Mining Data Streams

Big Data Analytics CSCI 4030

New Topic: Infinite Data

High dim. data	Graph data	**Infinite data**	Machine learning	Apps
Locality sensitive hashing	PageRank, SimRank	Filtering data streams	SVM	Recommender systems
Clustering	Community Detection	Queries on streams	Decision Trees	Association Rules
Dimensionality reduction	Spam Detection	Web advertising	Perceptron, kNN	Duplicate document detection

Data Streams

- **In many data mining situations, we do not know the entire data set in advance**

- **Stream Management** is important when the input rate is controlled **externally:**
  - Google queries
  - Twitter or Facebook status updates
- We can think of the **data** as **infinite** and **non-stationary** (the distribution changes over time)

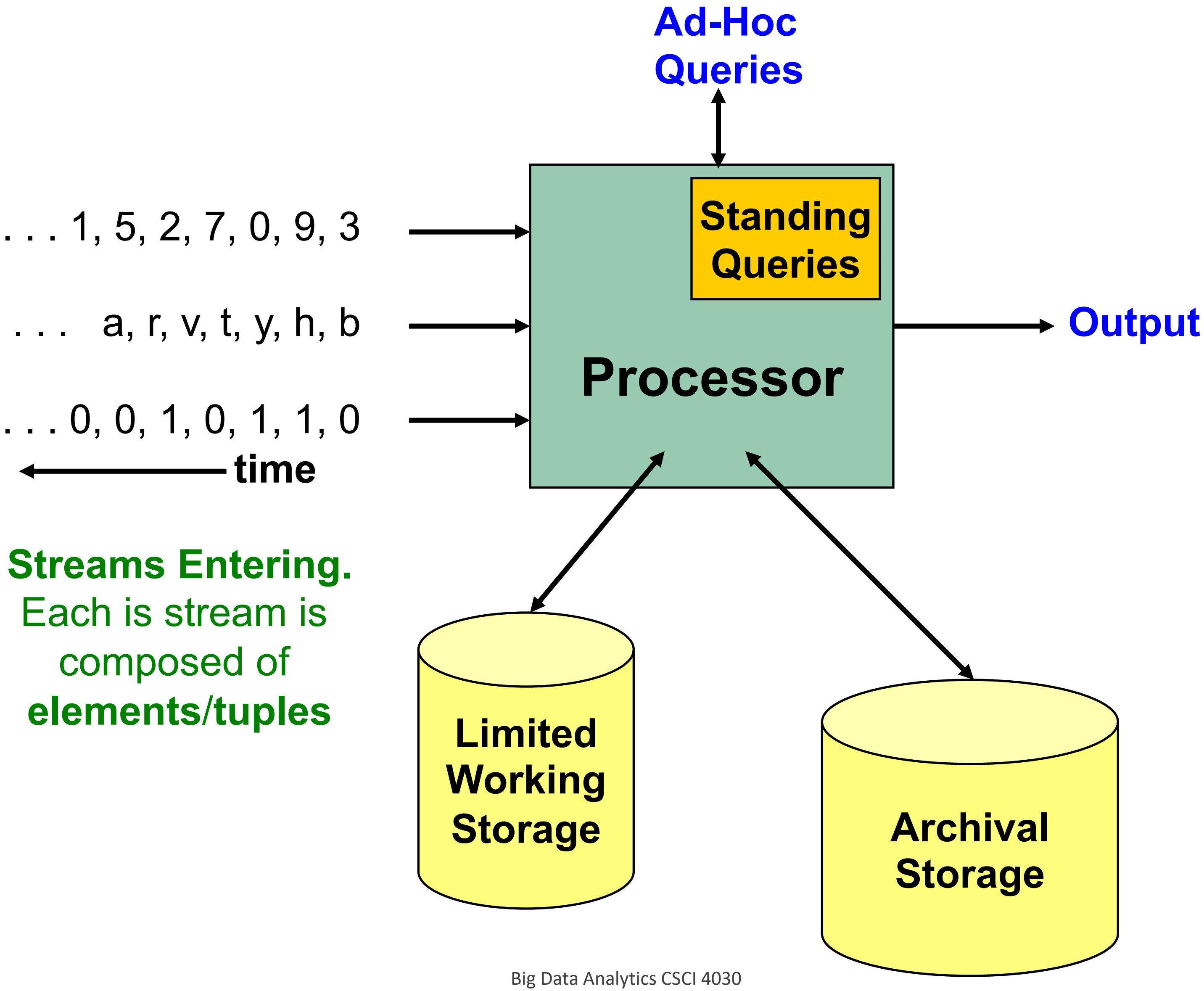
The Stream Model

- 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 memory?**

General Stream Processing Model



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

Side note: SGD is a Streaming Alg.

- **Stochastic Gradient Descent (SGD) is an example of a stream algorithm**
- **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
- **Idea: Do slow updates to the model**
  - **SGD** (SVM, Perceptron) makes small updates
  - **So:** First train the classifier on training data.
  - **Then:** For every example from the stream, we slightly update the model (using small learning rate)

Problems on Data Streams

- **Types of queries one wants on answer on a data stream:**
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows**
    - Number of items of type ***x*** in the last ***k*** elements of the stream

Problems on Data Streams

- **Types of queries one wants answer on a data 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
  - **Finding frequent elements**

Applications (1)

- **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

Applications (2)

- **Sensor Networks**
  - Many sensors feeding into a central controller
- **Telephone call records**
  - Data feeds into customer bills
- **IP packets monitored at a switch**
  - Gather information for optimal routing
  - Detect denial-of-service attacks

Sampling from a Data Stream: Sampling a fixed proportion

As the stream grows the sample also gets bigger

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?**

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 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

Problem with Naïve Approach

- Simple question: **What fraction of queries by a search engine average user are duplicates?**
  - Suppose each user issues ***x*** queries once and ***d*** queries twice (total of ***x*+2*d*** queries)
    - **Correct answer:** ***d*/(*x*+*d*)**
  - **Proposed solution: We keep 10% of the queries**
  - **The sample-based answer is..** $\frac{d}{10x+19d}$

Solution: Sample Users

Solution:

- Pick **1/10th** of **users** and take all their searches in the sample
- Use a hash function that hashes the user name or user id uniformly into 10 buckets
  - consider users that only hash only 1st bucket

Sampling from a Data Stream: Sampling a fixed-size sample

As the stream grows, the sample is of fixed size

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

- 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 ***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
- **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***

Queries over a (long) Sliding Window

Sliding Windows

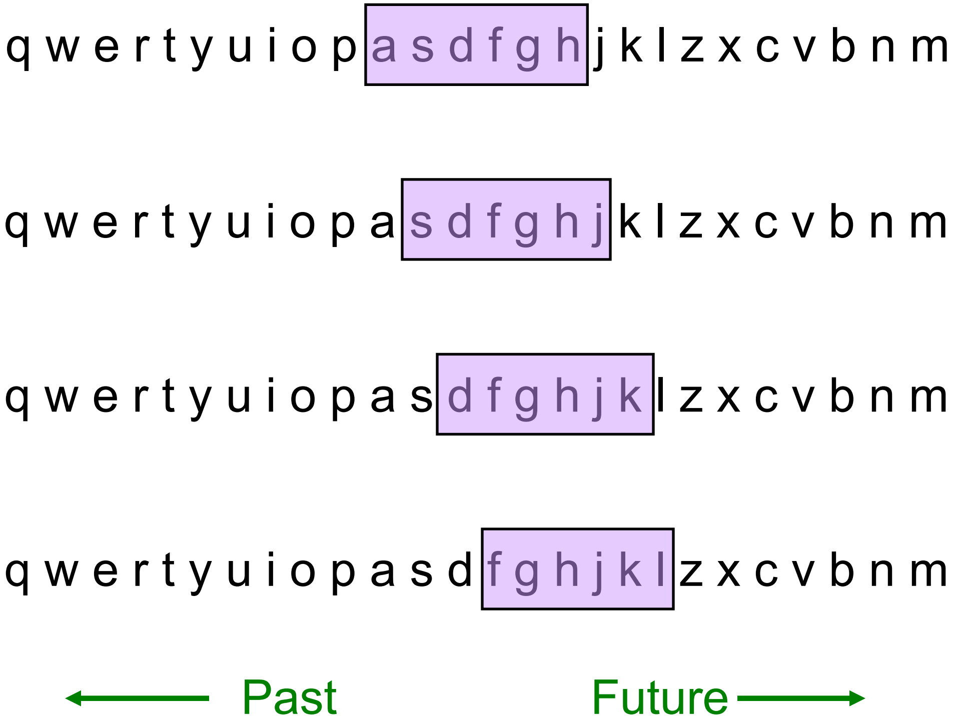
- A useful model of stream processing is that queries are about a ***window*** of length ***N***: the ***N*** most recent elements received

- **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

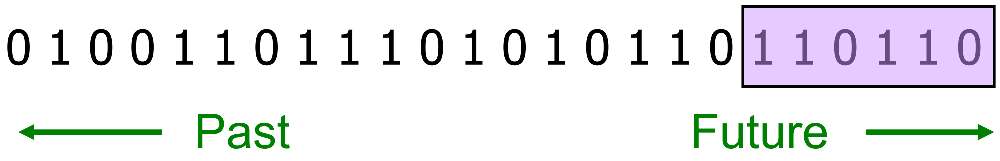
q w e r t y u i o p a s d f g h j k l z x c v b n m

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

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*** ≤ ***N***

- **Obvious solution:**
  Store the most recent ***N*** bits
  - When new bit comes in, discard the ***N*+1st** 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

← Past

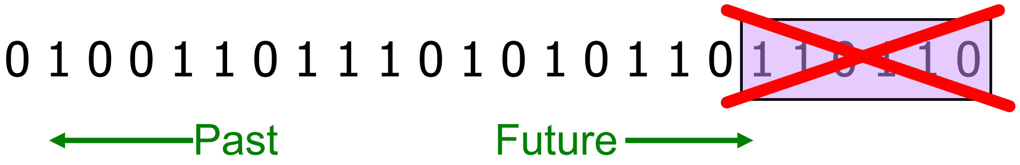
Future →

Suppose N=6

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~~

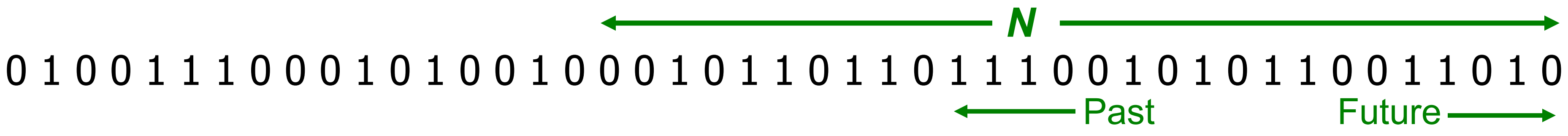
Past

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**



- **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?

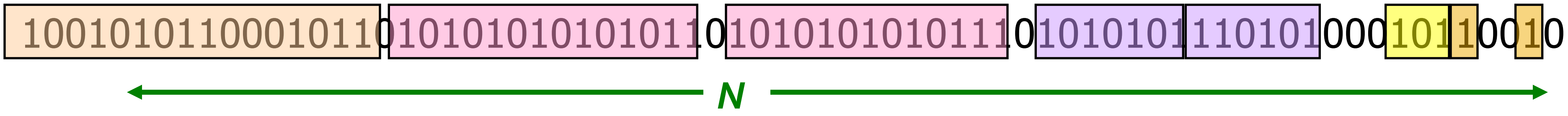
DGIM Method

- **DGIM solution that does not 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

DGIM Method

- **Idea:** 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

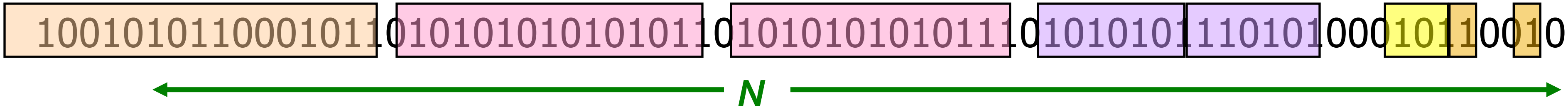
- Each bit in the stream has a ***timestamp***, starting **1**, **2,** …

- Record timestamps modulo ***N*** (**the window size**)

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]

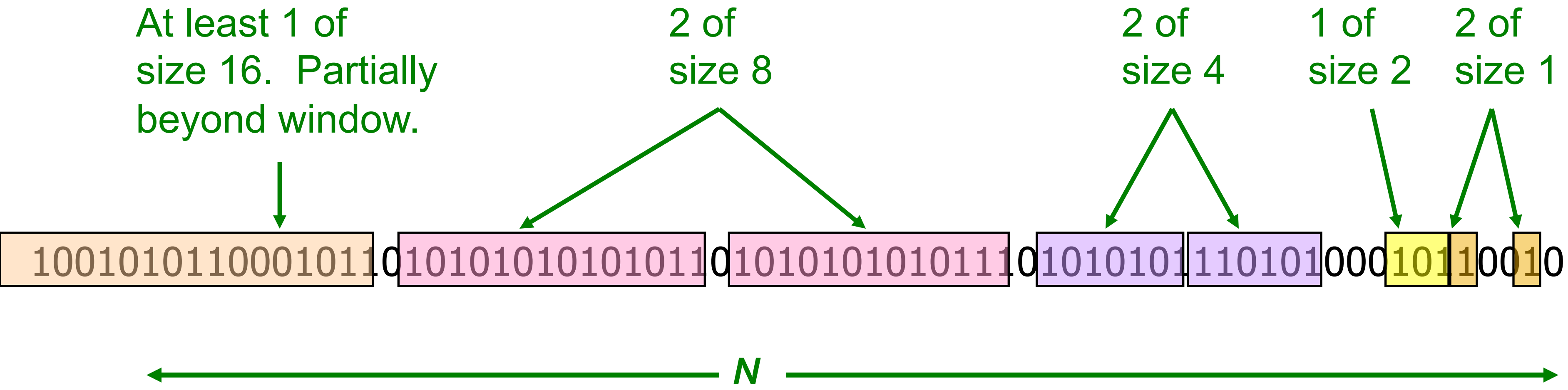
- **Constraint on buckets:**
  Number of **1s** must be a power of **2**
  - That explains the **O(log log *N*)** in **(B) above**



Representing a Stream by Buckets

- The **right end** of a bucket is always a position with a 1.
- 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



At least 1 of size 16. Partially beyond window.

2 of size 8

2 of size 4

1 of size 2

2 of size 1

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**

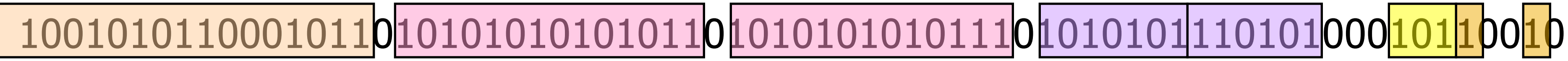
- **If the current bit is 0:**
  no other changes are needed

Updating Buckets (2)

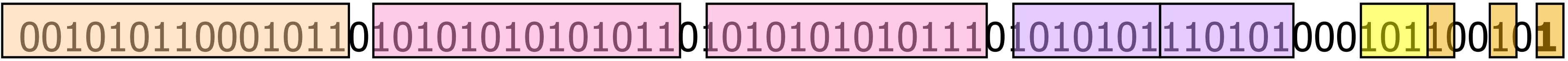
- **If the current bit is 1:**
  - **(1)** Create a new bucket of size **1**, for just this bit
    - **End timestamp = current time**
  - **(2)** If there are now **three buckets of size 1**, **combine the oldest two into a bucket of size 2**
  - **(3)** If there are now **three buckets of size 2**,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

Example: Updating Buckets

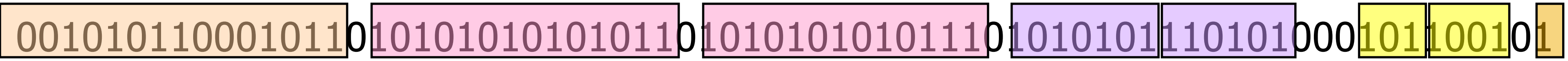
Current state of the stream:


1001010110001011010101010101010110101010101011101010101110101000101100 10

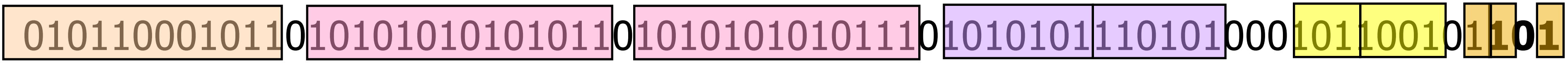
Bit of value 1 arrives


0010101100010110101010101010101101010101010101110101010111010100010110010 1

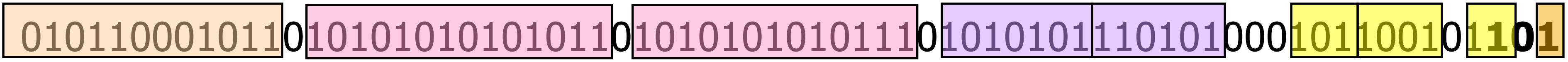
Two orange buckets get merged into a yellow bucket


00101011000101101010101010101011010101010101011101010101110101000101100101

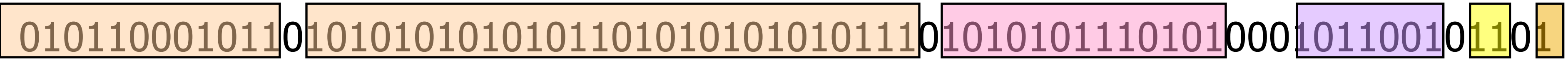
Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:


010110001011010101010101010110101010101010111010101011101010001011001011 01

Buckets get merged…


010110001011010101010101010110101010101010111010101011101010001011001011 01

State of the buckets after all merging..


0101100010110101010101010101101010101010101110101010111010100010110010 1101

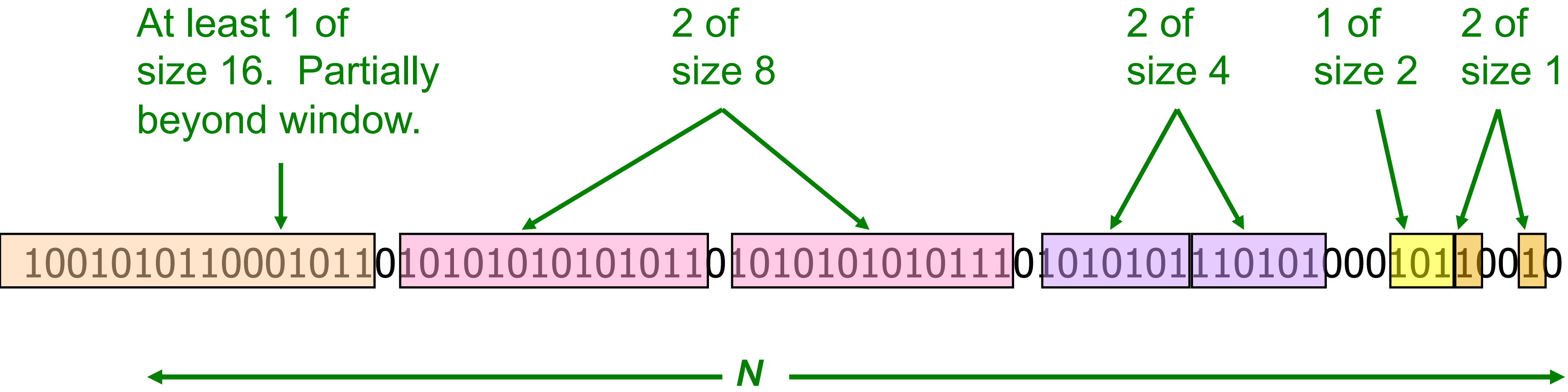
How to Query?

- **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



At least 1 of size 16. Partially beyond window.

2 of size 8

2 of size 4

1 of size 2

2 of size 1

10010101100010110101010101010110101010101011110101010111010100010110010

N

2x1 + 1x2 + 2x4 + 2x8 + 16/2 = 36

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**
  - DGIM

Next

- More algorithms for streams:
  - (1) Filtering a data stream: Bloom filters
    - Select elements with property **x** from stream
  - (2) Decaying Windows

(1) Filtering Data Streams

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

Applications

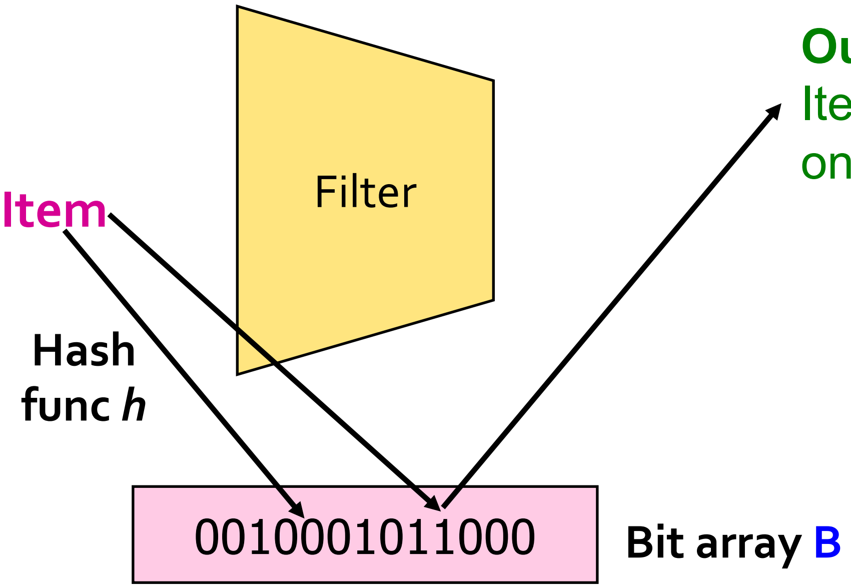
- **Example: Email spam filtering**
  - We know 1 billion “good” email addresses
  - If an email comes from one of these, it is **NOT** spam

- **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

First Cut Solution (1)

- **Given a set of keys *S* that we want to filter**
- Create a **bit array *B*** of ***n*** bits, initially all ***0*s**
- Choose a **hash function *h*** with range **[*0,n*)**
- Hash each member of $s \in S$ to one of ***n*** buckets, and set that bit to **1**, i.e., ***B[h(s)]=1***
- Hash each element ***a*** of the stream and output only those that hash to bit that was set to **1**
  - **Output *a* if B[h(a)] == 1**

First Cut Solution (2)



Output the item since it may be in *S*.
Item hashes to a bucket that at least one of the items in **S** hashed to.

Drop the item.
It hashes to a bucket set to **0** so it is surely not in ***S***.

- **Creates false positives but no false negatives**
  - If the item is in ***S*** we surely output it, if not we may still output it

First Cut Solution (3)

- **|S| = 1 billion email addresses**
  |B|= 1GB = 8 billion bits

- If the email address is in ***S***, then it surely hashes to a bucket that has the bit set to **1**, so it always gets through (***no false negatives***)

- Approximately **1/8** of the bits are set to **1**, so about **1/8th** of the addresses not in ***S*** get through to the output (***false positives***)

Bloom Filter

- Consider: **|S| = *m*, |B| = *n***
- Use ***k*** independent hash functions $h_1, \dots, h_k$
- **Initialization:**
  - Set **B** to all **0s**
  - Hash each element $s \in S$ using each hash function h_i, set **B[$h_i(s)$] = 1** (for each ***i* = *1,.., k***)

(**note:** we have a single array B!)

- **Run-time:**
  - When a stream element with key ***x*** arrives
    - If **B[$h_i(x)$] = 1** for all ***i* = 1,..., *k*** then declare that ***x*** is in ***S***
      - That is, ***x*** hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element ***x***

Bloom Filter: Wrap-up

- **Bloom filters guarantee no false negatives, and use limited memory**
  - Great for pre-processing before more expensive checks
- **Suitable for parallelization**
  - Hash function computations can be parallelized

Exponentially Decaying Windows

Exponentially Decaying Windows

- **Exponentially decaying windows: A heuristic for selecting likely frequent item(sets)**
  - **What are “currently” most popular movies?**
    - Instead of computing the raw count in last ***N*** elements
    - Compute a **smooth aggregation** over the whole stream
- If stream is $\boldsymbol{a_1, a_2, \ldots}$ and we are taking the sum of the stream, take the answer at time ***t*** to be:

$$= \sum_{i=1}^{t} a_i (1-c)^{t-i}$$

    - **c** is a constant, presumably tiny, like $\mathbf{10^{-6}}$ or $\mathbf{10^{-9}}$
- **When new a_{t+1} arrives:**
  Multiply current sum by **(1-c)** and add $\mathbf{a_{t+1}}$

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: $\sum_{i=1}^{t} \delta_i \cdot (1-c)^{t-i}$

    tiny, like 10^{-6} or 10^{-9}

    where **δ_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)
  - 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***

Summary

- **The Stream Data Model**: This model assumes data arrives at a processing engine at a rate that makes it infeasible to store everything in active storage.
- One strategy to dealing with streams is to maintain summaries of the streams, sufficient to answer the expected queries about the data.
- A second approach is to maintain a sliding window of the most recently arrived data.

Summary

- **Sampling of Streams**: To create a sample of a stream that is usable for a class of queries, we identify a set of key attributes for the stream.
- By hashing the key of any arriving stream element, we can use the hash value to decide consistently whether all or none of the elements with that key will become part of the sample.

Summary

- **Bloom Filters**: This technique allows us to filter streams so elements that belong to a particular set are allowed through, while most nonmembers are deleted.
- We use a large bit array, and several hash functions. Members of the selected set are hashed to buckets, which are bits in the array, and those bits are set to 1.
- To test a stream element for membership, we hash the element to a set of bits using each of the hash functions, and only accept the element if all these bits are 1.

Summary

- **Estimating the Number of 1's in a Window**: We can estimate the number of 1's in a window of 0's and 1's by grouping the 1's into buckets.
- Each bucket has a number of 1's that is a power of 2; there are one or two buckets of each size, and sizes never decrease as we go back in time.
- If we record only the position and size of the buckets, we can represent the contents of a window of size N with O(log^2N) space.

Summary

- **Answering Queries About Numbers of 1’s**: If we want to know the approximate numbers of 1’s in the most recent k elements of a binary stream
  - we find the earliest bucket B that is at least partially within the last k positions of the window
  - and estimate the number of 1’s to be the sum of the sizes of each of the more recent buckets plus half the size of B.
  - This estimate can never be off by more than 50% of the true count of 1’s.

Summary

- **Closer Approximations to the Number of 1’s**: By changing the rule for how many buckets of a given size can exist in the representation of a binary window
  - so that either r or r −1 of a given size may exist
  - we can assure that the approximation to the true number of 1’s is never off by more than 1/r.

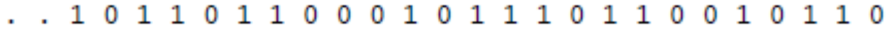
Summary

- **Exponentially Decaying Windows**: Rather than fixing a window size, we can imagine that the window consists of all the elements that ever arrived in the stream
  - but with the element that arrived t time units ago weighted by e^{-ct} for some time-constant c.

- Doing so allows us to maintain certain summaries of an exponentially decaying window easily.
  - For instance, the weighted sum of elements can be recomputed, when a new element arrives by multiplying the old sum by 1 – c and then adding the new element.

Quiz: DGIM Rules

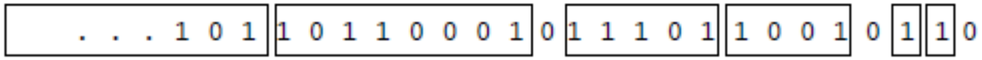
- Divide a following bit-stream into buckets following the DGIM rules.
  - Assume without the loss of generality there are two buckets of size 1 and one bucket of size 2.



```
. . 1 0 1 1 0 1 1 0 0 0 1 0 1 1 1 0 1 1 0 0 1 0 1 1 0
```

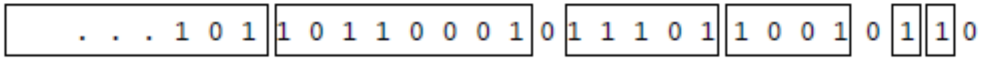
Quiz: Maintaining DGIM Conditions

- Suppose we start with buckets presented below and a 0 enters the stream. How the modified buckets will look like?



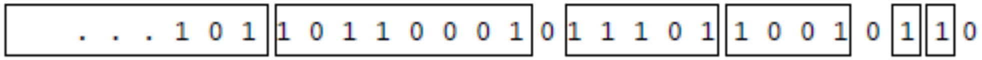
Quiz: Maintaining DGIM Conditions

- Suppose we start with buckets presented below and a 1 enters the stream. How the modified buckets will look like?



Quiz: Estimate number of 1's

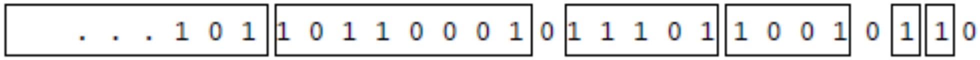
- Suppose the window is as shown below. Estimate the number of 1's for the last k positions, for *k* = 5.



. . . 1 0 1 1 0 1 1 0 0 0 1 0 1 1 1 0 1 1 0 0 1 0 1 1 0

Quiz: Estimate number of 1's

- Suppose the window is as shown below. Estimate the number of 1's for the last k positions, for *k* = 15.



Quiz: Bloom Filtering

- Assume
  - |S| = 1 billion email addresses
  - |B|= 2GB = 16 billion bits
  - Estimate the number of false positives in the bloom filtering by using a single hash function.

Quiz: Bloom Filtering

- Assume
  - |S| = 1 billion email addresses
  - |B|= 2GB = 16 billion bits
  - Estimate the number of false positives in the bloom filtering.
- Approximately 1/16 of the bits are set to 1, so about 1/16th of the addresses not in *S* get through to the output (false positives).

Quiz: Bloom Filtering

- How to improve the accuracy of the Bloom Filter?

Actions

- Review slides!
- Read Chapter 4 from course book.
  - You can find electronic version of the book on Blackboard.