Semester : VII Subject : Big Data Analytics Academic Year: 2024 – 2025

MODULE 4:

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.

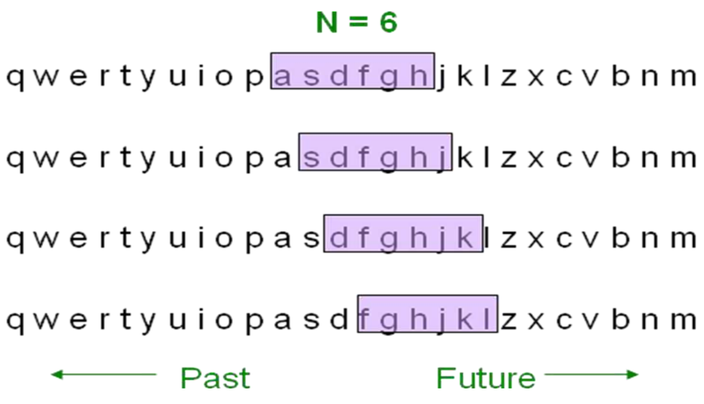
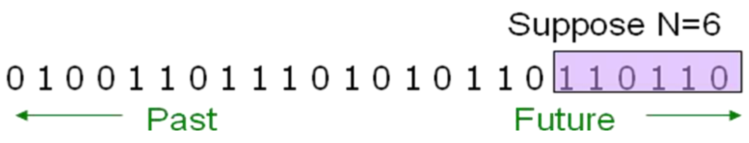

N = 6
q w e r t y u i o p a s d f g h j k l z x c v b n m
q w e r t y u i o p a s d f g h j k l z x c v b n m
q w e r t y u i o p a s d f g h j k l z x c v b n m
q w e r t y u i o p a s d f g h j k l z x c v b n m
Past
Future

Figure 1. Sliding window on stream

Counting Bits

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.

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

The DGIM 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.

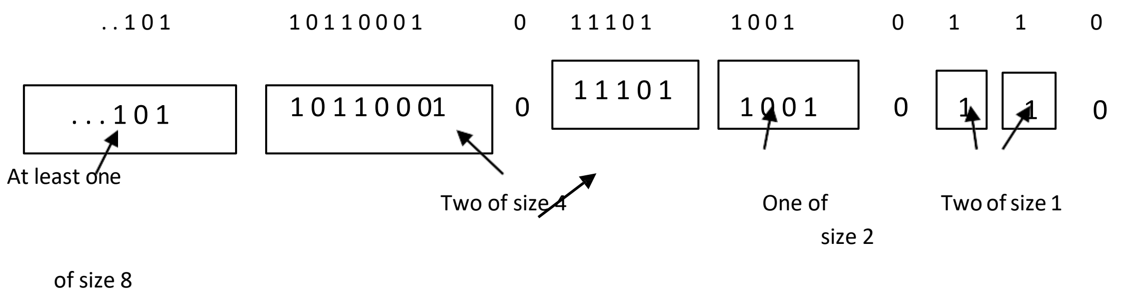
We divide the window into buckets, consisting of:

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).

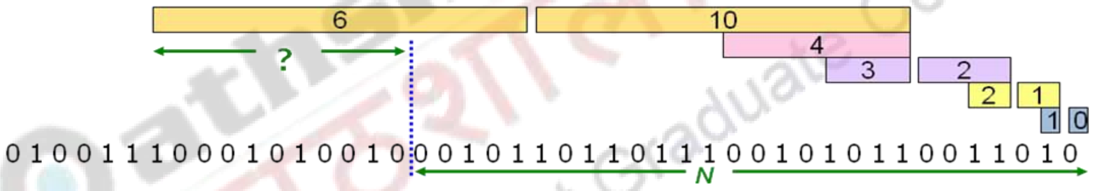


Advantages

- Stores only O($\log^2$ N) bits
  - O(log N)counts of log2N bits each
- Easy update as more bits enter
  - Error in count no greater than the number of 1's in the unknown area.

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.



1.3.2 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.

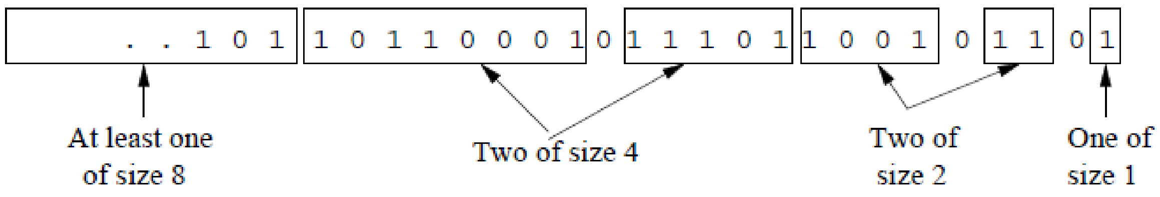


Figure 3: Modified buckets after a new 1 arrives in the stream

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.

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.

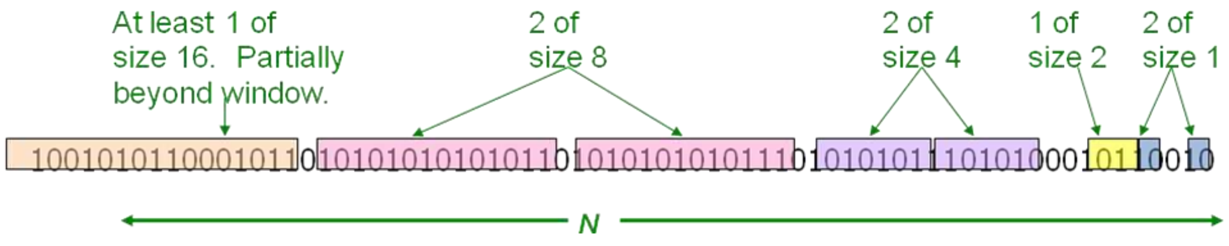


Figure 4. Bucketized Stream

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.

1.3.3 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.

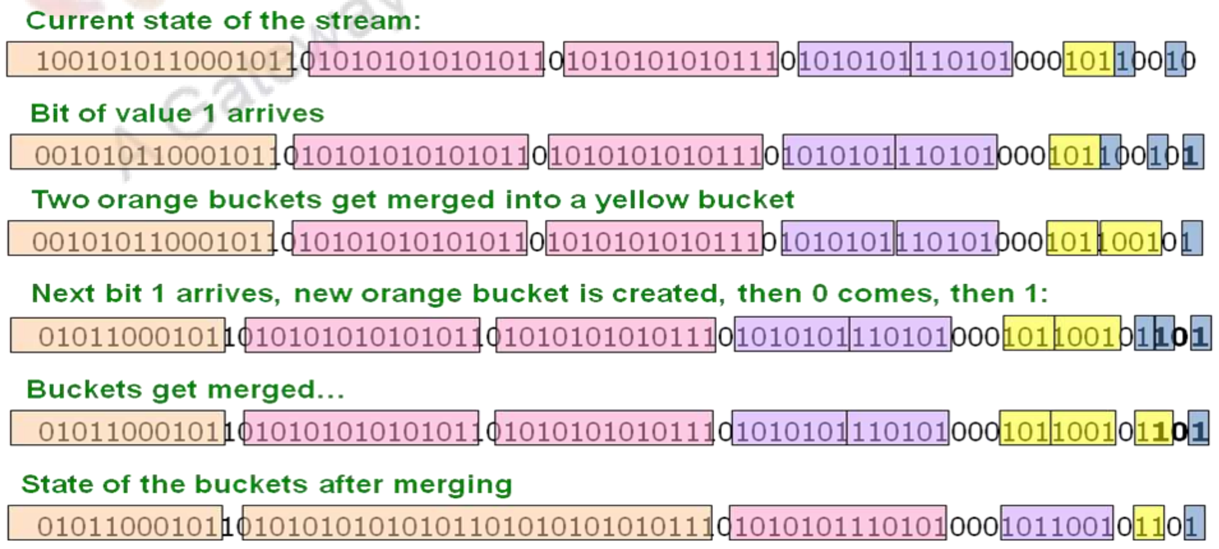

Current state of the stream:
Bit of value 1 arrives
Two orange buckets get merged into a yellow bucket
Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:
Buckets get merged…
State of the buckets after merging

Figure 5. Updating Buckets