DS 503: Advanced Data Analytics

Lecture 7: Streams

Instructor: Dr. Gagan Raj

How to deal with high volume streaming data?

- In many data analysis situations, we don’t know the entire dataset in advance
- **Stream Management** is important when the input rate is controlled **externally:**
  - Google queries
  - Twitter or Facebook status updates
  - User Activity on E-commerce platforms
  - Sensor activity
  - Event-driven cameras
- We can think of the **data** as **infinite** and **non-stationary** (the distribution changes over time)
- This is the fun part and why interesting algorithms are needed

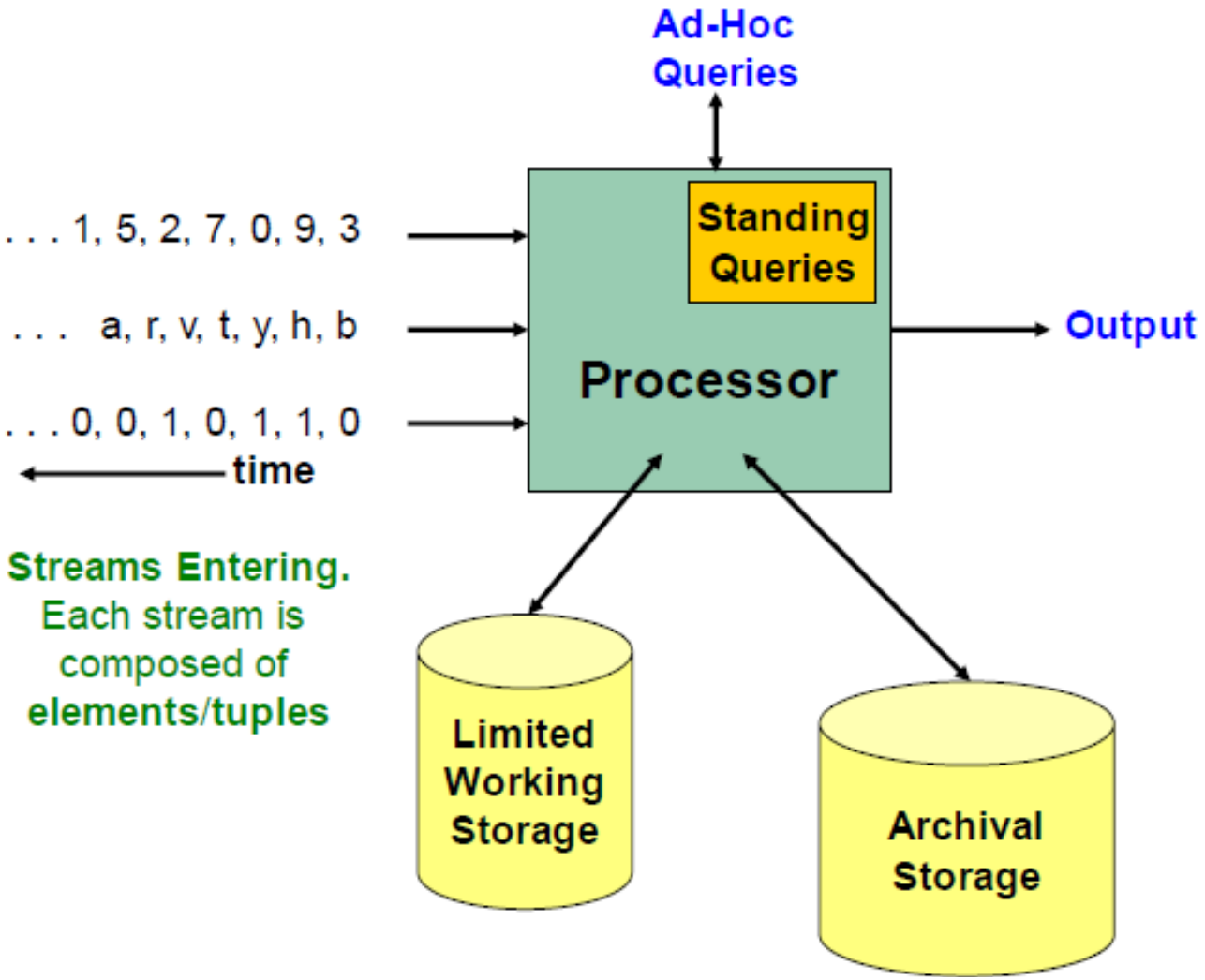
Problem statement: Streaming Model

- Input **elements** enter at a rapid rate, at one or more input ports (i.e., **streams**)
  - **We call elements of the stream tuples**
- **The system cannot store the entire stream accessibly**
- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

Side Note: Online Learning Algorithms

- **In Machine Learning we call this: Online Learning**
  - Allows for modeling problems where we have a continuous stream of data
  - We want an algorithm to learn from it and slowly adapt to the changes in data
- **Stochastic Gradient Descent (SGD) is an example of a streaming algorithm**
- **Idea: Do small updates to the model**
  - **Initial Training:** Train the classifier on training data (or random initialization with small weights)
  - **Compute Loss and Gradients**
  - **SGD** makes small updates
  - **Update Model:** For every example from the stream, we slightly update the model (using small learning rate)

General Stream Processing Model



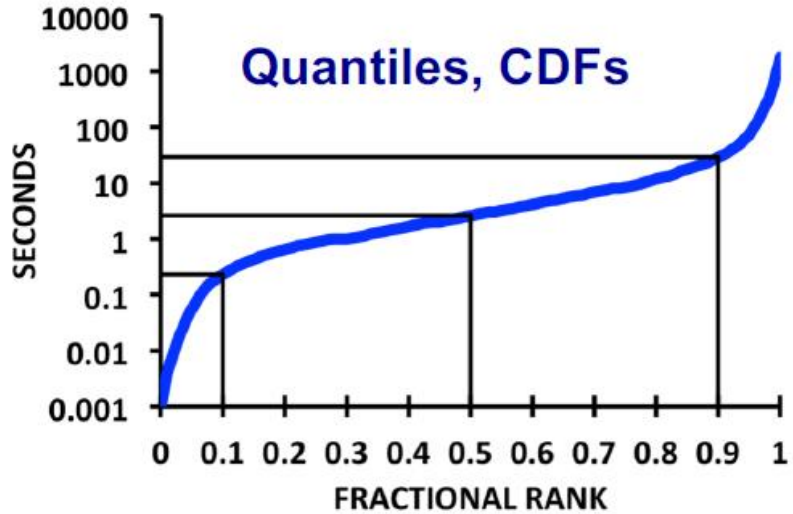
- In general, streams have to be processed in one pass
- The working storage (main memory) is typically limited and can't store the whole stream

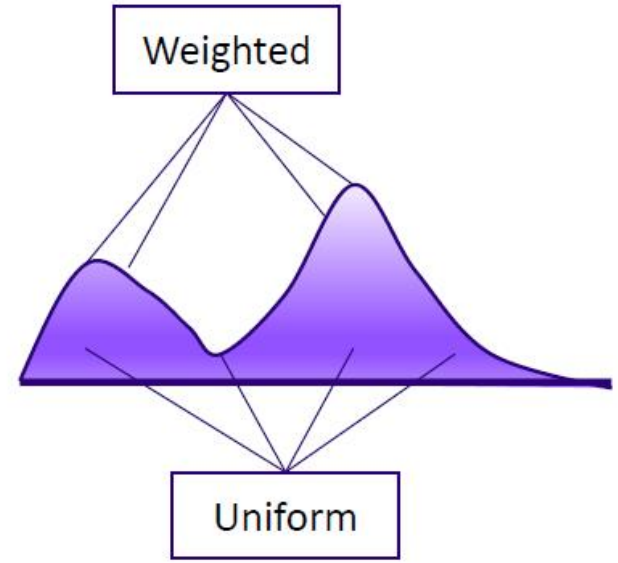
Examples of analysis on Data Streams

- Sampling data from a stream
  - Construct a random sample (weighted or unweighted)
- Queries over sliding windows
  - Number of items of type **x** in the last **k** elements of the stream
- Filtering a data stream
  - Select elements with property **x** from the stream
- Counting distinct elements
  - Number of distinct elements in the last **k** elements of the stream
- Estimating moments
  - Estimating the avg./std. dev. Of last **k** elements
- Finding frequent elements

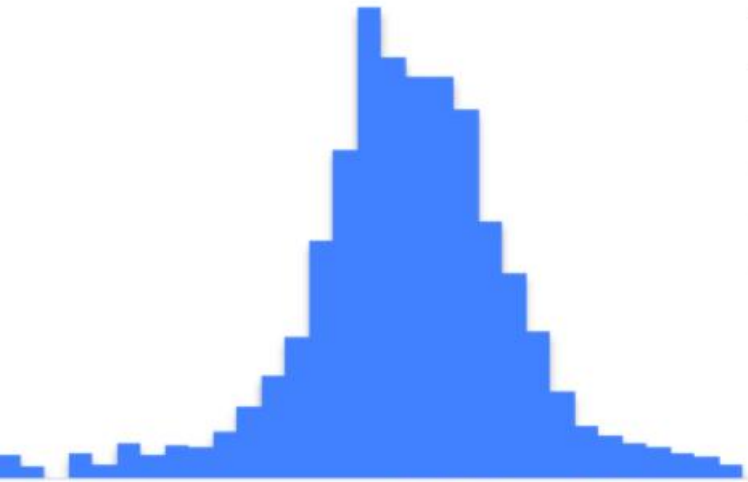
Some Very Common Queries …

Counting Unique Identifiers

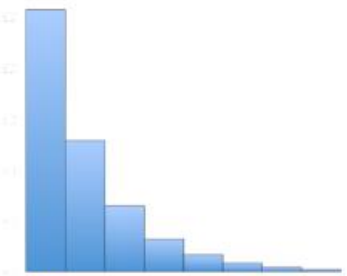




Reservoir Sampling



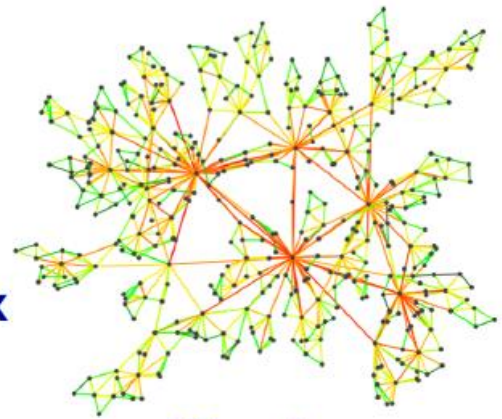
Histograms, PMFs, CDFs



Frequent Items

$$\begin{Bmatrix} 5 & \cdots & 2 \\ \vdots & \ddots & \vdots \\ 4 & \cdots & 3 \end{Bmatrix}$$

Vector & Matrix Operations: SVD, etc.



Graphs

… All Are Computationally Difficult

Use cases

- **Mining query streams**
  - Google wants to know what queries are more frequent today than yesterday {several hundred million queries/day}
  - An increase in queries for “sore throat” may help tracking of virus spread
- **Mining click streams**
  - Wikipedia wants to know which of its pages are getting an unusual number of hits in the past hour {billions of clicks/day}
- **Mining social network news feeds**
  - Look for trending topics on Twitter, Facebook
- **Sensor Networks**
  - Many sensors feeding into a central controller {e.g. million sensors sending 4 bytes every 0.1 sec}
- **Image Data**
  - Satellites images, surveillance cameras {City of London has six millions of cameras}
- **Telephone call records**
  - Data feeds into customer bills as well as settlements between telephone companies {internet backbone}
- **IP packets monitored at a switch**
  - Gather information for optimal routing, Detect denial-of-service attacks

Use cases continued

- **Smart Cities** - real-time traffic analytics, congestion prediction and travel time apps.
- **Oil & Gas** - real-time analytics and automated actions to avert potential equipment failures.
- **Security intelligence** for fraud detection and cybersecurity alerts. For example, detecting Smart Grid consumption issues, and SIM card misuse.
- **Industrial automation**, offering real-time analytics and predictive actions for patterns of manufacturing plant issues and quality problems.
- **For Telecoms,** real-time call rating, fraud detection and QoS monitoring from CDR (call detail record) and network performance data.
- **Cloud infrastructure** and web clickstream analysis for IT Operations.

Sampling from a Data Stream

- Since **we can not store the entire stream**, one obvious approach is to store a **sample**
- **Two different problems:**
  - **(1)** Sample a **fixed proportion** of elements in the stream (say 1 in 10)
  - **(2)** Maintain a **random sample of fixed size** over a potentially infinite stream
    - At any "time" ***k*** we would like a random sample of ***s*** elements of the stream 1..*k*
      - **What is the property of the sample we want to maintain?**
      - For all time steps ***k***, each of the ***k*** elements seen so far has equal prob. of being sampled
- Samples serve as representative elements for other downstream tasks

Sampling a fixed proportion

- **Problem 1: Sampling a fixed proportion**
- **Scenario:** Search engine query stream
  - **Stream of tuples:**(user, query, time)
  - **Answer questions such as: How often did a user run the same query in a single day**
  - Have space to store **1/10th** of query stream
- **Naïve solution:**
  - Generate a random integer in **[0…9]** for each query
  - Store the query if the integer is **0**, otherwise discard
- Answer questions using the stored random sample

Problem with the Naïve Approach

- **Simple question: What fraction of (unique) queries by an average search engine user are duplicates in a given period of time?**
  - Suppose each user issues ***x*** queries once and ***d*** queries twice (total of ***x*+2*d*** query instances)
    - **Correct answer:** $\frac{d}{x+d}$
  - **Proposed solution: We keep 10% of the queries**
    - Sample will contain approximately $\frac{x}{10}$ of the singleton queries
    - Only ***d*/100** pairs of duplicates
      - **d/100= 1/10 · 1/10 · d**
    - Of ***d*** "duplicates" ***18d/100*** appear exactly once
      - **18d/100 = ((1/10 · 9/10)+(9/10 · 1/10)) · d**
  - **So the sample-based answer is** $\frac{\frac{d}{10}}{\frac{x}{10}+\frac{18d}{100}+\frac{d}{100}} = \frac{d}{10x+19d} < \frac{d}{x+d}$

Solution: sample users instead of queries

- Basic Idea:
  - Pick $\frac{1}{10}$ of the users and take all their searches in the sample
  - Use a hash function that hashes the user name or user id uniformly into “k=10” buckets
- Generalized solution:
  - Stream of tuples with keys:
    - Key is some subset of each tuple’s components
      - E.g., tuple is (user, search, time); key is user
    - Choice of key depends on the application
  - To get a sample $\frac{a}{b}$ fraction of the stream:
    - Hash each tuple’s key uniformly into b buckets
    - Pick the tuple if its hash value is at most a

Hash table with **b** buckets, pick the tuple if its hash value is at most **a.**

How to generate a 30% sample?

Hash into b=10 buckets, take the tuple if it hashes to one of the first 3 buckets

Maintaining a fixed-size sample

- Problem 2: Fixed-size sample
- Suppose we need to maintain a random sample S of size exactly s tuples
  - E.g., main memory size constraint
- Why? We don't know the length of the stream in advance
- Suppose by time n we have seen n items
  - Each item in the sample S must be taken with equal probability $\frac{s}{n}$
- Example: Let us say s=2

  Stream: a b c x y z k y d e g …

  At n=5, each of the first 5 tuples (in red) are selected in sample with probability 0.4

  At n=8, each of the first 8 tuples (in red and green) are selected in the sample with probability 0.25

Difficulty: The sampling probability is dependent on the size of the stream.

Solution: Reservoir sampling

- Store all the first s elements of the stream to S
- Suppose we have seen n-1 elements, and now the n^{th} element arrives (n>s)
  - With probability s/n, keep the n^{th} element, else discard it {At least we are correct for this element}
  - If we have to keep the n^{th} element in this step, we need to discard something. Just discard one of the previously samples elements randomly from S

Claim: This algorithm maintains a sample S with the desired property: After n elements, sample contains each element seen so far with P= s/n (each duplicate is treated as a separate element)

Proof: By induction

Assume that after n elements, sample contains each element seen so far with P = s/n

We have to show that after seeing the n+1 element, the P = $\frac{s}{n+1}$ for each element

Proof

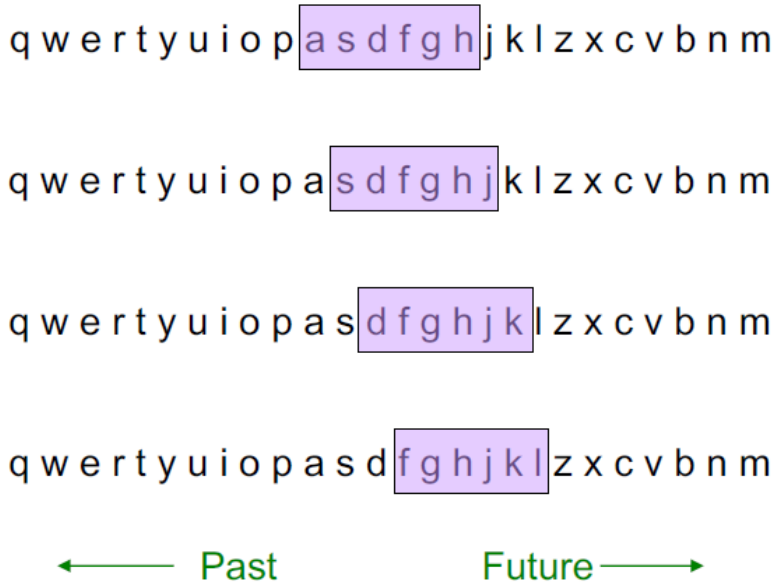
- Base Case:
  - After we see n=s elements, the sample S_s has the desired property.
  - Each of the n=s elements is in the sample with P=s/s =1
- Inductive Hypothesis:
  - After n elements, the S_n contains each element i seen so far with $P[i \in S_n]$ = s/n
- Now element n+1 arrives
- Inductive step:
  - For elements already in S_n, the probability that the algorithms keeps it in S_{n+1} is:
    - $P[i \in S_{n+1} | i \in S_n] = \left(1 - \frac{s}{n+1}\right) + \left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right) = \frac{n}{n+1}$
  - $P[i \in S_{n+1}]$ = P[i $\in S_n$]. $P[i \in S_{n+1} | i \in S_n] = \frac{s}{n} . \frac{n}{n+1} = \frac{s}{n+1}$ $\{i \in S_{n+1}\} \subset \{i \in S_n\}$

Queries over a long Sliding Window

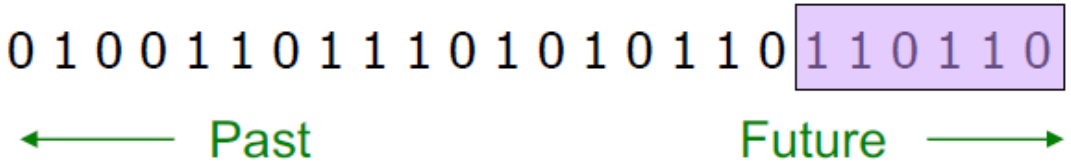
- A useful model of stream processing is that queries are about a ***window*** of length ***N*** – the ***N*** most recent elements received
- **Interesting case:** ***N*** is so large that the data cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
- **Amazon example:**
  - For every product **X** we keep 0/1 stream of whether that product was sold in the $\mathbf{n^{th}}$ transaction
  - We want answer queries, how many times have we sold **X** in the last **k** sales

Sliding Window

- Given a stream of items/symbols
- Most frequent item in the sliding window
- Here N=6



- Given a stream of **0**s and **1**s (for each item)
- Be prepared to answer queries of the form
- **How many 1s are in the last *k* bits?** For any ***k*** ≤ ***N***



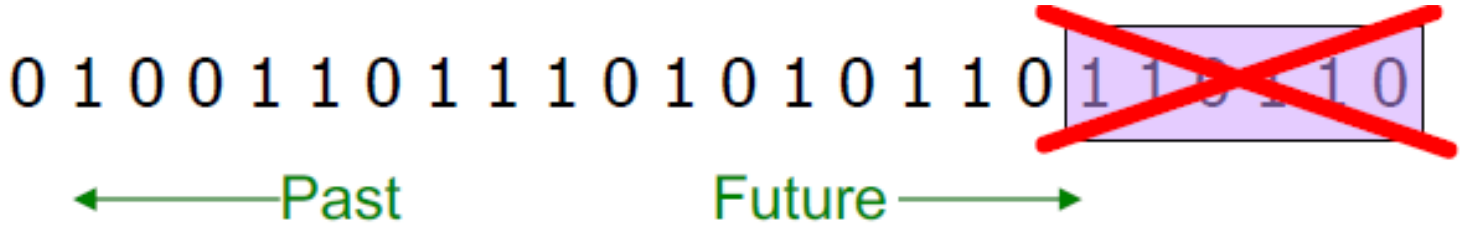
Obvious solution:

Store the most recent ***N*** bits

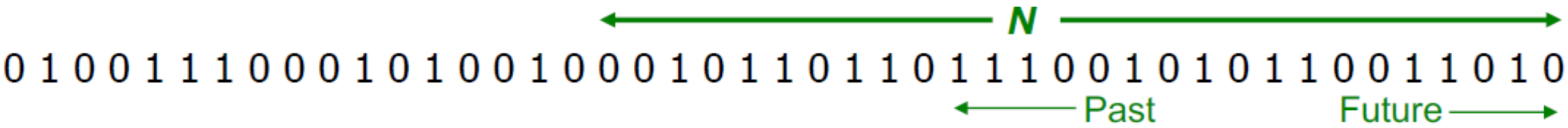
- When new bit comes in, discard the ***N*+1st** bit

Counting Bits

- **You can not get an exact answer without storing the entire window**
- **Real Problem:**
  - **What if we cannot afford to store *N* bits?**
  - Say we're processing many such streams and for each *N*=1 billion
- But we are happy with an approximate answer



Simple Solution with Uniformity Assumption



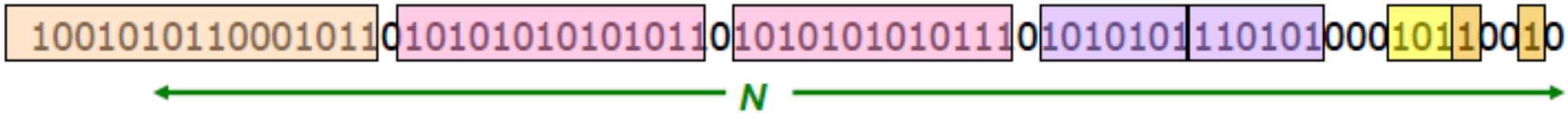
- Assume that the ratio of 1 remains the same over time
- Maintain two counters: O and Z to keep the counts of ones and zeros
- At any time, if you get the query, how many 1's in last N bits?
  - $N\left(\frac{O}{O+Z}\right)$
- But, what if the stream is non-uniform?
  - Distribution changes over time
  - We will need more counters, obviously

DGIM Method

- **DGIM solution that does not assume uniformity**
  - We store $\boldsymbol{O}(\log \boldsymbol{2N})$ bits per stream
- **Solution is approximate, never off by more than 50%**
  - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits
  - Error example: If we have 10 1s then 50% error means estimate = 10 +/- 5
- Basic Idea:
  - Summarize exponentially increasing regions of the stream, looking backward

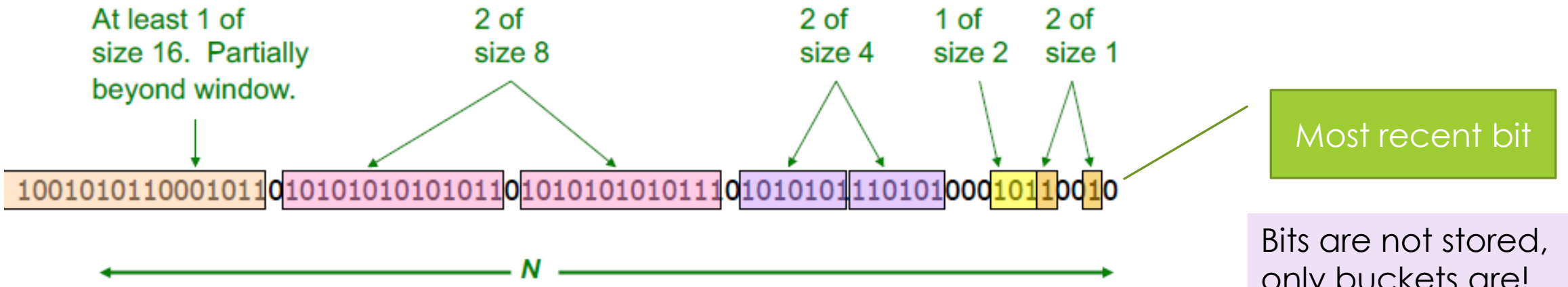
DGIM: Terms

- Timestamps:
  - Each bit in the stream has a timestamp, starting 1,2,…
  - Record the timestamps modulo N (the window size), so we can represent any relevant timestamp in N bits
- Buckets:
  - A bucket in the DGIM method is a record consisting of
    - The timestamp of its end [O(log N) bits]
    - Number of 1s in each bucket must be a power of 2
    - The number of 1s between its beginning and end [O(log log N) bits



Representing a stream by buckets

- We can have only one or two buckets with the same number of 1s (power-of 2)
  - Think of this as the binary representation of the 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



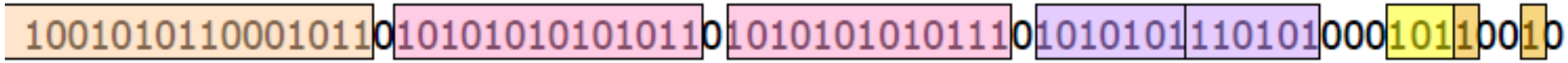
Bits are not stored, only buckets are!

Updating Buckets

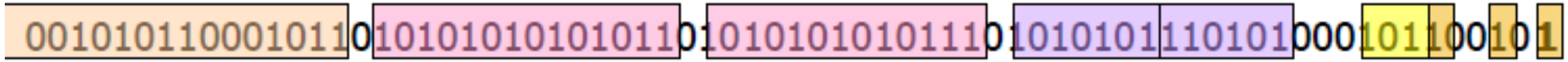
- When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to N time units before the current time
- Processing the current bit:
  - If the current bit is 0: No other changes needed
  - If current bit is 1:
    - Create a new bucket of size 1, for just this bit. End timestamp = current time
    - If there are now three buckets of size 1, combine the oldest two into a bucket of size 2 with timestamp of older one
    - If there are now three buckets of size 2, combine the oldest two into a bucket of size 4 with timestamp of older one
    - And so on …

Example: Updating Buckets

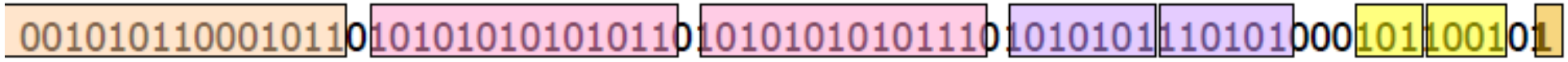
Current state of the stream:



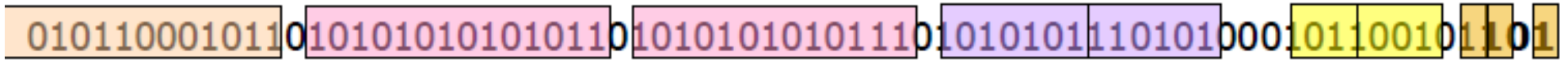
Bit of value 1 arrives



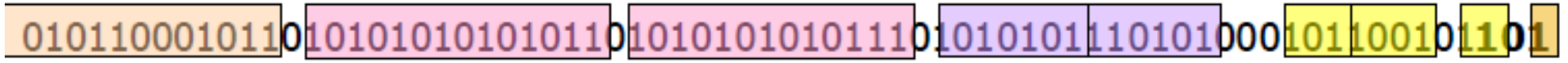
Two orange buckets get merged into a yellow bucket



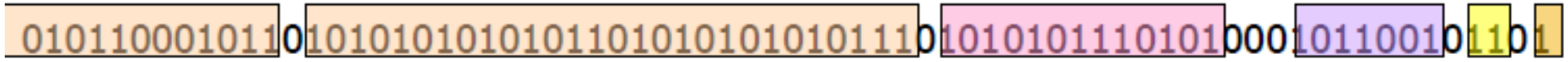
Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:



Buckets get merged…

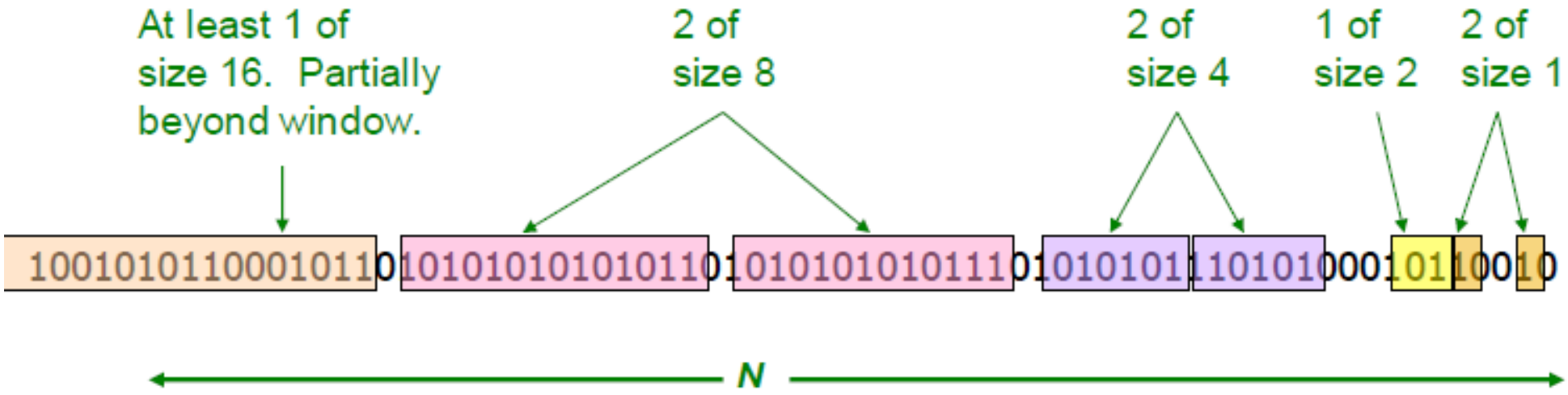


State of the buckets after merging



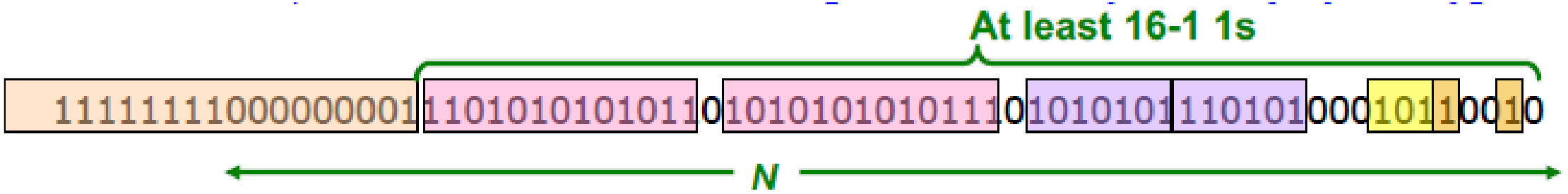
How to Query?

- **To estimate the number of 1s in the most recent *N* bits:**
  - **Sum the sizes of all buckets but the last (note "size" means the number of 1s in the bucket)**
  - **Add half the size of the last bucket (**we can also do other approximations**)**
- **Remember:** We do not know how many **1s** of the last bucket are still within the wanted window



Error Bound: Proof

- **Why is error at most 50%? Let's prove it!**
- Suppose the last bucket has size 2^r
- Then by assuming 2^{r-1} (i.e., half) of its **1s** are within the last bucket (but out of last-N bits), we make an error of at most $\pm 2^{r-1} - 1$
- 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 + .. +** $\mathbf{2^{r-1} = 2^r}$**-1**
- Thus, error at most **50%** **[=**$\mathbf{2^{r-1}/2^r}$**>** $\mathbf{(2^{r-1}-1)/(2^r-1)}$**]**

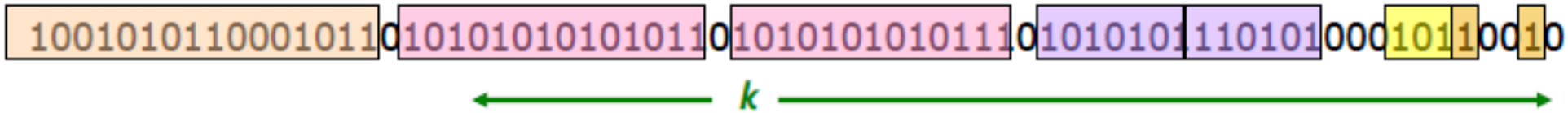


Further Reducing the Error

- 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*)**
  - see MMDS book for details
- By picking ***r*** appropriately, we can tradeoff between number of bits we store and the error

Extensions (Dynamic Window size)

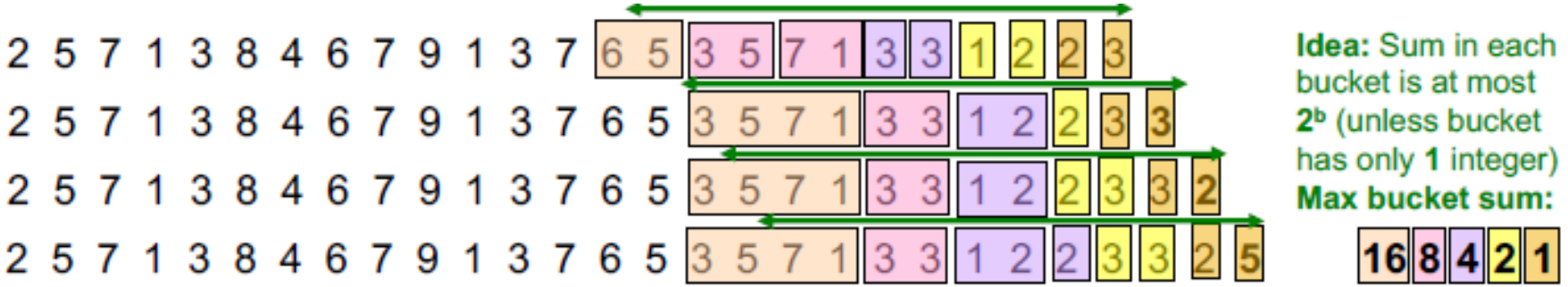
- Can we use the same trick to answer queries: **How many 1's in the last *k*?** where ***k* < *N***?
  - Find earliest bucket **B** that at overlaps with ***k***.
  - Number of **1s** is the **sum of sizes of more recent buckets + ½ size of B**



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

Extensions: Sum of last k elements

- **Stream of positive integers**
- **We want the sum of the last *k* elements**
  - **Application (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/stream
    - 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**



Counting Itemsets

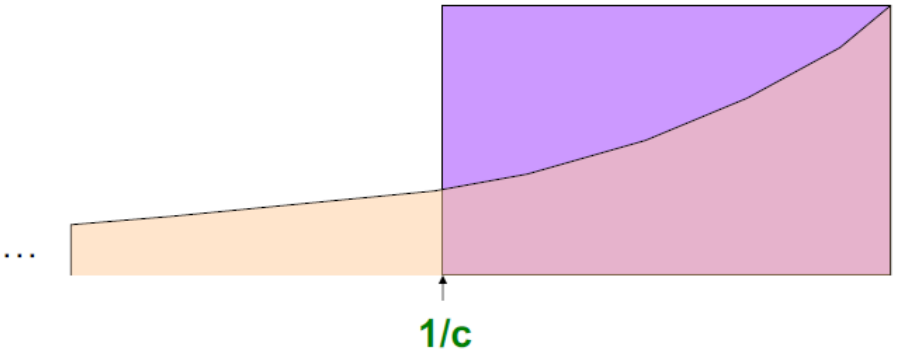
- **New Problem:** Given a stream, which items (or itemsets) appear more than *s* times in the window?
- **Possible solution:** Think of the stream of baskets as one binary stream per item/itemset
  - **1** = item/itemset present; **0** = not present
  - Use **DGIM** to estimate counts of **1**s for all items/itemsets
- **In principle, you could count frequent pairs or even larger sets**
- **Drawbacks:**
  - Only approximate counts are maintained
  - **Number of itemsets (all subsets of baskets/carts) is way too big!**

Exponentially Decaying Windows

- **Exponentially decaying windows: A heuristic for selecting likely frequent item(sets)**
  - **What are "currently" most popular items?**
    - Instead of computing the raw count in last ***N*** elements
    - Compute a **smooth aggregation** over the whole stream
- If stream is $a_1, a_2, \ldots$ and we are taking the sum of the stream, take the answer at time ***t*** to be: $S_1 = a_1$; $\mathrm{S_t} = \sum_{i=1}^{t} a_i (1-c)^{t-i}$
  - **c** is a constant, presumably tiny, like 10^{-6} or 10^{-9}
- **When new a_{t+1} arrives:**
  - Multiply current sum by **(1-c)** and add a_{t+1} ; $S_{t+1} = S_t(1-c) + a_{t+1}$

Example: Counting Items

- If each a_i is an "item" we can compute the **characteristic function** of each possible item ***x*** as an Exponentially Decaying Window
  - That is: $\sum_{i=1}^{t} \delta_i (1-c)^{t-1}$ where $\delta_i = 1$ if $a_i = x$ and 0 otherwise
- Imagine that for each item ***x*** we have a binary stream (**1** if ***x*** appears, **0** if ***x*** does not appear)
- ***We maintain the sum of each element***
- New item ***x*** arrives:
  - Multiply all counts by **(1-c)**
  - Add **+1** to count for element ***x***
- **Call this sum the "weight" of item *x***



Important property:

$$\sum_t (1-c)^t = \frac{1-(1-c)^t}{1-(1-c)} \sim \frac{1}{c}$$

Example: Counting items

- **What are "currently" most popular items?**
- **Suppose we want to find items of weight > ½**
  - **Important property:** Sum over all weights = **1/c** [$\because \frac{1}{c}(1-c)+1=\frac{1}{c}$]
- **Thus:**
  - There cannot be more than **2/c** items with weight of **½** or more
- Drop items with count less than 1/2
- So, **2/c** is a limit on the number of items being counted at any time

Suppose c=0.1
Current Counts: $\{S_x = 5, S_y = 2, S_z = 1, S_a = 0.8, S_b = 0.7, S_c = 0.5\}$
An item d comes next
Updated Counts: $\{S_x = 4.5, S_y = 1.8, S_d = 1, S_z = 0.9, S_a = 0.72, S_b = 0.63\}$
S_c $is\ dropped$!

Extension to Itemsets

- **Count (some) itemsets in an E.D.W.**
  - **What are currently "hot" itemsets?**
  - **Problem:** Too many itemsets to keep counts of all of them in memory
  - Count every subset: One basket of **20** items would initiate **1M** counts (2^20)
- **When a basket B comes in:**
  - Multiply all counts by **(1-c)**
  - For uncounted items in **B**, create new count
  - Add **1** to count of any item in **B** and to any **itemset** contained in **B** that is already being counted
  - **Drop counts < ½**
  - Initiate new counts for itemsets **only under certain condition** (next slide)

Initiation of New Counts for itemsets

- Start a count for an itemset $S \subseteq B$ if every proper subset of S had a count prior to arrival of basket B
  - **Intuitively:** If all subsets of **S** are being counted this means they are "**frequent/hot**" and thus S has a potential to be "**hot**"
- **Example:**
  - Start counting S**={i, j}** iff both **i** and **j** were counted prior to seeing B
  - Start counting S**={i, j, k}** iff **{i, j}**, **{i, k}**, and **{j, k}** were all counted prior to seeing B

Summary

- **Sampling a fixed proportion of a stream**
  - Sample size grows as the stream grows
- **Sampling a fixed-size sample**
  - Reservoir sampling
- **Counting the number of 1s in the last N elements**
  - Exponentially increasing windows
  - Extensions:
    - Number of 1s in any last k ($k < N$) elements
    - Sum of integers in the last N elements
- **Task: Which were the most popular recent items/itemsets?**
  - Can keep exponentially decaying counts for items and potentially larger itemsets
  - **Be conservative about starting counts of large sets**