CS 5614: (Big) Data Management Systems

B. Aditya Prakash

Lecture #14: Mining Streams 1
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)
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 (secondary) memory?
Side note: SGD is a Streaming Alg.

- **Stochastic Gradient Descent (SGD) is an example of a stream 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 slow updates to the model**
  - **SGD** (SVM, Perceptron) 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)
General Stream Processing Model

Streams Entering. Each stream is composed of elements/tuples.

... 1, 5, 2, 7, 0, 9, 3
... a, r, v, t, y, h, b
... 0, 0, 1, 0, 1, 1, 0

Processor

Ad-Hoc Queries

Standing Queries

Limited Working Storage

Archival Storage

Output

Prakash 2014
Problems on Data Streams

- Types of queries one wants on a data stream: (we’ll do 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
Problems on Data Streams

- Types of queries one wants on answer on a data stream: (we’ll do these next time)
  - Filtering a data stream
    - Select elements with property $x$ from the stream
  - Counting distinct elements
    - Number of distinct elements in the last $k$ elements of the stream
  - Estimating moments
    - Estimate avg./std. dev. of last $k$ elements
  - Finding frequent elements
Applications (1)

- **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
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
Sampling from a Data Stream: Sampling a fixed proportion

As the stream grows the sample also gets bigger
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)
  - (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
    - **What is the property of the sample we want to maintain?**
      For all time steps $k$, each of $k$ elements seen so far has equal prob. 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 $\frac{1}{10}$ 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: \( \frac{d}{x+d} \)
  - Proposed solution: We keep 10% of the queries
    - Sample will contain \( \frac{x}{10} \) of the singleton queries and \( \frac{2d}{10} \) of the duplicate queries at least once
    - But only \( \frac{d}{100} \) pairs of duplicates
      - \( \frac{d}{100} = \frac{1}{10} \cdot \frac{1}{10} \cdot d \)
    - Of \( d \) “duplicates” \( \frac{18d}{100} \) appear exactly once
      - \( \frac{18d}{100} = (\frac{1}{10} \cdot \frac{9}{10})+(\frac{9}{10} \cdot \frac{1}{10}) \cdot d \)
  - So the sample-based answer is
Solution: Sample Users

Solution:

- Pick $\frac{1}{10}$ 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
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

As the stream grows, 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 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: \([a, x, c, y, z, k, j, c, d, e, g, \ldots]\)

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 \( \frac{s}{n} \)
  - We need to show that after seeing element \( n+1 \) the sample maintains the property
    - Sample contains each element seen so far with probability \( \frac{s}{n+1} \)

- **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 \( \frac{s}{s} = 1 \)
Proof: By Induction

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

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

Element \( n+1 \) discarded  Element \( n+1 \) not discarded  Element in the sample not picked

- So, at time \( n \), tuples in \( S \) were there with prob. \( s/n \)
- Time \( n \rightarrow n+1 \), tuple stayed in \( S \) with prob. \( n/(n+1) \)
- So prob. tuple is in \( S \) at time \( n+1 = \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1} \)
QUERIES OVER A (LONG) SLIDING WINDOW
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:

  \[
  \text{Past} \quad \text{Future}
  \]

  \[
  \text{N} = 6
  \]
Problem:

- Given a stream of 0s and 1s
- Be prepared to answer queries of the form
  \textit{How many 1s are in the last } k \textit{ bits?} where } k \leq N \textit{ }

Obvious solution:

Store the most recent } N \textit{ bits

- When new bit comes in, discard the } N+1^{\text{st}} \textit{ bit

\begin{center}
\begin{tabular}{cccccccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
\textbf{Past} & & & & & & & & & & & \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{cccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
\textbf{Future} & & & & & & & & & & & \\
\end{tabular}
\end{center}

Suppose } N=6 \textit{ }

Prakash 2014
Counting Bits (2)

- You cannot 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?**
- **But, what if stream is non-uniform?**
  - What if distribution changes over time?
DGIM Method

- DGIM solution that does not assume uniformity

- We store $O(\log^2 n)$ 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 \textit{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?

- DGIM solution that does not assume uniformity
- We store $O(\log^2 N)$ 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
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 method**

- **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 1, 2, ...

- Record timestamps modulo $N$ (*the window size*), so we can represent any relevant timestamp in bits
DGIM: Buckets

- **bucket** in the DGIM method is a record consisting of:
  - (A) The timestamp of its end \( [O(\log N) \text{ bits}] \)
  - (B) The number of 1s between its beginning and end \( [O(\log \log N) \text{ bits}] \)

**Constraint on buckets:**
Number of 1s must be a power of 2
- That explains the \( O(\log \log N) \) in (B) above
Representing a Stream by Buckets

- 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

At least 1 of size 16. Partially beyond window.

Two of size 8

Two of size 4

One of size 2

Two of size 1

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

Prakash 2014
Updating Buckets (1)

- 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
Updating Buckets (2)

- 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:

1001010110001011 101010101010111 0 1010101010111 0 1010101110101 0 1011001100110

Bit of value 1 arrives

0010101110001011 101010101010111 0 1010101010111 0 1010101110101 0 0010110011011

Two orange buckets get merged into a yellow bucket

0010101110001011 101010101010111 0 1010101010111 0 1010101110101 0001011001101

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

0101100010110 101010101010111 0 1010101010111 0 1010101110101 0001011001101

Buckets get merged…

0101100010110 101010101010111 0 1010101010111 0 1010101110101 0001011001101

State of the buckets after merging

0101100010110 101010101010111 0 101010101010111 0 1010101110101 0001011001101
How to Query?

- To estimate the number of 1s in the most recent $N$ bits:
  1. Sum the sizes of all buckets but the last
     (note “size” means the number of 1s in the bucket)
  2. 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: Bucketized Stream

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

N
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 $1 + 2 + 4 + .. + 2^{r-1} = 2^r - 1$

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 + ½ 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?
Extensions

- Stream of positive integers
- We want the sum of the last $k$ elements
  - Amazon: Avg. price of last $k$ sales
- Solution:
  - (1) If you know all have at most $m$ bits
    - Treat $m$ bits of each integer as a separate stream
    - Use DGIM to count 1s in each integer $c_i$…estimated count for $i$-th bit
    - The sum is $\sum_{i=0}^{m-1} c_i 2^i$
  - (2) Use buckets to keep partial sums
    - Sum of elements in size $b$ bucket is at most $2^b$

Idea: Sum in each bucket is at most $2^b$ (unless bucket has only 1 integer)
Bucket sizes:
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

Prakash 2014