Mining Data Streams

CS246: Mining Massive Datasets

Jure Leskovec, Stanford University
Mina Ghashami, Amazon

http://cs246.stanford.edu

New Topic: Infinite Data

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

Datasets vs Data Streams

- So far we have worked with datasets where all data is available

- In contrast, in many data mining scenarios, we do not know the entire data in advance. This is called **data streams.**

- Think of data streams as **infinite data** arriving one element at a time

Data Streams

- **Examples:**
  - Google queries
  - Twitter posts or Facebook status updates
  - e-Commerce purchase data
  - Credit card transactions
- The input rate is controlled **externally:**
  - *Stream management* is important.
  - This is the fun part and why interesting algorithms are needed

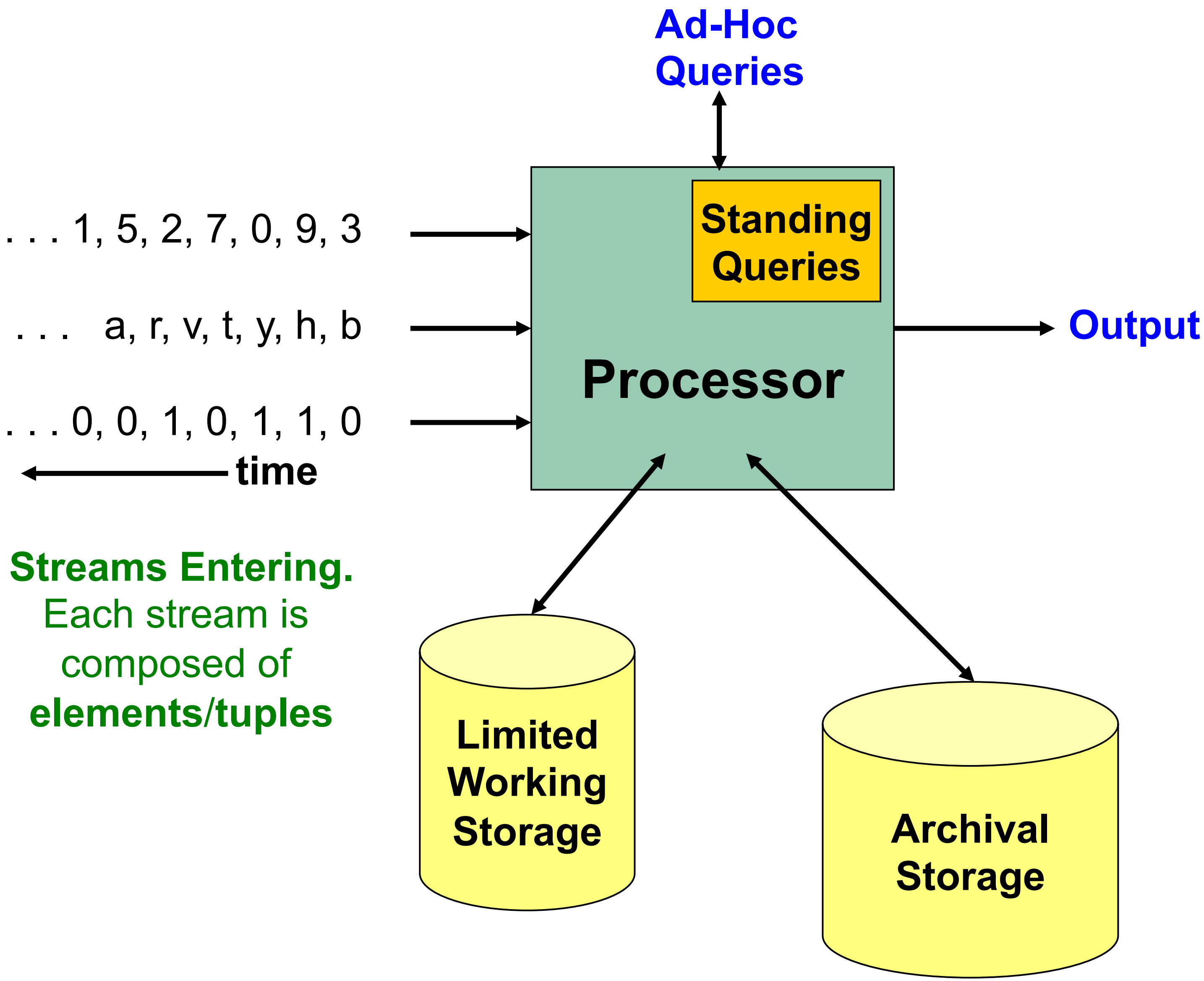
Applications (1)

- **Mining query streams**
  - Google wants to know what queries are more frequent today than yesterday
- **Mining click streams**
  - Wikipedia 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)

- **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

General Stream Processing Model



The Stream Model

- Input **elements** enter at a rapid rate, at one or more input ports (i.e., **streams**)
  - **Elements of the stream may be tuples**

- **The system cannot store the entire stream**

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

Side note: SGD is a Streaming Alg.

- **Stochastic Gradient Descent (SGD) is an example of a streaming algorithm**
- **In Machine Learning we call this: Online Learning**
  - Allows for modeling problems where we have a continuous stream of data
  - We want an algorithm to learn from it and slowly adapt to the changes in data
- **Idea: Do small updates to the model**
  - **SGD** makes small updates
  - **So:** First train the classifier on training data
  - **Then:** For every example from the stream, we slightly update the model (using small learning rate)

Problems on Data Streams

- **Types of queries one wants to answer on a data stream:**
  - **Sampling data from a stream**
    - Construct a random sample
  - **Filtering a data stream**
    - Select elements with property ***x*** from the stream
  - **Counting distinct elements**
    - Number of distinct elements in the last ***k*** elements of the stream
  - **finding most frequent elements**

Sampling from a Data Stream

Sampling from a Data Stream

- Why is sampling important?
  - Since we cannot store the entire stream, a representative **sample** can act like the stream
- **Two different problems:**
  - **(1)** Sample a **fixed proportion** of elements in the stream (say 1 in 10)
  - **(2)** Maintain a **random sample of fixed size** over a potentially infinite stream
    - At any "time" ***k*** we would like a random sample of ***s*** elements of the stream 1..*k*
      - **What is the property of the sample we want to maintain?** For all time steps ***k***, each of the ***k*** elements seen so far must have **equal probability** of being sampled

Sampling a Fixed Proportion

- **Problem 1: Sampling a fixed proportion**
  - E.g. sample 10% of the stream
  - As stream grows, sample grows

- **Naïve solution:**
  - Generate a random integer in **[0...9]** for each element
  - Store the element if the integer is **0**, otherwise discard

- **Any problem with this approach?**
  - Since elements of stream can be tuples, we have to be very careful how we sample them

Problem with Naïve Approach

- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Question:** What fraction of unique queries by an average user are duplicates?
    - Suppose each user issues ***x*** queries once and ***d*** queries twice (total of ***x*+2*d*** query instances) then the correct answer to the query is ***d/(x+d)***
  - **Proposed solution: We keep 10% of the queries**
    - Let's say at any point in time you have seen data of **n** users
    - Sample will contain ***n*(*x*+2*d*)/10** elements of the stream
    - Sample will contain ***nd*/100** pairs of duplicates
      - **n.d/100 = n.1/10 · 1/10 · d**
    - There are ***n(10x+19d)/100*** unique elements in the stream
      - ***n*(*x*+2*d*)/10 - *n d*/100 = *n*(10*x*+19*d*)/100**
- **So the sample-based answer is** $\frac{n\frac{d}{100}}{n\frac{10x}{100}+n\frac{19d}{100}} = \frac{d}{10x+19d}$

Sample underestimates

Solution: Sample Users

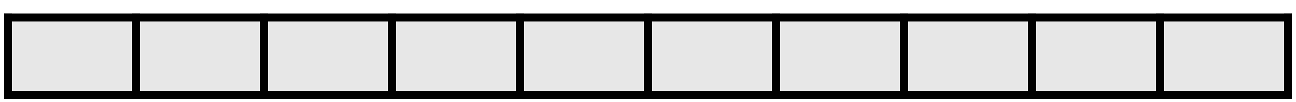
Solution:

- Don't sample *queries*, sample *users* instead
- Pick **1/10th** of **users** and take all their search queries in the sample

- Use a hash function that hashes the user name or user id uniformly into 10 buckets

Generalized Solution

- **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***



Hash table with **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

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

The sample is of fixed size

Maintaining a fixed-size sample

- **Problem 2: Fixed-size sample**
- **Suppose we need to maintain a random sample *S* of size exactly *s* tuples**
  - E.g., main memory size constraint
- **Why?** Don't know length of stream in advance
- **Suppose by time *n* we have seen *n* items**
  - **Each item is in the sample *S* with equal prob. *s/n***

How to think about the problem: say s = 2

Stream: a x c y z k c d e g…

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.

At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.

Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the 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 ***S*** contains each element seen so far with prob. ***s/n***
- **Inductive step:**
  - **New element *n+1* arrives, it goes to S with prob s/(n+1)**
  - **For all other elements currently in S:**
    - They were in ***S*** with prob. **s/n**
    - The probability that they remain in S:

$$\left(1-\frac{s}{n+1}\right)+\left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)=\frac{n}{n+1}$$

Element **n+1** discarded — Element **n+1** not discarded — Element in the sample not picked

    - tuples stayed in ***S*** with prob. **n/(n+1)**
- So ***P(tuple is in S at time n+1)*** $= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

Filtering Data Streams

Filtering Data Streams

- **Each element of data stream is a tuple**
- A filter **S** that is a list of keys
- **Determine which tuples of stream have key 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 at the same time on the stream

Applications

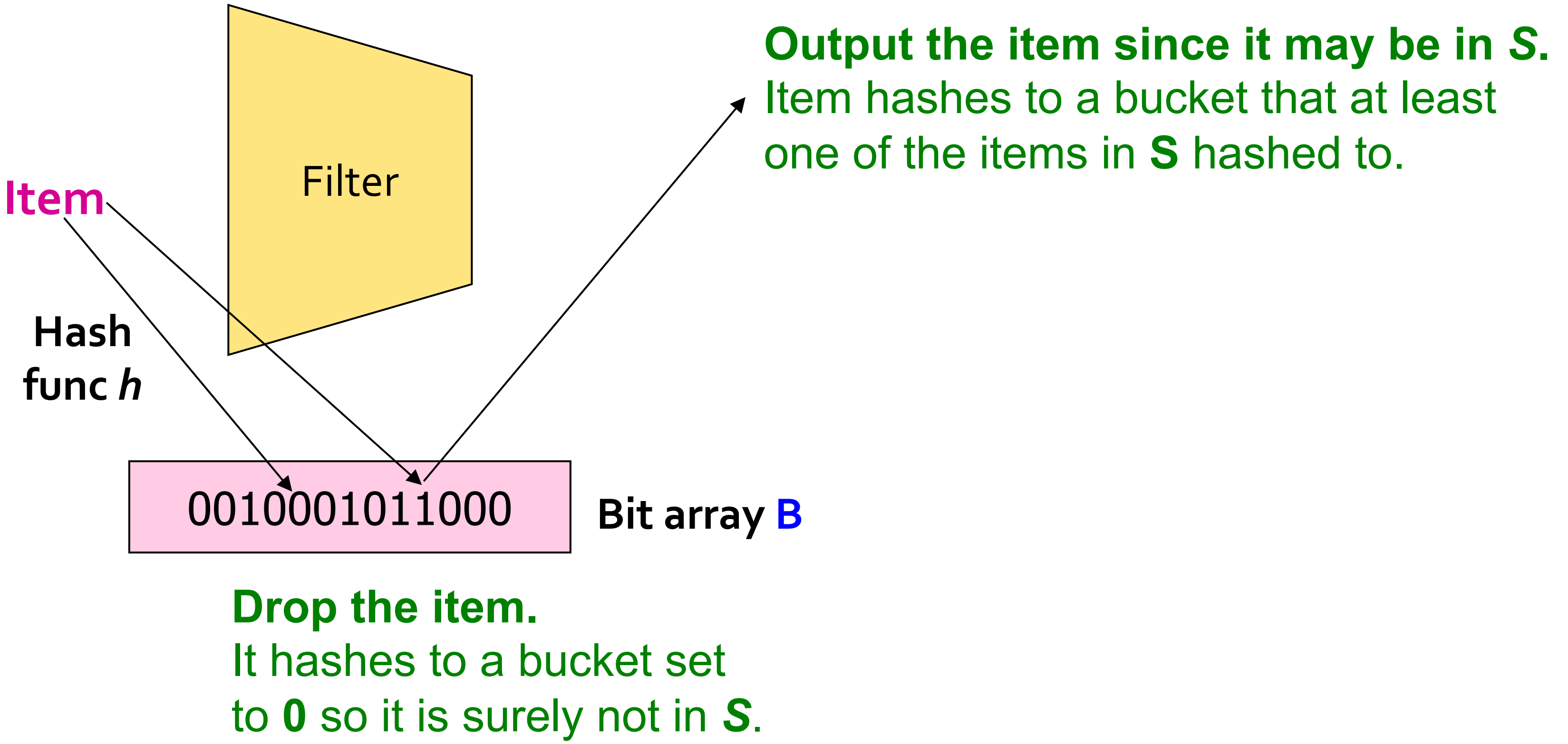
- **Example: Email spam filtering**
  - 1 million users, each user has 1000 "good" email addresses (trusted addresses)
  - If an email comes from one of these, it is **NOT** spam
- **Publish-subscribe systems**
  - You are collecting lots of messages (news articles)
  - People express interest in certain sets of keywords
  - Determine whether each message matches a user's interest
- **Content filtering**
  - You want to make sure the user does not see the same ad/recommendation multiple times

First Cut Solution (1)

Bloom Filter algorithm:
Given a set of keys *S* that we want to filter

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

First Cut Solution (2)



- **Creates false positives**
  - Items that are hashed to a 1 bucket may or may not be in S
- **but no false negatives**
  - Items that are hashed to 0 bucket are surely not in S

First Cut Solution (3)

- **|S| = 1 billion email addresses**

Naive Dictionary approach: 1 billion email address, every email address is ~20 characters long → 160 GB to store email addresses + overhead of dictionary → 200 GB!

Bloom Filter: |B| = 1GB = 8 billion bits

- If the email address is in ***S***, then it surely hashes to a bucket that has the bit set to **1**, so it always gets through (***no false negatives***)
- Approximately **1/8** of the bits are set to **1**, so about **1/8th** of the addresses not in ***S*** get through to the output (***false positives***)
  - Actually, less than **1/8th**, because more than one address might hash to the same bit

Analysis: Throwing Darts (1)

- **Let's do a more accurate analysis of number of false positives, we know that:**
  - **Fraction of 1s in array B = prob. of false positive**
- **Darts & Targets:** 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?**

Equals **1/e** as **n** $\rightarrow \infty$

Equivalent

$$1 - (1 - 1/n)^{n(m/n)}$$

$$1 - e^{-m/n}$$

Probability some target **X** not hit by a dart

Probability at least one dart hits target **X**

Approximation is especially accurate when n is large

Analysis: Throwing Darts (3)

- **Fraction of 1s in the array B = probability of false positive = $1 - e^{-m/n}$**

1 billion email addresses

1 GB = 8 billion bits

- **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**

- **To reduce false positive rate of bloom filter we use multiple hash functions**

Bloom Filter

- Consider: **|S| = *m keys*, |B| = *n bits***
- Use ***k*** independent hash functions $h_1, \ldots, h_k$
- **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 $i = 1, \ldots, k$)

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

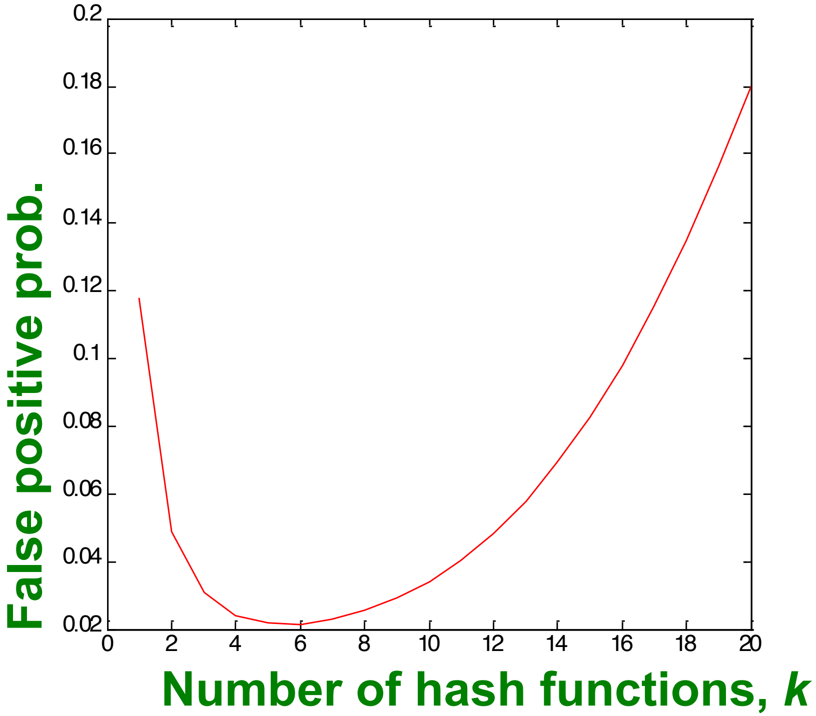
- **Run-time:**
  - When a stream element with key ***x*** arrives
    - If **B[$h_i(x)$] = 1** <u>for all</u> $i = 1, \ldots, k$ then declare that ***x*** is in ***S***
      - That is, ***x*** hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element ***x***

Bloom Filter – Analysis

- **What fraction of the bit vector B are 1s?**
  - Throwing $k \cdot m$ darts at n targets
  - So fraction of **1**s 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**: $(1 - e^{-1/4})^2$ = **0.0489**



- **What happens as we keep increasing *k*?**

- Optimal value of ***k***: $\frac{n}{m} \ln 2$
  - **In our case:** Optimal **k = 8 ln(2) = 5.54 ≈ 6**
    - **Error at k = 6**: $(1 - e^{-3/4})^6$ = **0.0216**

Optimal *k*: *k* which gives the lowest false positive probability

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 **B**s?
  - **It is the same:** $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping **1 big B** is simpler

Counting Distinct Elements

Counting Distinct Elements

- **Problem:**
  - Data stream consists of elements chosen from a universal set of size ***N***
  - Maintain a count of the number of distinct elements seen so far
- **Obvious approach:**
  Maintain a dictionary of elements seen so far
  - keep a hash table of all the distinct elements seen so far
  - What if number of distinct elements are huge?
  - What if there are many streams that need to be processed at once?

Applications

- **How many unique users a website has seen in each given month?**
  - Universal set = set of logins for that month
  - Stream element = each time someone logs in

- **How many different words are found at a site which is among the Web pages being crawled?**
  - Unusually low or high numbers could indicate artificial pages (spam?)

- **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 in every stream?**
  - **We have limited working storage**
- **We use a variety of hashing and randomization to get approximately what we want**
- **Estimate the count in an unbiased way**
- **Accept that the count may have a little error, but limit the probability that the error is large**

Flajolet-Martin(FM) Approach

- Estimates number of distinct elements by hashing elements to a bit-string that is sufficiently long
  - The length of the bit-string is large enough that it produces more result than size of universal set.
- **Idea**: hash elements to a binary string
  - the more different elements we see in the stream, the more different hash values we shall have.
  - Number of trailing **0**s in these hash values estimates number of distinct elements.

Flajolet-Martin(FM) Approach

- Pick a hash function ***h*** that maps each of the ***N*** elements to at least $\log_2 N$ bits
  - So hash values are binary strings
    E.g. for a stream element **a**, ***h(a) = 1100***
- Let ***r***(***a***) be the number of trailing **0s** in ***h***(***a***)
  - **r(a)** = position of first 1 counting from the right
    - E.g., for ***h(a) = 1100***, the ***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 rough and heuristic intuition why Flajolet-Martin works:**
  - ***h(a)*** hashes ***a*** with **equal prob.** to any of ***N*** values
  - All elements have **equal prob**. to have a tail of ***r*** zeros
  - The prob. of a given h(a) to have a tail of ***r*** zeros is:

$$\Pr(\text{a tail of } r \text{ zeros }) = 2^{-r}$$

    - About 50% of ***a***s hash to *****0**
    - About 25% of ***a***s hash to ****00**

Why It Works: More formally

- **Let m be the number of distinct elements seen so far**
- **Then the probability that we have at least one tail of r zeros is**

$$1 - (1 - 2^{-r})^m$$

Prob. no element has tail of ***r*** zeros.

Prob. that a given **h(a)** does not have a tail of ***r*** zeros

Why It Works: More formally

- Therefore ***pr(finding at least one tail of r zeros) =***
- $1-(1-2^{-r})^{m} = 1-(1-2^{-r})^{2^r(m2^{-r})} \approx 1-e^{-m2^{-r}}$
  - If ***m << 2^r^***, then prob. tends to **1**
    - $1-e^{-m2^{-r}} \approx 0$ as **$m/2^r \to 0$**
    - So, the probability of finding a tail of length ***r*** tends to **0**
  - If ***m >> 2^r^***, then prob. tends to **0**
    - $1-e^{-m2^{-r}} \approx 1$ as **$m/2^r \to \infty$**
    - So, the probability of finding a tail of length ***r*** tends to **1**
- **Thus, 2^R will almost always be around *m*!**

Why It Doesn't Work

- **$E[2^R]$ is actually infinite**
  - Probability halves when $R \to R+1$, but value doubles
- **Workaround involves using many hash functions h_i and getting many samples of R_i**
- **How are samples R_i combined?**
  - **Average?** What if one very large value 2^{R_i}?
  - **Median?** All estimates are a power of **2**
  - **Solution:**
    - Partition your samples into small groups
    - Take the median of groups
    - Then take the average of the medians

Counting Most-Common Recent Items

The Most-Common Recent Elements

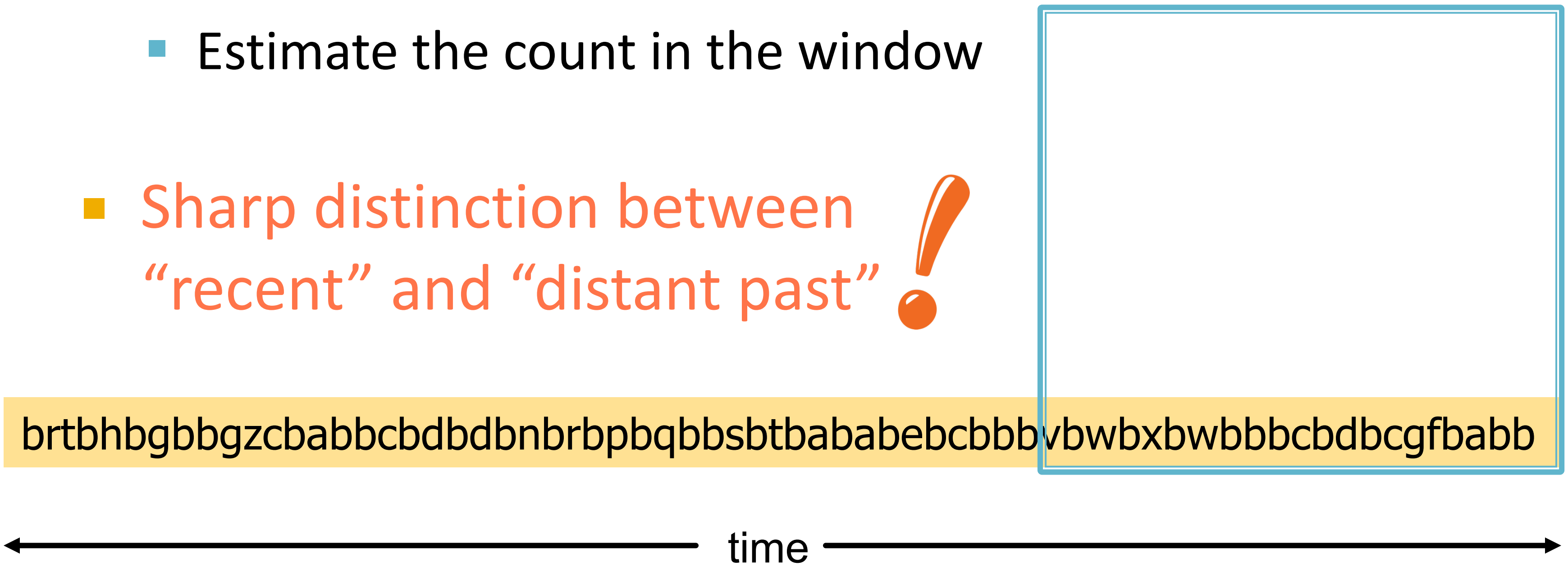
Two flavor of a problem:

1. Finding the most common elements
2. Finding the most common *"recent"* elements

- Example:
  - In a stream of movie tickets from all over the world, what are most popular movies "currently"?
  - In a stream of items sold at Amazon, what are most popular items "recently"?
  - In a stream of tweets, who are the most active users "currently"?

The Most-Common Recent Elements

- What is "recent"?
- One approach:
  - Get a sliding window of size N
  - Estimate the count in the window
- Sharp distinction between "recent" and "distant past"!



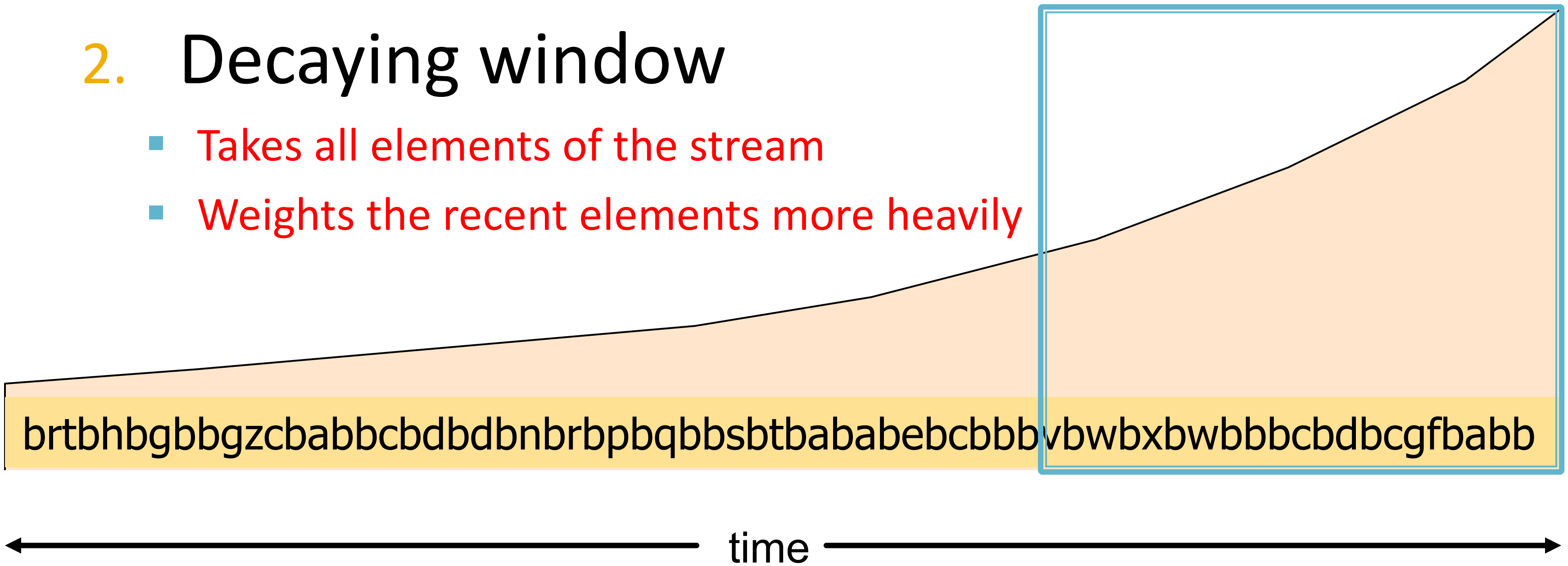
Exponentially Decaying Window

Solution: Exponentially decaying windows

Two type of windows:

1. Sliding window of fixed length
   - Holds last N elements

2. Decaying window
   - Takes all elements of the stream
   - Weights the recent elements more heavily



Exponentially Decaying Window

- **Computes a smooth aggregation over stream**
- If stream is $a_1, a_2, \ldots, a_t$ then the *exponentially decaying window* at time t is

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

$$= a_t + a_{t-1}(1-c) + a_{t-2}(1-c)^2 + \cdots$$

  - **c** is a constant, presumably tiny, like 10^{-6} or 10^{-9}
  - a_t is a non-negative integer in general
- **When new a_{t+1} arrives:**
  Multiply current sum by **(1-c)** and add a_{t+1}

Counting Items

- Given a stream of items, form a binary stream per item:
  - **1** = item present; **0** = not present

Stream of items:

brtbhbgbbgzcbabbcbdbdbnbrbpbqbbsbtbababebcbbbvbwbxbwbbbcbdbcgfbabbzdba

Binary stream for item "b"

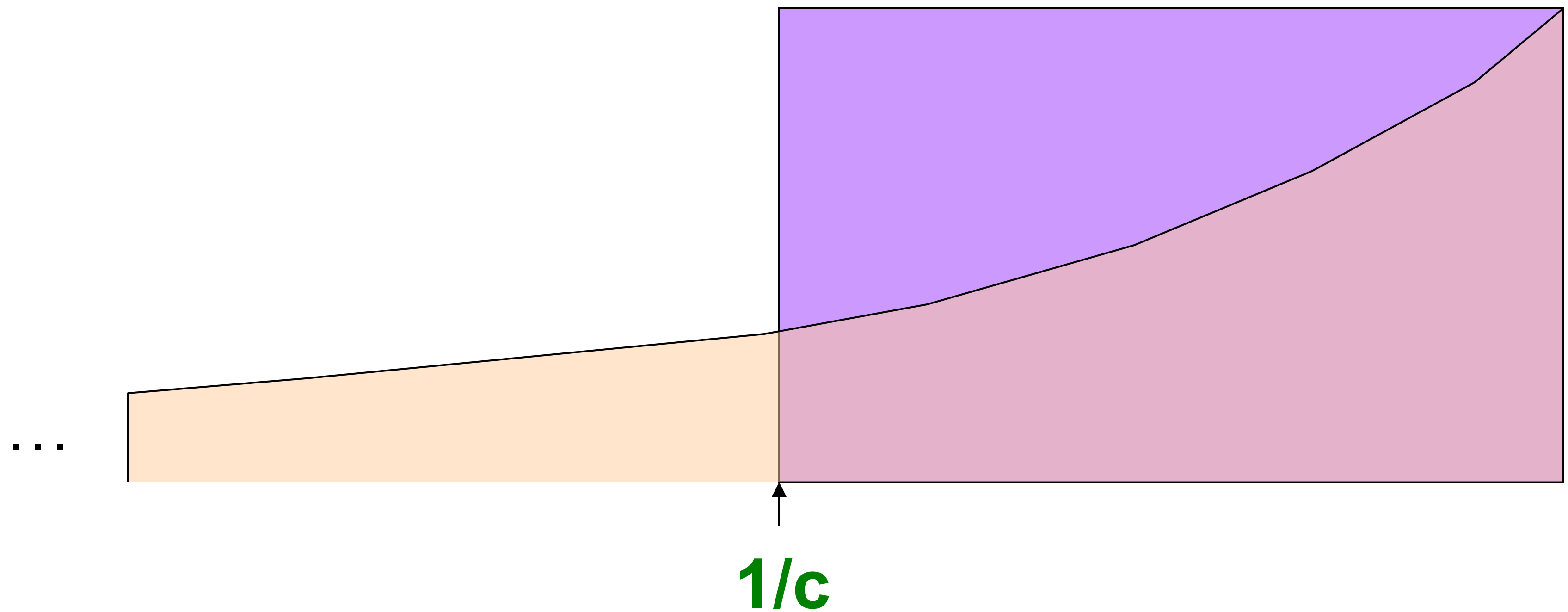
1001010110001011010101010101011010101010101110101010111010100010110010

Counting Items

- On all binary streams, compute exponentially decaying window
  - If each a_t is an "item" we can compute the **characteristic function** of each item x as an **Exponentially Decaying Window**:
  - That is: $\sum_{t=1}^{T} \delta_t \cdot (1-c)^{T-t}$
    where $\delta_t = 1$ if $a_t = x$, and 0 otherwise
  - **In other words:** Imagine that for each item x we have a binary stream (1 if x appears, 0 if x does not appear)
  - Then, when a new item a_t arrives:
    - Multiply the summation by $(1-c)$
    - Add **+1** to the summation if item = x
- **Call this sum the "weight" of item x**

Counting Items: Decaying Windows

Spreads out weights of the stream as far back as the stream goes



- **Important property:** Sum over all weights
  $\sum_t 1 \cdot (1-c)^t = 1/[1-(1-c)] = 1/c$

$$\sum_{k=0}^{n} z^k = \frac{1-z^{n+1}}{1-z}$$

Counting Individual Items

- **What are "currently" most popular movies?**
- **Suppose we want to find movies of weight > ½**
  - **Important property:** Sum over all weights $\sum_t \delta_t \cdot (1-c)^t$ is 1/[1 – (1 – c)] = $\mathbf{1/c}$
    - **That means that no item can have weight greater than 1/c**
    - The item will have weight $\mathbf{1/c}$ if its stream is [1,1,1,1,1…]. Note we have a separate binary stream for each item. So, at a given time only one item will have a δ_t=1, and other items will get a 0.
- **Thus:**
  - There cannot be more than $\mathbf{2/c}$ movies with weight of ½ or more
    - Why? Assume weight of item is ½. How many items *n* can we have so that the sum is <1/c; **Answer:** ½n<1/c → $n < 2/c$
- So, $\mathbf{2/c}$ is a limit on the number of movies being counted at any time

Counting Individual Items

- **Algorithm for finding items of weight > ½ :**

1. Keep ***2/c*** counters and initialize them to ***0***
2. When an item a_t arrives in the stream:
   - Multiply all counts by ***(1-c)***
   - Drop all counters whose ***count < 1/2***
   - If the new item is among the counters, increment its count by ***1***
   - Otherwise, if there is an empty counter assign it to a_t and set it to ***1***
3. At any point in the stream, the most common recent items are the ones in the counter set.

Extension to Itemsets

- **Extension: Count (some) itemsets**
  - **What are currently "hot" itemsets?**
    - **Problem:** Too many itemsets to keep counts of all of them in memory
- **When a basket B comes in:**
  - Multiply all counts by $(\mathbf{1} - \boldsymbol{c})$
  - For uncounted items in $\boldsymbol{B}$, create new count
  - Add $\mathbf{1}$ to count of any item in $\boldsymbol{B}$ and to any **itemset** contained in $\boldsymbol{B}$ that is already being counted
  - **Drop counts < ½**
  - Initiate new counts (next slide)

Initiation of New Counts

- Start a count for an itemset $S \subseteq B$ if every proper subset of ***S*** had a count prior to arrival of basket ***B.***
  - **Intuitively:** If all subsets of **S** are being counted this means they are "**frequent/hot**" and thus ***S*** has a potential to be "**hot**"
- **Example:**
  - Start counting ***S*=**{**i, j**} iff both **i** and **j** were counted prior to seeing ***B***
  - Start counting ***S*=**{**i, j, k**} iff {**i, j**}, {**i, k**}, and {**j, k**} were all counted prior to seeing ***B***

How many counts do we need?

- Counts for single items **< (2/c)·(avg. number of items in a basket)**
- **Counts for larger itemsets = ??**
- **But we are conservative about starting counts of large sets**
  - If we counted every set we saw, one basket of **20** items would initiate **1M** counts

Summary

- **Sampling a fixed proportion of a stream**
  - Sample size grows as the stream grows
- **Sampling a fixed-size sample**
  - Reservoir sampling
- **Check existence of a set of keys in the stream**
  - Bloom filter
- **Counting distinct elements in a stream**
  - Flajolet-Martin algorithm
- **Counting frequent elements in a stream**
  - Exponentially decaying window