More Stream Mining

Bloom Filters
Sampling Streams
Counting Distinct Items
Computing Moments

Jeffrey D. Ullman
Stanford University

Filtering Stream Content

- To motivate the Bloom-filter idea, consider a web crawler.
- It keeps, centrally, a list of all the URL's it has found so far.
- It assigns these URL's to any of a number of parallel tasks; these tasks stream back the URL's they find in the links they discover on a page.
- It needs to filter out those URL's it has seen before.

Role of the Bloom Filter

- A Bloom filter placed on the stream of URL's will declare that certain URL's have been seen before.
- Others will be declared new, and will be added to the list of URL's that need to be crawled.
- Unfortunately, the Bloom filter can have false positives.
  - It can declare a URL seen before when it hasn't.
- But if it says "never seen," then it is truly new.
- So we need to restart the filter periodically.

Example: Filtering Chunks

- Suppose we have a database relation stored in a DFS, spread over many chunks.
- We want to find a particular value v, looking at as few chunks as possible.
- A Bloom filter on each chunk will tell us certain values are there, and others aren't.
  - As before, false positives are possible.
- But now things are exactly right: if the filter says v is not at the chunk, it surely isn't.
  - Occasionally, we retrieve a chunk we don't need, but can't miss an occurrence of value v.

How a Bloom Filter Works

- A *Bloom filter* is an array of bits, together with a number of hash functions.
- The argument of each hash function is a stream element, and it returns a position in the array.
- Initially, all bits are 0.
- When input x arrives, we set to 1 the bits h(x), for each hash function h.

Example: Bloom Filter

- Use N = 11 bits for our filter.
- Stream elements = integers.
- Use two hash functions:
  - $h_1(x)$ =
    - Take odd-numbered bits from the right in the binary representation of x.
    - Treat it as an integer i.
    - Result is i modulo 11.
  - $h_2(x)$ = same, but take even-numbered bits.

Example – Continued

Stream element	h_1	h_2	Filter contents
			00000000000
25 = 11001	5	2	00100100000
159 = 10011111	7	0	10100101000
585 = 1001001001	9	7	10100101010

Note: bit 7 was already 1.

Bloom Filter Lookup

- Suppose element y appears in the stream, and we want to know if we have seen y before.
- Compute h(y) for each hash function y.
- If all the resulting bit positions are 1, say we have seen y before.
  - We could be wrong.
    - Different inputs could have set each of these bits.
- If at least one of these positions is 0, say we have not seen y before.
  - We are certainly right.

Example: Lookup

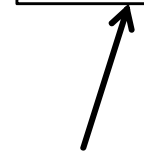
- Suppose we have the same Bloom filter as before, and we have set the filter to 10100101010.
- Lookup element y = 118 = 1110110 (binary).
- $h_1(y)$ = 14 modulo 11 = 3.
- $h_2(y)$ = 5 modulo 11 = 5.
- Bit 5 is 1, but bit 3 is 0, so we are sure y is not in the set.

Performance of Bloom Filters

- Probability of a false positive depends on the density of 1's in the array and the number of hash functions.
  - = (fraction of 1's)$^{\text{\# of hash functions}}$.
- The number of 1's is approximately the number of elements inserted times the number of hash functions.
  - But collisions lower that number slightly.

Throwing Darts

- Turning random bits from 0 to 1 is like throwing *d* darts at *t* targets, at random.
- How many targets are hit by at least one dart?
- Probability a given target is hit by a given dart = 1/t.
- Probability none of d darts hit a given target is $(1-1/t)^d$.
- Rewrite as $(\boxed{1-1/t})^{t(d/t)} \sim= e^{-d/t}$.



$\sim= 1/e$

Example: Throwing Darts

- Suppose we use an array of 1 billion bits, 5 hash functions, and we insert 100 million elements.
- That is, $t = 10^9$, and $d = 5*10^8$.
- The fraction of 0's that remain will be $e^{-1/2} = 0.607$.
- Density of 1's = 0.393.
- Probability of a false positive = $(0.393)^5 = 0.00937$.

Sampling a Stream

What Doesn't Work
Sampling Based on Hash Values

Sampling

- Unbiased
- Sampling process preserves the elements to be answered for a query

When Sampling Doesn't Work

- Suppose Google would like to examine its stream of search queries for the past month to find out what fraction of them were unique – asked only once.
- But to save time, we are only going to sample $1/10^{th}$ of the stream.
- The fraction of unique queries in the sample != the fraction for the stream as a whole.
  - In fact, we can't even adjust the sample's fraction to give the correct answer.

Example: Unique Search Queries

- The length of the sample is 10% of the length of the whole stream.
- Suppose a query is unique.
  - It has a 10% chance of being in the sample.
- Suppose a query occurs exactly twice in the stream.
  - It has an 18% chance of appearing exactly once in the sample.
- And so on … The fraction of unique queries in the stream is unpredictably large.

Sampling by Value

- My mistake: I sampled based on the position in the stream, rather than the value of the stream element.
- The right way: hash search queries to 10 buckets 0, 1,…, 9.
- Sample = all search queries that hash to bucket 0.
  - All or none of the instances of a query are selected.
  - Therefore the fraction of unique queries in the sample is the same as for the stream as a whole.

Controlling the Sample Size

- Problem: What if the total sample size is limited?
- Solution: Hash to a large number of buckets.
- Adjust the set of buckets accepted for the sample, so your sample size stays within bounds.

Example: Fixed Sample Size

- Suppose we start our search-query sample at 10%, but we want to limit the size.
- Hash to (say) 100 buckets, 0, 1,…, 99.
  - Take for the sample those elements hashing to buckets 0 through 9.
- If the sample gets too big, get rid of bucket 9.
- Still too big, get rid of 8, and so on.

Sampling Key-Value Pairs

- This technique generalizes to any form of data that we can see as tuples (K, V), where K is the "key" and V is a "value."
- Distinction: We want our sample to be based on picking some set of keys only, not pairs.
  - In the search-query example, the data was "all key."
- Hash keys to some number of buckets.
- Sample consists of all key-value pairs with a key that goes into one of the selected buckets.

Example: Salary Ranges

- Data = tuples of the form (EmpID, Dept, Salary).
- Query: What is the average range of salaries within departments?
- Key = Dept.
- Value = (EmpID, Salary).
- Sample picks some departments, has salaries for all employees of that department, including its min and max salaries.
- Result will be an unbiased estimate of the average salary range.

Counting Distinct Elements

Applications
Flajolet-Martin Approximation Technique
Generalization to Moments

Counting Distinct Elements

- Example:
  - Counting unique users in a website for a particular day – Amazon – login
  - Google – identify users by IP address

- Problem: a data stream consists of elements chosen from a set of size *n*. Maintain a count of the number of distinct elements seen so far.

- Obvious approach: maintain the set of elements seen in a hash table. (Size ????)

Applications

- How many different words are found among the Web pages being crawled at a site?
  - Unusually low or high numbers could indicate artificial pages (spam?).
- How many unique users visited Facebook this month?
- How many different pages link to each of the pages we have crawled.
  - Useful for estimating the PageRank of these pages.
    - Which in turn can tell a crawler which pages are most worth visiting.

Estimating Counts

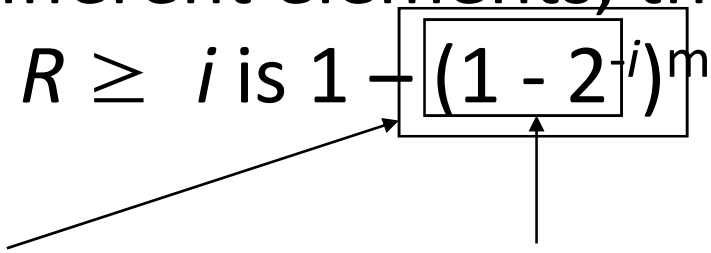
- Real Problem: what if we do not have space to store the complete set?
  - Or we are trying to count lots of sets at the same time.
- Estimate the count in an unbiased way.
- Accept that the count may be in error, but limit the probability that the error is large.
- Solution:
  - More machines and parallel processing
  - Flajolet – Martin algorithm – estimate the number of distinct elements with less memory than the number of distinct elements

Flajolet-Martin Approach

- Hash the elements of universal set to a bit string that is sufficiently long.
- F-M says more different elements we see in stream, the more different hash values we shall see.
- More different hash value then one of these values will be unusual. (value ends in many 0's)
- Pick a hash function h that maps each of the n elements to at least $\log_2 n$ bits.
- For each stream element 'a', let $r(a)$ be the number of trailing 0's in $h(a)$.
  - Called the *tail length*.
- Record R = the maximum $r(a)$ seen for any 'a' in the stream.
- Estimate of number of distinct elements (based on this hash function) = 2^R.

Why It Works

- The probability that a given $h(a)$ ends in at least i 0's is 2^{-i}.
- If there are m different elements, the probability that $R \geq i$ is $1 - (1 - 2^{-i})^m$.



Why It Works – (2)

- Since 2^{-i} is small, $1 - (1-2^{-i})^m \approx 1 - e^{-m2^{-i}}$.
- If $2^i >> m$, $1 - e^{-m2^{-i}} \approx 1 - (1 - m2^{-i}) \approx m/2^i \approx 0$.
- If $2^i << m$, $1 - e^{-m2^{-i}} \approx 1$.
- Thus, 2^R will almost always be around m.

First 2 terms of the Taylor expansion of e^x

Same trick as "throwing darts." Multiply and divide m by 2^{-i}. Prob of not finding a stream Element with as many as i '0' s at the end of its hash value

Why It Doesn't Work

- E(2^R) is, in principle, infinite.
  - Probability halves when R -> R+1, but value doubles.
- Workaround involves using many hash functions and getting many samples.
- How are samples combined?
  - Average? What if one very large value?
  - Median? All values are a power of 2.

Solution

- Partition your samples into small groups.
  - O(log n), where n = size of universal set, suffices.
- Take the average within each group.
- Then take the median of the averages.

- Exercise 4.4.1 : Suppose our stream consists of the integers 3, 1, 4, 1, 5, 9, 2, 6, 5. Our hash functions will all be of the form h(x) = ax+b mod 32 for some a and b. You should treat the result as a 5-bit binary integer. Determine the tail length for each stream element and the resulting estimate of the number of distinct elements if the hash function is:
- (a) h(x) = 2x + 1 mod 32.
- (b) h(x) = 3x + 7 mod 32.
- (c) h(x) = 4x mod 32.

Element	Hashed value	Binary	r	2^r	R

R=?

Number of Distinct element 2^R =?

- (a) h(x) = 2x + 1 mod 32. --- 1
- (b) h(x) = 3x + 7 mod 32. --- 4
- (c) h(x) = 4x mod 32. --- 4

- Average of 2^r =1, 3.22,6.22
- Median of Average = 3.22

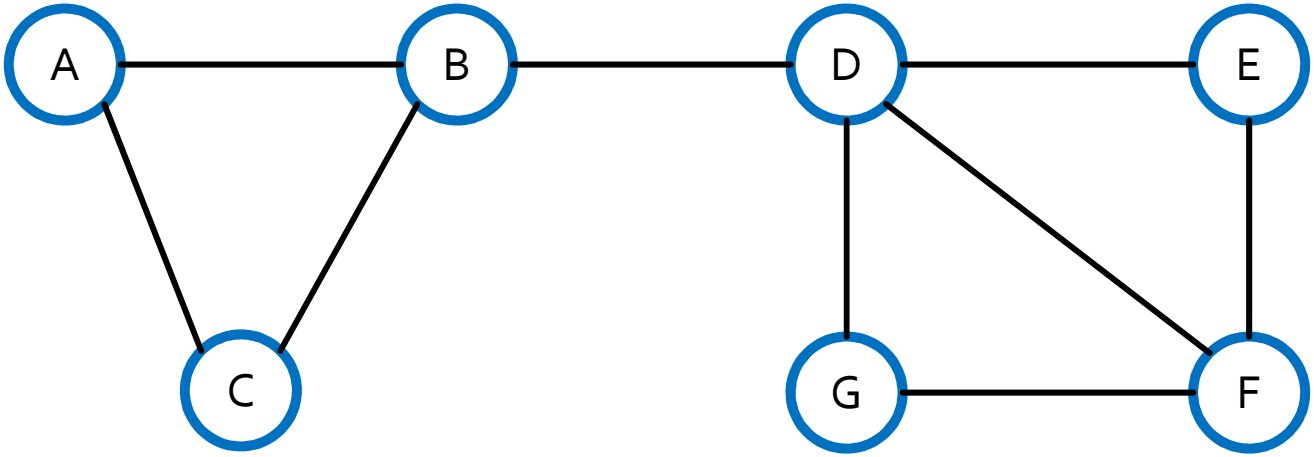
Application:Neighborhoods

Neighborhood of Distance d
Recursive Algorithm for Neighborhoods
Approximate Neighborhood Count

Neighbors and Neighborhoods

- If there is an edge between nodes u and v, then u is a *neighbor* of v and vice-versa.
- The *neighborhood* of node u at distance d is the set of all nodes v such that there is a path of length at most d from u to v.
  - Denoted n(u,d).
- Notice that if there are N nodes in a graph, then n(u,N-1) = n(u,N) = n(u,N+1) = … = all nodes reachable from u.

Example: Neighborhoods



n(E,0) = {E}; n(E,1) = {D,E,F}; n(E,2) = {B,D,E,F,G};
n(E,3) = {A,B,C,D,E,F,G}.

Why Neighborhoods?

- The sizes of neighborhoods of small distance measure the “influence” a person has in a social network.
  - Note it is the size of the neighborhood, not the exact members of the neighborhood that is important here.

Algorithm for Finding Neighborhoods

- n(u,0) = {u} for every u.
- n(u,d) is the union of n(v, d-1) taken over every neighbor v of u.
- Not really feasible for large graphs, since the neighborhoods get large, and taking the union requires examining the neighborhood of each neighbor.
  - To eliminate duplicates.
- Note: Another approach where we take the union of neighbors of members of n(u, d-1) presents similar problems.

Approximate Algorithm for Neighborhood Sizes

- The idea behind Flajolet-Martin lets you estimate the number of distinct elements in the union of several sets.
- Pick several hash functions; let h be one.
- For each node u and distance d compute the maximum tail length among all nodes in n(u,d), using hash function h.

Approximate Algorithm – (2)

- Remember: if R is the maximum tail length in a set of values, then 2^R is a good estimate of the number of distinct elements in the set.
- Since n(u,d) is the union of all neighbors v of u of n(v,d-1), the maximum tail length of members of n(u,d) is the largest of
  1. The tail length of h(u), and
  2. The maximum tail length for all the members of n(v,d-1) for any neighbor v of u.

Approximate Algorithm – (3)

- Thus, we have a recurrence (on d) for the maximum tail length of any neighbor of any node u, using any given hash function h.
- Repeat for some chosen number of hash functions.
- Combine estimates to get an estimate of neighborhood sizes, as for the Flajolet-Martin algorithm.

Moments

Surprise Numbers
AMS Algorithm

Generalization: Moments

- Generalization of problem of counting distinct elements in a stream – computing moments.
- Suppose a stream has elements chosen from a set of n values.
- Let m_i be the number of times value i occurs.
- The k^{th} *moment* is the sum of $(m_i)^k$ over all i.

Special Cases

- 0^{th} moment = number of different elements in the stream.
  - The problem just considered.
- 1^{st} moment = sum of counts of the numbers of elements = length of the stream.
  - Easy to compute.
- 2^{nd} moment = *surprise number* = a measure of how uneven the distribution is.

Example: Surprise Number

- Stream of length 100; 11 values appear.
- Unsurprising: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9. Surprise # = 910.
- Surprising: 90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1. Surprise # = 8,110.
- Naïve approach : keep a counter for each unique element in main memory
- Approximate approach: Keep a limited number of values in main memory and compute estimates from these values

AMS Method

- Alon-Matias-Szegedy
- Works for all moments; gives an unbiased estimate.
- We'll talk about only the 2nd moment.
- Based on calculation of many random variables *X*.
  - Each requires a count in main memory, so number is limited.

One Random Variable

- Assume stream has length *n*.
- Pick a random time to start, so that any time is equally likely.
- Let the chosen time have element *a* in the stream.
- *X* = *n* * ((twice the number of *a*'s in the stream starting at the chosen time) – 1).
  - Note: store *n* once, store count of *a*'s for each *X*.

Example

- Suppose the stream is a, b, c, b, d, a, c, d, a, b, d, c, a, a, b. The length of the stream is n = 15. Since a appears 5 times, b appears 4 times, and c and d appear three times each
- the second moment for the stream is 52+42+32+32 = 59. Suppose we keep three variables, X1, X2, and X3. Also, assume that at "random" we pick the 3rd, 8th, and 13th positions to define these three variables.

Example

- When we reach position 3, we find element c, so we set X1.element = c and X1.value = 1. Position 4 holds b, so we do not change X1. Likewise, nothing happens at positions 5 or 6. At position 7, we see c again, so we set X1.value = 2.
- At position 8 we find d, and so set X2.element = d and X2.value = 1. Positions 9 and 10 hold a and b, so they do not affect X1 or X2. Position 11 holds d so we set X2.value = 2, and position 12 holds c so we set X1.value = 3.
- At position 13, we find element a, and so set X3.element = a and X3.value = 1. Then, at position 14 we see another a and so set X3.value = 2. Position 15, with element b does not affect any of the variables
- X1.value = 3 and X2.value = X3.value = 2.

Solution

- Estimate (X) = n × (2 × X.value − 1).
- From X1 we derive the estimate n × (2 × X1.value − 1) = 15 × (2 × 3 − 1) = 75. The other two variables, X2 and X3, each have value 2 at the end, so their estimates are 15×(2 ×2−1) = 45.
- average of the three estimates is 55
- Actual estimate is 59

Working

- For each variable X, we store:
- 1. A particular element of the universal set, which we refer to as X.element , and
- 2. An integer X.value, which is the value of the variable. To determine the value of a variable X, we choose a position in the stream between 1 and n, uniformly and at random. Set X.element to be the element found there, and initialize X.value to 1. As we read the stream, add 1 to X.value each time we encounter another occurrence of X.element .

Higher order moments

- Second moment =n×(2v−1)
- 2v − 1 is the difference between v^2 and $(v - 1)^2$
- third moment = replace 2v−1 by $v^3-(v-1)^3$ = $3v^2-3v+1$. Then sum $_{(\ v=1\ \text{to}\ m)}$ $3v^2-3v+1 = m^3$.
- k^{th} moments for any $k \geq 2$ by turning value v = X.value into $n \times (v^k - (v - 1)^k)$.

Problem: Streams Never End

- We assumed there was a number *n*, the number of positions in the stream.
- But real streams go on forever, so *n* changes; it is the number of inputs seen so far.

Fixups

1. The variables *X* have *n* as a factor – keep *n* separately; just hold the count in *X*.
2. Suppose we can only store *k* counts. We cannot have one random variable X for each start-time, and must throw out some start-times as we read the stream.
   - Objective: each starting time *t* is selected with probability *k*/*n*.

Solution to (2)

- Choose the first k times for k variables.
- When the n^{th} element arrives ($n > k$), choose it with probability k/n.
- If you choose it, throw one of the previously stored variables out, with equal probability.
- Probability of each of the first n-1 positions being chosen:

$$(n-k)/n * k/(n-1) + k/n * k/(n-1) * (k-1)/k = k/n$$

n-th position not chosen

Previously chosen

n-th position chosen

Previously chosen

Survives

Final Remarks

- Thus, each variable has the second moment as its expected value.
- Use many (e.g., 100) such variables.
- Combine them as for Flajolet-Martin: average of groups of size O(log n), and then take the median of the averages.