Mining Data Streams (Part 1)

Mining of Massive Datasets
Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University
http://www.mmds.org

Queries over a (long) Sliding Window

Sliding Windows

- A useful model of stream processing is that queries are about a ***window*** of length ***N*** – the ***N*** most recent elements received
- **Interesting case:** ***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
- **Amazon example:**
  - For every product **X** we keep 0/1 stream of whether that product was sold in the **n**-th transaction
  - We want answer queries, how many times have we sold **X** in the last **k** sales

Sliding Window: 1 Stream

- **Sliding window on a single stream:** **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 →

Counting Bits (1)

- **Problem:**
  - Given a stream of **0**s and **1**s
  - Be prepared to answer queries of the form
    How many 1s are in the last *k* bits? where $k \leq N$
- **Obvious solution:**
  Store the most recent ***N*** bits
  - When new bit comes in, discard the ***N*+1**^st^ bit

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0   Suppose N=6

← Past   Future →

Counting Bits (2)

- **You can not get an exact answer without storing the entire window**

- **Real Problem:**
  What if we cannot afford to store *N* bits?
  - **E.g.**, we're processing 1 billion streams and
    ***N* = 1 billion**

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 ~~1 1 0 1 1 0~~

← Past   Future →

- **But we are happy with an approximate answer**

An attempt: Simple solution

- **Q: How many 1s are in the last *N* bits?**
- A simple solution that does not really solve our problem: **Uniformity assumption**

N

0 1 0 0 1 1 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 1 0 1 0

← Past Future →

- **Maintain 2 counters:**
  - ***S***: number of 1s from the beginning of the stream
  - ***Z***: number of 0s from the beginning of the stream
- **How many 1s are in the last N bits?** $N \cdot \frac{S}{S+Z}$
- **But, what if stream is non-uniform?**
  - What if distribution changes over time?

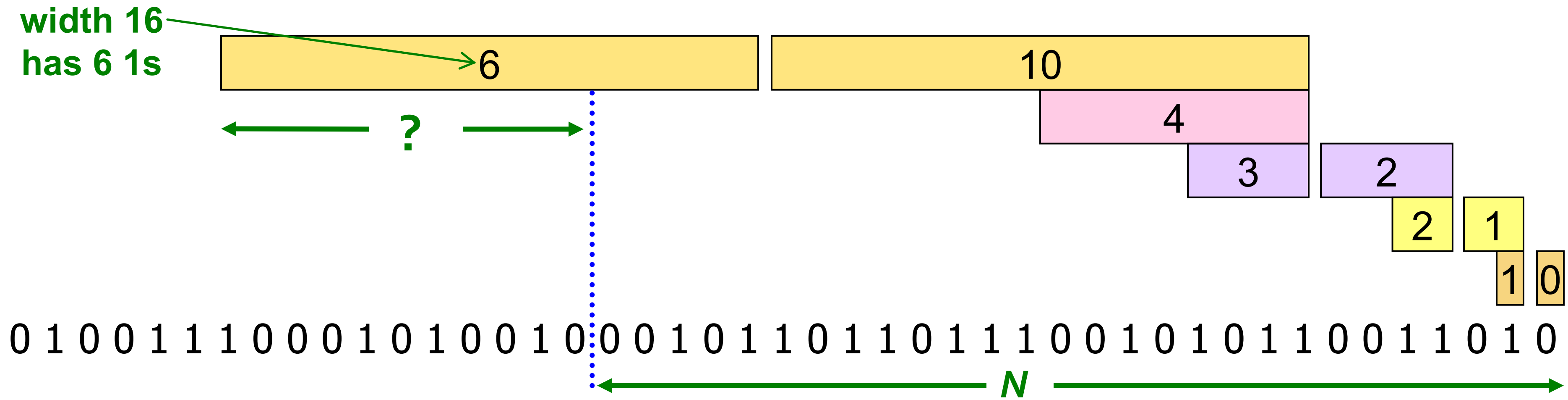
[Datar, Gionis, Indyk, Motwani]

DGIM Method

- **DGIM solution that does not assume uniformity**
- We store $O(\log^2 N)$ bits per stream
- **Solution gives approximate answer, never off by more than 50%**
  - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits

Idea: Exponential Windows

- **Solution that doesn't (quite) work:**
  - Summarize **exponentially increasing** regions of the stream, looking backward
  - Drop small regions if they begin at the same point as a larger region



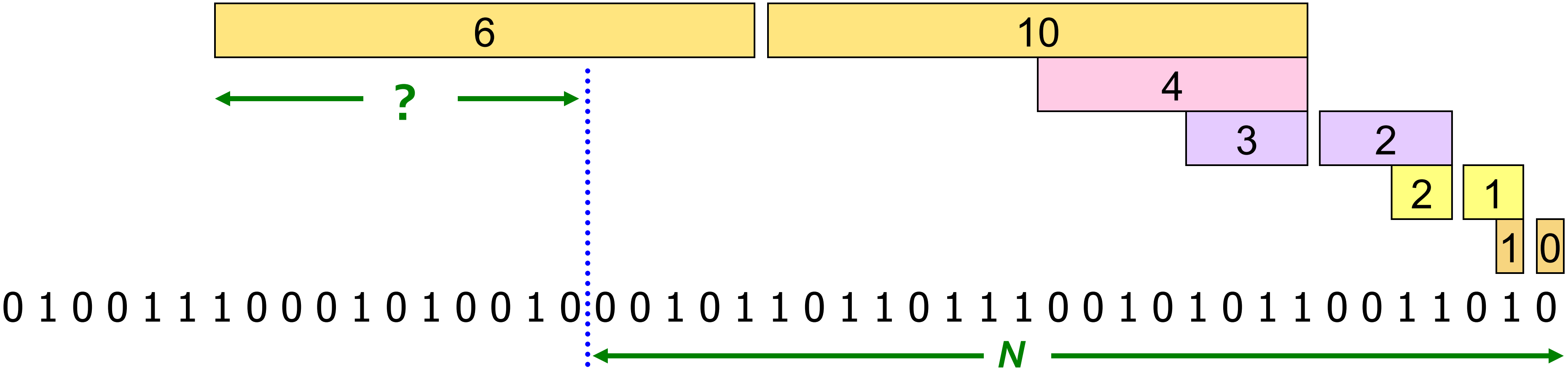
We can reconstruct the count of the last ***N*** bits, except we are not sure how many of the last **6 1s** are included in the ***N***

What's Good?

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

What's Not So Good?

- 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 at the end
- In that case, **the error is unbounded!**



[Datar, Gionis, Indyk, Motwani]

Fixup: DGIM method

- **Idea:** Instead of summarizing fixed-length blocks, summarize blocks with specific number of **1s**:
  - Let the block ***sizes*** (number of **1s**) increase exponentially

- **When there are few 1s in the window, block sizes stay small, so errors are small**

1001010110001011 0 1010101010101011 0 10101010101111 0 1010101 110101 000 101 1 00 1 0

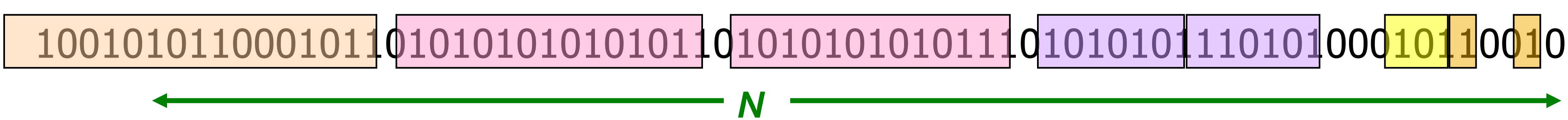
← N →

DGIM: Timestamps

- Each bit in the stream has a ***timestamp***, starting **1**, **2,** …
- Record timestamps modulo ***N*** (**the window size**), so we can represent any **relevant** timestamp in $\boldsymbol{O}(\log_2 \boldsymbol{N})$ bits

DGIM: Buckets

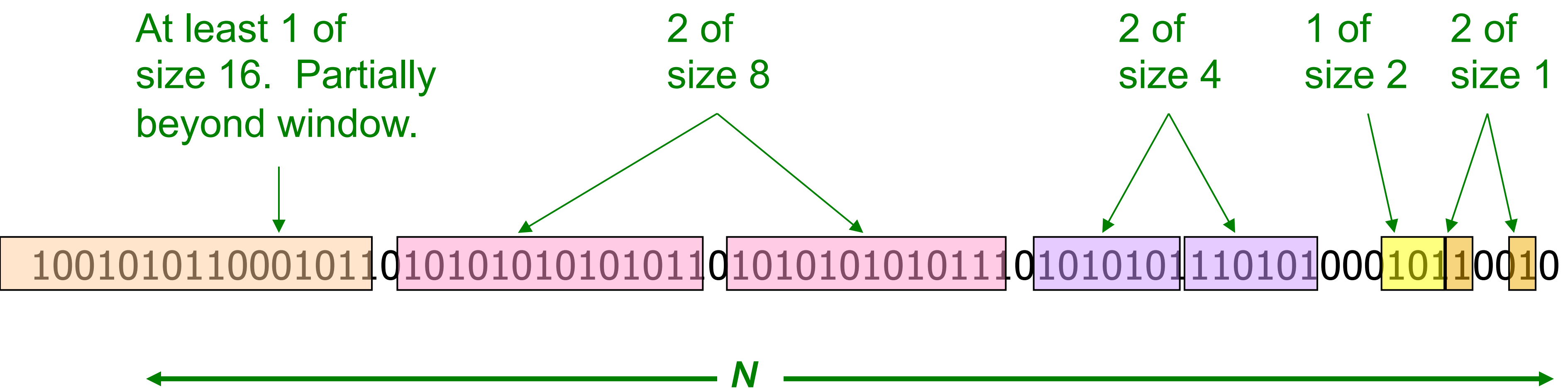
- A ***bucket*** in the DGIM method is a record consisting of:
  - **(A) The timestamp of its end** [O(log *N*) bits]
  - **(B) The number of 1s between its beginning and end** [O(log log *N*) bits]
- **Constraint on buckets:**
  Number of **1s** must be a power of **2**
  - That explains the **O(log log *N*)** in **(B) above**



Representing a Stream by Buckets

- 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
- Buckets disappear when their end-time is **>** ***N*** time units in the past

Example: Bucketized Stream



Three properties of buckets that are maintained:

- 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

Updating Buckets (1)

- 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
- **2 cases:** Current bit is **0** or **1**
- **If the current bit is 0:**
  no other changes are needed

Updating Buckets (2)

- **If the current bit is 1:**
  - **(1)** Create a new bucket of size **1**, for just this bit
    - **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 …**

Example: Updating Buckets

Current state of the stream:

10010101100010110101010101010101101010101010101111010101011101010001011001 0

Bit of value 1 arrives

001010110001011010101010101010110101010101011110101010111010100010110010 1

Two orange buckets get merged into a yellow bucket

001010110001011010101010101010110101010101011110101010111010100010110010 1

Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:

01011000101101010101010101011010101010101111010101011101010001011001011 01

Buckets get merged…

01011000101101010101010101011010101010101111010101011101010001011001011 01

State of the buckets after merging

01011000101101010101010101011101010101010111101010101110100010110010110 1

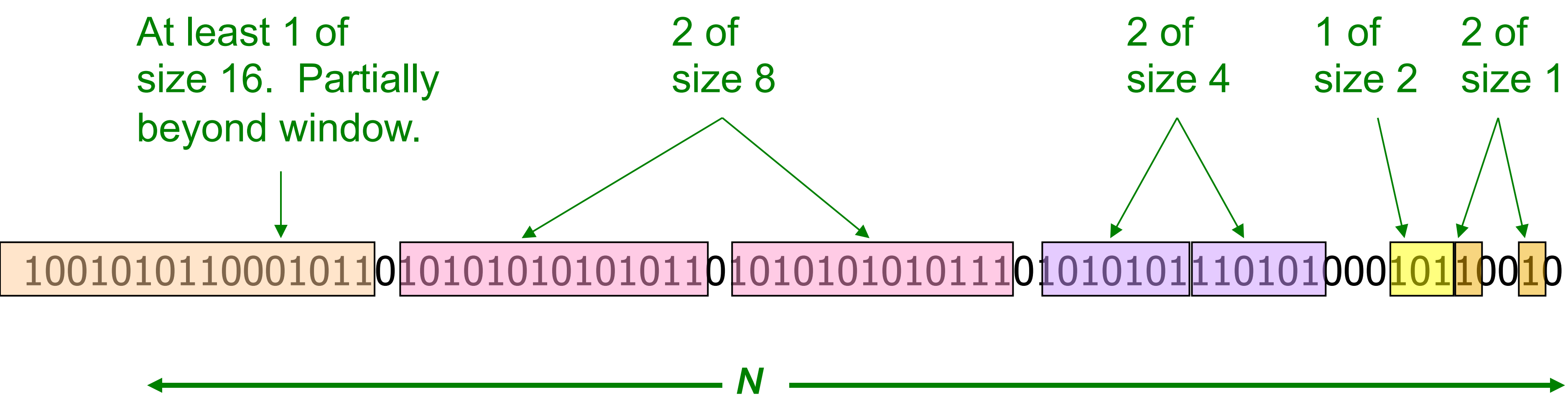
How to Query?

- **To estimate the number of 1s in the most recent *N* bits:**
  1. **Sum the sizes of all buckets but the last**

     (note "size" means the number of 1s in the bucket)
  2. **Add half the size of the last bucket**

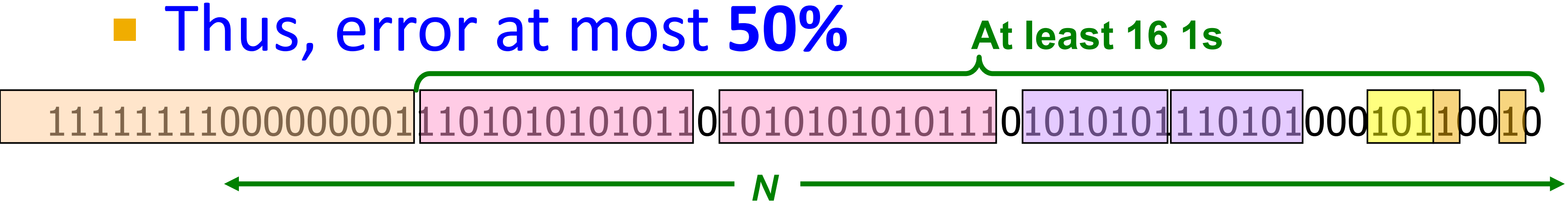
- **Remember:** We do not know how many **1s** of the last bucket are still within the wanted window

Example: Bucketized Stream



Error Bound: Proof

- **Why is error 50%? Let's prove it!**
- Suppose the last bucket has size $\mathbf{2^r}$
- Then by assuming $\mathbf{2^{r-1}}$ (i.e., half) of its **1s** are still within the window, we make an error of at most $\mathbf{2^{r-1}}$
- Since there is at least one bucket of each of the sizes less than $\mathbf{2^r}$, the true sum is at least
  $\mathbf{1 + 2 + 4 + .. + 2^{r-1} = 2^r - 1}$
- Thus, error at most **50%**



Further Reducing the Error

- 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
- **Error is at most *O*(1/*r*)**
- By picking ***r*** appropriately, we can tradeoff between number of bits we store and the error

Extensions

- Can we use the same trick to answer queries **How many 1's in the last *k*?** where ***k*** **<** ***N***?
  - **A:** 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**

100101011000101101010101010101011010101010101110101010111010100010110010

$\longleftarrow k \longrightarrow$

- **Can we handle the case where the stream is not bits, but integers, and we want the sum of the last *k* elements?**

Extensions

- **Stream of positive integers**
- **We want the sum of the last *k* elements**
  - **Amazon:** Avg. price of last **k** sales
- **Solution:**
  - **(1) If you know all have at most *m* bits**
    - Treat ***m*** bits of each integer as a separate stream
    - Use DGIM to count **1s** in each integer
    - The sum is $= \sum_{i=0}^{m-1} c_i 2^i$
  - **(2) Use buckets to keep partial sums**
    - **Sum of elements in size *b* bucket is at most 2^b**

c_i …estimated count for **i-th** bit

2 5 7 1 3 8 4 6 7 9 1 3 7 | 6 5 | 3 5 | 7 1 | 3 | 3 | 1 | 2 | 2 | 6
2 5 7 1 3 8 4 6 7 9 1 3 7 6 5 | 3 5 7 1 | 3 3 | 1 2 | 2 | 6 | 3
2 5 7 1 3 8 4 6 7 9 1 3 7 6 5 | 3 5 7 1 | 3 3 | 1 2 | 2 | 6 | 3 | 2
2 5 7 1 3 8 4 6 7 9 1 3 7 6 5 | 3 5 7 1 | 3 3 | 1 2 | 2 | 6 | 3 | 2 | 5

Idea: Sum in each bucket is at most **2^b** (unless bucket has only **1** integer)

Bucket sizes:

16	8	4	2	1

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