Lec 08–11: Mining Data Streams

COSC–254 – February 18–27, 2019

Outline

Data streams: motivation, applications, model(s), queries

Approximate *query answering*: reservoir sampling

Approximate *set membership*: Bloom filters

Approximate *distinct counting*: The Flajolet-Martin approach

Approximate *counting* on *sliding windows* : The DGIM Algorithm

Data streams

Sensor data: *continuously* transmit (measurements of) quantities of interest

temperature, location, traffic, stock prices, web search queries, …

Stream of data elements:

$$e_{t-2},\quad e_{t-1},\quad \underbrace{e_t}_{\text{Element seen at time } t\geq 0},\quad e_{t+1},\quad \ldots \xrightarrow{\text{time}}$$

Example: elements are *tuples* of $\underbrace{\text{(temperature, wind speed, humidity)}}_{\text{3-tuple}}$

$$\underbrace{(15^\circ, 20\text{mph}, 52\%)}_{e_t},\quad \underbrace{(18^\circ, 10\text{mph}, 64\%)}_{e_{t+1}},\quad \ldots$$

Data streams

The dataset is *never complete*: data points are appended at each timestep:

$$\underbrace{\mathcal{D}_{t+1}}_{\text{Dataset at time } t+1} = \underbrace{\mathcal{D}_t}_{\text{Dataset at time } t} \cup \{ \underbrace{e_{t+1}}_{\text{Element seen at time } t+1} \}$$

$D_0 = \emptyset, D_1 = \{e_1\}, \ldots$

TASK: *for each* t, compute quantity/ies of interest $q = f(\mathcal{D}_t)$ ((standing) *queries*).

EXAMPLE: wind-chill at each time t, average temperature $\underbrace{\text{over the past 7 days}}_{\textit{sliding window}}$.

Data streams

Properties of the data that make the task hard:

1) The data is essentially *infinite*;
2) Input elements arrive *very fast*
   (think: Instagram photos, stock prices, security camera frame)

Consequences:

1) *cannot store the entire stream* accessibly;
2) must *compute query answer fast*.

Stream processing model

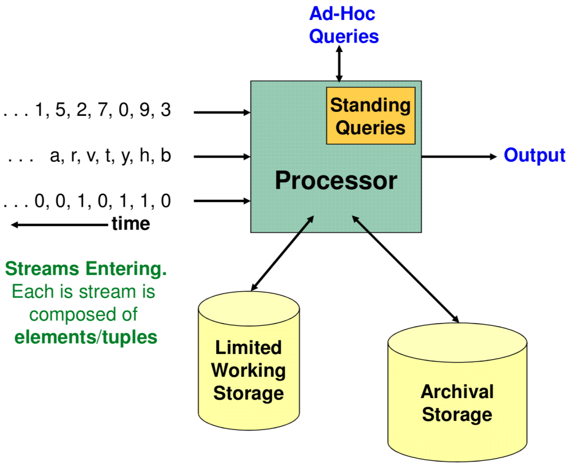


Figure from slides at http://mmds.org

Queries

Filtering: select all elements with property x

Counting distinct elements (possibly in the last k elements seen)

Moment estimation: estimate the average or the standard deviation
(possibly of the last k elements)

Find frequent elements

Applications

How many distinct users visited my website in the last month?

Mining streams of web search queries:
  what queries are *more frequent* today than yesterday?

Mining click streams:
  what web pages are getting an *unusual* number of hits in the past three hour?

Mining social network status updates:
  is there an earthquake happening right now in California? A protest in Cairo?

More applications

Sensor networks:

With a million of sensors sending 4 bytes every 1/10 of seconds,
you get a million data points per 1/10 of second, 3.5 terabytes per day.

IP packets monitored at a switch:

Is there a flow of packets that would benefit from different routing decision?

Are there unusual patterns in the flow ? (denial-of-service attacks)

Query answers

Answering queries *exactly* may not always be possible because of

1) the *limited working space*

2) computing the exact answer $q_t = f(\mathcal{D}_t)$ may *take too long*

EXAMPLE: Count the distinct elements.

Can't count exactly if the set of distinct elements is larger than the number of elements I can store

EXAMPLE: Mine the frequent itemsets from the last k elements.

Would take too long

Approximations

ISSUE: Impossible to compute the exact answer and compute it fast.

SOLUTION: Compute *approximate answer* $\tilde{q}_t = \tilde{f}(\mathcal{D}_t)$

$$\tilde{q}_t \approx q_t \quad \text{for every } t > 0$$

COMPUTER SCIENTIST TASK: Given a query f, design an algorithm $\tilde{f}$ that:

1) "approximate" f for *all possible* input datasets;

2) uses a *small working space*;

3) is *fast* in computing the approximation

Why shall we be happy with approximate answers?

1) We cannot compute anything else;

2) *High-quality* approximations are still *very useful*;

3) Exact answers have *little value* in a streaming setting;

Outline

✔ *Data streams*: motivation, applications, model(s), queries

Approximate *query answering*: reservoir sampling

Approximate *set membership*: Bloom filters

Approximate *distinct counting*: The Flajolet-Martin approach

Approximate *counting* on *sliding windows*: The DGIM Algorithm

Subs

We cannot store the whole stream? Let's store a *subset* $\mathcal{S}_t$ *of* $\mathcal{D}_t$

How to compute the approximate answer $\tilde{q}_t$?

Possible answer: $\tilde{q}_t = f(\mathcal{S}_t)$ (i.e., use the same f)

Example:

$f =$ average: $\tilde{q}_t$ works well on *some* subsets;

We need to build and keep a *representative subset*

Samples

Easiest way to build a representative subset (*sample*): select one *(uniformly) at random*

Uniform sampling:

- Each element has *equal probability* of being in the sample (being *sampled*)
  - Equivalently: each subset of a fixed size has equal probability of being the sample.

How to create a random sample?

Approach 1: select a *fixed proportion* of elements in the stream (e.g., 1 in 10)

Approach 2: Maintain a random sample of *fixed size*

Quick probability primer

If two events cannot happen simultaneously, they are called *disjoint*.

The probability that at least one of two (or more) disjoint events happens is the sum of their individual event probabilities.

We will always deal with *independent* events. The formal definition is not important for our purposes. What is important is that the probability of two or more independent events happening at the same time is the product of their individual event probabilities.

Sampling a fixed proportion

SCENARIO: Web search query stream:

$$(\text{user}_t, \text{search}_t), \qquad (\text{user}_{t-1}, \text{search}_{t+1}), \qquad \ldots$$

Query: What *fraction* of the typical user's queries are *repeated*?

Naïve approach to build the sample:

For each t, generate a random integer i_t from $[0 \ldots 9]$

Add the element e_t to $\mathcal{S}_t$ if $i_t = 0$.

Answering the query:

for each user u, count the fraction r_u of repeated queries in $\mathcal{S}_t$,

then take the average of r_u over the users

Issues with the naïve approach

Suppose *each user* issues x queries once, and d queries twice (total $x + 2d$ queries)

Correct query answer: $\dfrac{d}{(x+d)}$

The “typical” sample will contain (typical: informal way to say “in expectation”):

$x/10$ of the singleton queries

$d/100$ pairs of duplicates ($d/100 = d \times \underbrace{(1/10 \times 1/10)}_{\text{probability of sampling both queries appearances}}$)

$18d/100$ of the d duplicates, each appearing exactly once.

$$18d/100 = d \times (\underbrace{(1/10 \times 9/10) + (9/10 \times 1/10)}_{\text{probability of sampling exactly one of the query appearances}})$$

(Typical / Expected) naïve approach query answer:

$$\frac{\frac{d}{100}}{\frac{x}{10} + \frac{d}{100} + \frac{18d}{100}} = \frac{d}{10x + 19d}$$

Solution: sample users!

Pick $1/10^{th}$ of *users* and add *all* their searches to the sample.

How to decide whether a user is one of the “sampled” one?

Use a *hash function* h that hashes user names *uniformly* into 10 buckets
If the $h(\text{user}_t) = 0$, add $e_t = (\text{user}_t, \text{search}_t)$ to $\mathcal{S}_t$.

Generalized solution

Stream of tuples with *keys*:

- Key is a subset of the components of each tuple (e.g., user_t)
  - 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 $[0, \ldots, b-1]$
- Add the tuple to the sample if the hash value is less than a

Example: To generate a 30% sample, what is b and what is a?

- $b = 10$ and $a = 3$.

Fixed-size sample

A problem with the previous approach is that the size of the sample grows with time.

Our memory may not grow as fast. It may even be fixed to exactly s tuples.

How to build a *fixed-size random sample* that is *representative* of all elements seen so far?

For all time steps k,
each of the k elements seen so far *must have the same probability* of being in S_k.

Reservoir sampling

ALGORITHM:

$\mathcal{S} \leftarrow \emptyset$

If $t \leq s$, store the e_t in $\mathcal{S}$

Else `// i.e., when` $t > s$

flip a *biased coin* that has probability of head equal to s/t

If outcome is tail, discard e_t

Else `// i.e., when outcome is head`

choose an element of $\mathcal{S}$ *uniformly at random* and replace it with e_t

LEMMA:

At each time t, $\mathcal{S}_t$ is such that each element e_k, for $k \leq t$, has probability $\min\{1, s/t\}$ of being in $\mathcal{S}_t$.

PROOF: Next time

Decreasing probabilities

Why should the probability s/t of modifying the sample *decrease* as t grows?

Consider $Z = \{e_1, \ldots, e_t\}$.

There are $\binom{t-1}{s}$ subsets of Z of size s that *do not* contain e_t.

There are $\binom{t-1}{s-1}$ subsets of Z of size s that *do* contain e_t.

It holds:

$$\frac{\binom{t-1}{s}}{\binom{t-1}{s-1}} = \frac{(t-1)!}{s!(t-1-s)!}\frac{(s-1)!(t-s)!}{(t-1)!} = \frac{t}{s}$$

which grows with t: as t grows, there are more and more samples that *do not* contain e_t, so the probability of choosing a sample that contains e_t must go down (and it does!).

Proof: By induction

LEMMA:

At each time t, $\mathcal{S}_t$ is such that each element e_k, for $k \leq t$, has probability $\min\{1, s/t\}$ of being in $\mathcal{S}_t$.

PROOF IDEA:

1) Show that the property holds for all $t \leq s$ (*base case*).

2) Assume that the property holds for all t from 0 to a generic $z - 1 > s$ (*inductive hypothesis*

3) Show that the property holds for $t = z$ (*inductive step*)

Base case

For every $t \leq s$, $\mathcal{S}_t$ contains *all* elements seen so far:

$$\mathcal{S}_t = \{e_1, e_2, \ldots, e_t\}$$

So the probability of e_k of being in $\mathcal{S}_t$ is $1 = \min\{1, s/t\}$, for $k \leq t$

Inductive hypothesis / step

Inductive hypothesis: At time $t-1 > s$, each element e_k, for $k \leq t-1$, has probability $s/(t-1) = \min\{1, s/(t-1)\}$ of being in $\mathcal{S}_{t-1}$

Now element e_t arrives

Inductive step: The probability of any element already in $\mathcal{S}_{t-1}$ to be in $\mathcal{S}_t$ is:

$$\underbrace{\left(1-\frac{s}{t}\right)}_{\text{tail}} + \underbrace{\frac{s}{t}}_{\text{head}} \times \underbrace{\frac{s-1}{s}}_{\text{not selected}} = \frac{t-1}{t}$$

(This is the *conditional* probability of an element to be in S_t *given* that it was in S_{t-1})

The probability for any e_k, for $k < t$ to be in $\mathcal{S}_t$ is then

$$\underbrace{\frac{s}{t-1}}_{\text{prob. of being in } \mathcal{S}_{t-1}} \times \frac{t-1}{t} = \frac{s}{t}$$

For $k = t$, the probability is obviously s/t

Outline

✔ *Data streams*: motivation, applications, model(s), queries

✔ Approximate *query answering*: reservoir sampling

Approximate *set membership*: Bloom filters

Approximate *distinct counting*: The Flajolet-Martin approach

Approximate *counting* on *sliding windows*: The DGIM Algorithm

Hash functions

Given a *universe* U of *keys*, and a number of *buckets* b, a *hash* function h maps U to $\{0, 1, \ldots, b-1\}$

For a key $k \in U$, we call $h(k)$ the *hash (value)* of k

Typically, $|U| > b$, so there may be two keys x, y with $h(x) = h(y)$, i.e., a *collision*

There exists efficient ways to build *independent* hash functions that have "good" collision properties (*universal hashing*)

We will *assume* that $h(k)$ is chosen uniformly at random from $\{0, 1 \ldots, b-1\}$

Set membership

Given a fixed list of keys S, for each tuple t on the stream determine if $t \in S$.

Data structure for set membership: *hash table* for S

Data stream issue: *not enough memory* to store the hash table for S

Set membership application

Data stream issue: *not enough memory* to store the hash table for S

Application: you distribute news articles to 200million subscribers.

Each subscriber x is interested in a (potentially large) set S_x of keywords

For each news article t, for each subscriber x, determine whether the article matches their interest (i.e., if $t \in S_x$)

You cannot store 200million keywords sets exactly.

First cut solution

Given: A set S of m keys, (working) space to store n bits.

Initialization (before the stream starts):

- Create an array B of n bits, all set to 0
- *Pick* a hash function h over n buckets ($h : U \to \{0, 1, \ldots, n-1\}$)
- For each key $k \in S$, set $B[h(k)] = 1$

For each element e of the stream:

- Compute $h(e)$, and output e if and only if $B[h(e)] == 1$

First cut solution

Output e if and only if $B[h(e)] == 1$

Q: Do we may any mistake?

If $B[h(e)] == 0$, then surely $e \notin S$.

If $B[h(e)] == 1$, then we do not know:

There may have been a *collision*: $e \notin S$ but there exists $k \in S$ such that $h(k) = h(e)$.

But if $k \in S$, then $B[h(k)] == 1$, so we output k for sure.

One-sided error: no *false negatives*, potentially some *false positives*

How many false positives?

Recall assumption: $h(z)$ chosen uniformly at random from $\{0, \ldots, n-1\}$ for each $z \in U$

For $k \notin S$, the probability that k is a false positive is the probability that the bit $B[h(k)]$ is 1 (because one (or more) key z in S has $h(z) = h(e)$).

For any fixed bit $B[i]$, what is the probability that $B[i]$ is set to 1 due to some key in S?

$$p_{m,n} = 1 - \underbrace{\left(\underbrace{1 - \frac{1}{n}}_{\text{Pr. a \textit{specific} key } k \in S \text{ has } h(k) \neq i} \right)^m}_{\text{Pr. \textit{every} key } k \in S \text{ has } h(k) \neq i\text{, i.e., Pr. that } B[i] == 0} \approx 1 - \underbrace{e^{-m/n}}_{\text{because}(1-1/n)^n \approx e^{-1}}$$

The typical/expected fraction of false positives is $p_{m,n}$.

Bloom Filter

IDEA: Don't use just one hash function, use q *independent* $h_1, \ldots h_q$ hash funcs.

Initialization (before the stream starts):

Create *one* array B of n bits, all set to 0

Pick q *independent* hash functions $h_1, \ldots, h_1$ over n buckets

For each key $k \in S$, for each $i = 1, \ldots, q$, set $B[h_i(k)] = 1$

For each element e of the stream:

Compute $h_1(e), \ldots, h_q(e)$,

Output e if and only if $B[h_i(e)] == 1$ for all $i = 1, \ldots, q$

Analysis of the Bloom filter

The probability that a bit is set to 1 is now

$$1 - \left(1 - \frac{1}{n}\right)^{qm} \approx 1 - e^{-qm/n} \quad \text{(much higher than before)}$$

But $k \notin S$ is a false positive only if *all* q of the bits $h_1(k), \ldots, h_q(k)$ are set to 1.

The probability that all the q bits $h_1(k), \ldots, h_q(k)$ are set to 1 is

$$\left(1 - \left(1 - \frac{1}{n}\right)^{qm}\right)^q \approx \left(1 - e^{-qm/n}\right)^q$$

Hopefully, this probability of a false positive would be smaller than when we used a single hash function.

How many hash functions?

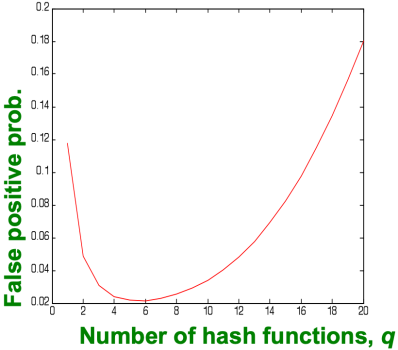
$$\left(1-\left(1-\frac{1}{n}\right)^{qm}\right)^{q} \approx \left(1-e^{-qm/n}\right)^{q}$$

$m = 10^9$, $n = 8 \times 10^9$

$q = 1$: $(1 - e^{-1/8})^1 = 0.1775$

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

What about higher values of q?



Optimal value of q: $\dfrac{n}{m}\ln 2$

For m and n as above, optimal $q = 8\ln 2 = 5.54 \approx 6$,

Prob. of a false positive for $q = 6$: $(1 - e^{6/8})^6 = 0.02156$

Image from slides at `http://mmds.org`

Application

You run a web caching service, but do not want to store web objects that are only requested once (3/4 of the URLs)

Keep a Bloom filter B of objects seen at least once:

- Every time you are requested for any object z, compute its hash(es) $h_1(z), \ldots, h_q(z)$;
- If *not* all the bits $h_1(z), \ldots, h_q(z)$ are 1, set them all to 1, fetch and return z, and *do not store* z in the cache;
- If all the bits $h_1(z), \ldots, h_q(z)$ are 1, look for z in the cache, and if z is not there, fetch z and store z in the cache.

Bloom filter saves bout 1/2 of disk writes $\longrightarrow$$$$

Union of Bloom filters

Take filters B_1 and B_2, for sets S_1 and S_2, using the same $h_1, \ldots, h_q$.

Bloom filter $B_{1\cup 2}$ for $S_1 \cup S_2$: set $B_{1\cup 2}[i]$ to 1 if either $B_1[i] = 1$ or $B_2[i] = 1$.

Outline

✔ *Data streams*: motivation, applications, model(s), queries

✔ Approximate *query answering*: reservoir sampling

✔ Approximate *set membership*: Bloom filters

Approximate *distinct counting*: The Flajolet-Martin approach

Approximate *counting* on *sliding windows*: The DGIM Algorithm

Counting distinct elements

The elements on the data stream come from a universe of size N.

At each time t, how many distinct elements have we seen so far?

Immediate solution: keep a *hash table* of the observed distinct elements

Applications:

Distinct active users, distinct products sold, distinct words in a web page, …

Small storage

What if we if do not have enough working space to maintain the hash table?

We want to *estimate* the distinct count in an *unbiased* way

Accept that the estimate will not be exact (duh!), but *with high probability* the error will be *small*.

Flajolet-Martin approach

Initialization:

Pick a hash function h from the N possible keys ($|U| = N$) to (at least) $\lceil \log_2 N \rceil$ bits.

I.e., $h : U \to \{0, \ldots, 2^{\lceil \log_2 N \rceil} - 1\}$, but we consider the *binary representation* of $h(k)$.

Initialize an integer variable R to 0

For each element e on the stream:

Compute $h(e)$, and let $r(e)$ be the number of *trailing zeros* in $h(e)$

E.g., if $h(e) = 0100$, then $r(e) = 2$

If $r(e) > R$, set $R = r(e)$ (i.e., $R_t = \max\{r(e_z), z \leq t\}$)

Estimation of number of distinct elements: 2^R

Analysis

What is the probability that $h(k)$ ends with *at least* z zeroes, for $z \in \{0, \ldots, \lceil \log_2 N \rceil\}$?

Assumption: h maps k to a bucket in $\{0, \ldots, 2^{\lceil \log_2 N \rceil} - 1\}$ chosen uniformly at random.

Equivalently: h maps k to a sequence of $\lceil \log_2 N \rceil - 1$ bits chosen uniformly at random.

How can we choose such a sequence uniformly at random?

Analysis

How do we chose a sequence of $\lceil \log_2 N \rceil - 1$ bits uniformly at random?

We flip one *unbiased* coin for each bit: if head, set the bit to 1, if tail, unset the bit to 0.

The probability that $h(k)$ ends with *at least* z zeroes is the probability that the z coins for the last z bits *all came out tail*, which is

$$\prod_{i=1}^{z} \Pr(\text{bit } i \text{ is zero}) = \left(\frac{1}{2}\right)^{z} = 2^{-z}$$

Analysis

$$\Pr(h(k) \text{ ends with at least } z \text{ zeroes}) = 2^{-z}$$

The probability that *none* of m (distinct) keys has a tail of at least z zeroes is

$$\Big(\underbrace{1 - 2^{-z}}_{\text{Pr. that } h(k) \text{ ends in } \textit{less} \text{ than } z \text{ zeroes}}\Big)^{m} \approx e^{-m/2^{z}}$$

Analysis

$$\Pr(\textit{none} \text{ of } m \text{ distinct keys has a tail of at least } z \text{ zeroes}) = e^{-m/2^z}$$

Let m_t be the number of distinct elements passed on the stream by time t
(we don't know m_t, the goal is to show that 2^{R_t} is a good estimate for it)

For any $z \in \{0, \ldots, \lceil \log_2 N \rceil - 1\}$, the probability that among these m_t elements there is *at least one* with a tail of at least z zeroes is

$$1 - e^{-m_t/2^z} = 1 - \frac{1}{e^{m_t/2^z}}$$

If $2^z \ll m_t$, this probability is ≈ 1 $\Rightarrow R_t > z$ with probability ≈ 1.

If $2^z \gg m_t$, this probability is ≈ 0 $\Rightarrow R_t < z$ with probability ≈ 1.

So 2_t^R is always close to m_t: our estimation is good!

Outline

✔ *Data streams*: motivation, applications, model(s), queries

✔ Approximate *query answering*: reservoir sampling

✔ Approximate *set membership*: Bloom filters

✔ Approximate *distinct counting*: The Flajolet-Martin approach

Approximate *counting* on *sliding windows*: The DGIM Algorithm

Sliding windows

Until now, queries were about the whole stream.

Shift focus to *recent* past: queries about the N most-recent elements.

Application: For every bug x, how many times have we seen x in the last N observations? There is a *window* of length N that *slides* over the stream.

At time t, the window w_t goes from e_{t-N+1} to e_t.

Sliding window

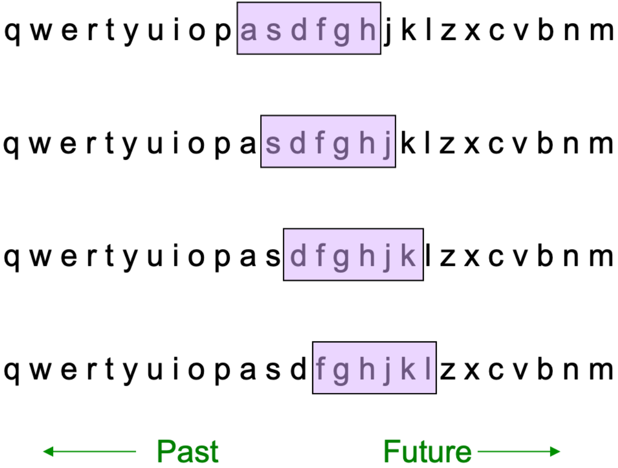


Image from slides at http://mmds.org.

Counting bits

Assume the stream elements come from $\{0, 1\}$ (bits):

$$0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad \dots$$

How many 1s are there in the last N bits?

If N is smaller than our working space (of size n), the solution is obvious:

Always store the most recent n bits in order of arrival.

We can even answer more queries than requested. Not an interesting situation.

If $N > n$, we *cannot* compute the *exact* count.

Must accept an *approximate answer*.

Counting bits: solution with assumption

Assumption: the bits are distributed *uniformly* across the stream:

there is a *fixed* probability π that the next bit will be a 1,

bits are *independent*

Expected number of 1s in the last N bits: $N\pi$.

Solution: Maintain 2 counters:

s: number of 1s from the *beginning* of the stream

z: number of 0s from the beginning of the stream ($s_t + z_t = t$)

At time t, *estimate* π as $\tilde{\pi}_t = \dfrac{s_t}{s_t + z_t}$, and the number of 1s in the last N as $N\tilde{\pi}_t$.

Why does it work? $\tilde{\pi}_t \xrightarrow{t\to\infty} \pi$, at a rate of $1/t^2$.

DGIM method

What if the uniformity assumption does not hold? (E.g., π changes over time)

Datar-Gionis-Indyk-Motwani: *no assumption* (i.e., *adversarial* stream)

GOOD PROPERTIES:

1) stores $O(\log^2 N)$ bits;

2) estimates $\tilde{m}_t$ are *never more than* 50% *off* from exact answer m_t:

$$\frac{|\tilde{m}_t - m_t|}{m_t} \leq \frac{1}{2}$$

(i.e., *2-approximation*):

Can reduce the error with more convoluted algos & more bits.

Blocks

A *block* is a contiguous subset of the stream, of some *length*.

The *start* of a block is farther in the past than the block's *end*

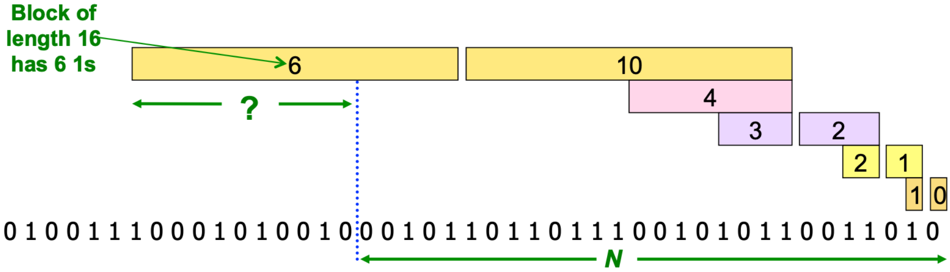
$$0 \quad 1 \quad \underbrace{\underset{\text{start}}{1} \quad 0 \quad 0 \quad 0 \quad \underset{\text{end}}{1}}_{\text{block of length } 5} \quad 1 \quad \ldots$$

Blocks *do not slide*: start and end are *fixed times* (e.g., 3 and 7)

The *weight* of a block is is the number of 1s in it.

Idea: Exponential windows

(Doesn't work, but has some merits)



At any time t, keep a set of blocks "covering" the window w_t.

For each block, keep a *summary*: start time, end time, weight.

The *oldest* block may *start* before the window starts:
We do not know *how many* of its 1s are *in the window*.

Timestamps

For each block, keep a *summary*: start times, end time, weight.

How much *space* does each summary take?

As t grows, storing the start/end times takes *more and more space*: not good!

Apart from the start of the oldest block, all times to be stored are in $[t-N+1, t]$

Let's split the stream into "days" of N "hours" (1 hour = 1 time unit)

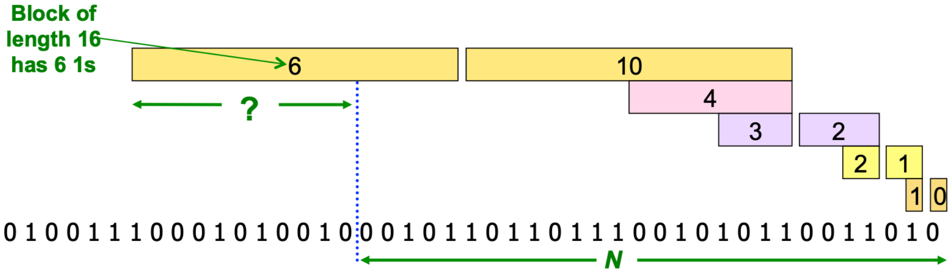
Store a time x as the *hour* of its day: as $x \mod N$.

E.g., for $N = 5$:

x	…	7	8	9	10	11	12	13	14	15	…
$x \mod 5$	…	2	3	4	0	1	2	3	4	0	…

We only need $O(\log N)$ bits for each time!

Idea: Exponential windows

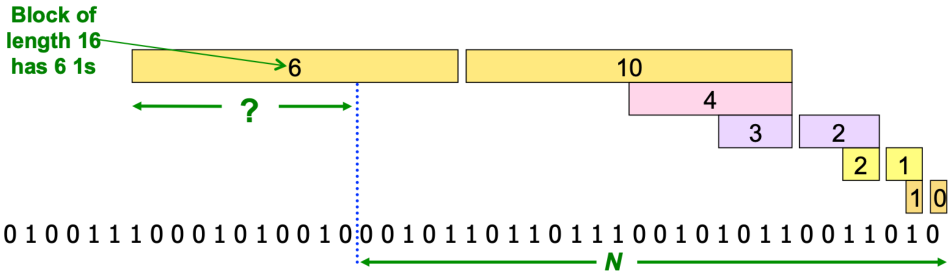


INVARIANT 1: For $i \geq 0$, keep at most *two* blocks of *pre-determined length* 2^i

(so $O(\log N)$ blocks, each taking $O(\log N)$ space $\longrightarrow O(log^2 N)$ total space)

INVARIANT 2: Blocks of length 2^i must *start more recently* than blocks of length 2^{i+1},
end no more in the past than them.

Idea: Exponential windows

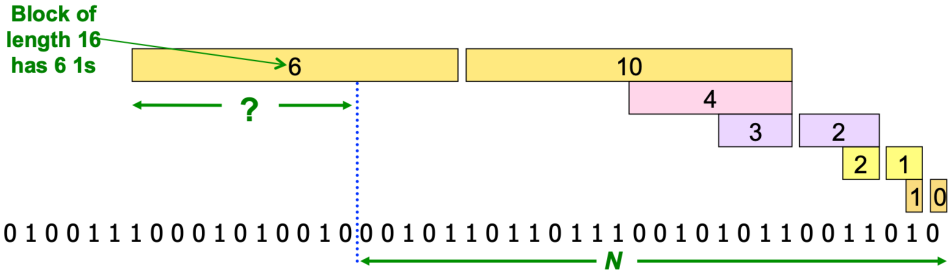


When a new bit arrives:

1) we may need to (*somehow*) *merge* blocks, in order to maintain the invariants;

2) the *start* of w_t may have past the *end* of the oldest block: *forget* the block.

Idea: Exponential windows

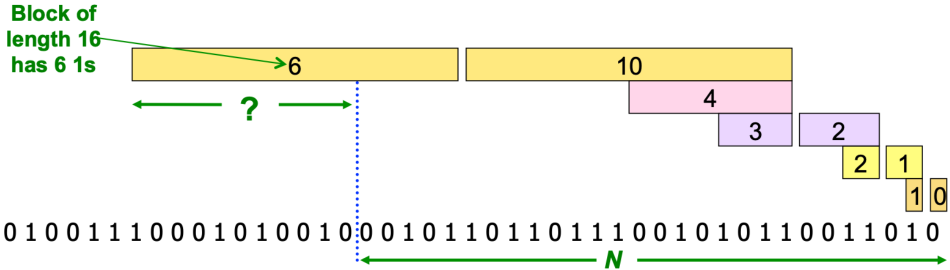


How to estimate the number m_t of 1s in the window w_t?

1) Find a set S of *non-overlapping* blocks *covering* the part of the window not covered by the oldest block

2) $\tilde{m}_t \leftarrow \sum_{\text{block } i \in S} (\text{weight of block } i) + \frac{1}{2} \cdot \text{weight of oldest block}$

Idea: Exponential windows



$$\tilde{m}_t \leftarrow \underbrace{\sum_{\text{block } i \in S} (\text{weight of block } i)}_{\text{exact, for the part of } w_t \text{ covered by } S} + \underbrace{\frac{1}{2} \cdot \text{weight of oldest block}}_{\text{possibly different than no. of 1s in the part of } w_t \text{ covered by the oldest block}}$$

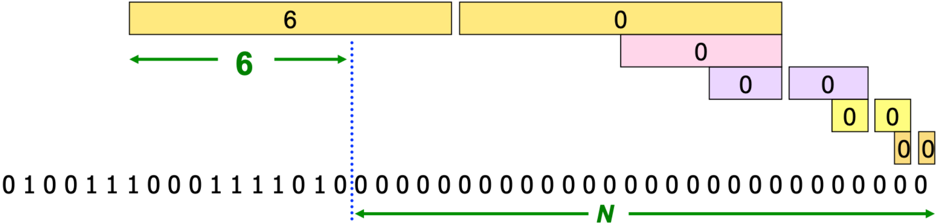
Error is

$$\frac{|(\text{1s in the part of } w_t \text{ covered by the oldest block}) - (\text{weight of the oldest block})/2|}{m_t}$$

Idea: Exponential windows

Error: $\dfrac{|(\text{1s in the part of } w_t \text{ covered by the oldest block}) - (\text{weight of oldest block})/2|}{m_t}$

How large can it be?



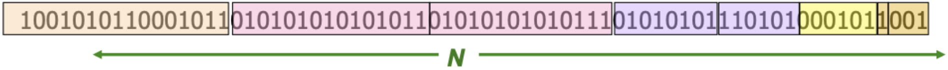
All the 1s in the oldest block could be *outside* w_t:

the error is *unbounded* (i.e., infinite).

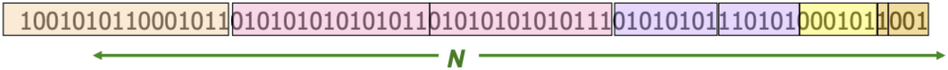
DGIM method

Don't keep blocks of *pre-determined length*,

rather keep blocks with a *pre-determined no. of 1s* (i.e., weight).



DGIM method



INVARIANTS:

1) All blocks have a weight that is a power of 2.

2) Block do *not overlap*, are *consecutive*, cover window (except final sequence of zeroes)

3) For each $i \geq 0$, there are *at most 2 block*s with weight 2^i

4) Older blocks have *weight no smaller* than that of more recent blocks

If there is a block of weight 2^i, there is at least a block of weight 2^j, for each $0 \leq j < i$

5) Blocks always *end with a 1*

DGIM method

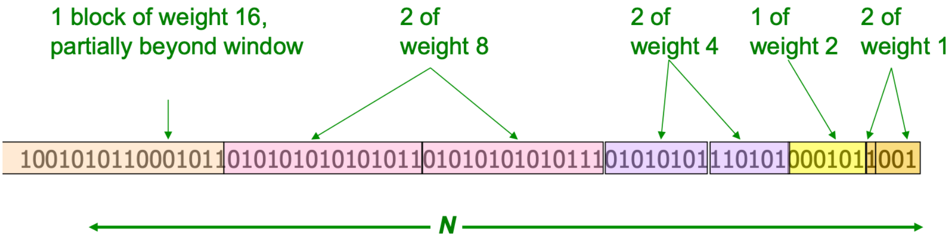
We still need to keep the summaries (*buckets*) of $O(\log N)$ blocks.

For each block we keep (in a *bucket*):

1) the time of the block's end (in $\mod N$ format)
(the time of the block's start is *implicit*, because the blocks are consecutive)

2) the weight of the block (as *exponent of 2*, takes space $O(\log \log N)$)

The stream as buckets: Example



Updating buckets

When a new element (bit) e_t arrives at time t:

1. Check the end-time z of the *oldest* bucket.

    1.a If $z == t \mod N$, then forget the oldest bucket

2.a If $e_t == 0$: do nothing

2.b If $e_t == 1$: see next slide.

Updating buckets

If $e_t == 1$:

create a new bucket of weight *1* just for bit e_t:

New bucket has end time $t \mod N$, weight-exponent 0

If there are now *three* buckets with weight-exponent 0,

Combine the *oldest* two into a bucket with weight-exponent 1

If there are now *three* buckets with weight-exponent 1,

Combine the *oldest* two into a bucket with weight-exponent 2

And so on …(we definitively stop at some point)

Updating buckets: example

Current state of the stream:

10010101100010110101010101010101010101010101011101010101110101000101100100

Bit of value 1 arrives

001010110001011010101010101010101010101010101110101010111010100010110010**1**

Two orange buckets get merged into a yellow bucket

0010101100010110101010101010101010101010101011101010101110101000101100101

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

010110001011010101010101010101010101010101011101010101110101000101100101**101**

Buckets get merged…

010110001011010101010101010101010101010101011101010101110101000101100101**101**

State of the buckets after merging

010110001011010101010101010101010101010101011101010101110101000101100101101

Querying the number of 1s

Estimate of m_t (no. of 1s in w_t):

$$\tilde{m}_t \leftarrow \underbrace{\sum_{\text{block } i \text{ not oldest}} (\text{weight of block } i)}_{\text{exact, for the part of } w_t \text{ covered by all blocks but the oldest}} + \underbrace{\frac{1}{2} \cdot \text{weight of oldest block}}_{\substack{\text{possibly different than no. of 1s in the part of } w_t \\ \text{covered by the oldest block}}}$$

The same as with blocks of pre-determined length. Do we get the same unbounded error?

Error bound: proof

If the oldest bucket has weight 2^r, the error is

$$\frac{|(\text{1s in the part of } w_t \text{ covered by the oldest block}) - 2^r/2|}{m_t} \leq \frac{2^{r-1}}{m_t}$$

If the oldest bucket has weight 2^r, then

$$m_t > \sum_{\text{block } i \text{ not oldest}} (\text{weight of block } i) \geq \sum_{j=0}^{r-1} 2^j = 2^r - 1$$

(strictly greater because the oldest block must end with a 1)
Then

$$\frac{2^{r-1}}{m_t} \leq \frac{2^{r-1}}{2^r} = \frac{1}{2}$$

i.e., we have a 2-approximation.

Can we do better?

What if we keep *more* than 1 or 2 buckets per weight?

What happens to the oldest (and *heaviest*) bucket?

It becomes *lighter* $\Rightarrow$ *smaller* error!

IDEA: keep either $r-1$ or r buckets for each weight ($r > 2$).

We may have any of $1, 2, 3, \ldots, r-1$ buckets of the *largest* weight

Relative error is at most $O(1/r)$.

What happens to the amount of *space* that we need?

It grows, because we have *more (lighter) buckets*.

Similarly to the Bloom filter, there is a *sweet spot* between space and error

Outline

- ✔ *Data streams*: motivation, applications, model(s), queries
- ✔ Approximate *query answering*: reservoir sampling
- ✔ Approximate *set membership*: Bloom filters
- ✔ Approximate *distinct counting*: The Flajolet-Martin approach
- ✔ Approximate *counting* on *sliding windows*: The DGIM Algorithm