Chapter 4

Mining Data Streams

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

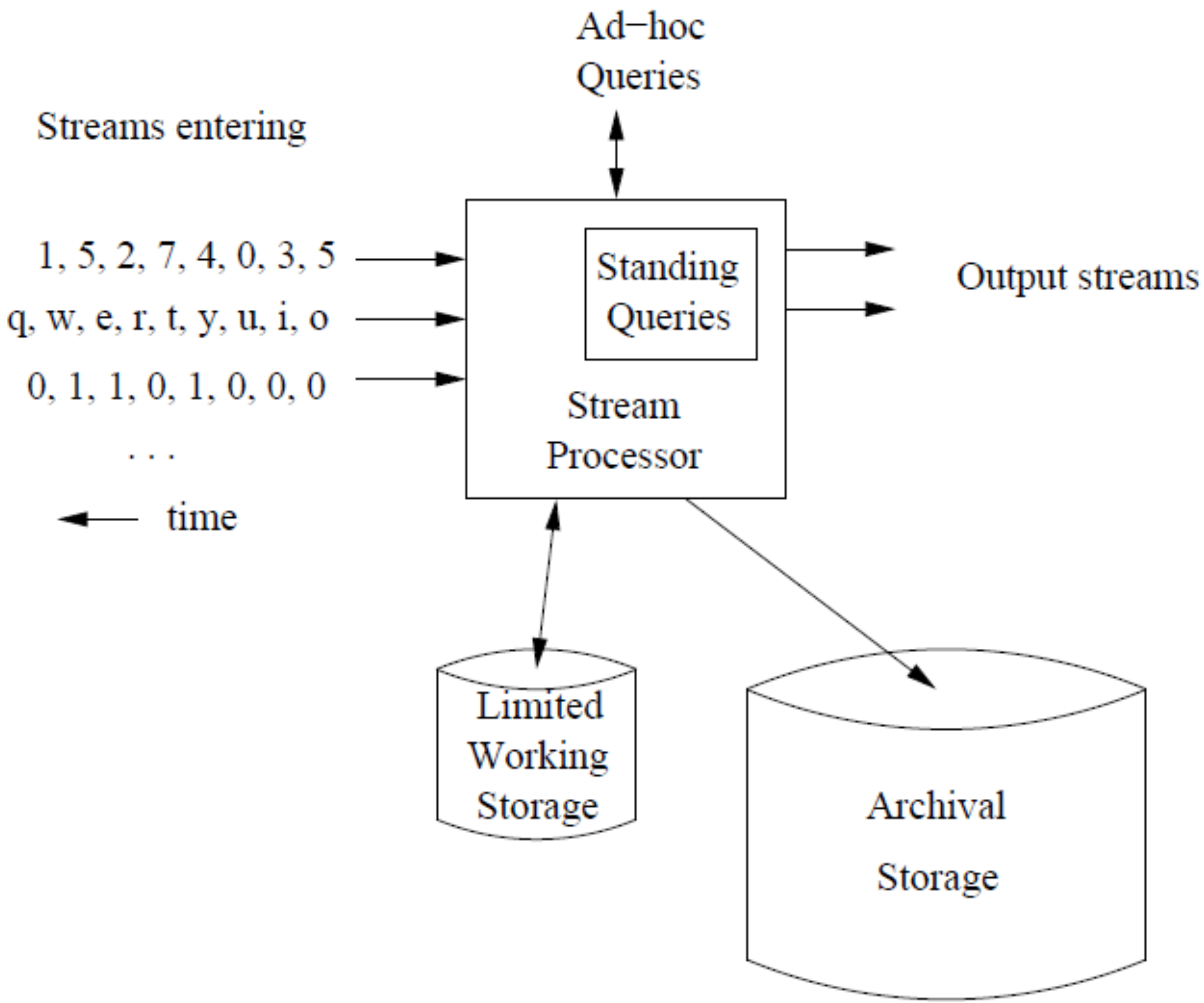
- We can think of the data as
  - ***Infinite***
  - ***non-stationary*** (i.e., the distribution changes over time)

Data Streams

- Data arrives in a ***stream*** or ***streams*** continuously
  - If it is not processed immediately or stored, then it is lost forever
  - The data arrives so rapidly that it is not feasible to store it all in active storage (e.g., in a conventional database)
- We consider ***summarization*** of a stream
  - How can we make a useful sample of a stream?
  - How can we filter a stream to eliminate undesirable elements?
  - How can we estimate the number of different elements in a stream?
- We also consider summarization of a stream using a ***window***
  - We look at only the last n elements of the stream

Data Stream Management System (DSMS)

- General architecture



Data Stream Model (1/2)

- 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

- Question
  - How do you make critical calculations about the stream using a ***limited*** amount of (secondary) memory?

Data Stream Model (2/2)

- Streams entering
  - They need not have the same date rates or data types
  - The time between elements of one stream need not be uniform
  - The arrival rate of a stream is not under the control of the system
- Archival storage (disk)
  - Streams may be archived in a large archival store
  - However, it is not possible to answer queries from the archival store
- Limited working storage (main memory or disk)
  - Summaries or parts of streams may be placed
  - It can be used for answering queries
  - It is of sufficiently limited capacity

Examples of Stream Sources

- Sensor data (temperature, GPS, height, etc.)
  - A sensor might send a reading every tenth of a second (3.5 MB/day)
  - There can be a million sensors (3.5 TB/day)

- Image data
  - Satellite (streams of many terabytes of images per day)
  - Surveillance cameras (streams of images at intervals like one second)

- Internet and Web traffic
  - A switching node receives streams of IP packets from many inputs
    - (ex) Detecting denial-of-service attacks or rerouting packets
  - Web sites receive streams of various types
    - (ex) Google (search queries), Facebook (clicks on its various links)

Query Type 1: Standing Queries

- Permanently execute and produce outputs at appropriate times
  - Also called "***continuous***" queries

- Examples
  - Output an alert whenever the temperature exceeds 25°C
    - Check only the most recent stream element
  - Each time a new reading arrives, produce the average of the 24 most recent readings
    - Store the 24 most recent stream elements in the working store
  - Produce the maximum temperature ever recorded by that sensor
    - Retain the maximum of all stream elements ever seen

Query Type 2: Ad-Hoc Queries (1/2)

- Asked once about the current state of a stream or streams
  - If we do not store all streams, we cannot answer arbitrary queries
  - If we have some idea of what kind of queries will be asked, we can prepare for them by storing ***appropriate parts*** or ***summaries*** of streams

- ***Sliding window***
  - A common approach to support ad-hoc queries
  - Two types
    - ***Tuple-based***: store the most recent n elements of a stream
    - ***Time-based***: store all the elements that arrived within the t time units
  - Then, we can treat the window as a relation and query it (e.g., SQL)
  - Of course, the stream management system must keep the window fresh
    - Deletes the oldest elements as new ones come in

Query Type 2: Ad-Hoc Queries (2/2)

- Example
  - Report the number of unique users over the past month

```
SELECT COUNT(DISTINCT(name))
FROM Logins
WHERE time >= t;
```

  - `Logins(name, time)`
    - A relation representing a window that is all logins in the most recent month
  - `t`
    - A constant representing the time one month before the current time

Issues in Stream Processing

- It is important that the stream-processing algorithm is executed in ***main memory***
  - Without or with only rare accesses to secondary storage
  - Because stream often deliver elements very rapidly

- However, the requirements of many streams can easily ***exceed*** the amount of available main memory

- Two generalizations about stream algorithms
  - Often, it is much more efficient to get an ***approximate answer*** than an exact solution
  - A variety of techniques related to ***hashing*** turn out to be useful to produce an approximate answer that is very close to the true result

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
  - 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
    - Estimate the average/standard deviation of last ***k*** elements
  - Finding frequent elements

Applications (1/2)

- 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
  - Look for trending topics on Twitter, Facebook

Applications (2/2)

- 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

Sampling Data in a Stream

Extracting reliable samples from a stream

Sampling Data in a Stream

- Why do we select a ***subset*** of a stream carefully?
  - We can ask queries about the selected subset and have the answers be ***statistically representative*** of the stream as a whole

- A motivating example
  - A search engine receives a stream of queries
    - Each query is in the form of $(user, query, time)$
  - Suppose we want to answer queries such as "what fraction of the typical user's queries were repeated over the past month?"
  - Assume also that we wish to store only **1/10th** of the stream elements

→ How should we select those **1/10th** of the stream elements?

Naïve Approach

- Generate a random integer from 0 to 9 for each query
- Store the tuple if the integer is 0, otherwise discard
- Then, each user has, on average, 1/10th of their queries stored
- However, this scheme gives us the ***wrong*** answer to some queries

Problem with Naïve Approach

- Consider the query asking for the fraction of repeated queries for a user
  - Suppose a user has issued s queries once and d queries twice, and no queries more than twice (a total of $s + 2d$ queries)
  - Correct answer = $d/(s + d)$

- If we use the naive approach (1/10th sample of queries)
  - Of the s queries, $s/10$ will appear in the sample
  - Of the d queries, only $d/100$ will appear ***twice*** in the sample
    - $d/100 = 1/10 \cdot 1/10 \cdot d$
  - Of the d queries, $18d/100$ will appear exactly ***once*** in the sample
    - $18d/100 = ((1/10 \cdot 9/10) + (9/10 \cdot 1/10)) \cdot d$
  - Thus, the answer = $\dfrac{d/100}{s/10 + d/100 + 18d/100} = \dfrac{d}{10s + 19d}$ (***wrong!***)

Obtaining a Representative Sample

- The example query ***can't*** be answered by taking 1/10th of queries
- Instead, we pick 1/10th of the ***users*** and take all their searches for the sample
  - Each time a query arrives, we hash the user into one of ten buckets (0…9)
  - If the user hashes to bucket 0, then we add the query to the sample, otherwise not
  - Note that we do not actually store the user in the bucket
- General way of obtaining a sample of a/b of the users
  - Hash users to b buckets ($0, 1, \ldots, b-1$)
  - Add the query to the sample if the hash value is less than a

General Sampling Problem

- Our stream consists of tuples with n components
  - Key: a subset of components used for selecting the sample
    - (ex) In the previous example, a tuple is ($user$, $query$, $time$) and the key is $user$
  - The choice of a key depends on the application
    - (ex) $user$, $query$, or ($user$, $search$, $time$)

- To take a sample of size a/b
  - Hash the key value for each tuple to b buckets
  - Accept the tuple for the sample if the hash value is less than a
  - If the key consists of more than one component, the hash function needs to combine the values for those components to make a single hash value
  - (ex) How to generate a 30% sample?
    - Hash into $b = 10$ buckets, take the tuple if it hashes to bucket 0, 1, or 2

Maintaining a Fixed-Size Sample (1/2)

- The sample will grow as more of the stream enters the system
  - (ex) As time goes on, more searches for the same users will be accumulated

- If there is a limit for how many tuples can be stored as sample, the fraction of key values must ***lower*** as time goes on

- We maintain a threshold t, which is initially set to $B-1$
  - B: the number of buckets for the hash function h
  - At all times, the sample contains those tuples whose key K satisfies $h(K) \leq t$
  - New tuples from the stream are added to the sample if and only if $h(K) \leq t$

Maintaining a Fixed-Size Sample (2/2)

- Suppose the number of stored tuples of the sample ***exceeds*** the allotted space

- Then, we lower t to $t-1$ and remove from the sample all those tuples whose key K hashes to t

- For efficiency, we can lower t by more than 1, and remove the tuples with serval of the highest hash values

- For further efficiency, we can maintain an index on the hash value
  - So we can find all those tuples whose key hashes to a particular value quickly

Reservoir Sampling

- Algorithm
  - 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 $\boldsymbol{s/n}$, 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, S contains each element seen so far with probability s/n

Proof (By Induction) (1/2)

- Assume that after n elements, the sample contains each element seen so far with probability s/n

- We need to show that after seeing element $(n+1)$ the sample maintains the property
  - i.e., sample contains each element seen so far with probability $s/(n+1)$

- Base case:
  - After we see $n = s$ elements, the sample S has the desired property
    - Each out of $n = s$ elements is in the sample with probability $s/s = 1$

Proof (By Induction) (2/2)

- Inductive hypothesis:
  - After n elements, S contains each element seen so far with probability s/n

- Inductive step:
  - Now element $(n + 1)$ arrives
  - For elements already in S, probability that the algorithm keeps it in S is:

Element $(n + 1)$ not discarded

$$\underbrace{\left(1-\frac{s}{n+1}\right)}_{\text{Element } (n+1) \text{ discarded}}+\left(\frac{s}{n+1}\right)\underbrace{\left(\frac{s-1}{s}\right)}_{\text{Element in S not picked}}=\frac{n}{n+1}$$

  - At time n, tuples in S were there with probability s/n
  - Time n → $(n + 1)$, tuple stayed in S with probability $n/(n + 1)$
  - So the probability that tuple is in S at time $(n + 1)$ $= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

Filtering Streams

Accepting those tuples in the stream that meet a criterion

Filtering Streams

- Accept ***only*** those tuples in the stream that meet a criterion
  - Another common process on streams
  - Rejected tuples are dropped

- Easy example
  - Select those tuples with the first component being less than 10
    - The selection criterion is a property of the tuple itself

- Harder example
  - Select those tuples that are a ***member*** of a given set
    - Especially hard when the set is ***too large*** to store in main memory

A Motivating Example

- Scenario
  - Suppose we have a set S of 10^9 allowed email addresses
  - We will accept only those emails sent from S (i.e., they are not spam)
  - The stream consists of pairs $(\mathit{email\ address}, \mathit{email})$
  - It is not reasonable to store S in main memory
    - 20 bytes (the size of the typical email address) $\times$ 10^9 = 20 GB

- Two approaches
  - Use disk accesses to determine the membership of each address

    → ***Not feasible!***
  - Devise a method that requires no more main memory than we have available, and yet will filter most of the undesired tuples

    → ***Bloom filtering***

First Cut Solution (1/2)

- Suppose we have 1 GB of available main memory (= 10^9 bytes)
- Use that main memory as a ***bit array*** of 8×10^9 bits
  - Hash each member of S to a bit, and set that bit to 1
  - All other bits of the array remain 0
  - Note that it is possible that two members of S hash to the same bit

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

- Devise a hash function h from email addresses to 8×10^9 buckets
- When a stream element arrives, we hash its email element
  - If the bit to which that email address hashes is 1, we accept the email
  - If the bit to which that email address hashes is 0, we drop the email

First Cut Solution (2/2)

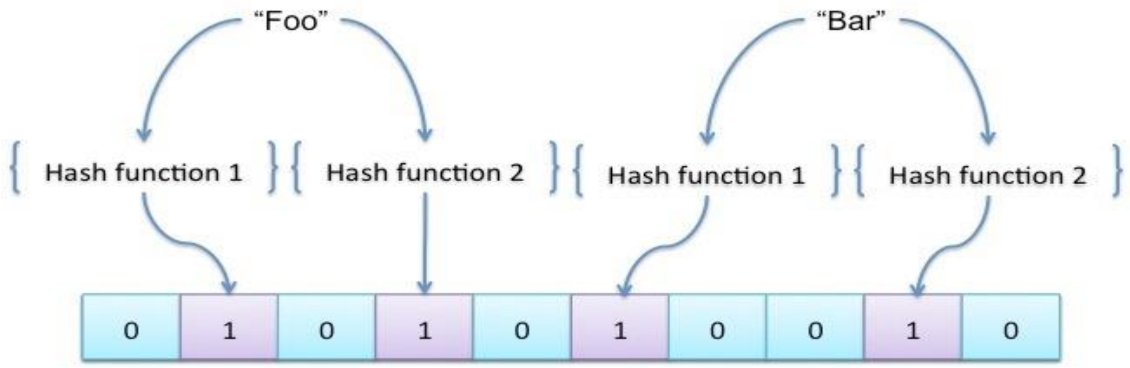
- ***No false negative***
  - If the email address is in S, we surely accept it
  - Because that email address surely hashes to a bucket that has bit set to 1
- ***But possible false positive***
  - If the email address is ***not*** in S, we may still accept it
  - Since there are 10^9 members of S, approximately 1/8th of the bit will be 1
    - Actually, less than 1/8, because two members of S may hash to the same bit
  - Thus, approximately 1/8th of the stream elements whose email address is not in S will happen to hash to a bit whose value is 1
- If we want to eliminate every spam
  - Only check for membership in S those that get through the filter
  - Use a cascade of filters (i.e., the general ***Bloom-filtering*** technique)

Definition: Bloom Filter (1/2)

- A bloom filter consists of
  - An array of n bits, initially all 0's
  - A ***collection*** of hash functions $h_1, h_2, \ldots, h_k$
    - Each hash function maps "key" values to n buckets
  - A set S of m key values
- The purpose of the Bloom filter
  - Allow through all stream elements whose keys are in S
  - Reject ***most*** of the stream elements whose keys are ***not*** in S

Definition: Bloom Filter (2/2)

- Initialization of the bit array
  - Take each key value is S and hash it using each of $h_1, h_2, \ldots, h_k$
  - Set to 1 each bit that is $h_i(K)$ for some h_i and some key value K in S



- Testing a key K that arrives in the stream
  - Check that ***all*** of $h_1(K), h_2(K), \ldots, h_k(K)$ are 1's in the bit array
  - If all are 1's, then let the stream element through
  - If one or more of these bits are 0, then reject the stream element
    - Because K could not be in S

Analysis of Bloom Filtering (1/4)

- If the key value is not S, it might still pass → ***false positives***

- We calculate the probability of a false positive, as function of
  - n: the bit-array length
  - m: the number of members in S
  - k: the number of hash functions

- The throwing darts model
  - Suppose we have x targets and y darts
  - Any dart is equally likely to hit any target
  - After throwing y darts, how many targets will be hit at least once?
  - ✓ In our case, targets = bits, darts = hash values of members in S

Analysis of Bloom Filtering (2/4)

- The probability that a given dart will not hit a given target
  - $(x - 1)/x$
- The probability that none of the y darts will hit a given target
  - $((x - 1)/x)^y = (1 - 1/x)^{x(y/x)} = e^{-y/x}$ ($\because$ $(1 - z)^{1/z} = e^{-1}$ for small z)
- Example: The probability that a given bit in the bit-array will be 1
  - $x = 8 \times 10^9$ (the number of targets = the number of bits)
  - $y = 10^9$ (the number of darts = the number of members of S)
  - The probability that a given target is not hit $= e^{-y/x} = e^{-1/8}$
  - The probability that a given target ***is*** hit $= 1 - e^{-1/8} \approx 0.1175 \approx 1/8$

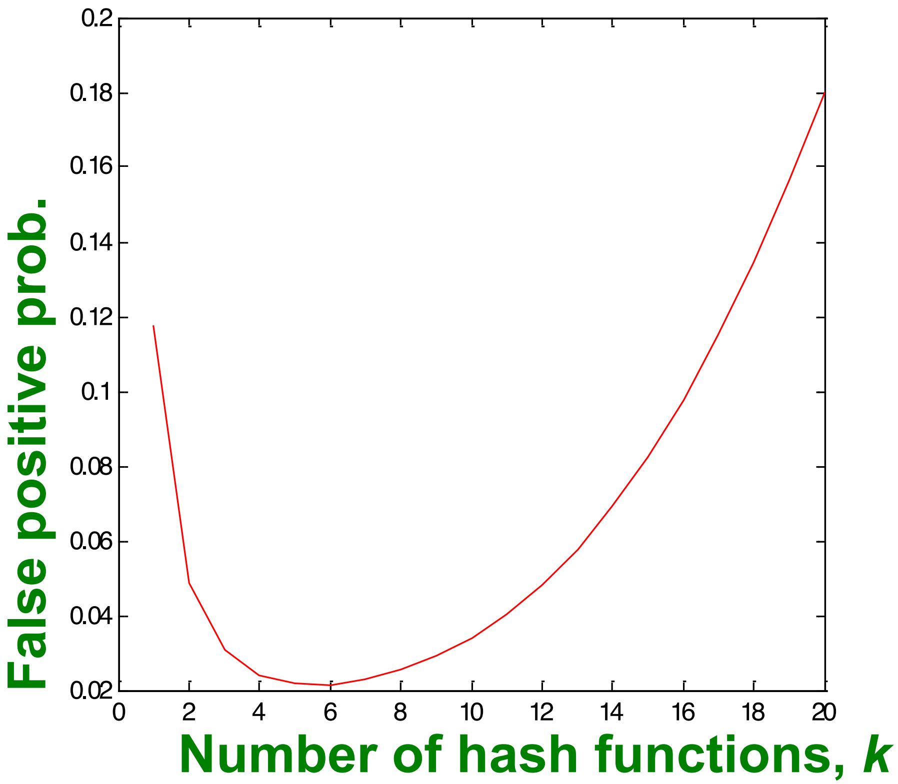
Analysis of Bloom Filtering (3/4)

- Generalization
  - The number of targets is $x = n$
  - The number of darts is $y = km$
  - The probability that a bit remains 0 is $e^{-km/n}$
  - The probability that a bit is 1 is $1 - e^{-km/n}$
  - So, the probability of a false positive is $\mathbf{(1 - e^{-km/n})^k}$
    - The hash values of all k hash functions are the same as 1

- Examples ($m = 10^9$, $n = 8 \times 10^9$)
  - $k = 1$: the probability of a false positive $= (1 - e^{-1/8}) = 0.1175$
  - $k = 2$: the probability of a false positive $= (1 - e^{-1/4})^2 = 0.0493$
    - When we used two different hash functions

Analysis of Bloom Filtering (4/4)

- What happens as we keep increasing k?



  - Optimal value of $k = n/m \cdot \ln(2)$
    - (ex) in our case, the optimal $k = 8 \cdot \ln(2) = 5.54 \approx 6$

Bloom Filter: Wrap-Up

- Guarantee ***no*** false negatives, and use ***limited*** memory
  - Great for pre-processing before more expensive checks
- Suitable for hardware implementation
  - Hash function computations can be parallelized
- Is it better to have ***1 big*** bit array or k ***small*** bit arrays?
  - It is the same: $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping 1 big bit array is simpler

Counting Distinct Elements in a Stream

Knowing how many different elements have appeared in the stream

Counting Distinct Elements in a Stream

- **The Count-Distinct Problem**
  - Suppose stream elements are chosen from some universal set
  - Maintain a count of the number of ***distinct*** elements seen so far

- Applications
  - How many unique users has seen in each month? (e.g., Amazon, Google)
    - Users may be identified by the IP address
    - Note that there are about 4 billion IP addresses!
  - 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?

Obvious Approach

- Keep in main memory ***all*** the elements seen so far
- Keep them in an efficient search structure
  - (ex) hash table or search tree
  - One can quickly check if a newly arrived element was already seen
- However, if the number of distinct elements is too great, then we ***cannot*** store them in main memory
- Several options
  - Store most of the data structure in second storage → ***too many disk I/O!***
  - Only ***estimate*** the number of distinct element but use ***much less*** memory
    → ***this section's topic***

Flajolet-Martin Algorithm (1/2)

- Hash each element of the stream to a ***bit-string*** of length N
  - The number of possible hash-values = 2^N
  - 2^N must be larger than the number of all elements in the universal set
  - (ex) 64 bits is sufficient to hash URL's

- Idea
  - The more different elements we seen in the stream, the more different hash-values we shall see
  - As we see more different hash-values, it becomes more likely that one of these values will be "***unusual***"
  - In our case, an unusual value is a value ending in many 0's
    - Of course, many other options exist

Flajolet-Martin Algorithm (2/2)

- Whenever we apply a hash function h to a stream element a, the bit-string $h(a)$ will end in some number of 0's, possibly none
  - Call this number the ***tail length*** for a, denoted by $r(a)$
  - (ex) if $h(a) = 1100$, then $r(a) = 2$

- Let R be the maximum tail length seen so far in the stream
  - That is, $R = \max_a r(a)$, over all the stream elements a seen so far

- Then, we estimate the number of distinct elements as $\mathbf{2^R}$
  - Why?

Very Very Rough and Heuristic Intuition

- $h(a)$ hashes a with ***equal*** probability to any of 2^N values

- Then $1/2^r$ fraction of all $h(a)$ have a tail of r zeros
  - About 1/2^1 of as hash to ****0
  - About 1/2^2 of as hash to ***00
  - About 1/2^3 of as hash to **000
  - So, for example, if we saw the longest tail of $r = 3$ (i.e., item hash ending *1000), then we have probably seen about 4 distinct items so far

- So, it takes to hash about $\boldsymbol{2^r}$ items before we see one with a tail of $\boldsymbol{r}$ zeros

Why It Works: More Formally (1/2)

- The probability that a stream element a has $h(a)$ ending in at least r 0's is $(1/2)^r = 2^{-r}$

$$h(a) \quad \boxed{1 \mid 0 \mid 1 \mid 1 \mid \ldots \mid \underbrace{0 \mid 0 \mid 0 \mid 0 \mid 0}_{r}}$$

- Suppose there are ***m*** distinct elements in the stream

- The probability of ***not*** finding a tail of length at least r
  - $(\boxed{1 - 2^{-r}})^m = ((1 - 2^{-r})^{2^r})^{m2^{-r}} \approx e^{-m2^{-r}}$

Probability that given $h(a)$ ends in fewer than r zeros

- The probability of finding a tail of length at least r
  - $1 - e^{-m2^{-r}}$

Why It Works: More Formally (2/2)

- We can conclude:
  - If $m >> 2^r$, the probability of finding a tail of length at least r approaches 1
    - $1 - e^{-m2^{-r}} \rightarrow 1 - e^{-\infty} \rightarrow 1$
  - If $m << 2^r$, the probability of finding a tail of length at least r approaches 0
    - $1 - e^{-m2^{-r}} \rightarrow 1 - e^{-0} \rightarrow 0$

- Thus, 2^R will almost always be around m
  - Recall that R is the largest tail length for any stream element
  - In other words, 2^R is unlikely to be either much too higher or much too lower than m

- Consequently, ***we estimate m as 2^R***

Combining Estimates

- m is always estimated as a power of 2
  - May not be accurate

- We may obtain a ***combined*** estimates by using ***many different*** hash functions $h_1, h_2, \ldots, h_k$ (i.e., $2^{R_1}, 2^{R_2}, \ldots, 2^{R_k}$)

- How can we combine $2^{R_1}, 2^{R_2}, \ldots, 2^{R_k}$?
  - Average? → What if one very large value 2^{R_i}? (the value of 2^R doubles)
  - Median? → All estimates are a power of 2 (what if m is between them?)

- Solution
  - Partition the hash functions into small groups
  - Take their averages
  - Then take the median of the averages

Space Requirements

- The only thing we need to keep in memory is one integer per hash function
  - i.e., R_i (the largest tail length seen so far) for each hash function h_i
- If we are processing only one stream, we could use million of hash functions
  - Far more than we need to get a close estimate
- In practice, the time it takes to compute hash values for each stream element would be more significant limitation on the number of hash functions we use

Counting Ones in a Window

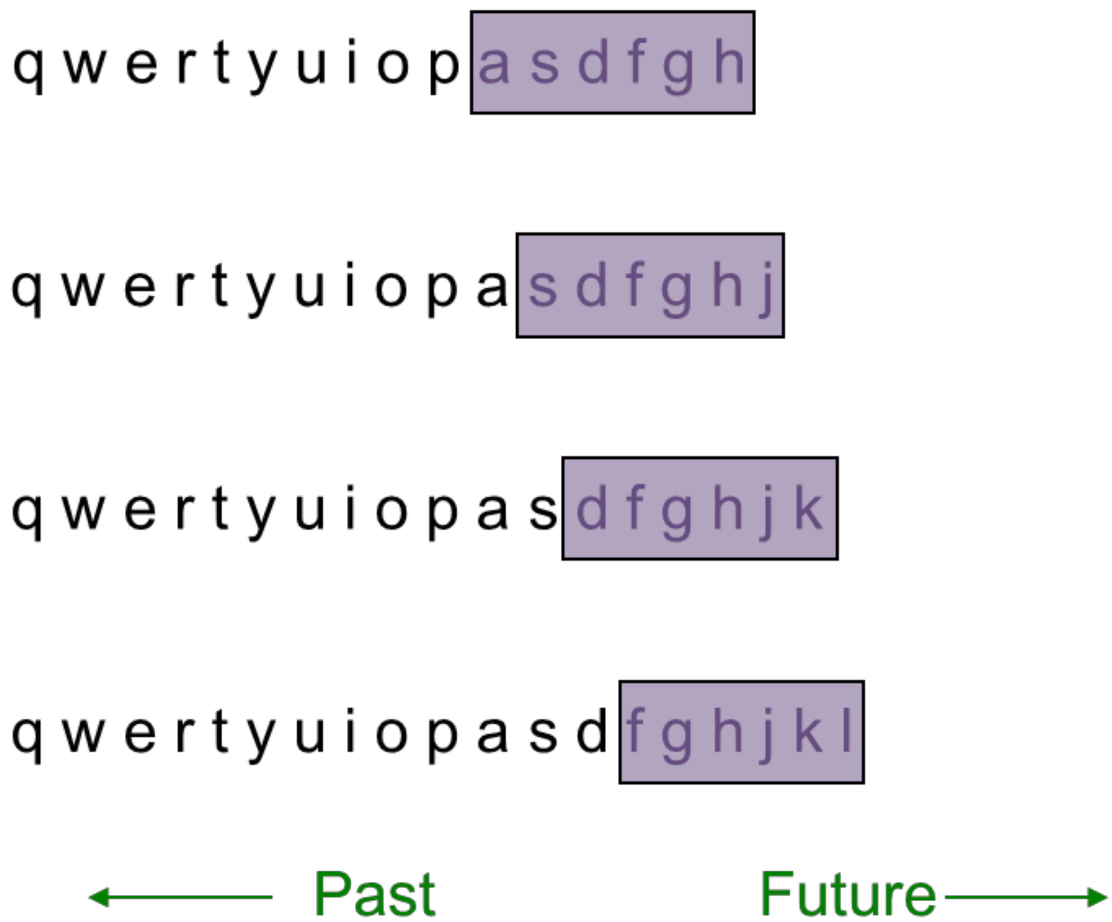
Answering “how many 1’s are there in the last k elements?”

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 to 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 Ones in a Window (1/2)

- Suppose we have a window of length N on a binary stream
  - That is, each stream element is 0 or 1
- We want at all times to be able to answer queries of the form "***How many 1's are there in the last k bits?***" for any $k \leq N$
- 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 $\boxed{1\ 1\ 0\ 1\ 1\ 0}$ Suppose $N = 6$

← Past Future →

Counting Ones in a Window (2/2)

- Real problem: ***What if we cannot store the entire window?***
  - (ex) we are processing a stream with a window of length $N = 10^9$

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 →

- Of course, you cannot get an exact answer without storing the entire window

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

An Attempt: Simple Solution

- Question: How many 1s are in the last k bits?
- Simple solution
  - We may make the ***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 →

  - We 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 k bits? → $\boldsymbol{k \cdot S/(S+Z)}$
  - But, what if stream is non-uniform?
    - What if distribution changes over time?

Datar-Gionis-Indyk-Motwani(DGIM) Algorithm

- Does ***not*** assume uniformity
- Uses $\boldsymbol{O(\log^2 N)}$ bits to represent a window of N bits
- Allows us to estimate the number of 1's in the window with an error of ***no more than 50%***
- This method can be improved to limit the error to any fraction $\varepsilon > 0$, and still uses only $\boldsymbol{O(\log^2 N)}$ bits
  - Although with a constant factor that grows as ε shrinks

DGIM: Timestamps

- Each bit of the stream has a ***timestamp***
  - The position in which it arrives (the first bit = 1, the second bit = 2, …)
- We represent timestamps modulo N
  - Since we only need to distinguish positions within the window of length N
  - So they can be represented by $\mathbf{log_2}N$ bits (to distinguish N different values)

The current window

Stream: 1 0 0 1 0 1 0 1 1 0 $\boxed{1\ 1\ 0\ 1}$

Timestamp modulo N: 1 2 3 0 1 2 3 0 1 2 3 0 1 2

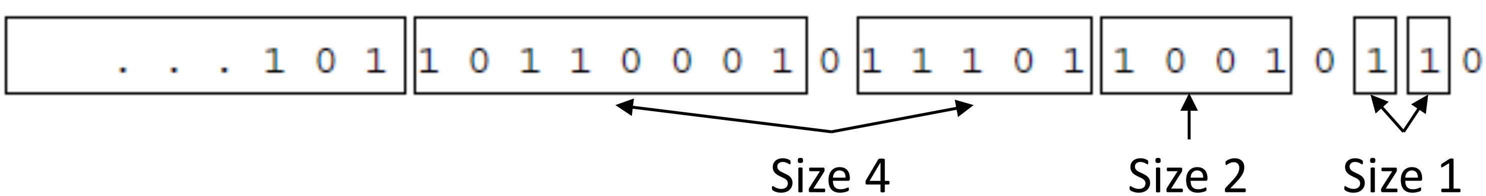
Suppose $N = 4$

DGIM: Buckets

- We divide the window into ***buckets***, consisting of:
  1. The timestamp of its right (most recent) end
     - $\log_2 N$ bits
  2. The number of 1's in the bucket (called the ***size*** of the bucket)
     - $\log_2 \log_2 N$ bits

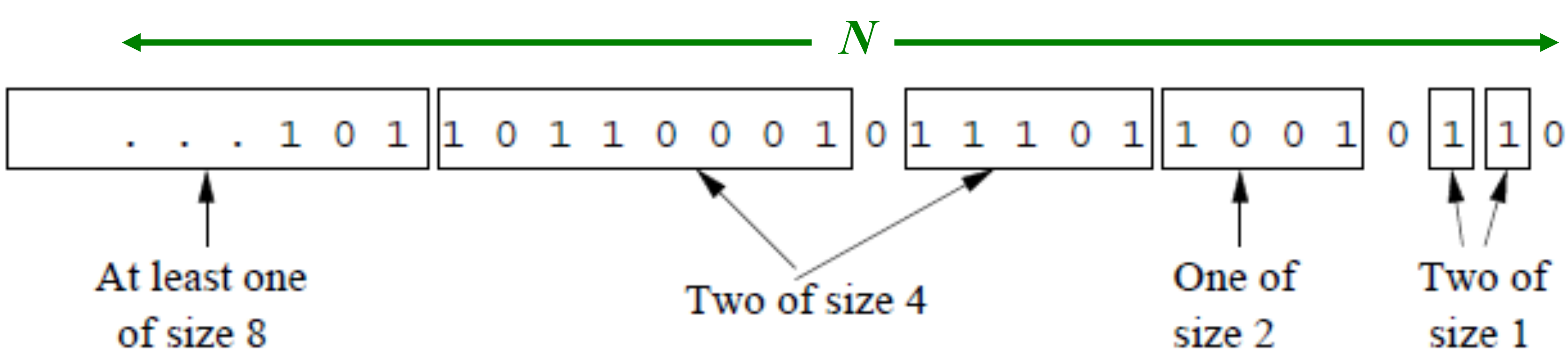
  → Thus, $O(\log_2 N)$ bits suffice to represent a bucket

- Constraint on buckets
  - The number of 1's must be ***a power of 2*** (i.e., 2^j)
    - We can represent this number by coding j in binary
    - Since j is at most $\log_2 N$, it requires $\log_2 \log_2 N$ bits



6 Rules to Represent 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 (buckets do not overlap)
- There are ***one*** or ***two*** buckets of any given size
- All sizes must be a power of 2
- The sizes of buckets increase as we move to the left



Storage Requirements

- Each bucket can be represented by $O(\log N)$ bits

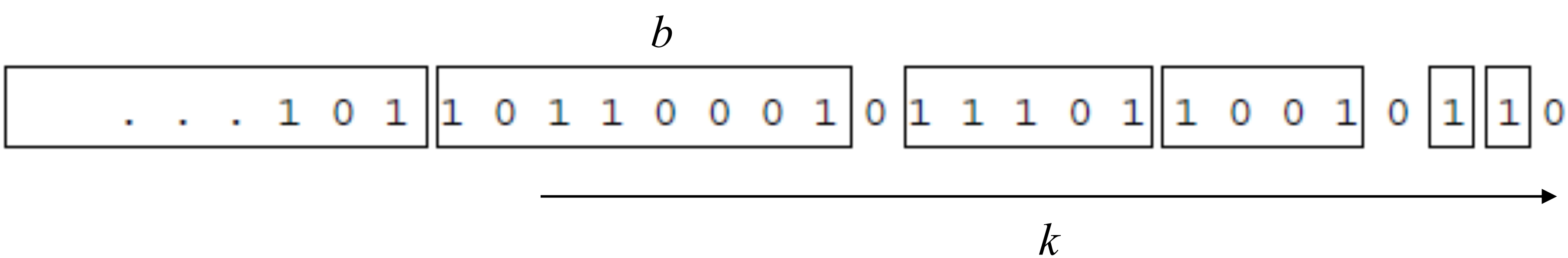
- There are $O(\log N)$ buckets
  - The total number of 1's in all the buckets cannot exceed N
  - That is, if the largest bucket is of size 2^j, then j cannot exceed $\log_2 N$
  - Thus, there are at most two buckets of all sizes from $\log_2 N$ down to 1
  - Consequently, the total number of buckets is $O(\log N)$

- Thus, the total space required for all the buckets is $\boldsymbol{O(\log^2 N)}$

Query Answering

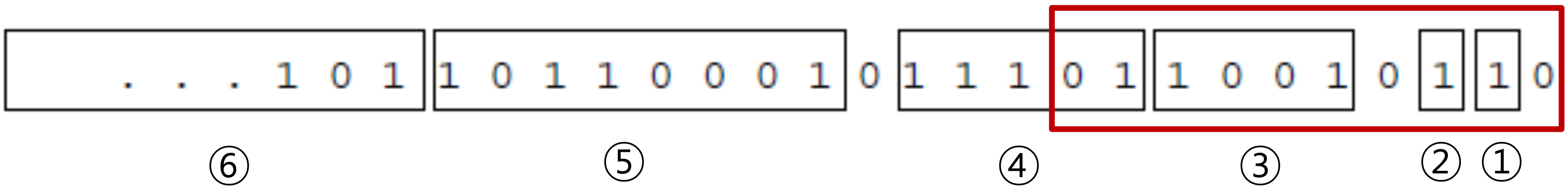
- Suppose we are asked how many 1's there are in the last k bits of the window ($1 \leq k \leq N$)

- Basic steps
  - Find the bucket b with the earliest timestamp that includes at least some of the k most recent bits
  - Sum the sizes of all the buckets to the right than bucket b
  - Add half the size of b itself
  - The result is the estimation of the number of 1's in the last k bits



(Ex) Query Answering

- How many 1's are there in the last 10 bits of window? ($k = 10$)



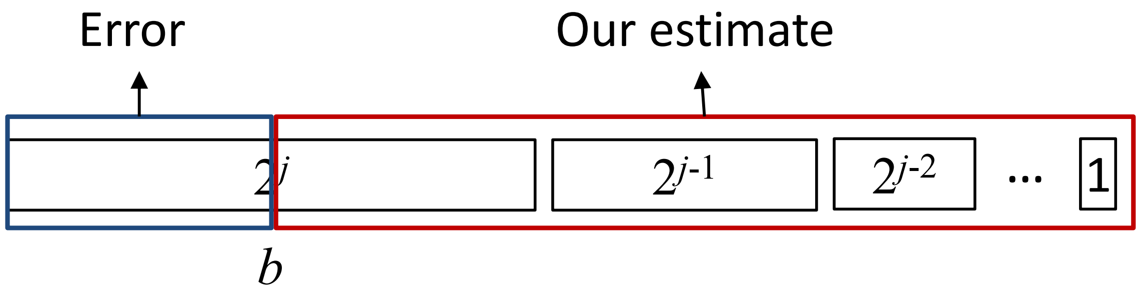
  – The bucket with the earliest timestamp that includes at least some of the 10 recent bits
    - ④
  – The sum of the sizes of all the buckets to the right than bucket ④
    - 2 + 1 + 1 = 4
  – Half the size of bucket ④
    - 4/2 = 2

- Thus, our estimate is 4 + 2 = 6 (*c.f.*, the correct answer is 5)

Error Bound (1/2)

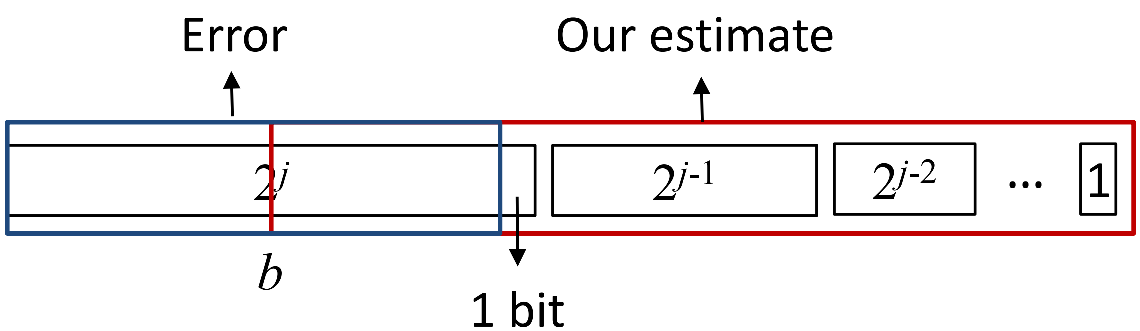
- How far from the correct answer c our estimate could be?
  - Let the largest bucket be of size 2^j

- **Case 1**: the estimate is less than c
  - In the worst case, all the 1's of b are within the range of the query, so the estimate misses half bucket b, or $\mathbf{2^{j-1}}$ 1's
  - In this case, c is at least $1 + 2^{j-1} + 2^{j-2} + \ldots + 1 = 2^j$
  - Thus, the error is at most $2^{j-1}/2^j = 50\%$ of c



Error Bound (2/2)

- **Case 2**: the estimate is greater than c
  - In the worst case, only the rightmost bit of bucket b is within range
  - In this case, $c = 1 + 2^{j-1} + 2^{j-2} + \ldots + 1 = 2^j$
  - Our estimation is $2^{j-1} + 2^{j-1} + 2^{j-2} + \ldots + 1 = 2^j + 2^{j-1} - 1$
  - Thus, the error is $((2^j + 2^{j-1} - 1) - 2^j)/2^j = (2^{j-1} - 1)/2^j < 50\%$ of c



- Thus, the error is at most **50%** of the correct answer

Maintaining DGIM Conditions (1/3)

- When a new bit comes in, we may need to modify the buckets
  - So they continue to represent the window and satisfy the 6 DGIM rules
- First, whenever a new bit enters:
  - Drop the oldest bucket if its timestamp has now reached the current timestamp $- N$
    - This bucket no longer has any of its 1's in the window of length N
  - Then we check whether the new bit is 0 or 1
- If the new bit is 0
  - No other changes are needed

Maintaining DGIM Conditions (2/3)

- If the new bit is 1
  - Create a new bucket of size 1, for just this bit
    - The timestamp of its right end = the current timestamp
  - If there are now three buckets of size 1
    - Combine the oldest two into a bucket of size 2
  - If there are now three buckets of size 2
    - Combine the oldest two into a bucket of size 4
  - And so on …
- This process takes $O(\log N)$ time
  - There are at most $\log_2 N$ different sizes
  - The combination of two buckets only requires constant time

(Ex) Maintaining DGIM Conditions

Current state of the stream:

10010101100010110101010101010101101010101010111101010101110101000101100100

Bit of value 1 arrives

0010101100010110101010101010101101010101010111101010101110101000101100101

Two orange buckets get merged into a yellow bucket

0010101100010110101010101010101101010101010111101010101110101000101100101

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

0101100010110101010101010101101010101010111101010101110101000101100101101

Buckets get merged…

0101100010110101010101010101101010101010111101010101110101000101100101101

State of the buckets after merging

0101100010110101010101010101101010101010111101010101110101000101100101101

Further Reducing the Error (1/2)

- Instead of allowing either 1 or 2 of each size bucket, suppose we allow either $r-1$ or r of each size bucket for some integer $r > 2$
  - Except for the buckets of size 1 and buckets of the largest size
  - There may be any number from 1 to r of buckets of these sizes

- The rule for combining buckets is essentially the ***same***
  - If we get $r+1$ buckets of size 2^j, combine the oldest two into a bucket of size 2^{j+1}
  - If that causes there to be $r+1$ buckets of size 2^{j+1}, continue combining buckets of larger sizes

Further Reducing the Error (2/2)

- However, we can get a stronger bound on the error
  - Because there are more buckets of smaller sizes
- The largest relative error
  - Occurs when only one 1 from the oldest bucket b is within the query range
  - Suppose bucket b is of size 2^j
  - The true count is at least $1 + (r-1)(2^{j-1} + 2^{j-2} + \ldots + 1) = 1 + (r-1)(2^j - 1)$
  - The overestimate is $2^{j-1} - 1$
- Thus, the fractional error is

$$\frac{2^{j-1} - 1}{1 + (r-1)(2^j - 1)} < \frac{1}{r-1} = O(1/r)$$

  - By picking r sufficiently large, we can limit the error to any desired $\varepsilon > 0$

Extensions (1/2)

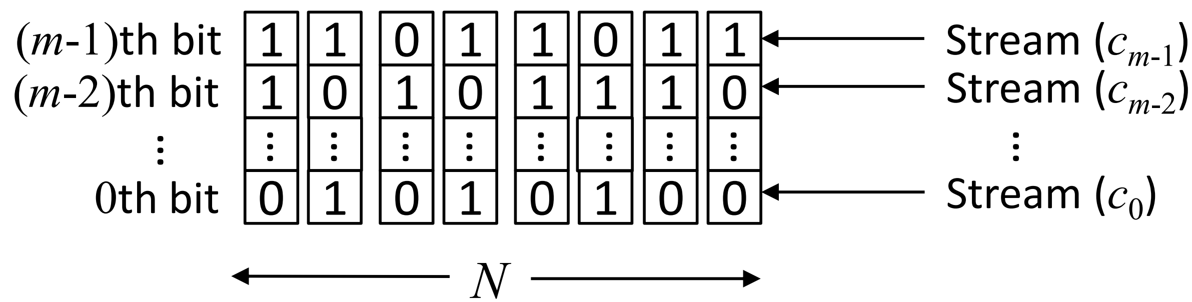
- Can we handle the case where the stream is not bits, but ***integers***, and we want the sum of the last k elements?
  – What if we represent each bucket by the sum of the integers therein and estimate the contribution of b by half its sum?
  – Our estimate may be ***meaningless*** if a stream contains both very large positive integers and very large negative integers

- However, we can handle the case where stream consists of only ***positive*** integers in the range 1 to 2^m for some m
  – Assume that each integer is represented by m bits
  – Then how? → In the next slide

Extensions (2/2)

- Solution
  - Treat each of the m bits of each integer as if it were a separate stream
  - Use DGIM to count the 1's in each bit
  - Let c_i be the count of the ith bit
    - Assume that 0th bit is the least significant bit
  - Then the sum of the integers is

$$\sum_{i=0}^{m-1} c_i 2^i$$



- The worst case
  - When all the c_i's are overestimated or all are underestimated by the same fraction

Decaying Windows

Weighting the recent elements more heavily

Decaying Windows

- Sometimes we want to weight the recent elements more ***heavily***

- The problem of most ***frequent*** elements
  - Consider a stream whose elements are the movie tickets purchased
  - We want to know what are "***currently"*** most popular movies
    - i.e., we want to discount the popularity of a move that was sold decades ago

- One solution
  - Imagine a bit stream for each movie
    - The ith bit is 1 if the ith ticket is for that movie, and 0 otherwise
  - Pick a window size N, which represents the recentness of move tickets
  - Then use DGIM to estimate the number of tickets for each movie
  - ***Drawbacks***: (1) Only approximate, (2) The number of items is way too big

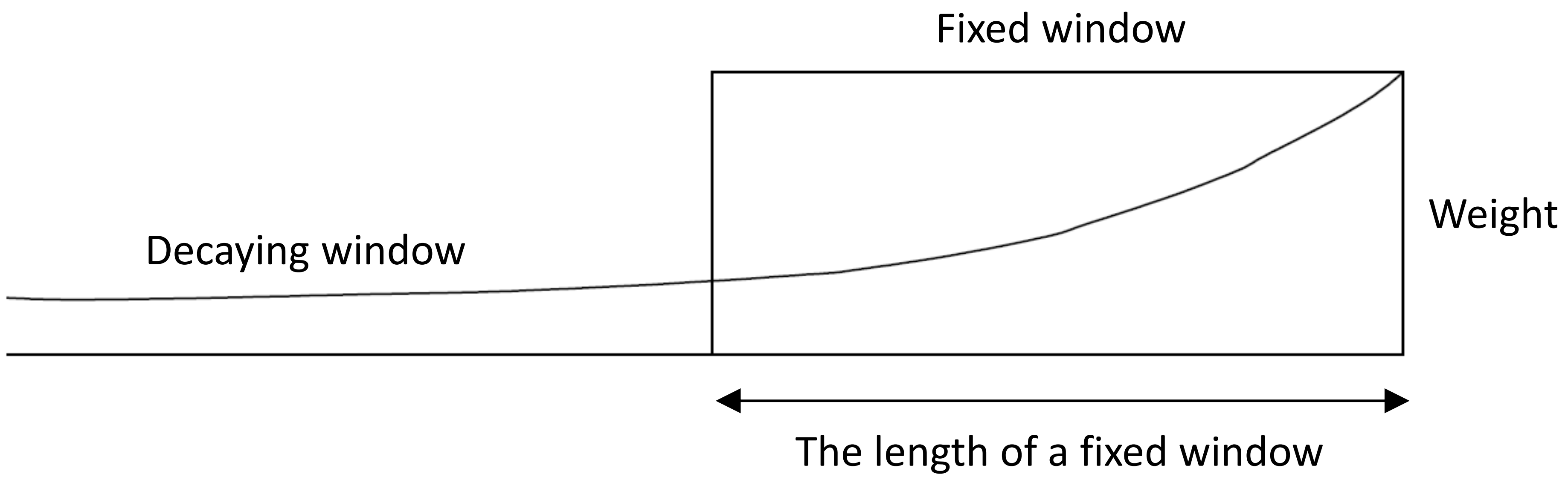
Definition: Exponentially Decaying Window

- Let a stream consist of $a_1, a_2, \ldots, a_t$
  - a_1 is the first element and a_t is the current element
- Let c be a small constant (such as 10^{-6} or 10^{-9})
- The ***exponentially decaying window*** is defined as follows:

$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^i$$

  - The weight of a stream element decreases as the stream goes
    - $a_t \rightarrow (1-c)^0, a_{t-1} \rightarrow (1-c)^1, a_{t-2} \rightarrow (1-c)^2, \ldots, a_1 \rightarrow (1-c)^{t-1}$

Fixed Windows vs. Decaying Windows



- Fixed window
  - Puts equal weight 1 on each of the most recent elements
  - Puts weight 0 on all previous elements

Updating An Exponentially Decaying Window

- When a new element a_{t+1} arrives
  - Multiply the current sum by $(1 - c)$
  - Add a_{t+1}

- The reason this method works
  - The weight of each of the previous elements is multiplied by $(1 - c)$
    - Each of them has now moved one position further from the current element
  - The weight of the current element is $(1 - c)^0 = 1$
    - So just adding a_{t+1} is the correct way

Finding the Most Popular Elements (1/2)

- Consider the problem of finding the most popular movies in a stream of ticket sales

- For each movie, we imagine a separate stream
  - 1, each time a ticket for the movie appears in the stream
  - 0, each time a ticket for some other movies arrives

- The current popularity of a movie
  - Measured by the ***decaying sum of the 1's*** $= \sum_{i=0}^{t-1} a_{t-i}(1-c)^i$
    - $a_{t-i} = 1$ or 0
  - That is, we use an exponentially decaying window with a constant c
    - You might think of c as 10^{-9}

Finding the Most Popular Elements (2/2)

- If the popularity score for a move goes below a threshold, its score is dropped from the counting
  - Because the number of possible movies in the stream is ***huge***, we do not want to record values for the unpopular movies
  - We assume that the threshold is $1/2$ (must be less than 1)

- When a new ticket arrives on the stream
  - For each movie whose score we are currently maintaining, multiply its score by $(1 - c)$
  - Suppose the new ticket is for movie M
    - If there is currently a score for M, add 1 to that score
    - If there is no score for M, create one and initialize it to 1
  - If any score is below the threshold $1/2$, drop that score

Limit on The Number of Scores

- In fact, the number of movies whose scores are maintained at any time is ***limited***

- Note that the sum of all scores is $1/c$

$$\sum_{i=0}^{\infty}(1-c)^i = 1 + (1-c) + (1-c)^2 + \cdots + 0 = 1/c$$

- There cannot be more than $2/c$ movies with score of $1/2$ or more
  - Or else the sum of the scores would exceed $1/c$

- Thus, $\mathbf{2/c}$ is a limit on the number of movies being counted at any time