Mining Data Streams I

Alexander Schönhuth

UNIVERSITÄT BIELEFELD
Faculty of Technology

Bielefeld University
May 12, 2022

TODAY

Overview

- Intro: A Data Stream Management Model
- Sampling Data in a Stream
- Filtering Streams: Bloom Filters
- Counting Distinct Elements: Flajolet-Martin algorithm

Learning Goals: Understand these topics and get familiarized

Mining Data Streams: Introduction

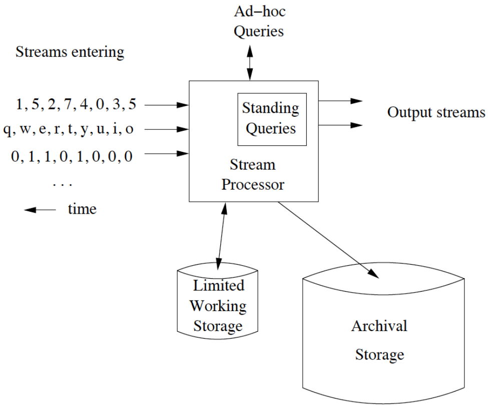
MINING DATA STREAMS: INTRODUCTION I

- *Situation:* Data arrives in a stream (or several streams)
  - Too much to be put in active storage (main memory, disk, database)
  - If not processed immediately or stored (in inaccesible archives), lost forever
- *Algorithms* involve some summarization of stream(s); e.g.
  - create useful samples of stream(s)
  - filter the stream(s)
  - focus on windows of appropriate length (last *n* elements)

Data Streams: Examples

- Sensor data:
  - Ocean data (temperature, height): terabytes per day
  - Tracking cars (location, speed)
- Image data from satellites
- Internet/web traffic
  - Switches that route data also decide on denial of service
  - Tracking trends via analyzing clicks

Data Stream Management System



A data stream management system

Adopted from `mmds.org`

Data Stream Queries

- *Standing queries*
  - need to be answered throughout time
  - Answers need to be updated when they change
  - Example: current or maximum ocean temperature
- *Ad-hoc queries*
  - ask immediate questions
  - *Example:* number of unique users of a web site in the last 4 weeks
  - Not all data can be stored/processed
    ☞ Only certain questions feasible
  - Need to prepare for queries
    ☞ For example, store data from sliding windows

DATA STREAM QUERIES

Issues

- Streams deliver elements rapidly: need to act quickly
- Thus, data to work on should fit in main memory
- New techniques required:

☞ Compute approximate, not exact answers

☞ Hashing is a useful technique

Sampling Elements from a Stream

Sampling Elements

- *Situation:*
  - Select subsample from stream to store
  - Subsample should be representative of stream as a whole
- *Running Example:*
  - Search engine processes stream of search queries
  - Stream consists of tuples (user,query,time)
  - Can store only 1/10-th of data
  - *Stream Query:* Fraction of repeated search queries?

Running Example: Pitfall

- *Running Example:*
  - *Stream Query:* Fraction of repeated search queries?

Naive and bad approach

- For each query, generate random integer from $[0, 9]$
- Keep only queries if 0 was generated
- *Scenario:* Suppose a user has issued
  - s queries one time
  - d queries two times
  - no queries more than two times
- *Correct answer* is $\frac{d}{d+s}$

Running Example: Pitfall

- *Running Example:*
  - *Stream Query:* Fraction of repeated search queries?

Naive and bad approach

- *Correct answer* is $\frac{d}{d+s}$
- But on randomly selected queries, we see that
  - Of one-time queries, $s/10$ appear to show once
  - Of two-time queries, $d/10 \times d/10$ appear to show twice
  - Of two-time queries, $d(1/10 \times 9/10) \times 2$ appear to show once
  - Resulting in *estimate*

$$\frac{0.01d}{(0.1s + 0.18d) + 0.01d} = \frac{d}{10s + 19d}$$

for repeated search queries, which is wrong for positive s, d

Running Example: Pitfall

- *Running Example:*
  - *Stream Query:* Fraction of repeated search queries?

Better approach

- For each user (not query!), generate random integer from $[0, 9]$
- Keep 1/10th of users, e.g. if 0 was generated
- Implement randomness by hashing users to 10 buckets
  - ☞ avoids storing for each user whether he was in or out
- For maintaining sample for a/b-th of data, use b buckets, and keep users in buckets 0 to $a-1$

RUNNING EXAMPLE: PITFALL

Better approach

- *General Sampling Problem:* Generalize from one-valued key to arbitrary-valued keys, keep a/b-th of (multi-valued) keys by the same technique
- *Reducing sample size:* On increasing amounts of data, ratio of data used for sample to be lowered
  - When lowering is necessary, decrease a by 1, so 0 to $a-2$ are still accepted
  - Remove all elements with keys hashing to $a-1$

Filtering Streams

Filtering Streams: Motivating Example

- *Problem:* Filter for data for which certain conditions apply
- Can be easy: data are numbers, select numbers of at most 10
- *Challenge:*
  - There is a set S that is too large to fit in main memory
  - Condition is too check whether stream elements belong to S

Filtering Streams: Motivating Example

Motivating Example: Email Spam

- Streamed data: pairs (email address, email text)
  - Set S is one billion (10^9) *approved (no spam!) addresses*
  - Only process emails from these addresses
    ☞ need to determine whether arbitrary address belongs to them
  - *But*, addresses cannot be stored in main memory
- *Option 1:* make use of disk accesses
- *Option 2 (preferrable):* Devise method without disk accesses, and determine set membership correctly in majority of cases
- *Solution:* “Bloom Filtering”

Bloom Filtering: Running Example

- Assume that main memory is 1 GB
- Bloom filtering: use main memory as bit array (of eight billion bits)
- Devise hash function h that hashes email addresses to eight billion buckets
- Hash each member of S (allowed email addresses) to one of the buckets
- Set bits of hashed-to buckets to 1, leave other bits 0
- About 1/8-th of bits are 1

Bloom Filtering: Running Example

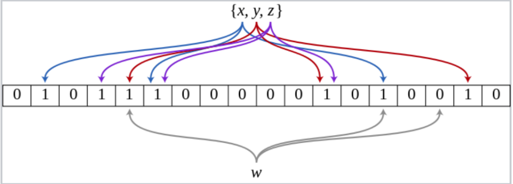
- Hash any new email address:
  - If hashed-to bit is 1, classify address as no spam
  - If hashed-to bit is 0, classify address as spam
- Each address hashed to 0 is indeed spam
- *But:* About 1/8-th of spam emails hash to 1
- So, not each address hashed to 1 is no spam
- 80% of emails are spam: filtering out 7/8-th is a big deal
- Filter cascade: filter out 7/8-th of (remaining) spam in each step

Bloom Filter: Definition

Definition [Bloom Filter]

A *Bloom filter* consists of

- A bit array B of n bits, initially all zero
- A set S of m key values
- Hash functions $h_1, \ldots, h_k$ hashing key values to bits of B
  - ☞ Number of buckets is n



A Bloom filter for set $S = \{x, y, z\}$ using three hash functions

From Wikipedia, by David Eppstein

Bloom Filter: Definition

Definition [Bloom Filter]

A *Bloom filter* consists of

- A bit array B of n bits, initially all zero
- A set S of m key values
- Hash functions $h_1, ..., h_k$ hashing key values to bits of B
  - ☞ Number of buckets is n

Bloom Filter Workflow

- *Initialization*
  - Take each key value $K \in S$
  - Set all bits $h_1(K), ..., h_k(K)$ to one
- *Testing keys:*
  - Take key K to be tested
  - Declare K to be a member of S if all $h_1(K), ..., h_k(K)$ are one

Bloom Filtering: Analysis

- If $K \in S$, all $h_1(K), ..., h_k(K)$ are one, so K passes
- If $K \notin S$, all $h_1(K), ..., h_k(K)$ could be one, so K mistakenly passes
  ☞ *False positive!*
- *Goal:* Calculate probability of false positives
- *For that*, calculate probability that bit is zero after initialization
- *Relates to* throwing y darts at x targets, where
  - Targets are bits in array, so $x = n$
  - Darts are members in S $(= m)$ times hash functions $(= k)$, which makes $y = km$

☞ What is the probability that target is not hit by any dart?

Bloom Filtering: Analysis

Throwing y darts at x targets:

- Probability that a given dart will not hit a given target is $(x-1)/x$
- Probability that none of the y darts will hit a given target is

$$(\frac{x-1}{x})^y = (1 - \frac{1}{x})^{x\frac{y}{x}} \quad (1)$$

- By $(1-\epsilon)^{1/\epsilon} = 1/e$ for small ϵ, we obtain that (1) is $e^{-y/x}$
- $x = n, y = km$: probability that a bit remains 0 is $e^{-km/n}$
- Would like to have fraction of 0 bits fairly large
- If k is about n/m, then probability of a 0 is e^{-1} (about 37%)
- In general, probability of false positive is k 1 bits, which evaluates as

$$(1 - e^{-\frac{km}{n}})^k \quad (2)$$

Counting Distinct Elements

The Flajolet-Martin Algorithm

Counting Distinct Elements: Problem

- *Problem:* Elements in streams can be identical
- *Question:* How many different elements has the stream brought along?
- *Model:* Consider the universal set of all possible elements
- Consider stream as a subset of the universal set
- *Question becomes:* What is the cardinality (size) of this subset?
- *Example:* Unique users of website
  - Amazon: determine number of users from user logins
  - Google: determine number of users from search queries

Counting Distinct Elements: Problem

- *Situation:* Stream picks elements from universal set
- *Question:* Size of subset of elements appearing in stream?
- *Obvious, but expensive:*
  - Keep stream elements in main memory
  - Store them in efficient search structure (hash table, search tree)
  - Works for sufficiently small amounts of distinct elements
- *If too many distinct elements, or too many streams:*
  - Use more machines ☞ Ok if affordable
  - Use secondary memory (disk) ☞ slow
  - *Here:* Estimate number of distinct elements using much less main memory than needed for storing all distinct elements
  - The *Flajolet-Martin algorithm* does this job

The Flajolet-Martin Algorithm

- *Central idea:* Hash elements to bit strings of sufficient length
  - For example, to hash URL's, 64-bit strings are sufficiently long
- *Intuition:*
  - The more different elements, the more different bit strings
  - The more different bit strings, the more "unusual" bit strings
  - Unusual here = bit string starts with many zeroes

Definition [Tail Length]

- Let h be the hash function that hashes stream elements a to bit strings $h(a)$
- The *tail length* of $h(a)$ is the number of zeroes by which it begins

The Flajolet-Martin Algorithm

Definition [Tail Length]

- Let h be the hash function that hashes stream elements a to bit strings $h(a)$
- The *tail length* of $h(a)$ is the number of zeroes by which it begins
- *Alternatively:* $h(a)$ number of zeroes a string ends with

Flajolet Algorithm

- Let A be the set of stream elements
- Let

$$R := \max_{a \in A} h(a) \tag{3}$$

  be the maximum tail length observed among stream elements
- *Estimate* 2^R for the number of distinct elements in the stream

FLAJOLET-MARTIN ALGORITHM: EXAMPLE

15 users

User	Hashed Bitstring
sean	01111101
todd	11010001
aaron	10000111
kat	01110001
don	01011010
sara	01000001
linda	01010011
eric	<u>0000</u>1001
jack	01101001
steph	10001100
terry	00111110
tim	00010000
wanda	11110001
chris	01101110
jane	00010010

→ Approximate Count = $2^4 = 16$

Because the longest leading sequence of zeros is 4 bits long, we can say that there may be approximately 16 users

Hashing user names to 8-bit strings

From `towardsdatascience.com`

Flajolet-Martin Algorithm: Explanation

- Probability that bit string $h(a)$ starts with r zeroes is 2^{-r}
- Probability that none of m distinct elements has tail length at least r is

$$(1-2^{-r})^m = ((1-2^{-r})^{2^r})^{m2^{-r}} \overset{(1-\epsilon)^{1/\epsilon}\approx 1/e}{=} e^{-m2^{-r}} \tag{4}$$

- Let $P_{m,r} := 1-(1-2^{-r})^m \approx 1-e^{-m2^{-r}}$ be the probability that for m stream elements, the maximum tail length R observed is at least r.
- Conclude:
  - For $m >> 2^r$, it holds that $P_{m,r}$ approaches 1
  - For $m << 2^r$, it holds that $P_{m,r}$ approaches 0
  - So, 2^R is unlikely to be much larger or much smaller than m

Flajolet-Martin Algorithm: Combining Estimates

- *Idea:* Use several hash functions $h_k, k = 1, ..., K$
- Combine their estimates $X_k, k = 1, ..., K$
- *Pitfall 1: Averaging*
  - Let p_r be the probability that the maximum tail length of h_k is r
  - One can compute that

$$p_r \geq \frac{1}{2} p_{r-1} \geq ... \geq 2^{-r+1} p_1 \geq 2^{-r} p_0$$

  - So $E(X_k)$, the expected value of X_k computes as

$$E(X_k) = \sum_{r \geq 0} p_r 2^r \geq p_o \sum_{r \geq 0} 2^{-r} 2^r = p_0 \sum_{r \geq 0} 1 = \infty$$

  - Therefore $\frac{1}{K} \sum_{k=1}^{K} E(X_k)$ the expected value of the average of the X_k turns out to be infinite as well
  - *Conclusion:* Overestimates spoil averaging

Flajolet-Martin Algorithm: Combining Estimates

- *Idea:* Use several hash functions $h_k, k = 1, ..., K$
- Combine their estimates $X_k, k = 1, ..., K$
- *Pitfall 2: Computing Medians*
  - The median is always a power of two
    ☞ makes only very limited sense
- *Solution:*
  - Group the hash functions into small groups and take averages within groups
  - Estimate m as median of group averages
  - Groups should be of size $C \log_2 m$ for some small C
- *Space Requirements:* Need to store only value of X_k, requiring little space as a maximum

MATERIALS / OUTLOOK

- See *Mining of Massive Datasets*: section 2.6; sections 4.1–4.4
- As usual, see `http://www.mmds.org/` in general for further resources
- For deepening your understanding, consider voluntary *homework*: read 2.6.7 and try to make sense of this
- Next lecture: "Mining Data Streams II / PageRank I"
  - See *Mining of Massive Datasets* 4.5–4.7; 5.1–5.2