e-PGPathshala
Subject : Computer Science
Paper: Data Analytics
Module No 21: CS/DA/21 - Data Analytics - Mining Streams - Part IV
Quadrant 1 – e-text

1.1 Introduction

This chapter gives an overview of the basics of sliding widow and method of counting bits in a data stream.

1.2 Learning Outcomes

- Learn the basics of sliding window
- Understand the basics of counting bits
- Know basics of DGIM approach for counting ones

1.3 Sliding Windows

Sliding window is a useful model of stream processing in which the queries are about a *window* of length *N* – the *N* most recent elements received. In certain cases *N* is so large that the data cannot be stored in memory, or even on disk, or, there are so many streams that windows for all cannot be stored. Sliding window is also known as windowing.

Consider a sliding window of length N=6 on a single stream as shown in figure 1. As the stream content varies over time the sliding window highlights new stream elements.

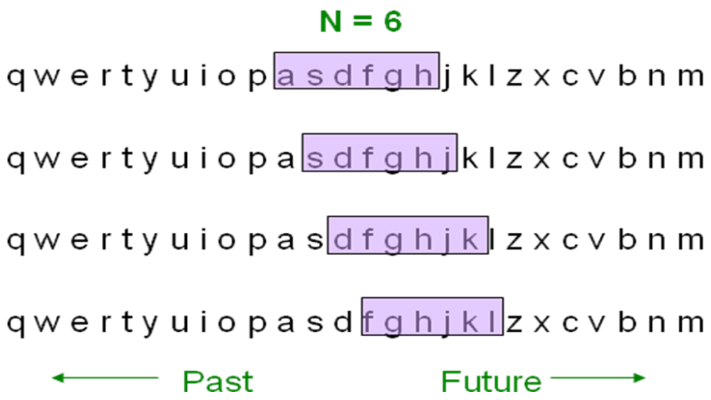


Figure 1. Sliding window on stream

Example1: Consider Amazon online transactions. For every product X we keep 0/1 stream of whether that product was sold in the n-th transaction. A query like, “how many times have we sold X in the last k sales?” and an answer for it can be derived using sliding window concept.

1.3 Counting Bits

To begin, suppose we want to be able to count exactly the number of 1’s in the last k bits for any k ≤ N. Then we claim it is necessary to store all N bits of the window, as any representation that used fewer than N bits could not work. In proof, suppose we have a representation that uses fewer than N bits to represent the N bits in the window. Since there are 2^N sequences of N bits, but fewer than 2^N representations, there must be two different bit strings w and x that have the same representation. Since w 6= x, they must differ in at least one bit. Let the last k −1 bits of w and x agree, but let them differ on the k^{th} bit from the right end.

Example 1 : If w = 0101 and x = 1010, then k = 1, since scanning from the right, they first disagree at position 1. If w = 1001 and x = 0101, then k = 3, because they first disagree at the third position from the right.

Suppose the data representing the contents of the window is whatever sequence of bits represents both w and x. Ask the query “how many 1’s are in the last k bits?” The query-answering algorithm will produce the same answer, whether the window contains w or x, because the algorithm can only see their representation. But the correct answers are surely different for these two bit-strings. Thus, we have proved that we must use at least N bits to answer queries about the last k bits for any k

In fact, we need N bits, even if the only query we can ask is “how many 1’s are in the entire window of length N?” The argument is similar to that used above. Suppose we use fewer than N bits to represent the window, and therefore we can find w, x, and k as above. It might be that w and x have the same number of 1’s. However, if we follow the current window by any N − k bits, we will have a situation where the true window contents resulting from w and x are identical except for the leftmost bit, and therefore, their counts of 1’s are unequal. However, since the representations of w and x are the same, the representation of the window must still be the same if we feed the same bit sequence to these representations. Thus, we can force the answer to the query “how many 1’s in the window?” to be incorrect for one of the two possible window contents.

1.4 The Datar-Gionis-Indyk-Motwani Algorithm(DGIM)

Here we discuss an algorithm called DGIM. This version of the algorithm uses $O(\log^2 N)$ bits to represent a window of N bits, and allows us to estimate the number of 1's in the window with an error of no more than 50%.

To begin, each bit of the stream has a timestamp, the position in which it arrives. The first bit has timestamp 1, the second has timestamp 2, and so on. Since we only need to distinguish positions within the window of length N, we shall represent timestamps modulo N, so they can be represented by log2 N bits. If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo N, then we can determine from a timestamp modulo N where in the current window the bit with that timestamp is.

We divide the window into buckets, consisting of:

1. The timestamp of its right (most recent) end
2. The number of 1's in the bucket. This number must be a power of 2, and we refer to the number of 1's as the size of the bucket

To represent a bucket, we need $\log_2 N$ bits to represent the timestamp (modulo N) of its right end. To represent the number of 1's we only need $\log_2 \log_2 N$ bits. The reason is that we know this number i is a power of 2, say 2^j, so we can represent i by coding j in binary. Since j is at most $\log_2 N$, it requires $\log_2 \log_2 N$ bits. Thus, O(logN) bits suffice to represent a bucket.

There are six rules that must be followed when representing a stream by buckets:

- The right end of a bucket is always a position with a 1.
- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left (back in time).

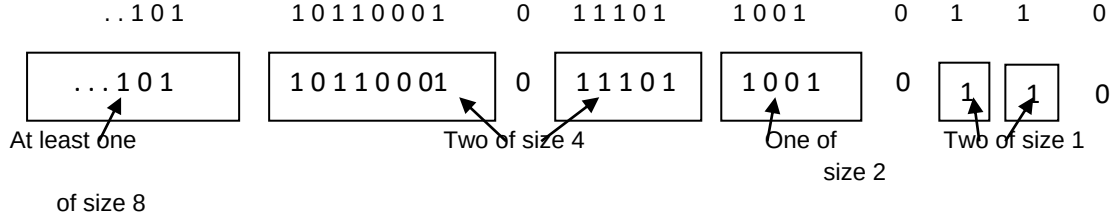


Figure 2: A bit-stream divided into buckets following the DGIM rules

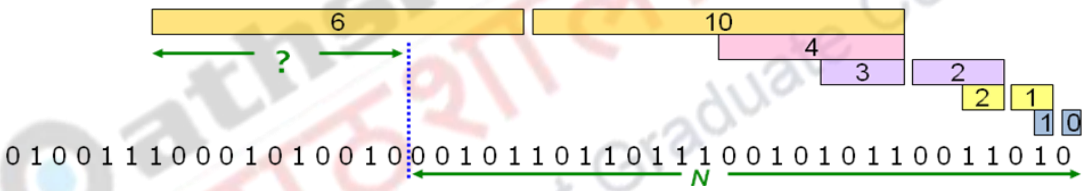
In principle, you could count frequent pairs or even larger sets the same way as one stream per itemset. But, DGIM provides only an approximate estimation and also the number of itemsets is way too big.

1.4.1 Advantages

- Stores only $O(\log^2 N)$ bits
    - $O(\log N)$counts of $\log_2 N$ bits each
- Easy update as more bits enter
    - Error in count no greater than the number of 1's in the unknown area.

1.4.2 Drawbacks

- As long as the **1s** are fairly evenly distributed, the error due to the unknown region is small – no more than 50%
- But it could be that all the 1s are in the unknown area (indicated by "?" in the below figure) at the end. In that case, the error is unbounded.



1.4.3 Maintaining the DGIM Conditions

Suppose we have a window of length N properly represented by buckets that satisfy the DGIM conditions. When a new bit comes in, we may need to modify the buckets, so they continue to represent the window and continue to satisfy the DGIM conditions. First, whenever a new bit enters:

Check the leftmost (earliest) bucket. If its timestamp has now reached the current timestamp minus N, then this bucket no longer has any of its 1's in the window. Therefore, drop it from the list of buckets. Now, we must consider whether the new bit is 0 or 1. If it is 0, then no further change to the buckets is needed. If the new bit is a 1, however, we may need to make several changes. First:

• Create a new bucket with the current timestamp and size 1.

If there was only one bucket of size 1, then nothing more needs to be done. However, if there are now three buckets of size 1, that is one too many. We fix this problem by combining the leftmost (earliest) two buckets of size 1.

• To combine any two adjacent buckets of the same size, replace them by one bucket of twice the size. The timestamp of the new bucket is the timestamp of the rightmost (later in time) of the two buckets.

Combining two buckets of size 1 may create a third bucket of size 2. If so,we combine the leftmost two buckets of size 2 into a bucket of size 4. That, in turn, may create a third bucket of size 4, and if so we combine the leftmost two into a bucket of size 8. This process may ripple through the bucket sizes, but there are at most log2 N different sizes, and the combination of two adjacent buckets of the same size only requires constant time. As a result, any new bit can be processed in O(logN) time.

Suppose we start with the buckets of Fig. 2 and a 1 enters. First, the leftmost bucket evidently has not fallen out of the window, so we do not drop any buckets. We create a new bucket of size 1 with the current timestamp, say t. There are now three buckets of size 1, so we combine the leftmost two. They are replaced with a single bucket of size 2. Its timestamp is t − 2, the timestamp of the bucket on the right (i.e., the rightmost bucket that actually appears in Fig. 3.

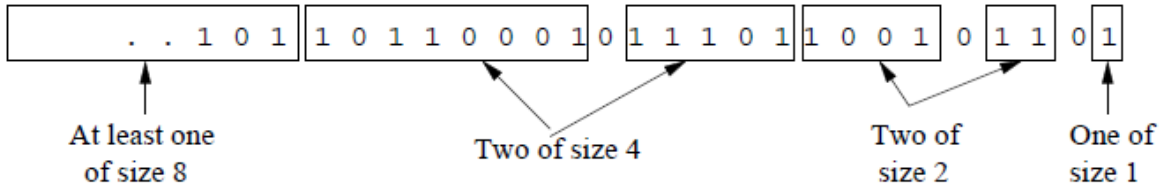


Figure 3: Modified buckets after a new 1 arrives in the stream

There are now two buckets of size 2, but that is allowed by the DGIM rules. Thus, the final sequence of buckets after the addition of the 1 is as shown in figure 3.

1.4.4 DGIM: Buckets

A ***bucket*** in the DGIM method is a record consisting of:

- (A) The timestamp of its end [O(log *N*) bits]
- (B) The number of 1s between its beginning and end [O(log log *N*) bits]

Constraint on buckets are, Number of 1s must be a power of 2 and O(log log *N)* in above mentioned (B)

The stream are represented as buckets as: Either one or two buckets with the same power-of-2 number of 1s, Buckets do not overlap in timestamps, Buckets are sorted by size, Earlier buckets are not smaller than later buckets and Buckets disappear when their end-time is > N time units in the past.

Example: Consider the Bucketized Stream given in figure 4.

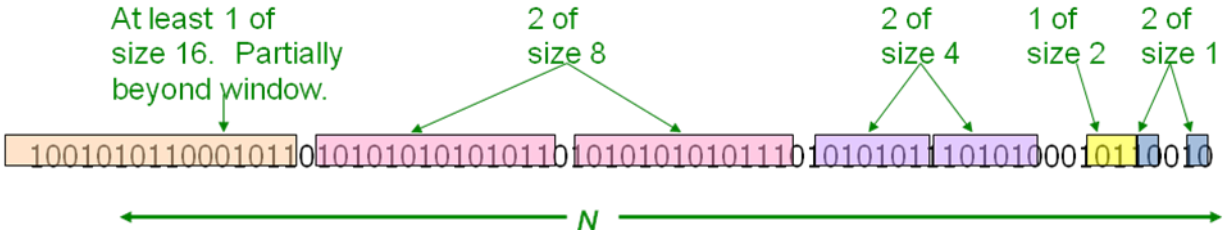


Figure 4. Bucketized Stream

Three properties of buckets that are maintained: (1) Either one or two buckets with the same power-of-2 number of 1s, (2) Buckets do not overlap in timestamps and (3) Buckets are sorted by size.

1.4.5 Updating Buckets

When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to *N* time units before the current time. There are 2 cases that might arise:

Case i: Current bit is 0 or 1. If the current bit is 0: no other changes are needed.
Case ii: **If the current bit is 1:**

(1) Create a new bucket of size 1, for just this bit and set End timestamp = current time
(2) If there are now three buckets of size 1, combine the oldest two into a bucket of size 2
(3) If there are now three buckets of size 2, combine the oldest two into a bucket of size 4
(4) And so on …

The above procedure is shown diagrammatically in figure 5.

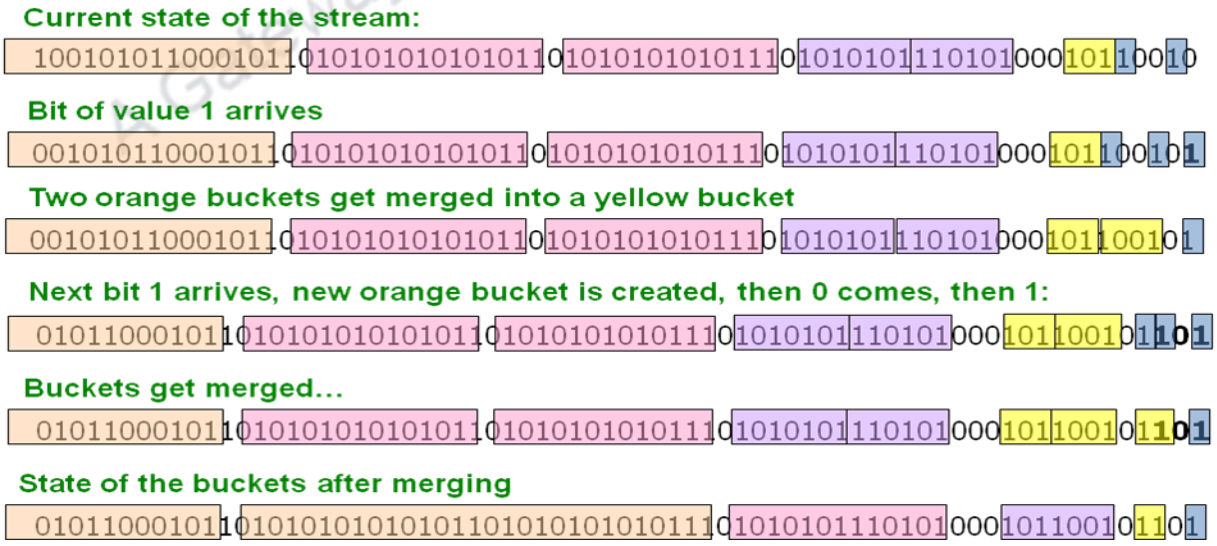


Figure 5. Updating Buckets

1.5 Extensions to the Counting of Ones

It is natural to ask whether we can extend the technique of this section to handle aggregations more general than counting 1's in a binary stream. An obvious direction to look is to consider streams of integers and ask if we can estimate the sum of the last k integers for any $1 \leq k \leq N$, where N, as usual,is the window size.

It is unlikely that we can use the DGIM approach to streams containing both positive and negative integers. We could have a stream containing both very large positive integers and very large negative integers, but with a sum in the window that is very close to 0. Any imprecision in estimating the values of these large integers would have a huge effect on the estimate of the sum, and so the fractional error could be unbounded.

For example, suppose we broke the stream into buckets as we have done, but represented the bucket by the sum of the integers therein, rather than the count of 1's. If b is the bucket that is partially within the query range, it could be that b has, in its first half, very large negative integers and in its second half, equally large positive integers, with a sum of 0. If we estimate the contribution of b by half its sum, that contribution is essentially 0. But the actual contribution of that part of bucket b that is in the query range could be anything from 0 to the sum of all the positive integers. This difference could be far greater than the actual query answer, and so the estimate would be meaningless. On the other hand, some other extensions involving integers do work.

1.6 Querying bucketized streams and error

To estimate the number of 1s in the most recent N bits, we can sum the sizes of all buckets but the last (note "size" means the number of 1s in the bucket), then add half the size of the last bucket. We must remember the fact that we do not know how many 1s of the last bucket are still within the wanted window.

Suppose the last bucket has size 2^r , then by assuming 2^{r-1} (i.e., half) of its 1s are still within the window, we make an error of at most 2^{r-1} . Since there is at least one bucket of each of the sizes less than 2^r, the true sum is at least $1 + 2 + 4 + .. + 2^{r-1} = 2^r - 1$. Thus, error at most 50%. Instead of maintaining 1 or 2 of each size bucket, we allow either r-1 or r buckets ($r > 2$).Except for the largest size buckets; we can have any number between 1 and r of those the error is at most O(1/r). By picking r appropriately, we can tradeoff between number of bits we store and the error.

1.7 Applications of DGIM

Same approach can be used to answer queries How many 1's in the last k? where $k < N$. We can find earliest bucket B that at overlaps with k. Number of 1s is the sum of

sizes of more recent buckets + ½ size of B. We can also handle the case where the stream is not bits, but integers, and we want the sum of the last k elements.

Suppose we have stream of positive integers and we want the sum of the last k elements, for example, find the average price of last k sales, and the stream consists of only positive integers in the range 1 to 2^m for some m. We can treat each of the m bits of each integer as if it were a separate stream. We then use the DGIM method to count the 1's in each bit. Suppose the count of the ith bit (assuming bits count from the low-order end, starting at 0) is ci. Then the sum of the integers is

$$\sum_{i=0}^{m-1} C_i 2^i$$

to estimate each C_i with fractional error at most ǫ, then the estimate of the true sum has error at most €. The worst case occurs when all the C_i's are over estimated or all are underestimated by the same fraction.

Summary

- Counting the number of 1s in the last N elements
    - Exponentially increasing windows
    - Extensions:
        - Number of 1s in any last k (k < N) elements
        - Sums of integers in the last N elements