

COMP9313: Big Data Management



Lecturer: Xin Cao

Course web site: <http://www.cse.unsw.edu.au/~cs9313/>

Chapter 9: 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** and **non-stationary** (the distribution changes over time)

Characteristics of Data Streams

- Traditional DBMS: data stored in *finite, persistent data sets*
- Data Streams: distributed, continuous, unbounded, rapid, time varying, noisy, . . .
- Characteristics
 - Huge volumes of continuous data, possibly infinite
 - Fast changing and requires fast, real-time response
 - Random access is expensive—single scan algorithm (can only have one look)
 - Store only the summary of the data seen thus far

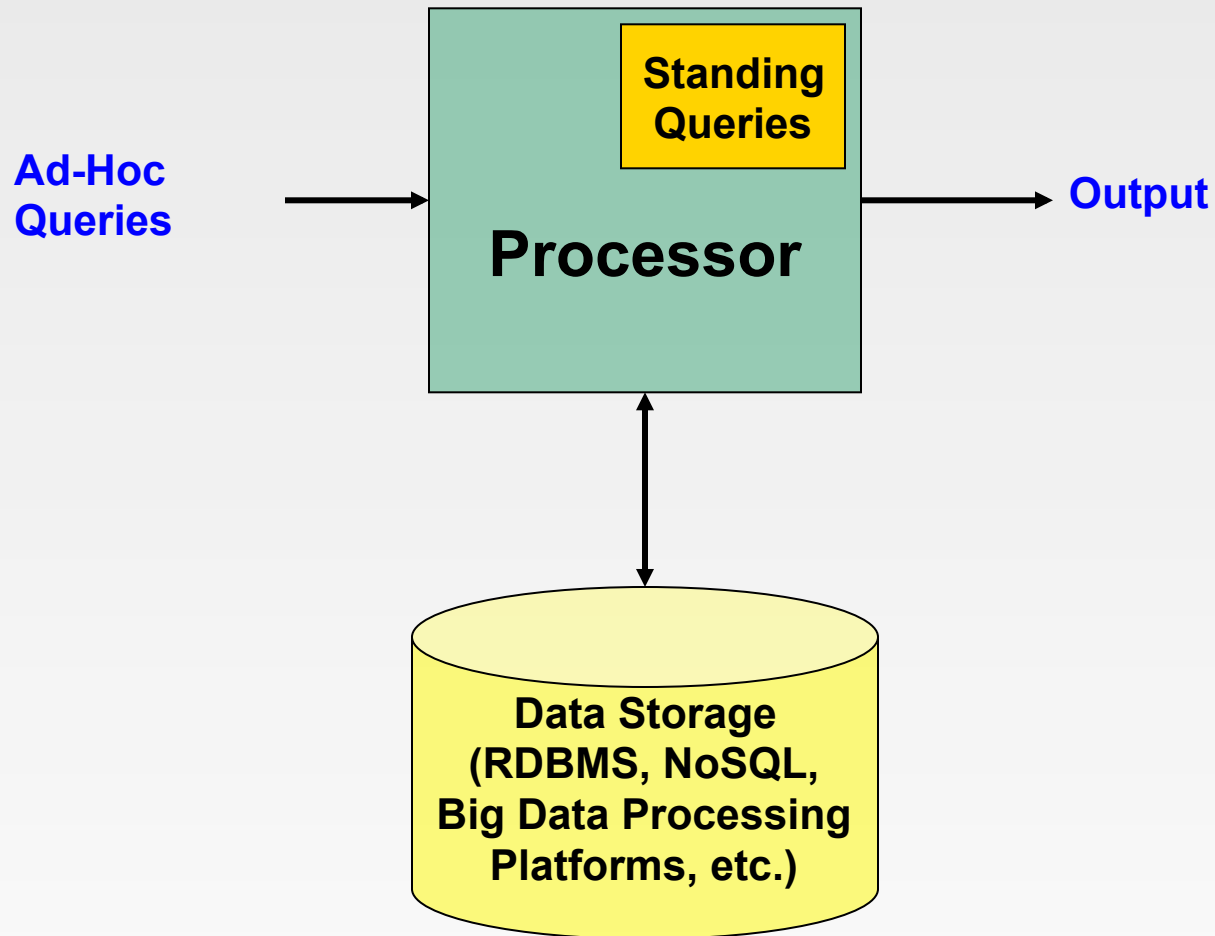
Massive Data Streams

- Data is *continuously growing* faster than our ability to store or index it
- There are 3 Billion Telephone Calls in US each day, 30 Billion emails daily, 1 Billion SMS, IMs
- Scientific data: NASA's observation satellites generate billions of readings each per day
- IP Network Traffic: up to 1 Billion packets per hour per router. Each ISP has many (hundreds) routers!
-

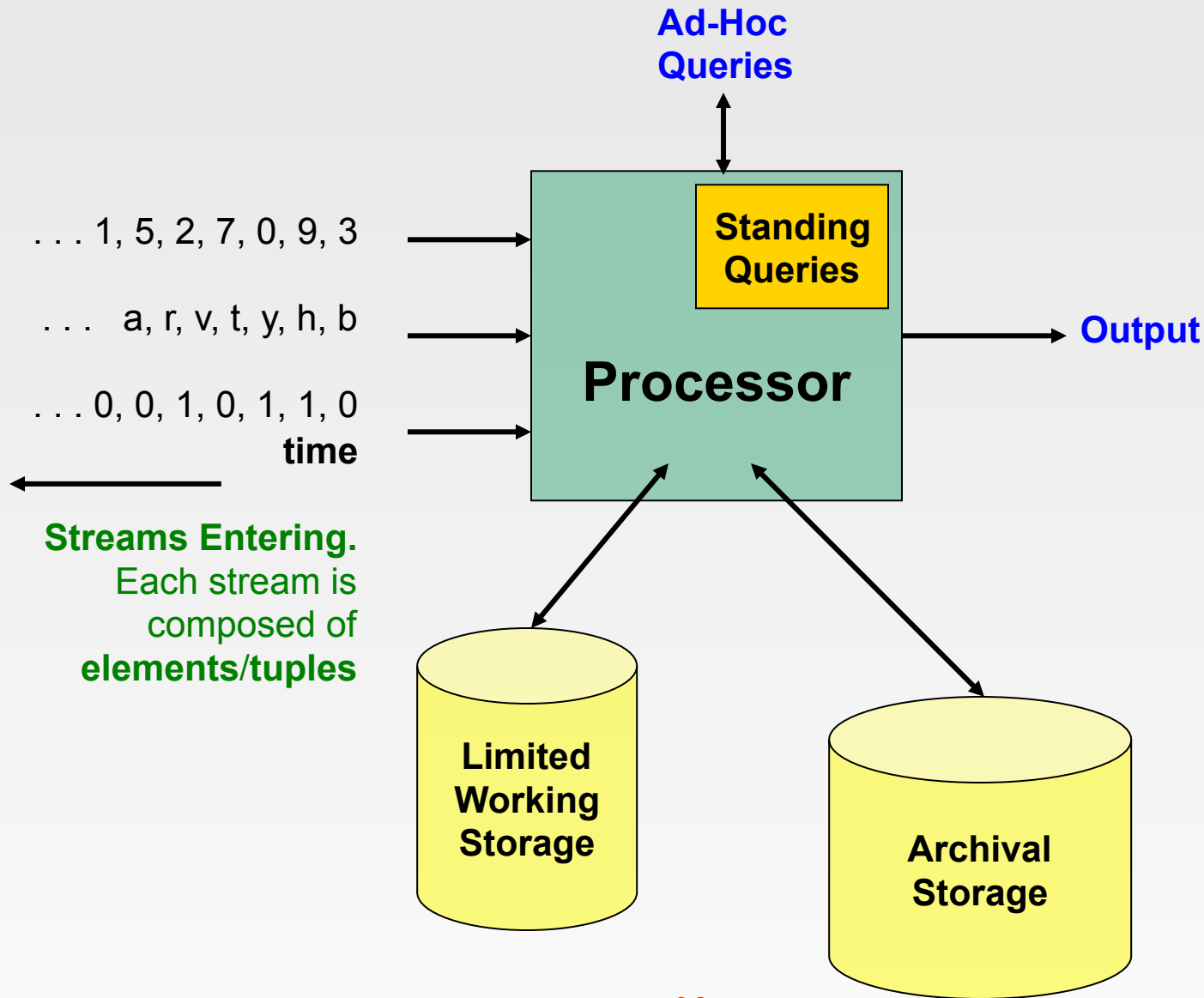
The Stream Model

- 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?**

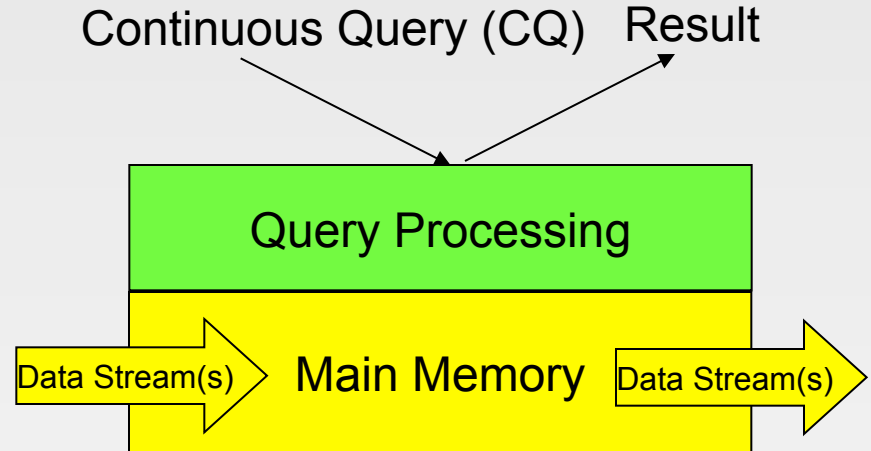
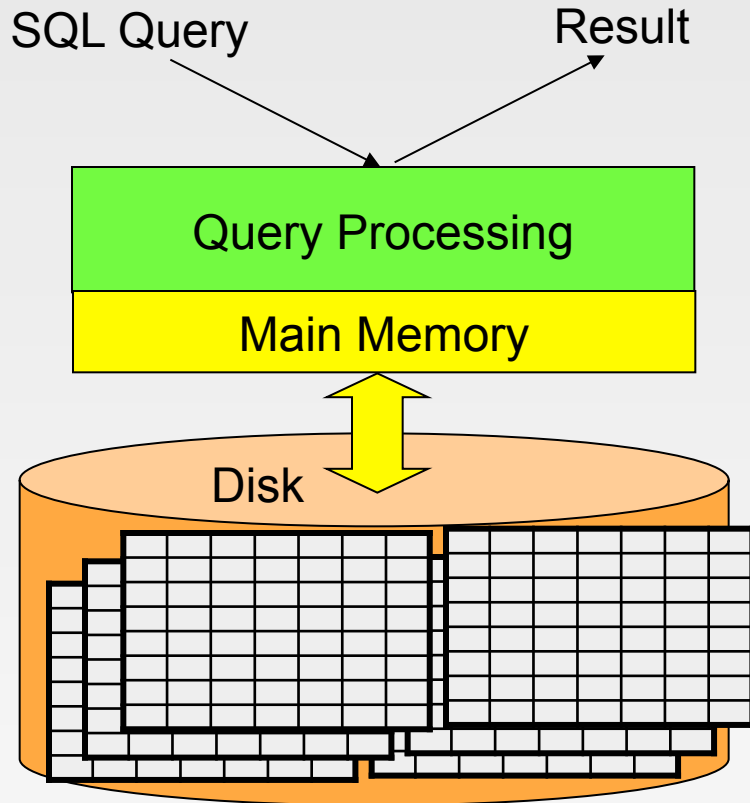
Database Management System (DBMS) Data Processing



General Data Stream Management System (DSMS) Processing Model



DBMS vs. DSMS #1



DBMS vs. DSMS #2

■ Traditional DBMS:

- stored sets of relatively **static** records with no pre-defined notion of time
- good for applications that require **persistent data storage** and **complex querying**

■ DSMS:

- support on-line analysis of rapidly **changing** data streams
- data stream: real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items, too large to store entirely, no ending
- **continuous** queries

DBMS vs. DSMS #3

DBMS

- Persistent relations
(relatively static, stored)
- One-time queries
- Random access
- “Unbounded” disk store
- Only current state matters
- No real-time services
- Relatively low update rate
- Data at any granularity
- Assume precise data
- Access plan determined by query processor, physical DB design

DSMS

- Transient streams
(on-line analysis)
- Continuous queries (CQs)
- Sequential access
- Bounded main memory
- Historical data is important
- Real-time requirements
- Possibly multi-GB arrival rate
- Data at fine granularity
- Data stale/imprecise
- Unpredictable/variable data arrival and characteristics

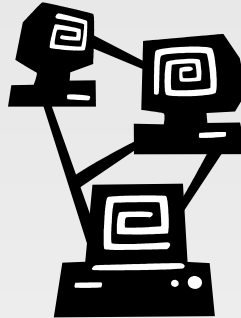
Problems on Data Streams

- Types of queries one wants on answer on a data stream: (we'll learn these today)
 - 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

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

Example: IP Network Data

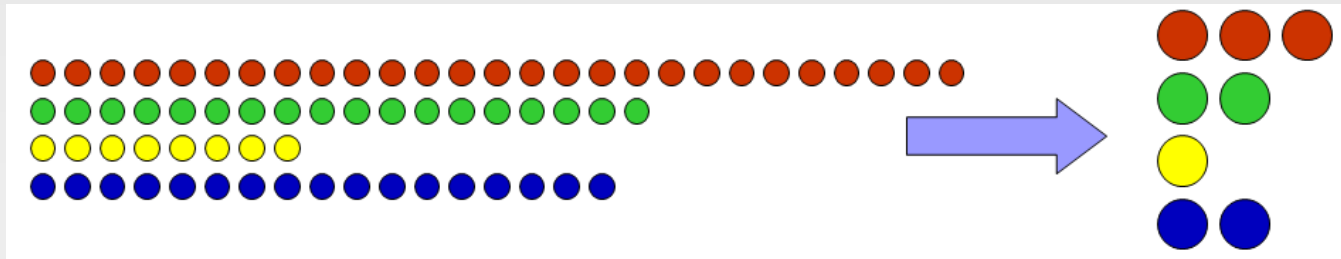


- Networks are sources of massive data: the **metadata** per hour per IP router is gigabytes
- Fundamental problem of data stream analysis:
 - Too much information to store or transmit
- So process data as it arrives
 - One pass, small space: the data stream approach
- **Approximate answers** to many questions are OK, if there are guarantees of result quality

Part 1: Sampling Data Streams

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)
 - ▶ As the stream grows the sample also gets bigger
 - (2) Maintain a **random sample of fixed size** over a potentially infinite stream
 - ▶ As the stream grows, the sample is of fixed size
 - ▶ At any “time” t we would like a random sample of s elements
 - **What is the property of the sample we want to maintain?**
For all time steps t , each of t elements seen so far has equal probability 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 $1/10^{\text{th}}$ 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 an average search engine user are duplicates?
 - Suppose each user issues x queries once and d queries twice (total of $x+2d$ queries)
 - ▶ **Correct answer:** $d/(x+d)$
 - **Proposed solution: We keep 10% of the queries**
 - ▶ Sample will contain $x/10$ of the singleton queries and $2d/10$ of the duplicate queries at least once
 - ▶ But only $d/100$ pairs of duplicates
 - $d/100 = 1/10 \cdot 1/10 \cdot d$
 - ▶ Of d “duplicates” $18d/100$ appear exactly once
 - $18d/100 = ((1/10 \cdot 9/10) + (9/10 \cdot 1/10)) \cdot d$
 - **So the sample-based answer is** $d/100 / x/10 + d/100 + 18d/100 = d / 10x + 19d$
 $\neq d/(x+d)$

Solution: Sample Users

Solution:

- Pick $1/10^{\text{th}}$ 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
 - We hash each user name to one of ten buckets, 0 through 9
 - If the user hashes to bucket 0, then accept this search query for the sample, and if not, then not.

Generalized Problem and Solution

- Problem: Give a data stream, take a sample of fraction a/b .
- 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 application
- To get a sample of 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**



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?** 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.

Note that the same item is treated as different tuples at different timestamps

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 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

■ 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

Proof: By Induction

■ We prove this by induction:

- 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
 - ▶ 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

- Inductive hypothesis: After n elements, the sample \mathbf{S} contains each element seen so far with prob. s/n
- **Now element $n+1$ arrives**
- **Inductive step:** For elements already in \mathbf{S} , probability that the algorithm keeps it in \mathbf{S} is:

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

- So, at time n , tuples in \mathbf{S} were there with prob. s/n
- Time $n \rightarrow n+1$, tuple stayed in \mathbf{S} with prob. $n/(n+1)$
- So prob. tuple is in \mathbf{S} at time $n+1 = s/n \cdot n/n+1 = s/n+1$

Part 2: Querying Data Streams

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 = 7

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

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

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

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

← Past Future →

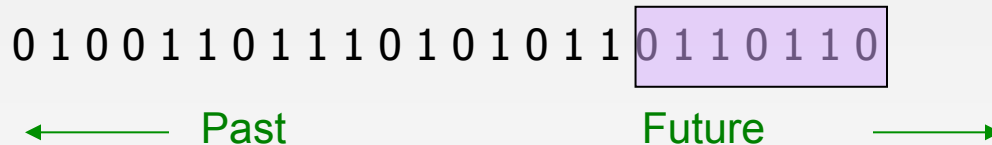
Counting Bits (1)

■ Problem:

- Given a stream of 0s and 1s
- 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 $N+1^{\text{st}}$ bit



Suppose $N=7$

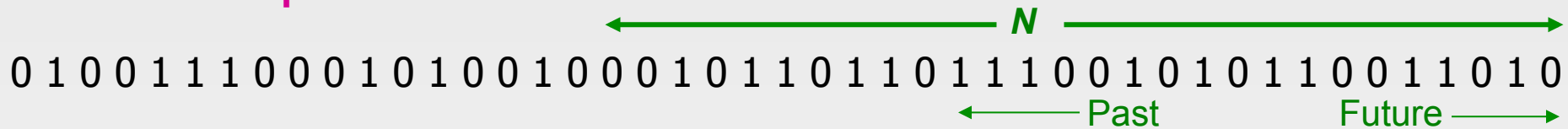
Counting Bits (2)

- 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? $N \cdot S / (S + Z)$
- **But, what if stream is non-uniform?**
 - What if distribution changes over time?

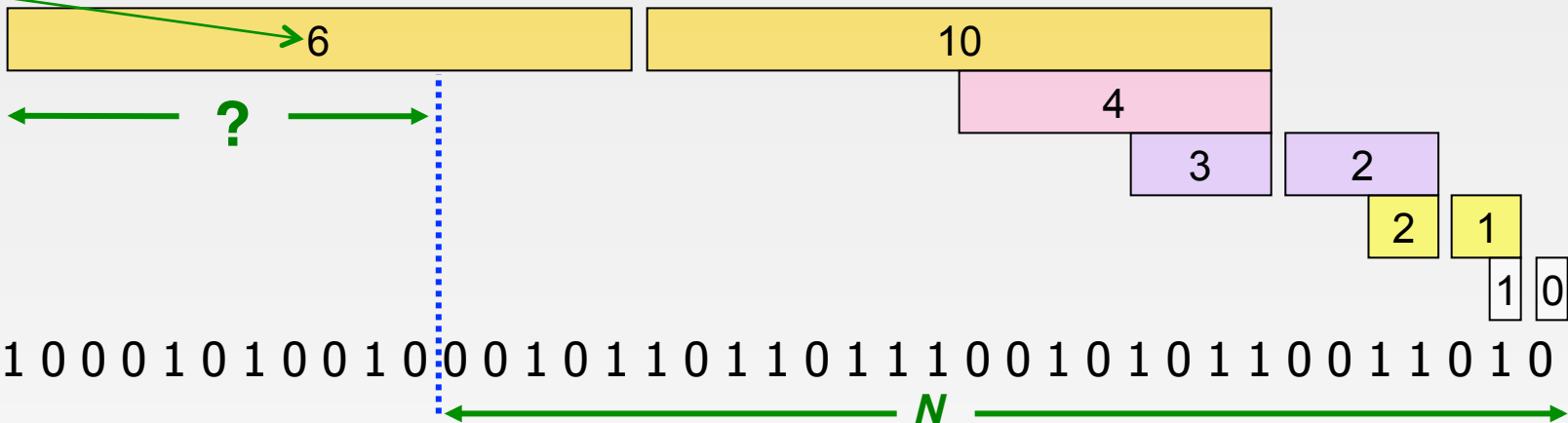
The Datar-Gionis-Indyk-Motwani (DGIM) Algorithm

- Maintaining Stream Statistics over Sliding Windows (SODA'02)
- DGIM solution that does not assume uniformity
- We store $\mathcal{O}(\log 2N)$ 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

Window of width 16 has 6 1s



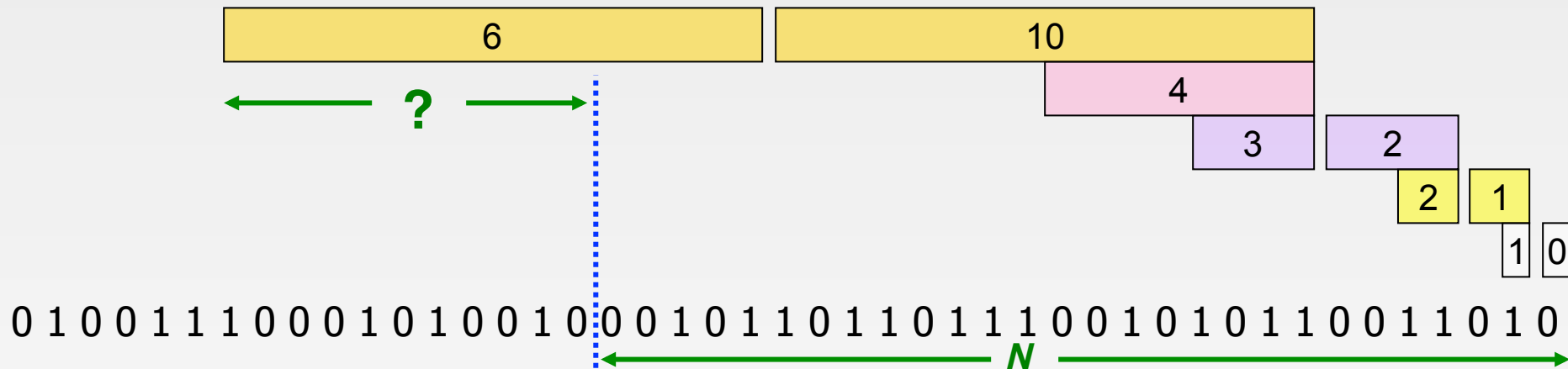
We can reconstruct the count of the last N bits, except we are not sure how many of the last 6 1s are included in the N

What's Good?

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

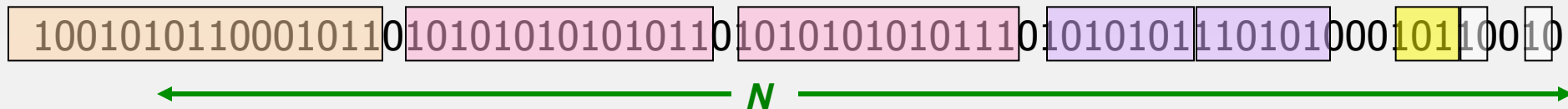
What's Not So Good?

- 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 Algorithm

- **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

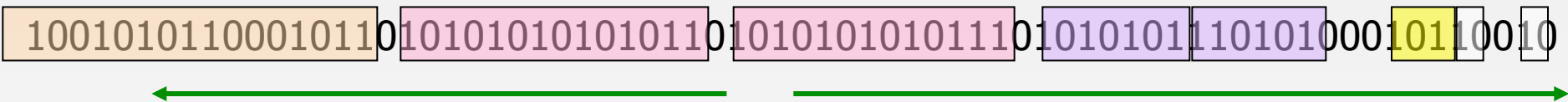


DGIM: Timestamps

- Each bit in the stream has a timestamp, starting from 1, 2, ...
- Record timestamps modulo N (the window size), so we can represent any relevant timestamp in $O(\log_2 N)$ bits
 - E.g., given the windows size 40 (N), timestamp 123 will be recorded as 3, and thus the encoding is on 3 rather than 123

DGIM: Buckets

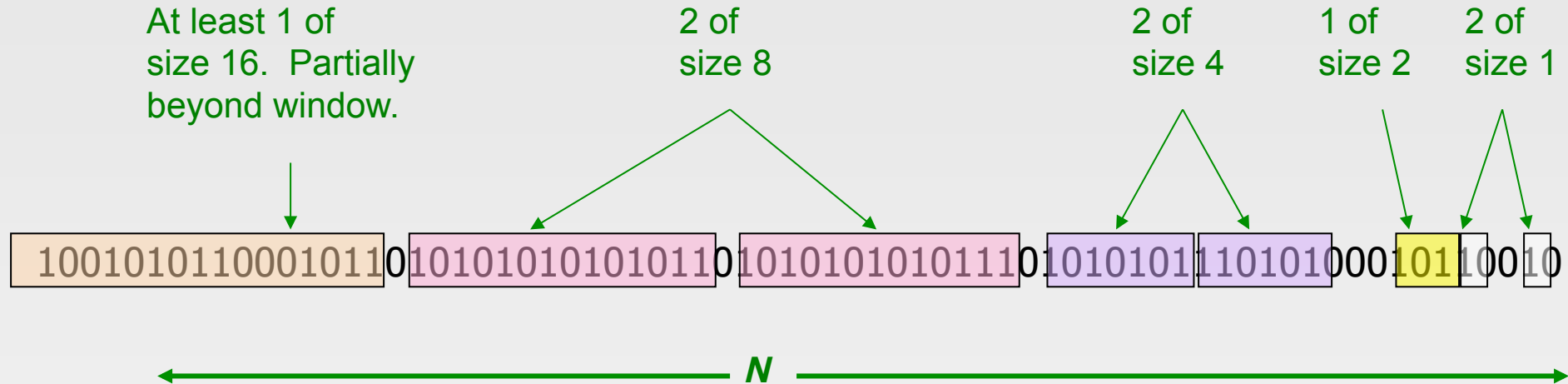
- A bucket in the DGIM method is a record consisting of:
 - (A) The timestamp of its end [$\mathcal{O}(\log N)$ bits]
 - (B) The number of 1s between its beginning and end [$\mathcal{O}(\log \log N)$ bits]
- Constraint on buckets:
 - Number of 1s must be a power of 2
 - That explains the $\mathcal{O}(\log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



Representing a Stream by Buckets

- The right end of a bucket is always a position with a 1
- Every position with a 1 is in some bucket
- 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

- 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
- 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:

100101011000101101010101010101101010101010111010101011101010111010100010110010

Bit of value 1 arrives

0010101100010110101010101010110101010101011101010111010101110101000101100101

Two white buckets get merged into a yellow bucket

00101011000101101010101010101101010101010111010101110101000101100101

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

01011000101101010101010101011010101010111010101110101000101100101101

Buckets get merged...

01011000101101010101010101011010101010111010101110101000101100101101

State of the buckets after merging

01011000101101010101010101011010101010111010101110101000101100101101

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

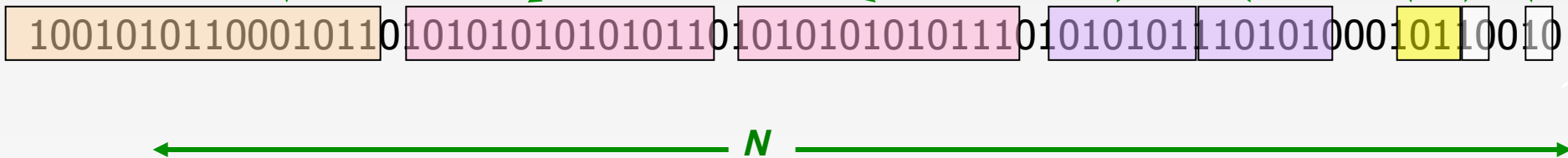
At least 1 of
size 16. Partially
beyond window.

2 of
size 8

2 of
size 4

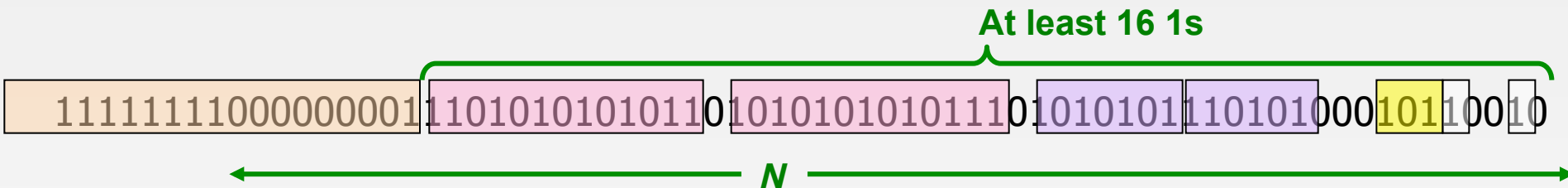
1 of
size 2

2 of
size 1



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
 $1 + 2 + 4 + \dots + 2^{r-1} = 2^r - 1$
- Thus, error at most 50%

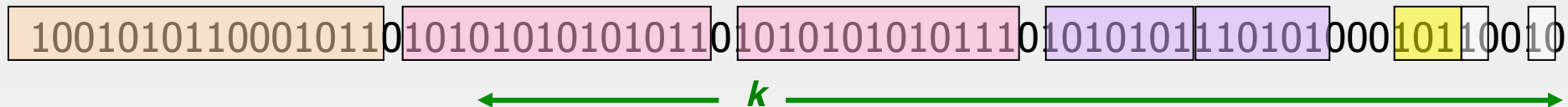


Further Reducing the Error

- Instead of maintaining 1 or 2 of each size bucket, we allow either $r-1$ or r buckets ($r > 2$)
 - Except for the largest size buckets; we can have any number between 1 and r of those
- Error is at most $O(1/r)$
- By picking r appropriately, we can tradeoff between number of bits we store and the error

Extensions

- Can we use the same trick to answer queries **How many 1's in the last k ?** where $k < N$?
 - **A:** Find earliest bucket **B** that at overlaps with k .
Number of **1s** is the **sum of sizes of more recent buckets + $\frac{1}{2}$ size of B**



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

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

Part 3: 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: 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

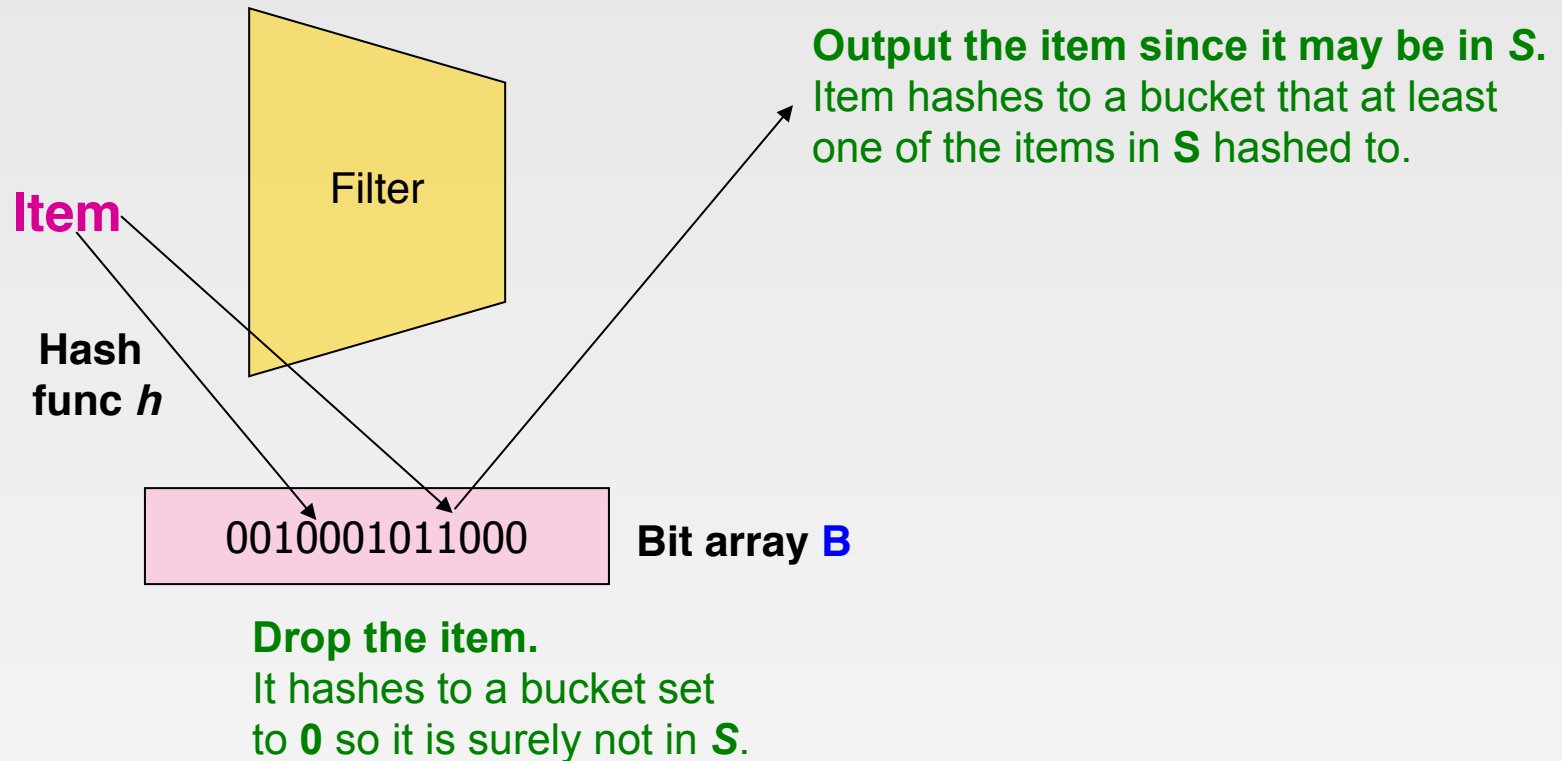
- Example: Email spam filtering
 - We know 1 billion “good” email addresses
 - If an email comes from one of these, it is **NOT** spam

- 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

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 0s
- 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)



■ 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)

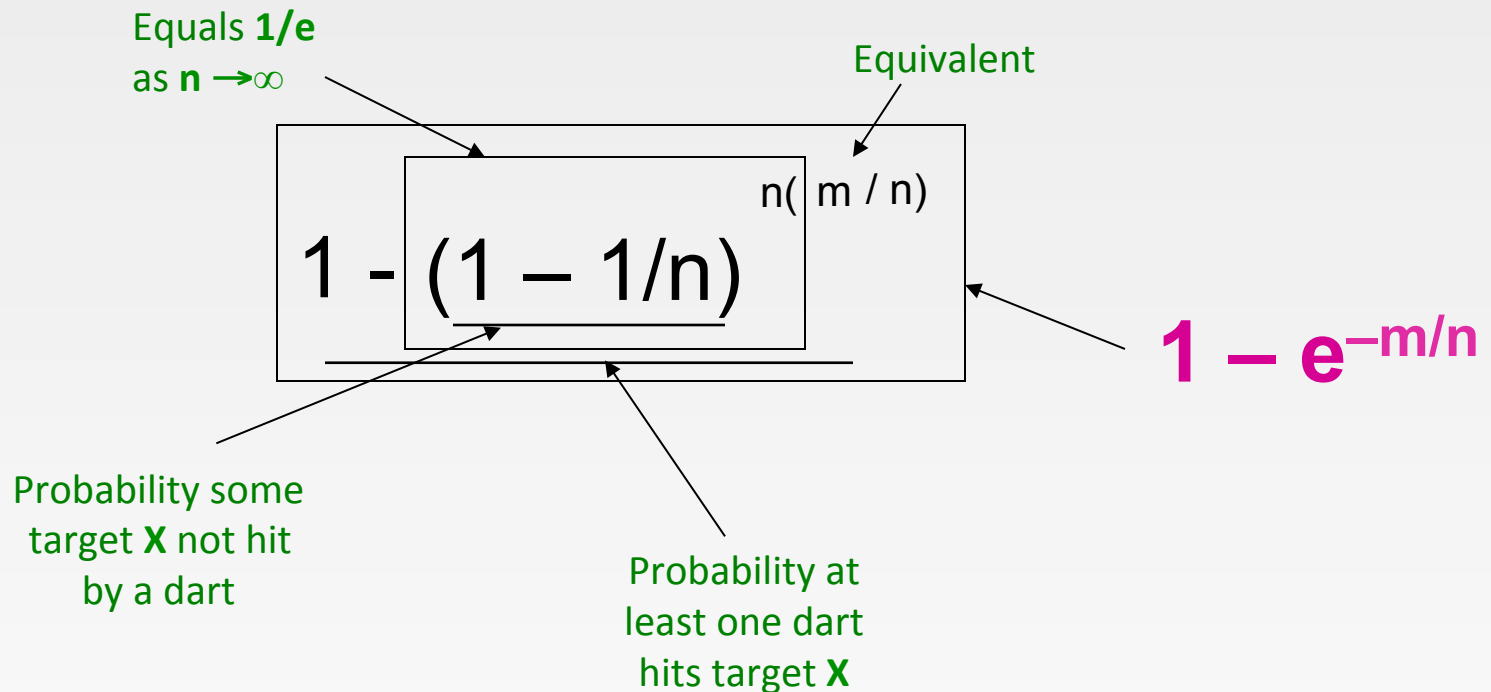
- $|S| = 1$ billion email addresses
 $|B| = 1\text{GB} = 8$ billion bits
- If the email address is in S , then it surely hashes to a bucket that has the big set to 1, so it always gets through (*no false negatives*)
 - False negative: a result indicates that a condition failed, while it actually was successful
- Approximately $1/8$ of the bits are set to 1, so about $1/8$ th of the addresses not in S get through to the output (*false positives*)
 - False positive: a result that indicates a given condition has been fulfilled, when it actually has not been fulfilled
 - Actually, less than $1/8$ th, because more than one address might hash to the same bit
 - Since the majority of emails are spam, eliminating $7/8$ th of the spam is a significant benefit

Analysis: Throwing Darts (1)

- More accurate analysis for the number of **false positives**
- **Consider:** If we throw m darts into n equally likely targets, **what is the probability that a target gets at least one dart?**
- **In our case:**
 - **Targets** = bits/buckets
 - **Darts** = hash values of items

Analysis: Throwing Darts (2)

- We have m darts, n targets
- What is the probability that a target gets at least one dart?



Analysis: Throwing Darts (3)

- Fraction of 1s in the array **B**
= probability of false positive = $1 - e^{-m/n}$
- **Example:** 10^9 darts, $8 \cdot 10^9$ targets
 - Fraction of 1s in **B** = $1 - e^{-1/8} = 0.1175$
 - ▶ Compare with our earlier estimate: $1/8 = 0.125$

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 $i = 1, \dots, k$)
- Run-time:
 - When a stream element with key x arrives
 - ▶ If $B[h_i(x)] = 1$ for all $i = 1, \dots, k$ 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 Example

- Consider a Bloom filter of size $m=10$ and number of hash functions $k=3$. Let $H(x)$ denote the result of the three hash functions.

- The 10-bit array is initialized as below

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- Insert x_0 with $H(x_0) = \{1, 4, 9\}$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

- Insert x_1 with $H(x_1) = \{4, 5, 8\}$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

- Query y_0 with $H(y_0) = \{0, 4, 8\} \Rightarrow ???$

- Query y_1 with $H(y_1) = \{1, 5, 8\} \Rightarrow ???$ **False positive!**

- Another Example: <https://lilmlib.github.io/bloomfilter-tutorial/>

Bloom Filter – Analysis

- What fraction of the bit vector B are 1s?
 - Throwing $k \cdot m$ darts at n targets
 - So fraction of 1s is $(1 - e^{-km/n})$
- But we have k independent hash functions and we only let the element x through if all k hash element x to a bucket of value 1
- So, false positive probability = $(1 - e^{-km/n})^k$

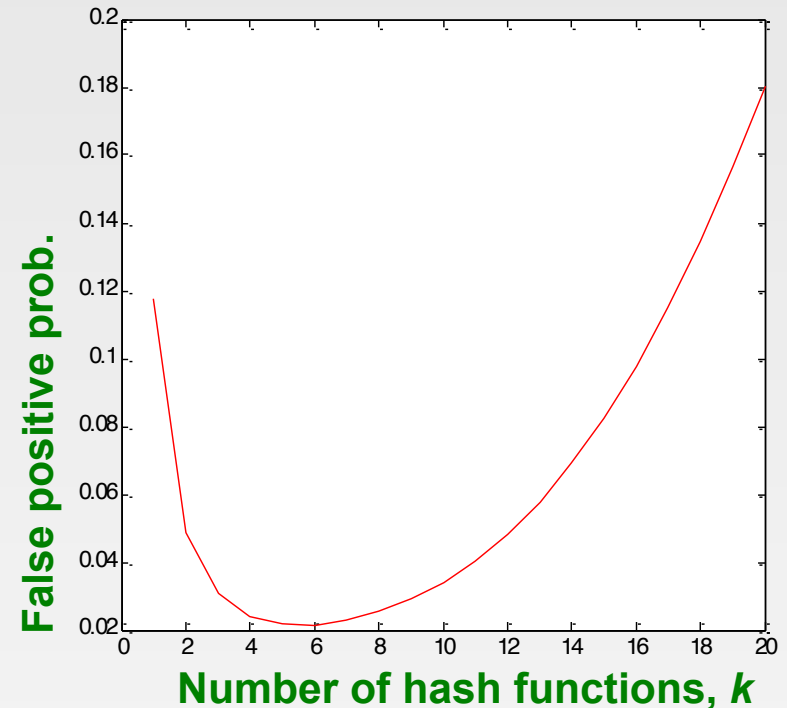
Bloom Filter – Analysis (2)

■ $m = 1$ billion, $n = 8$ billion

● $k = 1$: $(1 - e^{-1/8}) = 0.1175$

● $k = 2$: $(1 - e^{-1/4})^2 = 0.0493$

■ What happens as we keep increasing k ?



■ “Optimal” value of k : $n/m \ln(2)$

● In our case: Optimal $k = 8 \ln(2) = 5.54 \approx 6$

▶ Error at $k = 6$: $(1 - e^{-1/6})^2 = 0.0235$

Bloom Filter: Wrap-up

- Bloom filters 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 B or k small Bs?
 - It is the same: $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
 - But keeping 1 big B is simpler

References

- Chapter 4, Mining of Massive Datasets.

End of Chapter 9

Part 4: Counting Data Streams (Sketch)

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

■ Example:

Data stream:

3 2 5 3 2 1 7 5 1 2 3 7

Number of distinct values: 5

- ## ■ 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

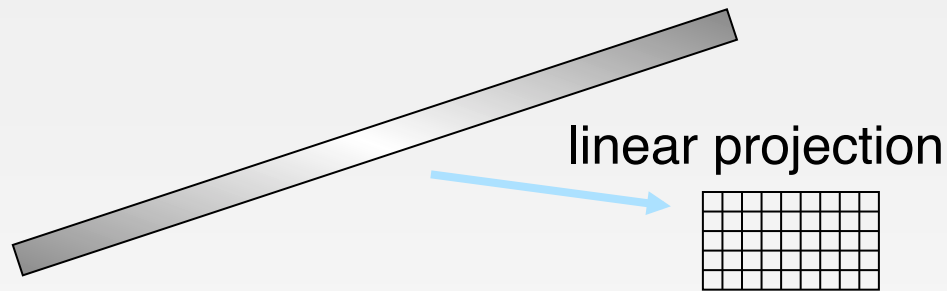
- How many different words are found among the Web pages being crawled at a site?
 - Unusually low or high numbers could indicate artificial pages (spam?)
- How many different Web pages does each customer request in a week?
- How many distinct products have we sold in the last week?

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

Sketches

- Sampling does not work!
 - If a large fraction of items aren't sampled, don't know if they are all same or all different
- Sketch: a technique takes advantage that the algorithm can “see” all the data even if it can't “remember” it all
- Essentially, sketch is a linear transform of the input
 - Model stream as defining a vector, sketch is result of multiplying stream vector by an (implicit) matrix



Flajolet-Martin Sketch

- Probabilistic Counting Algorithms for Data Base Applications. 1985.
- 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

Why It Works: Intuition

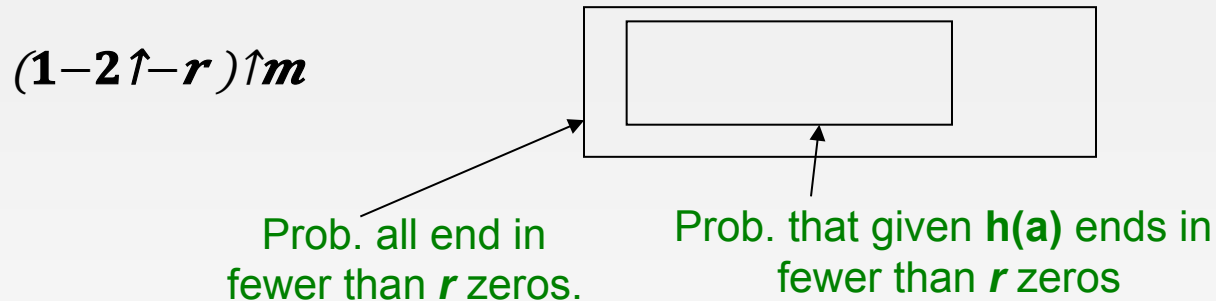
- Very very rough and heuristic intuition why Flajolet-Martin works:
 - $h(a)$ hashes a with **equal prob.** to any of N values
 - Then $h(a)$ is a sequence of $\log_2 N$ bits, where 2^{-r} fraction of all a s have a tail of r zeros
 - ▶ About 50% of a s hash to *****0**
 - ▶ About 25% of a s hash to ****00**
 - ▶ So, if we saw the longest tail of $r=2$ (i.e., item hash ending ***100**) then we have probably seen **about 4** distinct items so far
 - So, it takes to hash about 2^r items before we see one with zero-suffix of length r

Why It Works: More formally

- Formally, we will show that **probability of finding a tail of r zeros:**
 - Goes to **1** if $m \gg 2^r$
 - Goes to **0** if $m \ll 2^r$where m is the number of distinct elements seen so far in the stream
- Thus, 2^R will almost always be around m !

Why It Works: More formally

- The probability that a given $h(a)$ ends in at least r zeros is 2^{-r}
 - $h(a)$ hashes elements uniformly at random
 - Probability that a random number ends in at least r zeros is 2^{-r}
- Then, the probability of **NOT** seeing a tail of length r among m elements:

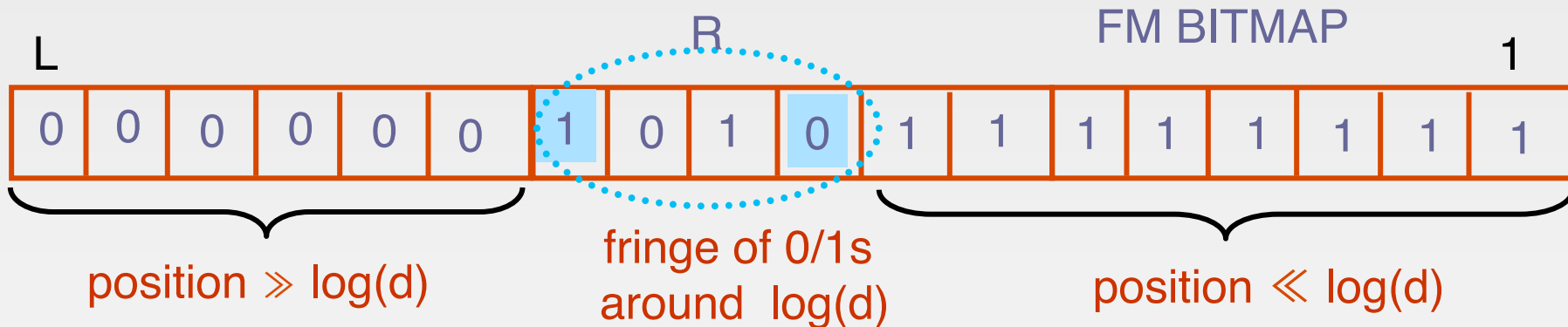


Why It Works: More formally

- **Note:** $(1 - 2^{-r})^m = (1 - 2^{-r})^{2^r (m2^{-r})} \approx e^{-m2^{-r}}$
- **Prob. of NOT finding a tail of length r is:**
 - If $m \ll 2^r$, then prob. tends to 1
 - ▶ $(1 - 2^{-r})^m \approx e^{-m2^{-r}} = 1$ as $m/2^r \rightarrow 0$
 - ▶ So, the probability of finding a tail of length r tends to 0
 - If $m \gg 2^r$, then prob. tends to 0
 - ▶ $(1 - 2^{-r})^m \approx e^{-m2^{-r}} = 0$ as $m/2^r \rightarrow \infty$
 - ▶ So, the probability of finding a tail of length r tends to 1
- **Thus, 2^R will almost always be around m !**

Flajolet-Martin Sketch

- Maintain FM Sketch = bitmap array of $L = \log N$ bits
 - Initialize bitmap to all 0s
 - For each incoming value a , set $\text{FM}[r(a)] = 1$
- If d distinct values, expect $d/2$ map to $\text{FM}[1]$, $d/4$ to $\text{FM}[2]$...



- Use the leftmost 1: $R = \max_a r(a)$
- Use the rightmost 0: also an indicator of $\log(d)$
 - ▶ Estimate $d = c2^R$ for scaling constant $c \approx 1.3$ (original paper)
- Average many copies (different hash functions) improves accuracy