CS60021: Scalable Data Mining

Streaming Algorithms

Sourangshu Bhattacharya

Data Streams

- **In many data mining situations, we do not know the entire data set in advance**

- **Stream Management** is important when the input rate is controlled **externally:**
  - Google Trends
  - Twitter or Facebook status updates

- We can think of the **data** as **infinite** and **non-stationary** (the distribution changes over time)

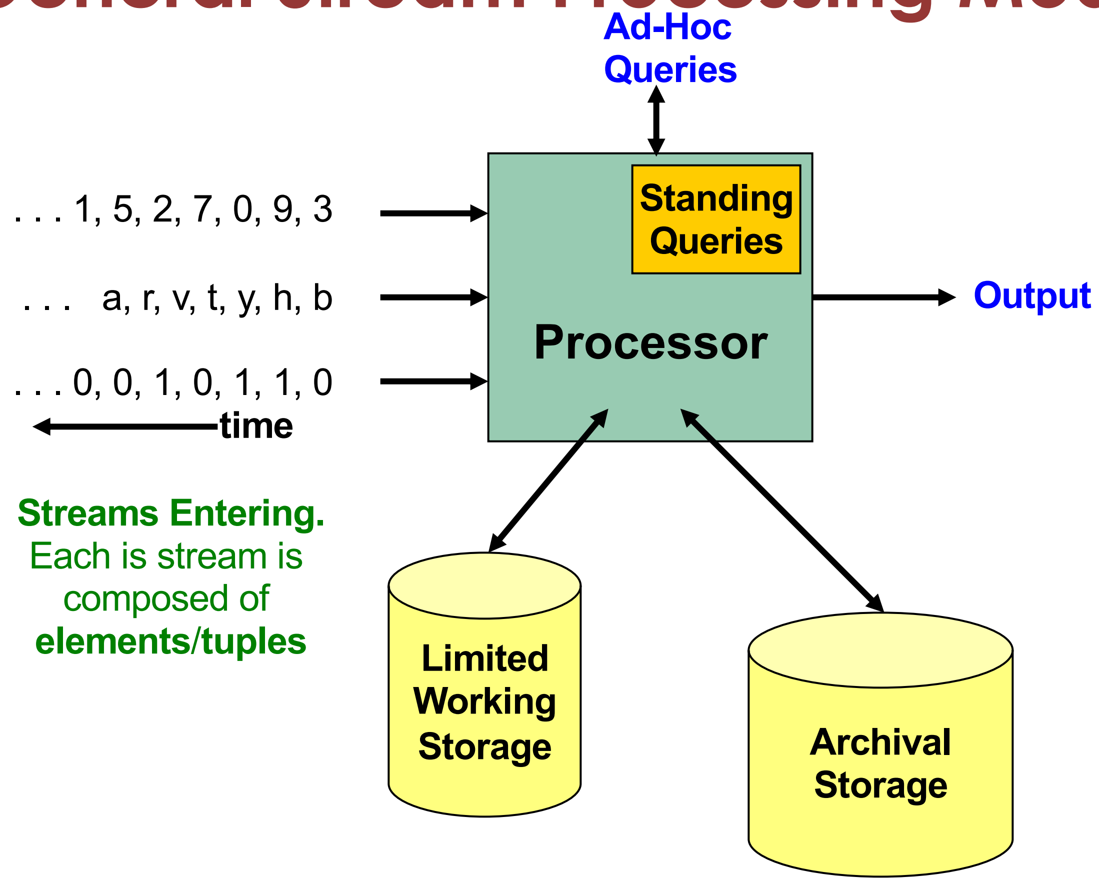
The Stream Model

- Input **elements** enter at a rapid rate,
  at one or more input ports (i.e., **streams**)
  - **We call elements of the stream tuples**

- **The system cannot store the entire stream accessibly**

- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

General Stream Processing Model



Reservoir Sampling

Maintaining a fixed-size sample

- **Problem: Fixed-size sample**
- **Suppose we need to maintain a random sample *S* of size exactly *s* tuples**
  - E.g., main memory size constraint
- **Why?** Don't know length of stream in advance
- **Suppose at time *n* we have seen *n* items**
  - **Each item is in the sample *S* with equal prob. *s/n***

How to think about the problem: say s = 2
Stream: a x c y z k c d e g…

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.
At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.
Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the n^{th} element arrives ($n > s$)
    - With probability ***s/n***, keep the n^{th} element, else discard it
    - If we picked the n^{th} element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random

- **Claim:** This algorithm maintains a sample ***S*** with the desired property:
  - After ***n*** elements, the sample contains each element seen so far with probability ***s/n***

Proof: By Induction

- **We prove this by induction:**
  - Assume that after ***n*** elements, the sample contains each element seen so far with probability ***s/n***
  - We need to show that after seeing element ***n+1*** the sample maintains the property
    - Sample contains each element seen so far with probability ***s/(n+1)***
- **Base case:**
  - After we see **n=s** elements the sample **S** has the desired property
    - Each out of **n=s** elements is in the sample with probability ***s/s = 1***

Proof: By Induction

- **Inductive hypothesis:** After ***n*** elements, the sample ***S*** contains each element seen so far with prob. ***s/n***
- **Now element *n+1* arrives**
- **Inductive step:** For elements already in ***S***, probability that the algorithm keeps it in ***S*** is:

$$\left(1-\frac{s}{n+1}\right)+\left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)=\frac{n}{n+1}$$

Element **n+1** discarded

Element **n+1** not discarded

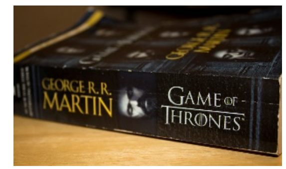
Element in the sample not picked

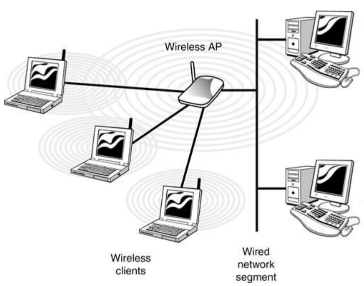
- So, at time ***n,*** tuples in ***S*** were there with prob. **s/n**
- Time ***n→n+1,*** tuple stayed in ***S*** with prob. **n/(n+1)**
- So prob. tuple is in ***S*** at time ***n+1*** $= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

Bloom Filters

Querying

ISBN present in collection?





IP seen by switch?

10.0.21.102

Solutions

- Universe U, but need to store a set of n items, $n \ll |U|$
- Hash table of size m:
  - Space $O(n \log |U|)$
  - Query time $O\left(\frac{n}{m}\right)$

Exact Solutions

- Universe U, but need to store a set of n items, $n \ll |U|$
- Hash table of size m:
  - Space $O(n \log |U|)$
  - Query time $O\left(\frac{n}{m}\right)$
- Bit array of size $|U|$
  - Space = $|U|$
  - Query time $O(1)$

Querying, Monte Carlo style

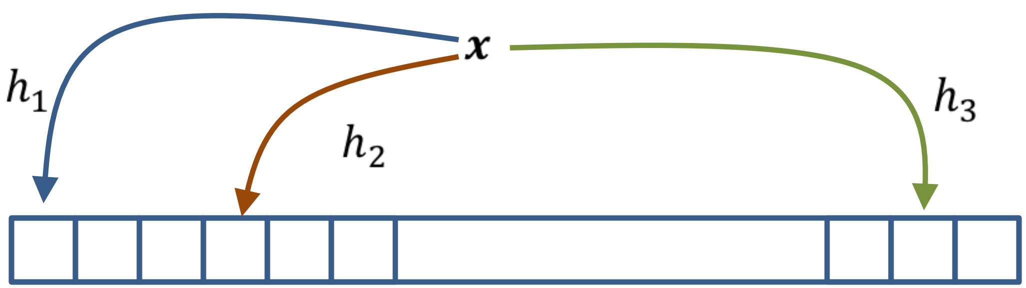
- In hash table construction, we used random hash functions
  - we never return incorrect answer
  - query time is a random variable
  - These are Las Vegas algorithms

- In Monte-Carlo randomized algorithms, we are allowed to return incorrect answers with (small) probability, say, δ

Bloom filter

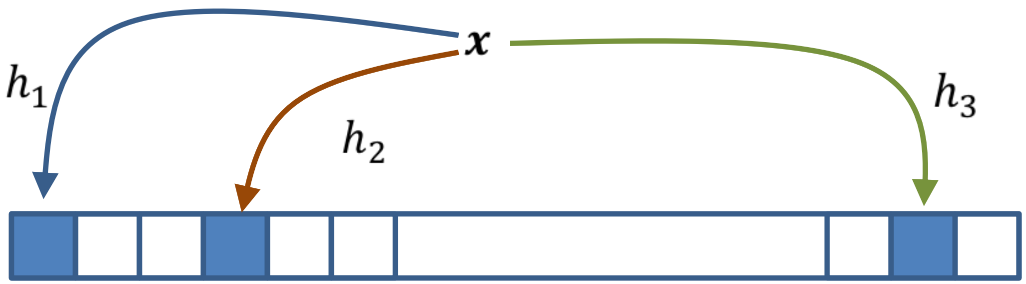
[Bloom, 1970]

- A bit-array B, $|B| = m$
- k hash functions, $h_1, h_2, \dots, h_k$, each $h_i \in U \to [m]$



Bloom filter

- A bit-array B, $|B| = m$
- k hash functions, $h_1, h_2, \ldots, h_k$, each $h_i \in U \to [m]$



Operations

- $Initialize(B)$
  - for $i \in \{1, .. m\}$, $B[i] = 0$
- $Insert\ (B, x)$
  - for $i \in \{1, .. k\}$, $B[h_i(x)] = 1$
- $Lookup\ (B, x)$
  - If $\bigwedge_{i \in \{1,\ldots k\}} B[h_i(x)]$, return PRESENT, else ABSENT

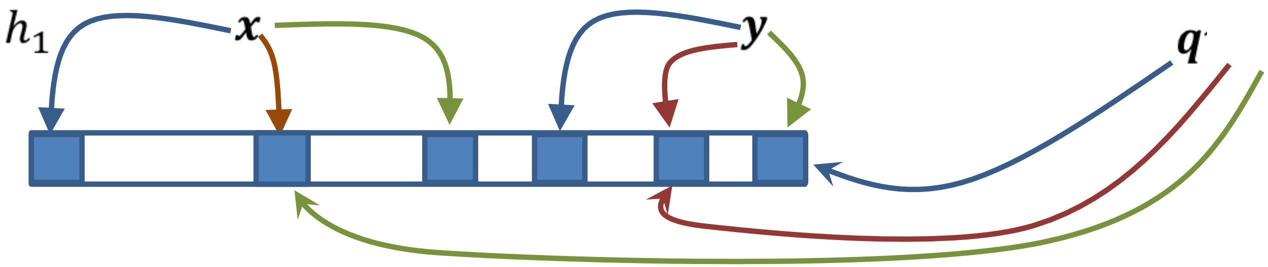
Bloom Filter

- If the element x has been added to the Bloom filter, then $Lookup(B, x)$ always return PRESENT

Bloom Filter

- If the element x has been added to the Bloom filter, then $Lookup(B, x)$ always return PRESENT

- If x has not been added to the filter before?
  - $Lookup$ sometimes still return PRESENT



Designing Bloom Filter

- Want to minimize the probability that we return a false positive
- Parameters $m = |B|$ and $k =$ number of hash functions
- $k = 1 \;\Rightarrow$ normal bit-array

- What is effect of changing k?

Effect of number of hash functions

- Increasing k
  - Possibly makes it harder for false positives to happen in $Lookup$ because of $\Lambda_{i \in \{1,\ldots k\}} B[h_i(x)]$
  - But also increases the number of filled up positions
- We can analyse to find out an "optimal k"

False positive analysis

- $m = |B|$, n elements inserted
- If x has not been inserted, what is the probability that $Lookup(B, x)$ returns PRESENT?

False positive analysis

- $m = \ |B|, \ n$ elements inserted
- If x has not been inserted, what is the probability that $Lookup(B, x)$ returns PRESENT?
- Assume $\{h_1, h_2, \dots h_k\}$ are independent and $\Pr[h_i(\cdot) = j] = \frac{1}{m}$ for all positions j

False positive analysis

- Probability of a bit being zero:

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

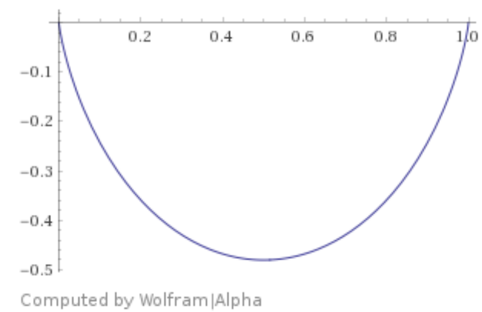
- The expected number of zero bits is given by: $me^{-kn/m}$.
- $P[lookup(B, x) = PRESENT] = \left(1 - e^{-\frac{kn}{m}}\right)^k$
- We can choose k to minimize this probability.

Choosing number of hash functions

- $p = e^{-kn/m}$
- Log (False Positive) =

$$\log(1 - p)^k = k\log(1-p) = -\frac{m}{n}\log(p)\log(1-p)$$

Minimized at $p = \frac{1}{2}$, i.e. $k = \mathrm{m}\log(2)/\mathrm{n}$



Bloom filter design

- This "optimal" choice gives false positive = $2^{-m\log(2)/n}$

- If we want a false positive rate of δ , set $m = \left\lceil \frac{\log\left(\frac{1}{\delta}\right) n}{\log^2(2)} \right\rceil$

Example: If we want 1% FPR, we need 7 hash functions and total $10n$ bits

Applications

- Widespread applications whenever small false positives are tolerable

- Used by browsers
  - to decide whether an URL is potentially malicious: a BF is used in browser, and positives are actually checked with the server.

- Databases e.g. BigTable, HBase, Cassandra, Postgrepsql use BF to avoid disk lookups for non-existent rows/columns

- Bitcoin for wallet synchronization….

Handling deletions

- Chief drawback is that BF does not allow deletions

[Fan et al 00]

- Counting Bloom Filter
  - Every entry in BF is a small counter rather than a single bit
  - $Insert(x)$ increments all counters for $\{h_i(x)\}$ by 1
  - $Delete(x)$ decrements all $\{h_i(x)\}$ by 1
  - maintains 4 bits per counter
  - False negatives can happen, but only with low probability

References:

- Mining massive Datasets by Leskovec, Rajaraman, Ullman, Chapter 4.

- Primary reference for this lecture
    - Survey on Bloom Filter, Broder and Mitzenmacher 2005, https://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf
    - http://www.firatatagun.com/blog/2016/09/25/bloom-filters-explanation-use-cases-and-examples/

- Others
    - Randomized Algorithms by Mitzenmacher and Upfal.