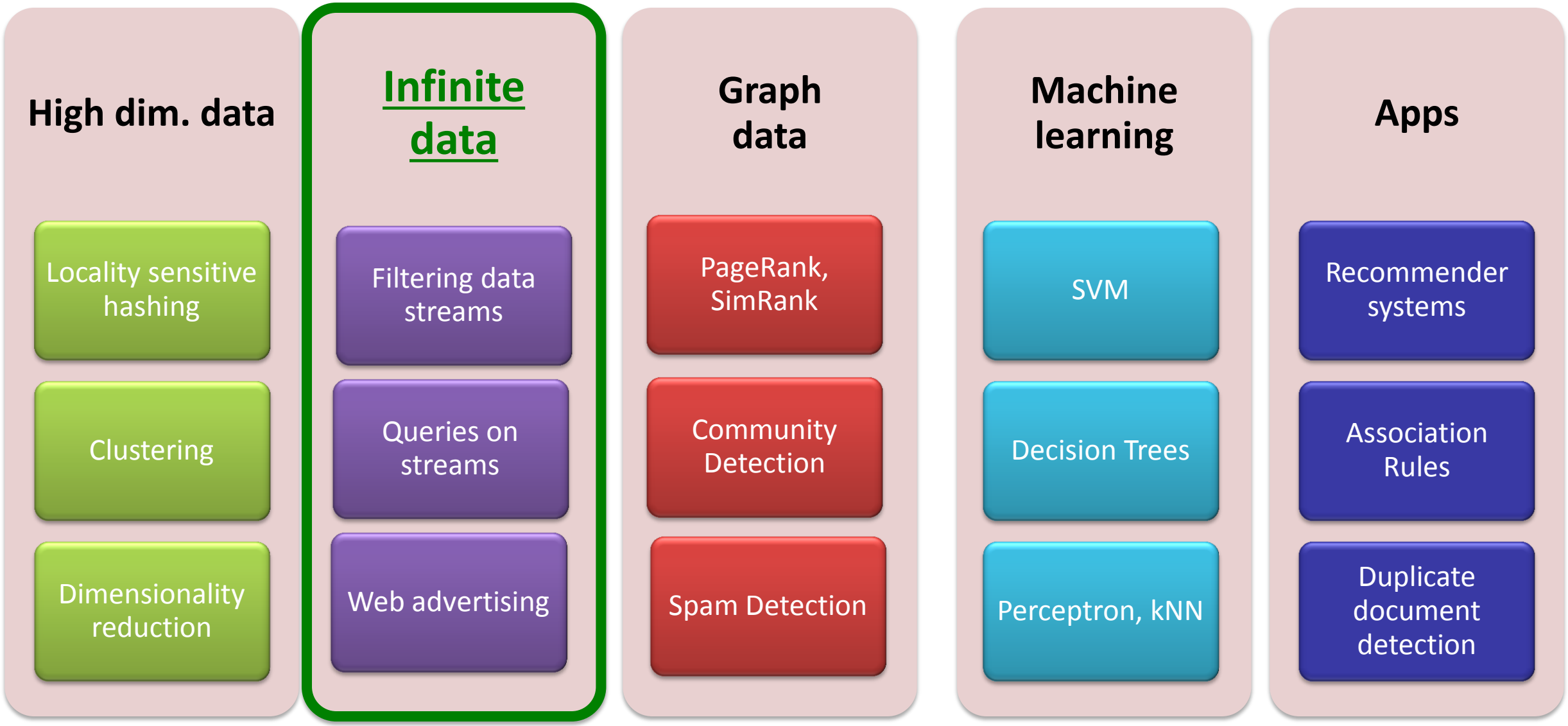
Data Stream Mining

New Topic: Infinite Data



Data Management Vs Stream Management

- In a DBMS, Input is under the control of the programming staff.
- Imagine a factory with 500 sensors capturing 10 KB of information every second, in one hour is captured nearby 36 GB of information and 432 GB daily. This massive information needs to be analysed in real time (or in the shortest time possible) to detect irregularities or deviations in the system and quickly react.
- In many data mining situations, we do not know the entire data set in advance. Therefore Stream Management is important when the input rate is controlled externally:

➢ Example: Google queries

Twitter or Facebook status updates

Stream Model

- Data arrives as sequence of items. Sometimes continuously and at high speed. Therefore We can think of the **data** as **infinite** and **non stationary** (the distribution changes over time)
- Input **elements** enter at a rapid rate, at one or more input ports (i.e., **streams**)
  - **We call elements of the stream tuples**
- Can't store them all in main memory.
- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

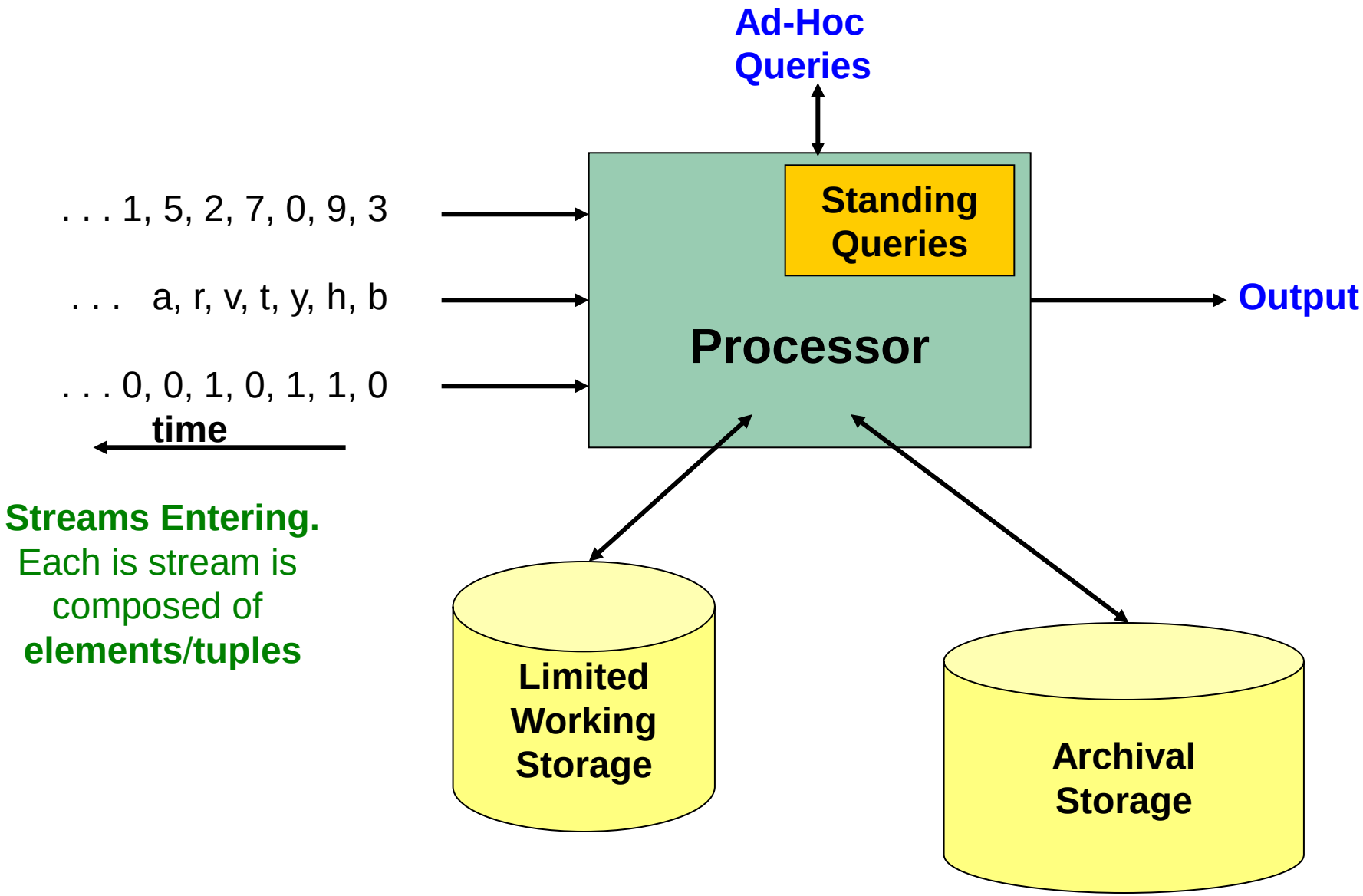
Stream Mining

- Stream Mining enables the analysis of massive quantities of data in real time using bounded resources
- Data Stream Mining is the process of extracting knowledge from continuous rapid data records which comes to the system in a stream. A data stream is an ordered sequence of instances that in many applications of data stream mining can be read only once or a small number of times using limited computing and storage capabilities.
- Data Stream Mining fulfil the following characteristics:
- **Continuous Stream of Data**. High amount of data in an infinite stream. we do not know the entire dataset
- **Concept Drifting**. The data change or evolves over time
- **Volatility of data**. The system does not store the data received (Limited resources). When data is analysed it’s discarded or summarised

Two Forms of Query

- **Ad-hoc queries:** Normal queries asked one time about streams.
  - Example: What is maximum value seen so far in stream S?
- **Standing queries:** Queries that are, in principle, asked about the stream at all times.
  - Example: Report each new maximum value ever seen in stream S.

General Stream Processing Model



Streams Entering.
Each is stream is composed of **elements/tuples**

Applications

- **Mining query streams:** Google wants to know what queries are more frequent today than yesterday
- **Mining click streams:** Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- **Mining social network news feeds:** E.g., look for trending topics on Twitter, Facebook
- **Sensor Networks :** Many sensors feeding into a central controller
- **Telephone call records** : Data feeds into customer bills as well as settlements between telephone companies
- **IP packets monitored at a switch:** Gather information for optimal routing, Detect denial-of-service attacks

Applications

- Examples of data streams include
  - computer network traffic,
  - phone conversations,
  - ATM transactions,
  - web searches,
  - sensor data.
- Data stream mining can be considered a subfield of data mining, machine learning, and knowledge discovery.

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
      - **What is the property of the sample we want to maintain?**
        For all time steps ***k***, each of ***k*** elements seen so far has equal prob. of being sampled

Sampling a Fixed Proportion

- **Problem 1: Sampling 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 days**
  - 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

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?** Don't know length of stream in advance
- **Suppose at time *n* we have seen *n* items**
  - **Each item is in the sample *S* with equal prob. *s/n***

How to think about the problem: say s = 2
Stream: a x c y z k c d e g…
At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.
At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.
Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random

Solution: Fixed Size Sample

- Problem: Maintain a uniform sample x from a stream of unknown length.
- **Algorithm (a.k.a. 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
    - If we picked the n^{th} element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the ***nth*** element arrives (***n > s***)
    - With probability ***s/n***, keep the ***nth*** element, else discard it
    - If we picked the ***nth*** element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random
- **Claim:** This algorithm maintains a sample ***S*** with the desired property:
  - After ***n*** elements, the sample contains each element seen so far with probability ***s/n***

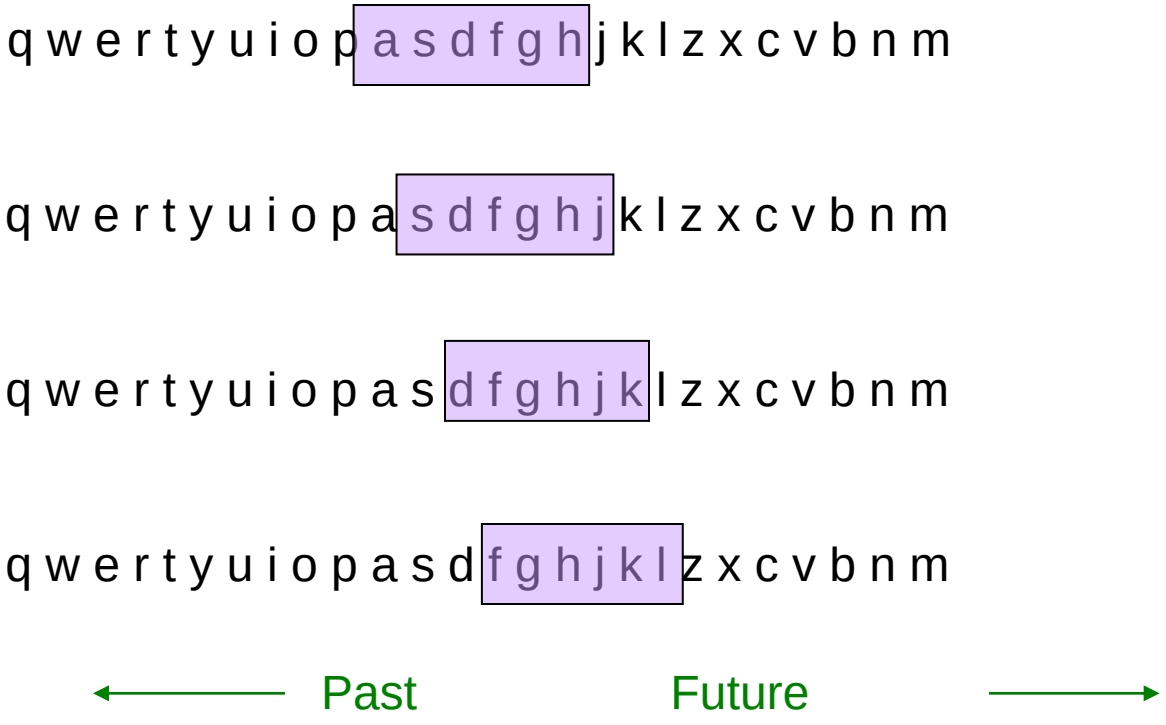
Sliding Windows

- 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

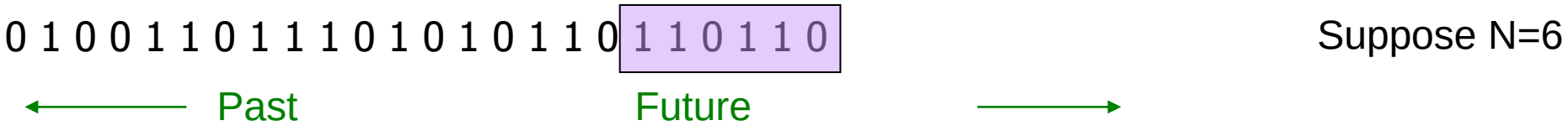
- **Sliding window on a single stream:**

N = 6



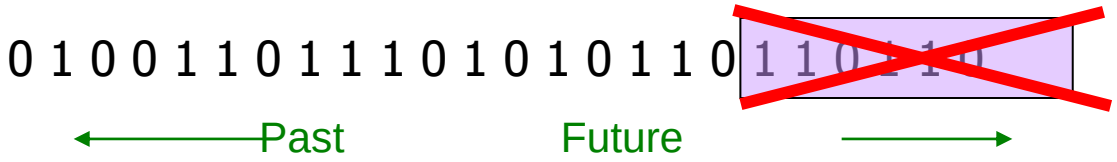
Counting Bits (1)

- **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



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**



- **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**

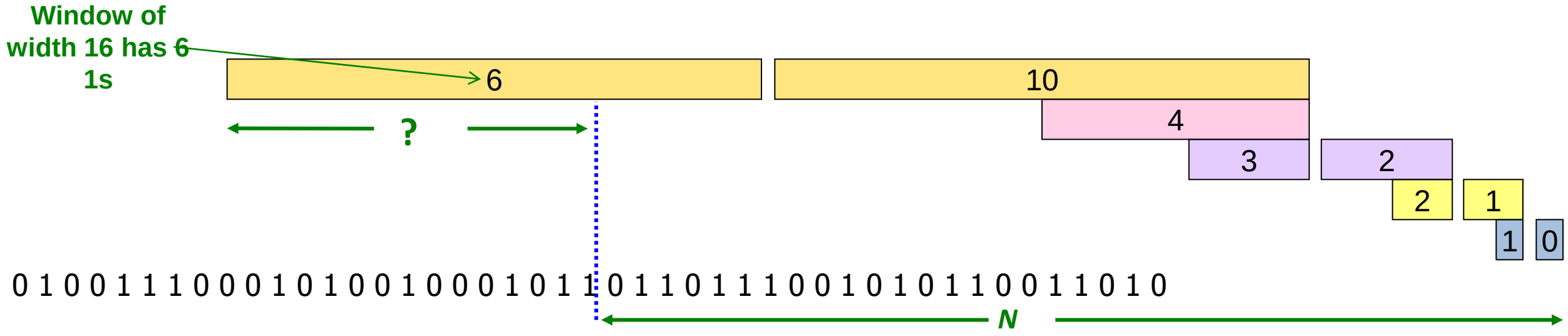
- **Maintain 2 counters:**
  - ***S***: number of 1s from the beginning of the stream
  - ***Z***: number of 0s from the beginning of the stream
- **How many 1s are in the last N bits?** $N \cdot \frac{S}{S+Z}$
- **But, what if stream is non-uniform?**
  - What if distribution changes over time?

DGIM Method

- **DGIM solution that does <u>not</u> assume uniformity**
- We store $\boldsymbol{O}(\log^2 \boldsymbol{N})$ 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



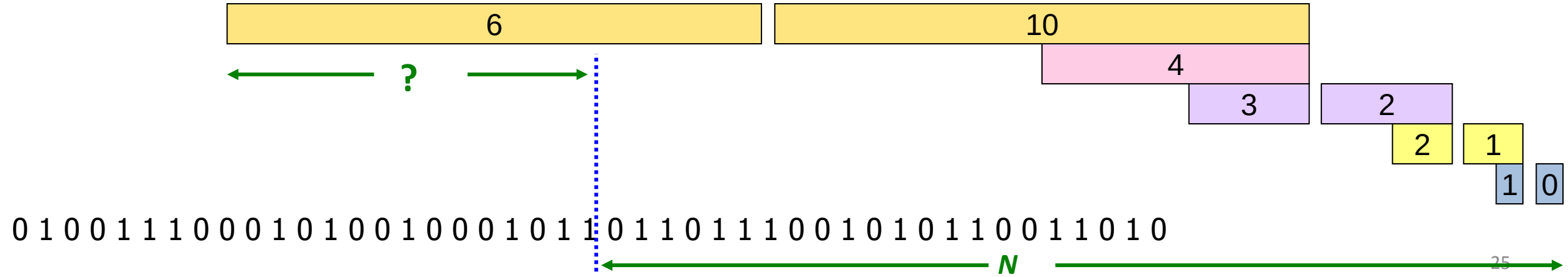
We can reconstruct the count of the last ***N*** bits, except we are not sure how many of the last **6 1s** are included in the ***N***

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?

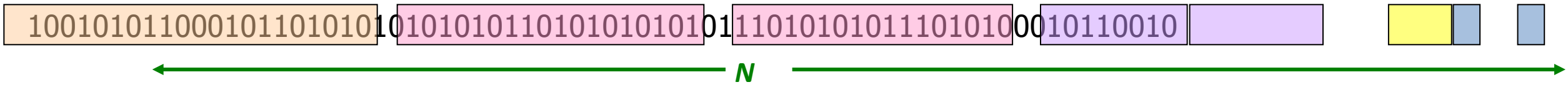
- As long as the **1s** are fairly evenly distributed, the error due to the unknown region is small – **no more than 50%**
- But it could be that all the **1s** are in the unknown area at the end
- In that case, **the error is unbounded!**



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**



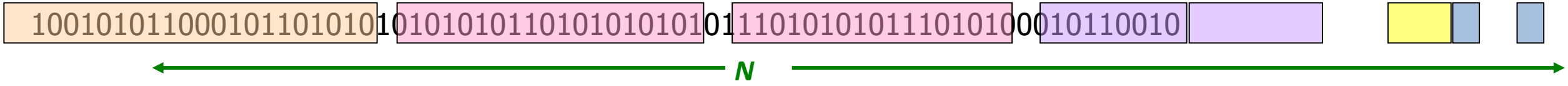
DGIM: Timestamps

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

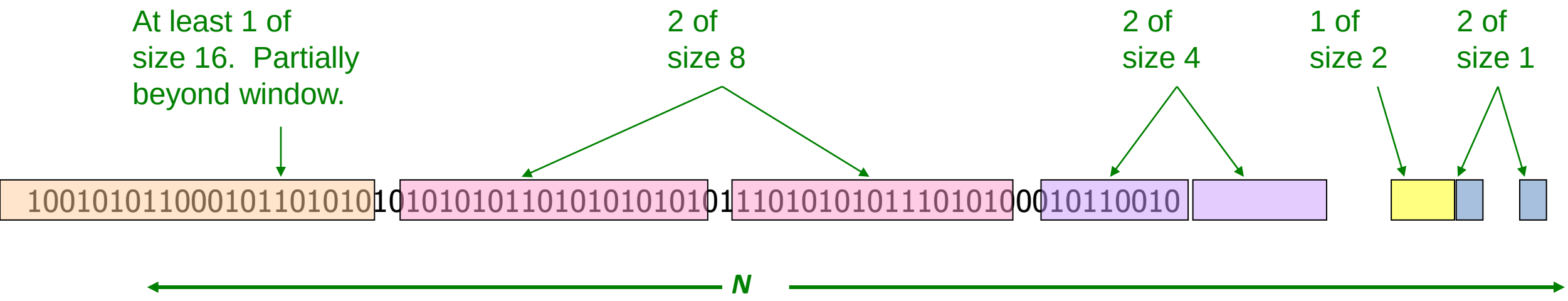
- **Constraint on buckets:**
  Number of **1s** must be a power of **2**
  - That explains the **O(log log *N)*** in **(B) above**



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)

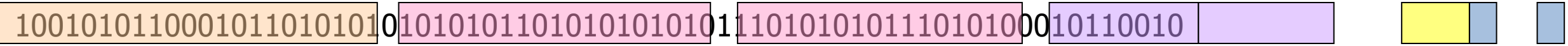
- When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to ***N*** time units before the current time
- **2 cases:** Current bit is **0** or **1**
- **If the current bit is 0:**
  no other changes are needed

Updating Buckets (2)

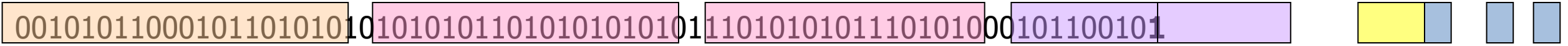
- **If the current bit is 1:**
  - **(1)** Create a new bucket of size **1**, for just this bit
    - **End timestamp = current time**
  - **(2)** If there are now **three buckets of size 1**, **combine the oldest two into a bucket of size 2**
  - **(3)** If there are now **three buckets of size 2**,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

Example: Updating Buckets

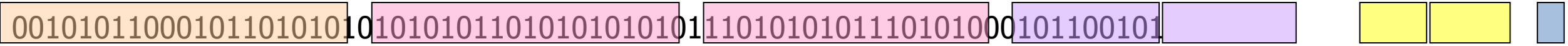
Current state of the stream:



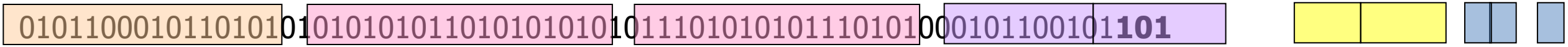
Bit of value 1 arrives



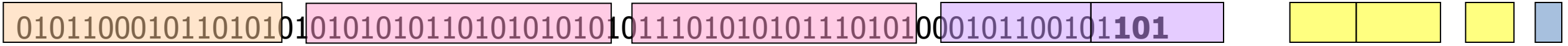
Two orange buckets get merged into a yellow bucket



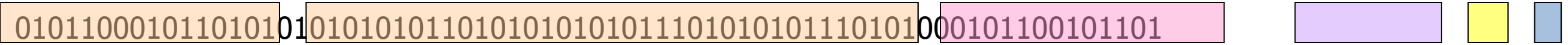
Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:



Buckets get merged…

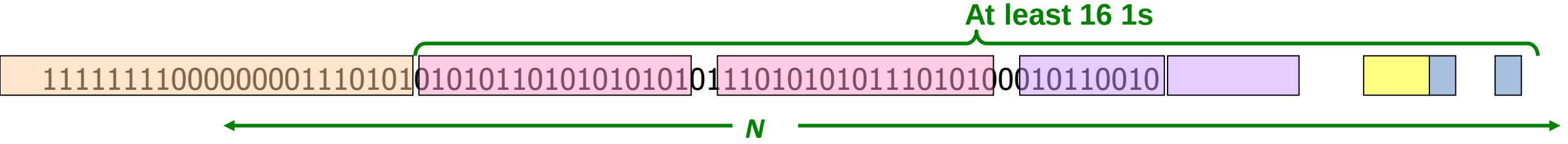


State of the buckets after merging



Error Bound: Proof

- **Why is error 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 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%**



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
    - Sums of integers in the last N elements

Mining Data Streams (Part 2)

More algorithms for streams

- **More algorithms for streams:**
  - **(1) Filtering a data stream: Bloom filters**
    - Select elements with property **x** from stream
  - **(2) Counting distinct elements: Flajolet-Martin**
    - Number of distinct elements in the last ***k*** elements of the stream
  - **(3) Estimating moments: AMS method**
    - Estimate std. dev. of last ***k*** elements
  - **(4) Counting frequent items**

Filtering Data Streams

- **Each element of data stream is a tuple**
- Given a list of keys **S**
- **Determine which tuples of stream are in *S***

- **Obvious solution: Hash table**
  - But suppose we **do not have enough memory** to store all of ***S*** in a hash table
    - E.g., we might be processing millions of filters on the same stream

Applications

- **Example: Email spam filtering**
  - We know 1 billion “good” email addresses
  - If an email comes from one of these, it is **NOT** spam
- **Publish-subscribe systems**
  - You are collecting lots of messages (news articles)
  - People express interest in certain sets of keywords
  - Determine whether each message matches user’s interest

Counting Distinct Elements

Counting Distinct Elements

- **Problem:**
  - Data stream consists of a universe of elements chosen from a set of size ***N***
  - Maintain a count of the number of distinct elements seen so far
- **Obvious approach:**
  Maintain the set of elements seen so far
  - That is, keep a hash table of all the distinct elements seen so far

Applications

- **How many different words are found among the Web pages being crawled at a site?**
  - Unusually low or high numbers could indicate artificial pages (spam?)
- **How many different Web pages does each customer request in a week?**
- **How many distinct products have we sold in the last week?**

Using Small Storage

- Real problem: What if we do not have space to maintain the set of elements seen so far?
- Estimate the count in an unbiased way
- Accept that the count may have a little error, but limit the probability that the error is large

Flajolet-Martin Approach

- Pick a hash function ***h*** that maps each of the ***N*** elements to at least $\log_2 N$ bits

- For each stream element ***a***, let ***r(a)*** be the number of trailing **0s** in ***h(a)***
  - **r(a)** = position of first 1 counting from the right
    - E.g., say ***h(a) = 12***, then ***12*** is ***1100*** in binary, so ***r(a) = 2***
- Record ***R* = the maximum *r*(*a*) seen**
  - **R = max$_a$ r(a)**, over all the items ***a*** seen so far

- **Estimated number of distinct elements = 2^R**

Thank you.