e-PGPathshala
Subject : Computer Science
Paper: Data Analytics
Module No 19: CS/DA/19 - Data Analytics - Mining Streams - II
Quadrant 1 – e-text

1.1 Introduction

A common process on streams is selection or filtering. This chapter gives an overview on algorithms for streams, filtering and counting of streams.

1.2 Learning Outcomes

- Learn the algorithms for streams
- Understand the filtering and counting streams
- Know basics of filtering and counting distinct element algorithms for stream

1.3 Algorithms for streams

In computer science, **streaming algorithms** are algorithms for processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes (typically just one). These algorithms have limited memory available to them (much less than the input size) and also limited processing time per item.

Streaming algorithms have several applications in networking such as monitoring network links for elephant flows, counting the number of distinct flows, estimating the distribution of flow sizes, and so on. They also have applications in databases, such as estimating the size of a join. Few algorithms for streams are:

1. **Filtering a data stream:** Bloom filters is a famous and commonly used filtering algorithm for streams where elements with property x are selected from stream and others are ignored.
2. **Counting distinct elements: Flajolet-Martin** which counts the number of distinct elements in the last k elements of the stream.
3. **Estimating moments: AMS method to e**stimate std. dev. of last k elements
4. **Counting frequent items**

1.4 Filtering Data Streams

Due to the nature of data streams, stream filtering is one of the most useful and practical approaches to efficient stream evaluation, whether it is done implicitly by the system to guarantee the stability of the stream processing under overload conditions, or explicitly by the evaluating procedure. In this section we will review some of the filtering techniques commonly used in data stream processing.

A common process on streams is selection or filtering. We want to accept those tuples in the stream that meet a criterion. Accepted tuples are passed to another process as a stream, while other tuples are dropped. If the selection criterion is a property of the tuple that can be calculated (e.g., the first component is less than 10), then the selection is easy to do. The problem becomes harder when the criterion involves lookup for membership in a set. It is especially hard, when that set is too large to store in main memory. In this section, we shall discuss the technique known as “Bloom filtering” as a way to eliminate most of the tuples that do not meet the criterion.

1.4.1 Applications

Email spam filtering is one best example. We know 1 billion “good” email addresses and if an email comes from one of these, it is **NOT** spam.

Publish-subscribe systems where lots of messages (news articles) are collected and people express interest in certain sets of keywords and determine whether each message matches user’s interest.

Solution:

1) **Given a set of keys *S* that we want to filter**
2) Create a **bit array *B*** of ***n*** bits, initially all ***0*s**
3) Choose a **hash function *h*** with range **[*0,n*)**
4) Hash each member of $s \in S$ to one of ***n*** buckets, and set that bit to **1**, i.e., ***B[h(s)]=1***
5) Hash each element ***a*** of the stream and output only those that hash to bit that was set to **1**
   - **Output *a* if B[h(a)] == 1**
   - **Output the item since it may be in *S*.**

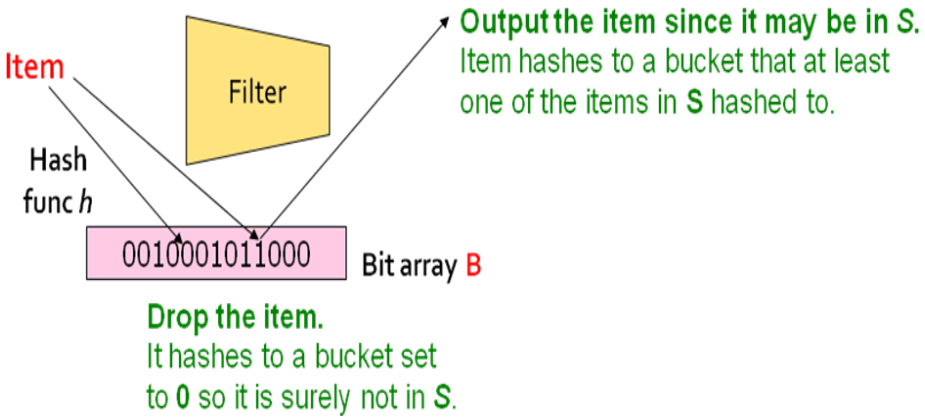

Figure 1. Filtering Stream

Blooms filter creates false positives but no false negatives, i.e., if the item is in ***S*** we surely output it, if not we may still output it.

Suppose |S| = 1 billion email addresses, |B|= 1GB = 8 billion bits, then,If the email address is in ***S***, then it surely hashes to a bucket that has the big set to **1**, so it always gets through (***no false negatives***). Approximately **1/8** of the bits are set to **1**, so about **1/8th** of the addresses not in ***S*** get through to the output (***false positives***). Actually, less than **1/8th**, because more than one address might hash to the same bit

False positives and false negatives are concepts analogous to type I and type II errors in statistical hypothesis testing, where a positive result corresponds to rejecting the null hypothesis, and a negative result corresponds to not rejecting the null hypothesis. The terms are often used interchangeably, but there are differences in detail and interpretation.

1.4.2 The Bloom Filter

A Bloom filter consists of:

1. An array of n bits, initially all 0's.

2. A collection of hash functions h_1, h_2, . . . , h_k. Each hash function maps "key" values to n buckets, corresponding to the n bits of the bit-array.

3. A set S of m key values.

The purpose of the Bloom filter is to allow through all stream elements whose keys are in S, while rejecting most of the stream elements whose keys are not in S. To initialize the bit array, begin with all bits 0. Take each key value in S and hash it using each of the k hash functions. Set to 1 each bit that is $h_i(K)$ for some hash function h_i and some key value K in S. To test a key K that arrives in the stream,

check that all of $h_1(K)$, $h_2(K)$, . . . , $h_k(K)$ are 1's in the bit-array. If all are 1's, then let the stream element through. If one or more of these bits are 0, then K could not be in S, so reject the stream element.

1.4.3 Analysis of Bloom Filtering

If a key value is in S, then the element will surely pass through the Bloom filter. However, if the key value is not in S, it might still pass. We need to understand how to calculate the probability of a false positive, as a function of n, the bit-array length, m the number of members of S, and k, the number of hash functions. The model to use is throwing darts at targets. Suppose we have x targets and y darts. Any dart is equally likely to hit any target. After throwing the darts, how many targets can we expect to be hit at least once? The analysis goes as follows:

The probability that a given dart will not hit a given target is (x − 1)/x. The probability that none of the y darts will hit a given target is $((x-1)/x)^y$. We can write this expression as $(1-1/x)^{x(y/x)}$. Using the approximation $(1-€)^{1/€}$ = 1/e for small €,we conclude that the probability that none of the y darts hit a given target is $e^{-y/x}$.

In general, Bloom filters guarantee no false negatives, and use limited memory and hence it is great for pre-processing before more expensive checks. It is suitable for hardware implementation by using hash function, computations can be infact parallelized. Is it better to have 1 big B or *k* small Bs which is the same as $(1 – e^{-km/n})^k$ vs. $(1 – e^{-m/(n/k)})^k$. But keeping 1 big B is simpler.

1.5 Counting Distinct Elements

The count-distinct problem is the problem of finding the number of distinct elements in a data stream with repeated elements. This is a well-known problem with numerous applications

1.5.1 The Count-Distinct Problem

Suppose stream elements are chosen from some universal set. We would like to know how many different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past.

Consider a Web site gathering statistics on how many unique users it has seen in each given month. The universal set is the set of logins for that site, and a stream element is generated each time someone logs in. This measure is appropriate for a site like Amazon, where the typical user logs in with their unique login name. A similar problem is a Web site like Google that does not require login to issue a search query, and may be able to identify users only by the IP address from which they send the query. There are about 4 billion IP addresses, sequences of four 8-bit

bytes will serve as the universal set in this case. The obvious way to solve the problem is to keep in main memory a list of all the elements seen so far in the stream. Keep them in an efficient search structure such as a hash table or search tree, so one can quickly add new elements and check whether or not the element that just arrived on the stream was already seen. As long as the number of distinct elements is not too great, this structure can fit in main memory and there is little problem obtaining an exact answer to the question how many distinct elements appear in the stream.

However, if the number of distinct elements is too great, or if there are too many streams that need to be processed at once (e.g., Yahoo! wants to count the number of unique users viewing each of its pages in a month), then we cannot store the needed data in main memory. There are several options. We could use more machines, each machine handling only one or several of the streams. We could store most of the data structure in secondary memory and batch stream elements so whenever we brought a disk block to main memory there would be many tests and updates to be performed on the data in that block. Or we could use the strategy to be discussed in this section, where we only estimate the number of distinct elements but use much less memory than the number of distinct elements.

1.5.2 The Flajolet-Martin Algorithm

It is possible to estimate the number of distinct elements by hashing the elements of the universal set to a bit-string that is sufficiently long. The length of the bit-string must be sufficient that there are more possible results of the hash function than there are elements of the universal set. For example, 64 bits is sufficient to hash URL's. We shall pick many different hash functions and hash each element of the stream using these hash functions. The important property of a hash function is that when applied to the same element, it always produces the same result. This property was also essential for the sampling technique.

The idea behind the Flajolet-Martin Algorithm is that the more different elements we see in the stream, the more different hash-values we shall see. As we see more different hash-values, it becomes more likely that one of these values will be "unusual." The particular unusual property we shall exploit is that the value ends in many 0's, although many other options exist. Whenever we apply a hash function h to a stream element **a**, the bit string h(a) will end in some number of 0's, possibly none. Call this number the tail length for a and h. Let R be the maximum tail length of any **a** seen so far in the stream. Then we shall use estimate 2R for the number of distinct elements seen in the stream.

This estimate makes intuitive sense. The probability that a given stream element a has **h(a)** ending in at least r 0's is 2^{-r}. Suppose there are **m** distinct elements in the

stream. Then the probability that none of them has tail length at least **r** is $(1 - 2^{-r})^m$. This sort of expression should be familiar by now. We can rewrite it as $((1 - 2^{-r})^{2r})^{m2^{-r}}$. Assuming **r** is reasonably large, the inner expression is of the form $(1 - \epsilon)^{1/\epsilon}$, which is approximately $1/\epsilon$. Thus, the probability of not finding a stream element with as many as r 0's at the end of its hash value is $e^{-m2^{-r}}$. We can conclude:

1. If m is much larger than 2^r, then the probability that we shall find a tail of length at least r approaches 1.
2. If m is much less than 2^r, then the probability of finding a tail length at least r approaches 0.

We conclude from these two points that the proposed estimate of m, which is 2^R (recall R is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

1.6 Estimating Moments

In this section we consider a generalization of the problem of counting distinct elements in a stream. The problem, called computing "moments," involves the distribution of frequencies of different elements in the stream. We shall define moments of all orders and concentrate on computing second moments, from which the general algorithm for all moments is a simple extension.

1.6.1 Definition of Moments

Suppose a stream consists of elements chosen from a universal set. Assume the universal set is ordered so we can speak of the i^{th} element for any i. Let mi be the number of occurrences of the i^{th} element for any i. Then the k^{th}-order moment (or just k^{th} moment) of the stream is the sum over all i of $(m_i)^k$.

Example: The 0th moment is the sum of 1 for each mi that is greater than 0. That is, the 0th moment is a count of the number of distinct elements in the stream.
The 1st moment is the sum of the m_i's, which must be the length of the stream. Thus, first moments are especially easy to compute; just count the length of the stream seen so far.

The second moment is the sum of the squares of the m_i's. It is sometimes called the surprise number, since it measures how uneven the distribution of elements in the stream is. To see the distinction, suppose we have a stream of length 100, in which eleven different elements appear. The most even distribution of these eleven elements would have one appearing 10 times and the other ten appearing 9 times each. In this case, the surprise number is $102 + 10 \times 92 = 910$. At the other extreme, one of the eleven elements could appear 90 times and the other ten appear 1 time each. Then, the surprise number would be $902 + 10 \times 12 = 8110$.

	**Case Studies**
A) http://ceur-ws.org/Vol-1019/paper_34.pdf	
Filter-Stream Named Entity Recognition: A Case Study at the MSM 2013 Concept Extraction Challenge upon the densities of cars behind green and red lights and the current cycle time.	
B)http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.6276&rep=rep1 &type=pdf	
Counting distinct elements in a data stream: Here three algorithms are presented to count the number of distinct elements in a data stream to within a factor of 1 ±€. Our algorithms improve upon known algorithms for this problem, and offer a spectrum of time/space tradeoffs.	

Summary

- **Filtering streams find wide usage in many scenarios**
- **Counting distinct elements by Flajolet-Martin approach is very effective for search applications.**