1 \rightarrow 4 \text{ distinct elements}$$

Applications:

- Distinct IP addresses clicking on an ad or visiting a site.
- Distinct values in a database column (for estimating sizes of joins and group bys).
- Number of distinct search engine queries.
- Counting distinct motifs in large DNA sequences.

Google Sawzall, Facebook Presto, Apache Drill, Twitter Algebird

HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

Hashing for Distinct Elements (variant of Flajolet-Martin):

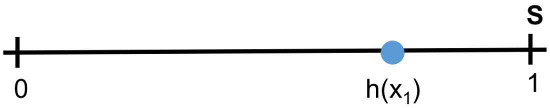
- Let $\mathbf{h} : U \to [0,1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$

HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

Hashing for Distinct Elements (variant of Flajolet-Martin):

- Let $\mathbf{h} : U \to [0, 1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$

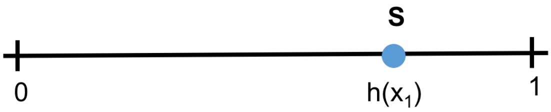


HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

Hashing for Distinct Elements (variant of Flajolet-Martin):

- Let $\mathbf{h} : U \to [0, 1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$

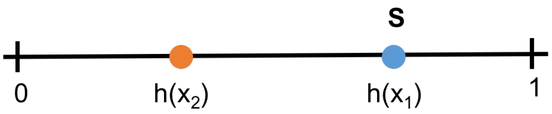


HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

Hashing for Distinct Elements (variant of Flajolet-Martin):

- Let $\mathbf{h} : U \to [0,1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$

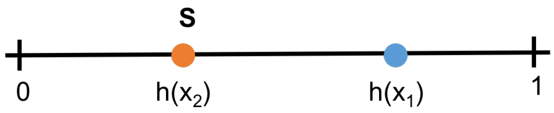


HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

Hashing for Distinct Elements (variant of Flajolet-Martin):

- Let $\mathbf{h} : U \to [0,1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$

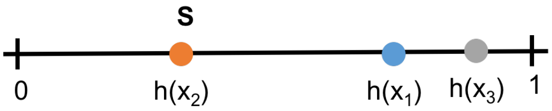


HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

Hashing for Distinct Elements (variant of Flajolet-Martin):

- Let $\mathbf{h} : U \to [0,1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$

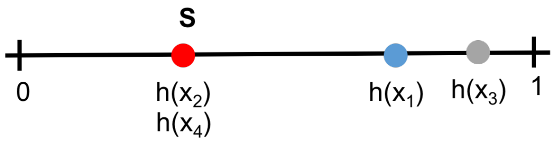


HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

Hashing for Distinct Elements (variant of Flajolet-Martin):

- Let $\mathbf{h} : U \to [0, 1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$

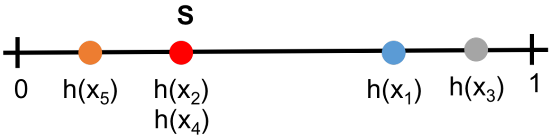


HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

Hashing for Distinct Elements (variant of Flajolet-Martin):

- Let $\mathbf{h} : U \to [0, 1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$

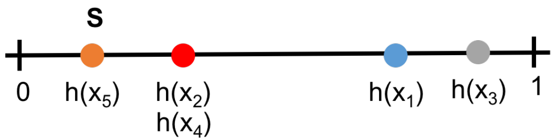


HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements.

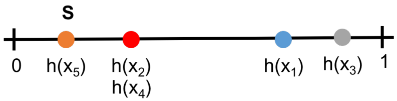
Hashing for Distinct Elements (variant of Flajolet-Martin):

- Let $\mathbf{h} : U \to [0, 1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$



Hashing for Distinct Elements:

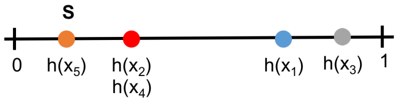
- Let $\mathbf{h} : U \to [0,1]$ be a random hash function (with a real valued output)
- $s := 1$
- For $i = 1, \ldots, n$
  - $s := \min(s, \mathbf{h}(x_i))$
- Return $\tilde{d} = \frac{1}{s} - 1$



- After all items are processed, s is the minimum of d points chosen uniformly at random on $[0,1]$. Where $d = \#$ distinct elements.
- Intuition: The larger d is, the smaller we expect s to be.
- Same idea as Flajolet-Martin algorithm and HyperLogLog, except they use discrete hash functions.

PERFORMANCE IN EXPECTATION

s computed by hashing algorithm is the minimum of d values chosen randomly in $[0, 1]$.



$$\mathbb{E}[s] = \frac{1}{d+1} \text{ (Interesting to prove to yourself.)}$$

- So estimate of $\tilde{d} = \frac{1}{s} - 1$ is correct if s exactly equals its expectation.
- If $|s - \mathbb{E}[s]| \leq \epsilon \cdot \mathbb{E}[s]$ for any $\epsilon \in (0, 1/2)$ can show:

$$(1 - 2\epsilon)d \leq \tilde{d} \leq (1 + 4\epsilon)d$$

.

Next Time:

- Complete the analysis of hashing algorithm for distinct elements.
- Use a min-of-hashes technique for a different problem: estimating the similarity between two bit strings.

- A key idea behind audio fingerprint search (Shazam), document search (plagiarism and copyright violation detection), recommendation systems, etc.

Questions?