

COMPSCI 514: Algorithms for Data Science

Cameron Musco

University of Massachusetts Amherst. Fall 2026.

Lecture 6

quiz section?
graduate part 1

In-Class Quiz 2

$$\left[\begin{array}{ccc} X_1 & X_2 & \dots & X_n \\ -1 & 1 & \dots & 1 \end{array} \right] \quad X = \sum_{i=1}^n X_i \quad \mathbb{E}[X] = \sum_{i=1}^n \mathbb{E}[X_i] = \sum 0 = 0$$

$$\text{Var}[X] = \sum_{i=1}^n \text{Var}(X_i) = \sum_{i=1}^n \mathbb{E}[X_i^2 - (\mathbb{E}[X_i])^2] = \sum_{i=1}^n 1 = n$$

Question 1: Consider independent random variables $X_1, \dots, X_n$, each taking value -1 with probability $1/2$ and 1 with probability $1/2$. Let $X = \sum_{i=1}^n X_i$. What are $\mathbb{E}[X]$ and $\text{Var}[X]$?

$$Y_i = \frac{X_i + 1}{2}$$

$$Y_i = \{0, 1\}$$

Question 2: Use Chebyshev's inequality to prove an upper bound on $\Pr[|X| \geq 5\sqrt{n}]$. If you wanted to prove a tighter bound, what concentration bound(s) taught in class could you potentially use?

$$\Pr[|X| \geq 5\sqrt{n}] \leq \frac{\text{Var}(X)}{(5\sqrt{n})^2} = \frac{n}{25n} = \frac{1}{25}$$

- Bernstein's, Chernoff, higher moments (1 answer sufficient)

- Problem Set 1 is due this Friday..
- My office hours this week will be **Wednesday**
10:30am-11:30am

Lecture Pacing:

- Way too Fast – 0 13
- A Bit too Fast – 1 17
- Just Right – 1 16
- Too Slow – 1 1
- Way Too Slow – 0 0

Last Time

Last Class:

- Exponential concentration bounds
- Bernstein inequality and the Chernoff bound
- Connection to the central limit theorem.
- Application to random hashing – $O(\log n)$ maximum load per bucket when hashing n items into n buckets.

$$\Pr(X - \mathbb{E}X \geq s) \leq e^{-s^2}$$

This Class:

- Bloom filters: random hashing to maintain a large set in small space.

Chernoff Bound Analysis

We showed that when hashing n items into n buckets of a hash table, the maximum load is $O(\log n)$ with high probability? How did we come to that answer?

Chernoff Bound Analysis

We showed that when hashing n items into n buckets of a hash table, the maximum load is $O(\log n)$ with high probability? How did we come to that answer?

- independent indicators*
- Let X_i = $\sum_{j=1}^n X_{i,j}$ be the number of items in bucket i .
 - $\mathbb{E}[X_i] = 1$ and by a Chernoff bound, for any $\delta > 0$

$$\Pr[X_i > 1 + \delta] \leq \Pr[|X_i - 1| > \delta] \leq 2e^{-\frac{\delta^2}{2+\delta}} \approx e^{-\delta}.$$

$$\Pr[X_i > 1 + \delta] = \underset{A}{\Pr[X_i - 1 > \delta]} \Rightarrow \underset{B}{\Pr[|X_i - 1| > \delta]}$$

Chernoff Bound Analysis

We showed that when hashing n items into n buckets of a hash table, the maximum load is $O(\log n)$ with high probability? How did we come to that answer?

- If I hash n items to n buckets except with 1% prob, there are at most $1 + \ln(100n) = O(\log n)$ items per bucket*
- Let $X_i = \sum_{j=1}^n X_{i,j}$ be the number of items in bucket i .
 - $\mathbb{E}[X_i] = 1$ and by a Chernoff bound, for any δ ,

One bucket prob.

$$\Pr[X_i > 1 + \delta] \leq \Pr[|X_i - 1| > \delta] \leq 2e^{-\frac{\delta^2}{2+\delta}} \approx e^{-\delta}$$

$$\Pr[\max X_i \geq 1 + \delta] = \Pr[\cup (X_i \geq 1 + \delta)] \leq \sum \Pr[X_i \geq 1 + \delta]$$

buckets that may be over-loaded

- By a union bound, $\Pr[\max X_i > 1 + \delta] \leq \underline{n \cdot e^{-\delta}}$.
- So, e.g, to have $\Pr[\max X_i > 1 + \delta] \leq \underline{\frac{1}{100}}$ we need to set:

$$n \cdot e^{-\delta} \leq \frac{1}{100} \implies e^{-\delta} \leq \frac{1}{100n} \implies \delta = \ln(100n) = O(\log n).$$

c/gebraik.

$$\begin{aligned} -\delta &\leq \log\left(\frac{1}{100n}\right) \\ \delta &\geq \log(100n) \end{aligned}$$

Bloom Filters

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).

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.

↳ classic solutions: hash table

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. What data structure solves this problem? hash tables

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. **What data structure solves this problem?**

• 01

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

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

- IF $x \in S$, algorithm should always report $query(x) = 1$
(no false negatives)

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. **What data structure solves this problem?**

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

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

Solution: Bloom filters (repeated random hashing). Will use much less space than a hash table.

Bloom Filters

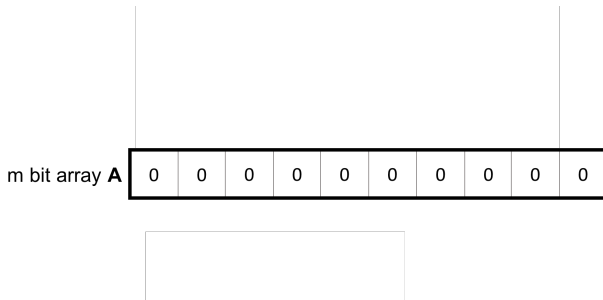
Chose k independent random hash functions $\mathbf{h}_1, \dots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

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

Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

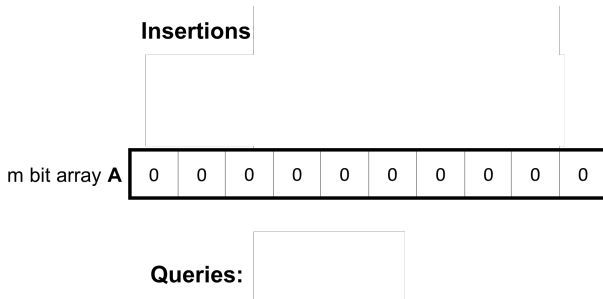
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

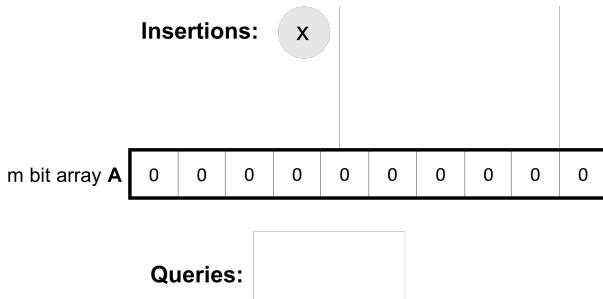
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.

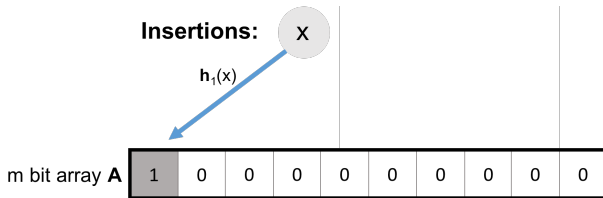


Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.

$k=3$

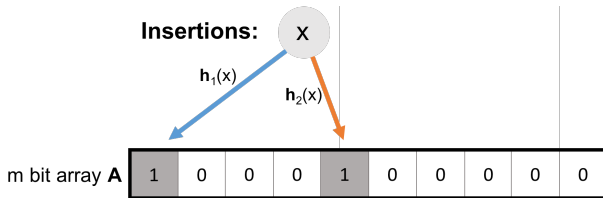


Queries:

Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.

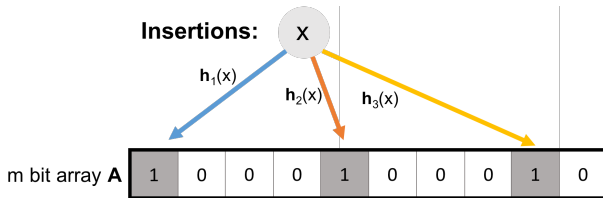


Queries:

Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.

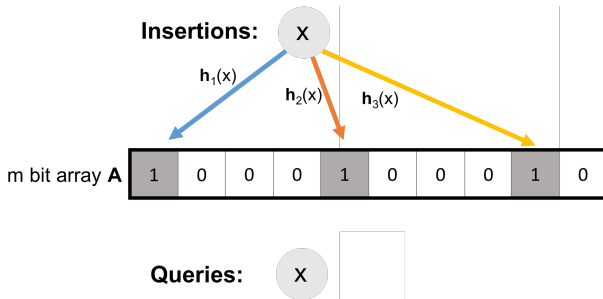


Queries:

Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

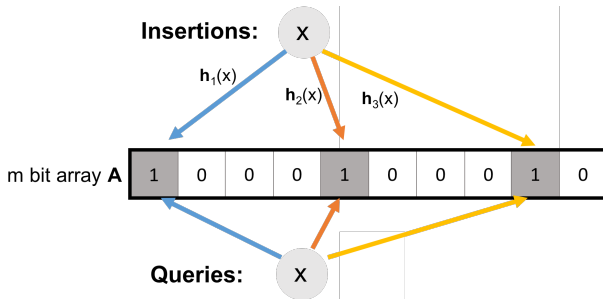
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

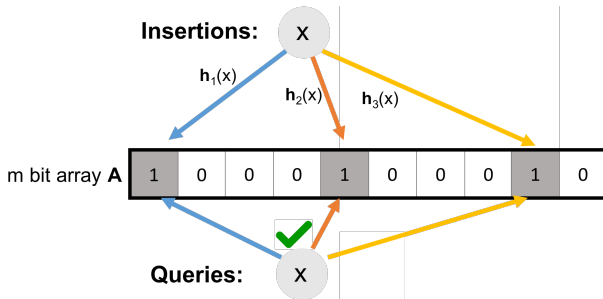
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

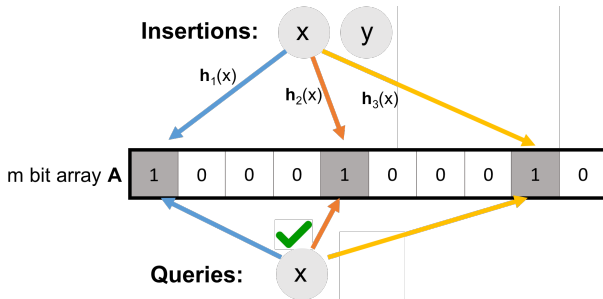
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

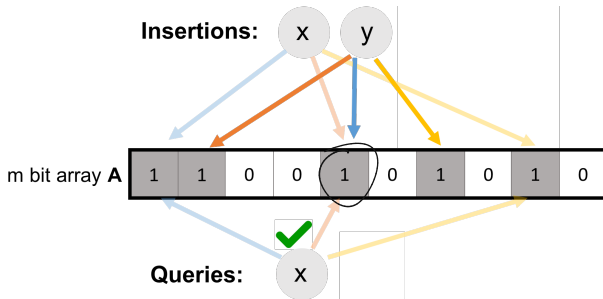
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

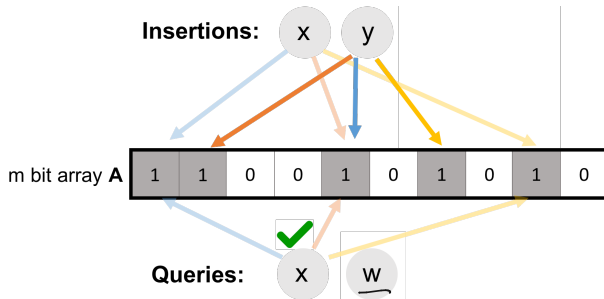
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

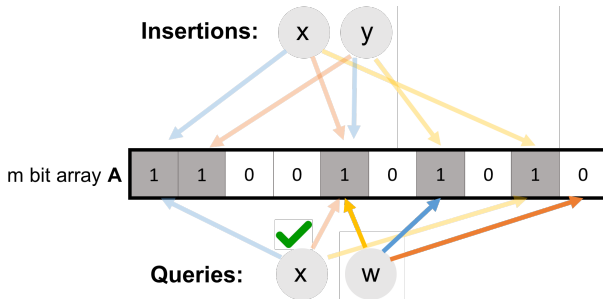
- Maintain an array A containing m bits, all initially 0.
- *insert*(x): set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- *query*(x): return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

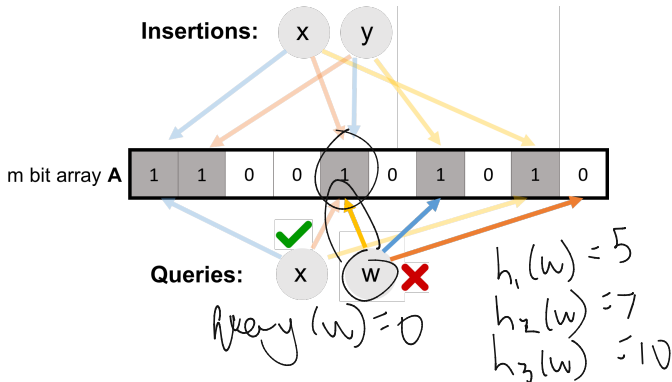
- Maintain an array A containing m bits, all initially 0.
- $insert(x)$: set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- $query(x)$: return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

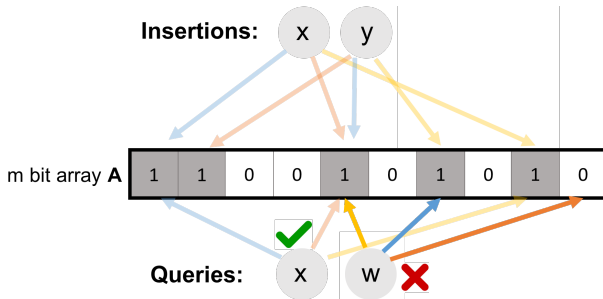
- Maintain an array A containing m bits, all initially 0.
- $insert(x)$: set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- $query(x)$: return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



Bloom Filters

Chose k independent random hash functions $h_1, \dots, h_k$ mapping the universe of elements $U \rightarrow [m]$.

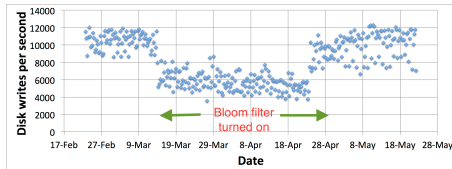
- Maintain an array A containing m bits, all initially 0.
- $insert(x)$: set all bits $A[h_1(x)] = \dots = A[h_k(x)] := 1$.
- $query(x)$: return 1 only if $A[h_1(x)] = \dots = A[h_k(x)] = 1$.



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

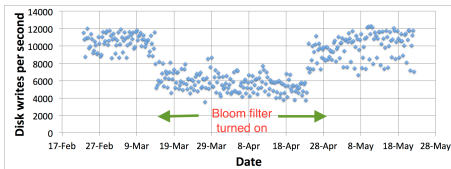
Applications: Caching

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 fill over 75% of cache.



Applications: Caching

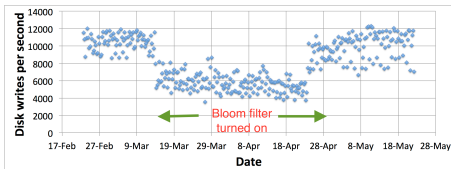
Akamai (Boston-based company serving 15 – 30% of all web traffic) applies bloom filters to prevent caching of ‘one-hit-wonders’ – pages only visited once fill over 75% of cache.



- 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.

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 fill 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.

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.

Movies

	5			1	4				
		3					5		
				4					
		5							5
	1			2					

Users

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.

Movies

	5			1	4				
		3					5		
Users					4				
		5							5
	1			2					

- 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 via an out of memory access), check the bloom filter, which is stored in memory.

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.

Movies

	5			1	4				
		3					5		
Users									
					4				
		5							5
	1			2					

- 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 via 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.

More Applications

- **Database Joins:** Quickly eliminate most keys in one column that don't correspond to keys in another.

• **Recommendation systems:** Bloom filters are used 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 $O(k)$.

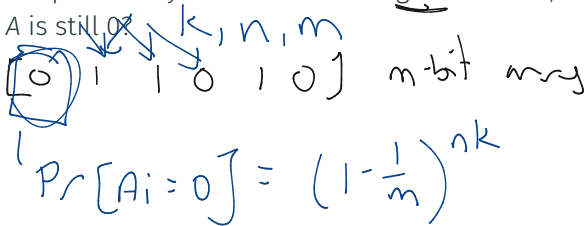
Analysis

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?

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?



The diagram shows an m -bit array represented as $[0 \ 1 \ 1 \ 0 \ 1 \ 0]$. The first bit, 0, is circled in blue. Above the array, the parameters k, n, m are written in blue. Three blue arrows point from these parameters to the first bit: one from k to the first bit, one from n to the first bit, and one from m to the first bit. To the right of the array, the text "m-bit array" is written in blue. Below the array, the probability formula is written in blue: $\Pr[A_i = 0] = \left(1 - \frac{1}{m}\right)^{nk}$.

$$\Pr[A_i = 0] = \left(1 - \frac{1}{m}\right)^{nk}$$

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 .

$$\Pr(A[i] = 0) = \Pr(h_1(x_1) \neq i \cap \dots \cap h_k(x_k) \neq i \cap h_1(x_2) \neq i \cap \dots \cap h_k(x_2) \neq i \cap \dots)$$

/ $n \cdot k$ events have to happen

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 .

$$\begin{aligned}\Pr(A[i] = 0) &= \Pr(h_1(x_1) \neq i \cap \dots \cap h_k(x_k) \neq i \\ &\quad \cap h_1(x_2) \neq i \dots \cap h_k(x_2) \neq i \cap \dots) \\ &= \underbrace{\Pr(h_1(x_1) \neq i) \times \dots \times \Pr(h_k(x_1) \neq i) \times \Pr(h_1(x_2) \neq i) \dots}_{k \cdot n \text{ events each occurring with probability } 1-1/m}\end{aligned}$$

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 .

$$\begin{aligned}\Pr(A[i] = 0) &= \Pr(h_1(x_1) \neq i \cap \dots \cap h_k(x_k) \neq i \\ &\quad \cap h_1(x_2) \neq i \cap \dots \cap h_k(x_2) \neq i \cap \dots) \\ &= \underbrace{\Pr(h_1(x_1) \neq i) \times \dots \times \Pr(h_k(x_1) \neq i) \times \Pr(h_1(x_2) \neq i) \dots}_{k \cdot n \text{ events each occurring with probability } 1 - 1/m} \\ &= \left(1 - \frac{1}{m}\right)^{kn}\end{aligned}$$

$n \uparrow$ $k \uparrow$ prob $\uparrow$
 $n \uparrow$ $k \uparrow$ prob $\downarrow$

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?

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

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

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?

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

Handwritten derivation:

$$\left(1 - \frac{1}{m}\right)^{n \cdot \frac{k}{1}} \approx \left(e^{-\frac{1}{m}}\right)^{\frac{kn}{1}} = e^{-\frac{kn}{m}}$$

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

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?

$$\Pr(\text{query}(w)=1) = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

n : total number items in filter, m : number of bits in filter, k : number of random hash functions, $h_1, \dots, h_k$: hash functions, A : bit array, δ : false positive rate.

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?

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

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

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?

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

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

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?

$$\begin{aligned}\Pr(A[\mathbf{h}_1(w)] = \dots = A[\mathbf{h}_k(w)] = 1) \\&= \Pr(A[\mathbf{h}_1(w)] = 1) \times \dots \times \Pr(A[\mathbf{h}_k(w)] = 1) \\&= \left(1 - e^{-\frac{kn}{m}}\right)^k \quad \text{Actually Incorrect!}\end{aligned}$$

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

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?

$$\begin{aligned}\Pr(A[\mathbf{h}_1(w)] = \dots = A[\mathbf{h}_k(w)] = 1) \\&= \Pr(A[\mathbf{h}_1(w)] = 1) \times \dots \times \Pr(A[\mathbf{h}_k(w)] = 1) \\&= \left(1 - e^{-\frac{kn}{m}}\right)^k \quad \text{Actually Incorrect! Dependent events.}\end{aligned}$$

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