Learning Goals

- State the purpose of Bloom filters
- Understand the tradeoff between space and accuracy in Bloom filters
- Analyze the performance of a Bloom filter

Membership Checking

- Sometimes we'd like to check very quickly very quickly whether an element is in it.

Membership Checking

- Sometimes we'd like to check very quickly very quickly whether an element is in it.
- Scenario: An ISP (Internet Service Provider), when loading a webpage, may need to check if certain media files are cached

Membership Checking

- Sometimes we'd like to check very quickly very quickly whether an element is in it.
- Scenario: An ISP (Internet Service Provider), when loading a webpage, may need to check if certain media files are cached
    - Modern Internet has multiple levels of caching

Membership Checking

- Sometimes we'd like to check very quickly very quickly whether an element is in it.
- Scenario: An ISP (Internet Service Provider), when loading a webpage, may need to check if certain media files are cached
    - Modern Internet has multiple levels of caching
    - More contents are delivered by *Content Delivery Networks* (CDNs) such as Akamai and Fastly than from original servers

Membership Checking

- Sometimes we'd like to check very quickly very quickly whether an element is in it.
- Scenario: An ISP (Internet Service Provider), when loading a webpage, may need to check if certain media files are cached
  - Modern Internet has multiple levels of caching
  - More contents are delivered by *Content Delivery Networks* (CDNs) such as Akamai and Fastly than from original servers
- We'd like to use $O(n)$ space, but we can tolerate a few errors.

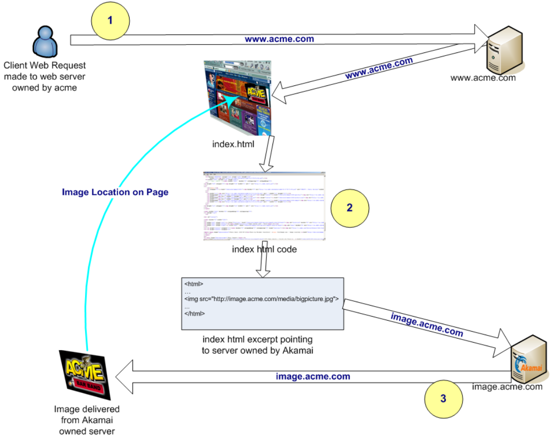
Membership Checking

- Sometimes we'd like to check very quickly very quickly whether an element is in it.
- Scenario: An ISP (Internet Service Provider), when loading a webpage, may need to check if certain media files are cached
  - Modern Internet has multiple levels of caching
  - More contents are delivered by *Content Delivery Networks* (CDNs) such as Akamai and Fastly than from original servers
- We'd like to use $O(n)$ space, but we can tolerate a few errors.
- Space requirement of hash table: even if we only store the keys, it would take $\Omega(n \log U)$ space.

Membership Checking

- Sometimes we'd like to check very quickly very quickly whether an element is in it.
- Scenario: An ISP (Internet Service Provider), when loading a webpage, may need to check if certain media files are cached
    - Modern Internet has multiple levels of caching
    - More contents are delivered by *Content Delivery Networks* (CDNs) such as Akamai and Fastly than from original servers
- We'd like to use $O(n)$ space, but we can tolerate a few errors.
- Space requirement of hash table: even if we only store the keys, it would take $\Omega(n \log U)$ space.

Illustration of Content Delivery Network (CDN)



```
<html>
...
<img src="http://image.acme.com/media/bigpicture.jpg">
...
</html>
```

Credit: Kim Meyrick — http://en.wikipedia.org/wiki/Image:Akamaiprocess.png

Bloom Filters

- The following variant of hashing is named after Burton Bloom.

Bloom Filters

- The following variant of hashing is named after Burton Bloom.
- A *Bloom filter* consists of an array $B[0, \ldots, m-1]$ bits, together with k hash functions $h_1, \cdots, h_k : U \to \{0, \ldots, m-1\}$.

Bloom Filters

- The following variant of hashing is named after Burton Bloom.
- A *Bloom filter* consists of an array $B[0, \ldots, m-1]$ bits, together with k hash functions $h_1, \cdots, h_k : U \to \{0, \ldots, m-1\}$.
- For the purpose of theoretical analysis, we assume $h_1, \cdots, h_k$ are mutually independent, ideal random functions.

Bloom Filters

- The following variant of hashing is named after Burton Bloom.
- A *Bloom filter* consists of an array $B[0, \ldots, m-1]$ bits, together with k hash functions $h_1, \cdots, h_k : U \to \{0, \ldots, m-1\}$.
- For the purpose of theoretical analysis, we assume $h_1, \cdots, h_k$ are mutually independent, ideal random functions.
  - Recall: the universal hashing function we constructed in the last lecture are not ideal random functions.

Bloom Filters

- The following variant of hashing is named after Burton Bloom.
- A *Bloom filter* consists of an array $B[0, \ldots, m-1]$ bits, together with k hash functions $h_1, \cdots, h_k : U \to \{0, \ldots, m-1\}$.
- For the purpose of theoretical analysis, we assume $h_1, \cdots, h_k$ are mutually independent, ideal random functions.
  - Recall: the universal hashing function we constructed in the last lecture are not ideal random functions.
  - In particular, for $x \neq y$ in U, and $p, q \in \{0, \ldots, m-1\}$, $\mathbf{Pr}_h[h(x) = p, h(y) = q] \neq \frac{1}{m^2}$.

Bloom Filters

- The following variant of hashing is named after Burton Bloom.
- A *Bloom filter* consists of an array $B[0, \ldots, m-1]$ bits, together with k hash functions $h_1, \cdots, h_k : U \to \{0, \ldots, m-1\}$.
- For the purpose of theoretical analysis, we assume $h_1, \cdots, h_k$ are mutually independent, ideal random functions.
  - Recall: the universal hashing function we constructed in the last lecture are not ideal random functions.
  - In particular, for $x \neq y$ in U, and $p, q \in \{0, \ldots, m-1\}$, $\mathbf{Pr}_h[h(x) = p, h(y) = q] \neq \frac{1}{m^2}$.
  - Later in the course, we'll see hash functions that guarantee pairwise independence.

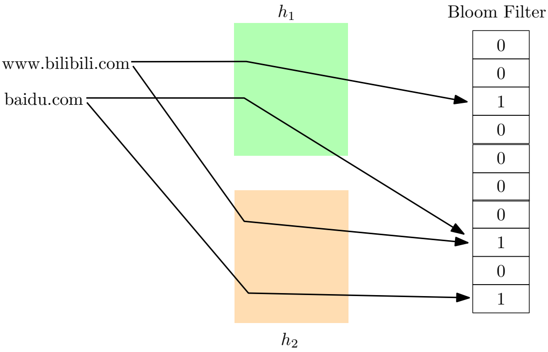
Bloom Filters

- The following variant of hashing is named after Burton Bloom.
- A *Bloom filter* consists of an array $B[0, \ldots, m-1]$ bits, together with k hash functions $h_1, \cdots, h_k : U \to \{0, \ldots, m-1\}$.
- For the purpose of theoretical analysis, we assume $h_1, \cdots, h_k$ are mutually independent, ideal random functions.
  - Recall: the universal hashing function we constructed in the last lecture are not ideal random functions.
  - In particular, for $x \neq y$ in U, and $p, q \in \{0, \ldots, m-1\}$, $\mathbf{Pr}_h[h(x) = p, h(y) = q] \neq \frac{1}{m^2}$.
  - Later in the course, we'll see hash functions that guarantee pairwise independence.
- For every entry $x \in S$, mark $B[h_1(x)] = \cdots = B[h_k(x)] = 1$.

Bloom Filters

- The following variant of hashing is named after Burton Bloom.
- A *Bloom filter* consists of an array $B[0, \dots, m-1]$ bits, together with k hash functions $h_1, \cdots, h_k : U \to \{0, \dots, m-1\}$.
- For the purpose of theoretical analysis, we assume $h_1, \cdots, h_k$ are mutually independent, ideal random functions.
  - Recall: the universal hashing function we constructed in the last lecture are not ideal random functions.
  - In particular, for $x \neq y$ in U, and $p, q \in \{0, \dots, m-1\}$, $\mathbf{Pr}_h[h(x) = p, h(y) = q] \neq \frac{1}{m^2}$.
  - Later in the course, we'll see hash functions that guarantee pairwise independence.
- For every entry $x \in S$, mark $B[h_1(x)] = \cdots = B[h_k(x)] = 1$.
- When checking the membership of a key x, return "$\textsc{Yes}$" if $B[h_1(x)] = \cdots = B[h_k(x)] = 1$; if any of these is 0, return "$\textsc{No}$".

Illustration



Analysis

- Whenever we answer NO, we are always correct.

Analysis

- Whenever we answer NO, we are always correct.
- When we answer YES, there is some chance we are wrong.

Analysis

- Whenever we answer NO, we are always correct.
- When we answer YES, there is some chance we are wrong.
    - Such an error is called a *false positive*.
- The probability that a bit in B remains 0 is $(1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$.

Analysis

- Whenever we answer NO, we are always correct.
- When we answer YES, there is some chance we are wrong.
  - Such an error is called a *false positive*.
- The probability that a bit in B remains 0 is $(1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$.
  - Denote $e^{-kn/m}$ by p, then $k = -\frac{m}{n} \ln p$.

Analysis

- Whenever we answer NO, we are always correct.
- When we answer YES, there is some chance we are wrong.
  - Such an error is called a *false positive*.
- The probability that a bit in B remains 0 is $(1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$.
  - Denote $e^{-kn/m}$ by p, then $k = -\frac{m}{n} \ln p$.
- For a key not in S, the probability of a false positive is roughly $(1 - p)^k = (1 - e^{-kn/m})^k$.

Analysis

- Whenever we answer NO, we are always correct.
- When we answer YES, there is some chance we are wrong.
  - Such an error is called a *false positive*.
- The probability that a bit in B remains 0 is $(1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$.
  - Denote $e^{-kn/m}$ by p, then $k = -\frac{m}{n} \ln p$.
- For a key not in S, the probability of a false positive is roughly $(1-p)^k = (1 - e^{-kn/m})^k$.
- Minimize this probability by minimizing its logarithm: $\ln[(1-p)^k] = k \ln(1-p) = -\frac{m}{n} \ln p \ln(1-p)$.

Analysis

- Whenever we answer $\textsc{No}$, we are always correct.
- When we answer $\textsc{Yes}$, there is some chance we are wrong.
  - Such an error is called a *false positive*.
- The probability that a bit in B remains 0 is $(1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$.
  - Denote $e^{-kn/m}$ by p, then $k = -\frac{m}{n} \ln p$.
- For a key not in S, the probability of a false positive is roughly $(1 - p)^k = (1 - e^{-kn/m})^k$.
- Minimize this probability by minimizing its logarithm: $\ln[(1 - p)^k] = k \ln(1 - p) = -\frac{m}{n} \ln p \ln(1 - p)$.
- By symmetry this is minimized at $p = \frac{1}{2}$, so $k = \ln 2 \cdot (m/n)$.

Analysis

- Whenever we answer NO, we are always correct.
- When we answer YES, there is some chance we are wrong.
  - Such an error is called a *false positive*.
- The probability that a bit in B remains 0 is $(1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$.
  - Denote $e^{-kn/m}$ by p, then $k = -\frac{m}{n} \ln p$.
- For a key not in S, the probability of a false positive is roughly $(1-p)^k = (1 - e^{-kn/m})^k$.
- Minimize this probability by minimizing its logarithm: $\ln[(1-p)^k] = k \ln(1-p) = -\frac{m}{n} \ln p \ln(1-p)$.
- By symmetry this is minimized at $p = \frac{1}{2}$, so $k = \ln 2 \cdot (m/n)$.
- The false positive rate is roughly $(1/2)^{\ln 2 \cdot (m/n)} \approx (0.61850)^{m/n}$.

Overall Performance

- If we'd like to achieve false positive rate $\delta > 0$, we should have $m = \Theta(n \log(1/\delta))$ and use $k = \lceil \log(1/\delta) \rceil$ hash functions.

Overall Performance

- If we'd like to achieve false positive rate $\delta > 0$, we should have $m = \Theta(n \log(1/\delta))$ and use $k = \lceil \log(1/\delta) \rceil$ hash functions.
- To achieve 1% false positive rate, we'll use a Bloom filter of $10n$ bits and 7 hash functions.

Overall Performance

- If we'd like to achieve false positive rate $\delta > 0$, we should have $m = \Theta(n \log(1/\delta))$ and use $k = \lceil \log(1/\delta) \rceil$ hash functions.
- To achieve 1% false positive rate, we'll use a Bloom filter of $10n$ bits and 7 hash functions.
- Using a filter of $32n$ bits (equivalent to one integer per entry) and 22 hash functions, we achieve false positive rate of about $2 \cdot 10^{-7}$.

Overall Performance

- If we'd like to achieve false positive rate $\delta > 0$, we should have $m = \Theta(n \log(1/\delta))$ and use $k = \lceil \log(1/\delta) \rceil$ hash functions.
- To achieve 1% false positive rate, we'll use a Bloom filter of $10n$ bits and 7 hash functions.
- Using a filter of $32n$ bits (equivalent to one integer per entry) and 22 hash functions, we achieve false positive rate of about $2 \cdot 10^{-7}$.
- In practice, the theoretical assumption we made works pretty well, even though the hash functions are not ideal random functions.

Overall Performance

- If we'd like to achieve false positive rate $\delta > 0$, we should have $m = \Theta(n \log(1/\delta))$ and use $k = \lceil \log(1/\delta) \rceil$ hash functions.
- To achieve 1% false positive rate, we'll use a Bloom filter of $10n$ bits and 7 hash functions.
- Using a filter of $32n$ bits (equivalent to one integer per entry) and 22 hash functions, we achieve false positive rate of about $2 \cdot 10^{-7}$.
- In practice, the theoretical assumption we made works pretty well, even though the hash functions are not ideal random functions.
- The idea of increased efficiency at the cost of some fault toleration is a recurring theme in handling with big data.

Overall Performance

- If we'd like to achieve false positive rate $\delta > 0$, we should have $m = \Theta(n \log(1/\delta))$ and use $k = \lceil \log(1/\delta) \rceil$ hash functions.
- To achieve 1% false positive rate, we'll use a Bloom filter of $10n$ bits and 7 hash functions.
- Using a filter of $32n$ bits (equivalent to one integer per entry) and 22 hash functions, we achieve false positive rate of about $2 \cdot 10^{-7}$.
- In practice, the theoretical assumption we made works pretty well, even though the hash functions are not ideal random functions.
- The idea of increased efficiency at the cost of some fault toleration is a recurring theme in handling with big data.
- This clever use of hash functions will also reappear later in the course.