Data Stream Analytics

V. CHRISTOPHIDES
University of Crete &
INRIA Paris

Fall 2016

Big Data: Velocity

- Traffic control
- Credit card transaction flows
- IP network monitoring
- Click streams
- Power consumption
- Phone calls
- Weather forecasting
- Financial Tickers

- Measurement data streams: monitor evolution of entity states
- Transactional data streams: log interactions between entities
The Big Challenge

- Data is continuously growing faster than our ability to store or index it!
- Requires infinite storage
- And running time

Data in Rest vs. Data in Motion

**Data in Rest:**
- Finite, persistent relations
- Relatively low update rate (static records)
- Historical data required many times
- No pre-defined notion of time
- Data at any granularity
- Mainly precise data

**Data in Motion:**
- Infinite, transient streams
- Constantly changes (mostly additions) possibly with a multi-GB arrival rate
- Mostly only freshest data (current state) used
- Ordered either implicitly by arrival time or explicitly by timestamps
- Data at fine granularity
- Data stale/imprecise
The Data Streaming Setting

- In many real situations, we do not know the entire data set in advance
  - Data streams: continuous, ordered, unbounded, time varying, noisy,…
  - Traditional DBMS: data stored in finite, persistent data sets
- Data Stream Management (DSM) is important when the order and rate in which input items arrive, is controlled externally
- DSM characteristics
  - Huge volumes of continuous data, possibly infinite
  - Fast changing and requires real-time response
  - Limited memory to store the data (less than linear in the input size)
  - Random access is expensive—sequential scan processing
  - Algorithms can have at most one pass on data (or in very few passes) and may rely on a structure reducing data seen thus far

Data Stream Analysis

- Must analyse data in motion:
  - Scientific research (monitor environment, species)
  - System management (spot faults, drops, failures)
  - Business intelligence (marketing rules, new offers)
  - Revenue protection (phone fraud, service abuse)
  - Reduce energy footprint (smart power grids, energy-aware consumers)
- else, why even measure this data?
- Typically: simple functions of the stream are computed and used as input to other algorithms
  - how often does an item appear in a stream?
  - how many distinct elements are in the stream?
  - what are the top-k most frequent items (aka Heavy Hitters)?
Using Traditional Databases

- Data-passive query-active paradigm

The Data Streams Paradigm

- Data-active query-passive paradigm

Streams Entering. Each stream is composed of elements/tuples
Data Stream Management System (DSMS)

- **User/Application**
- **Streamed Result**
- **Register (Long Standing) Query**
- **Stream Query Processor**
- **Limited Working Space (Memory and/or Disk)**
- **Archival Storage**

**Data Stream Management System (DSMS)**

**DBMS versus DSMS**

- **Resource** (memory, disk, per-tuple computation) rich
  - "Unbounded" disk store
  - Random/Sequential access

- **One-time (snapshot) queries**
  - No real-time query answering

- **Sophisticated query processing**, access plan determined by query optimizer and physical DB design
  - Query Evaluation: Arbitrary
  - Query Plan: Fixed

- **Resource** (memory, per-tuple computation) limited
  - Bounded main memory
  - Sequential access

- **Continuous (long-lasting) queries**
  - Real-time query answering

- **Unpredictable/variable data arrival and characteristics**
  - Reasonably complex, near real time, query processing
  - Query Evaluation: One pass
  - Query Plan: Adaptive
Stream Data Models

- **Basic “vanilla” model**: sequential
  - $S = \langle s_1, s_2, \ldots, s_m \rangle$ with elements drawn from $[n] := 1, 2, \ldots, n$

- **Frequency vectors**: computing some statistical property from the multi-set of items in the input stream
  - $F = (f_1, f_2, \ldots, f_n)$ where $f_j = |i: a_i = j|$ with $f$ starting at 0

- **Turnstile model**: elements can “arrive” and “depart” from the multi-set by variable amounts (updates)
  - upon receiving $a_i=(j, c)$, update $f_j \leftarrow f_j + c$

- **Cash register model**: only positive updates are allowed (increments)
  - E.g., web server traffic
Motivating Application:
ISP Network Measurement Data

ISP Network Measurement Data

- ISPs were first to hit many “big data” problems
- Networks are sources of truly massive data streams arriving at rapid rates:
  - Telephony call detail records
    - Originating and receiving telephone number, duration, …
  - 24x7 IP traffic flow records generated by routers
    - Source and destination IP address of packet flows, #packets, #bytes,…
  - Protocol transitions
    - Handovers of mobile device between wireless base stations
- Example: AT&T/Sprint collect ~100 Terabyte of Cisco NetFlow data each day
  - Used in network management over a range of timescales: from months (network planning) to seconds (network attack detection)
  - Often shipped off-site to data warehouse for off-line analysis
Packet-Level Data Streams

- Single 2Gb/sec link; say avg packet size is 50bytes
- Number of packets/sec = 5 million
- Time per packet = 0.2 microsec
- If we only capture header information per packet: src/dest IP, time, no. of bytes, etc. – at least 10bytes
  - Space per second is 50Mb
  - Space per day is 4.5Tb per link
  - ISPs typically have hundreds of links!

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Duration</th>
<th>Bytes</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1.0.2</td>
<td>16.2.3.7</td>
<td>12</td>
<td>20K</td>
<td>http</td>
</tr>
<tr>
<td>18.6.7.1</td>
<td>12.4.0.3</td>
<td>16</td>
<td>24K</td>
<td>http</td>
</tr>
<tr>
<td>13.9.4.3</td>
<td>11.6.8.2</td>
<td>15</td>
<td>20K</td>
<td>http</td>
</tr>
<tr>
<td>15.2.2.9</td>
<td>17.1.2.1</td>
<td>19</td>
<td>40K</td>
<td>http</td>
</tr>
<tr>
<td>12.4.3.8</td>
<td>14.8.7.4</td>
<td>26</td>
<td>56K</td>
<td>http</td>
</tr>
<tr>
<td>10.5.1.3</td>
<td>13.0.0.1</td>
<td>27</td>
<td>100K</td>
<td>ftp</td>
</tr>
<tr>
<td>11.1.0.6</td>
<td>10.3.4.5</td>
<td>32</td>
<td>300K</td>
<td>ftp</td>
</tr>
<tr>
<td>19.7.1.2</td>
<td>16.5.5.8</td>
<td>18</td>
<td>80K</td>
<td>ftp</td>
</tr>
</tbody>
</table>

IP session data (collected using Cisco NetFlow)

- Analyzing packet content streams – whole different ballgame!!

Network Data Processing

- Network-wide Reporting
  - How many bytes were sent between a (@S, @D) pair of IP addresses?
  - What fraction of network IP addresses are active?
  - How many distinct (@S, @D) pairs have been seen by specific routers
  - List the top (most frequent) 1000 (@S, @D) pairs over the last month, last week, last day, last hour
- Performance/Reliability Troubleshooting
  - What is the average duration of an IP session?
  - What is the median of the number of bytes in each IP session?
- Fraud Detection
  - List all sessions that transmitted more than 1000 bytes
  - Identify all sessions whose duration was more than twice the normal
- Security/Denial of Service
  - List all IP addresses that have witnessed a sudden spike in traffic distribution overnight
  - Identify IP addresses involved in more than 1000 sessions
IP Flow Records Streams

- **IP Flow**: set of packets with common key observed close in time
- **Flow Key**: IP src/dst address, TCP/UDP ports, ToS,… [64 to 104+ bits]
- **Flow Records**:
  - Protocol level summaries of flows, compiled and exported by routers
  - Flow key, packet & byte counts, first/last packet time, some router state
  - Realizations: Cisco Netflow, IETF Standards
- **Scale**: 100’s TeraBytes of flow records daily are generated in a large ISP
- **Used to manage network over range of timescales**:
  - Capacity planning (months),…., detecting network attacks (seconds)
- **Analysis tasks**
  - **Easy**: timeseries of predetermined aggregates (e.g. address prefixes)
  - **Hard**: fast queries over exploratory selectors, history, communications subgraphs

Abstraction: Keyed Data Streams

- **Data Model**: items are keyed weights
  - Items \((x,k)\): Weight \(x\); key \(k\)
    - Example 1: items= packets, \(x=\) bytes, \(k=\) src/dst IP
    - Example 2: items= flows, \(x=\) packets or bytes, \(k=\) src/dst IP, TCP/UDP port
    - Example 3: items= search queries, \(x =\) execution time, \(k=\) user id
- **Stream of keyed weights**, \(\{(x_i,k_i): i = 1,2,\ldots,n\}\)
- **Generic query**: subset sums
  - \(X(S) = \sum_{i \in S} x_i\) for \(S \subset \{1,2,\ldots,n\}\) i.e. total weight of index subset \(S\)
  - Typically \(S = S(K) = \{i: k_i \in K\}\) : items with keys in \(K\)
    - Example 1, 2: \(X(S(K))\) = total bytes to given IP dest address/UDP port
    - Example 3: \(X(S(K))\) = total query execution time for a set of users
Basic Techniques for Stream Data Processing

- Many stream-based applications strive for high-quality approximate answers!
- Methodology: Data Reduction (trade-off between storage & accuracy)
  - Use a much smaller data summary ($O(\log^k N)$ space) than their original data set ($O(N)$ space)
  - Compute an approximate answer $d$ within a small error range $\varepsilon$: find some $d$ of the true answer $D$ so that $(1-\varepsilon)D < d < (1+\varepsilon)D$ with probability $1-\delta$

- Major data reduction methods
  - Sampling
    - General & agnostic to the analysis to be done
  - Sliding windows
    - Good for processing only fresh data
  - Synopses or Sketches
    - Good for processing also historical data
  - Other transform/dictionary based summarization methods (Wavelets, Fourier Transform, Discrete Cosine Transform, Histograms)
    - Not incrementally updatable, high overhead

Interesting Problems on Data Streams

- 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
- Estimating moments
  - Estimate avg./std. dev. of last $k$ elements
- Finding frequent elements
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.
  - Then we can ask queries about the selected subset and have the answers be statistically representative of the stream as a whole.
- Sampling as a Mediator of Constraints

Data Characteristics
(Heavy Tails, Correlations)

Resource Constraints
(Bandwidth, Storage, CPU)

Query Requirements
(Ad Hoc, Accuracy, Aggregates, Speed)
Two Different Problems

1. Sample a fixed proportion of elements in a stream (say 1 in 10) for which we know its length in advance.

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.

Problem 1: Sampling a Fixed Proportion

- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as:** What fraction of a typical user’s queries were repeated over the past month?
  - Have space to store 1/10th of the query stream

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

Example from the (now) infamous AOL query log (in a stream those tuples arrive sorted by time)
Problem with Naïve Approach

- Simple question: What fraction of queries by an average user of a search engine are duplicates?

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

  - A user

    - $x$ queries submitted once
    - $d$ queries submitted twice
    - 0 queries more than twice

  - Correct answer: $d/(x+d)$

Problem with Naïve Approach

- Proposed solution: We keep 10% of the queries, sample will contain:

  - #queries sampled that were issued once (singleton queries):
    - $x/10$
  
  - #queries sampled twice that were issued twice (duplicate queries):
    - $d/100 = 1/10 \times 1/10 \times d$
    (d times the probability that both occurrences of the query will be in the 1/10th sample)
  
  - #queries sampled once that were issued twice (non sampled duplicate queries):
    - $18d/100 = ((1/10 \times 9/10) + (9/10 \times 1/10)) \times d$
    (the probability that one of the two occurrences will be in the 1/10th sample, while the other is in the 9/10th that is not selected)

- So the sample-based answer is the #queries sampled twice/all sampled queries

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

\[ \frac{d}{10x + 19d} \text{ vs } \frac{d}{x+d} \]

---

Solution: Sample Users

- Pick \( \frac{1}{10} \)th of users and take all their searches in the sample.
- Use a hash function that hashes the user name or id uniformly into 10 buckets.
  - If the user hashes to bucket 0, then accept this search query for the sample, and if not, then not.
- We use the hash function as a random-number generator, with the important property that, when applied to the same user several times, we always get the same “random” number.
  - That is, without storing the in/out decision for any user, we can reconstruct that decision any time a search query by that user arrives.
- Lesson: even in simple settings, convincing counter-examples exist.
  - Always think about the analytic query!
Generalized Solution

- Stream of tuples with keys:
  - Key is some subset of each tuple’s components
    - e.g., tuple is (user, query, time); key is user
  - Choice of key depends on application
- To get a sample of \( \frac{a}{b} \) fraction of the stream:
  - Hash each tuple's key uniformly into \( b \) buckets
  - Pick the tuple if its hash value is at most \( a \)

Example: 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
Problem 2: Maintaining a Fixed-size Sample

- Suppose we need to maintain a random sample $S$ of size exactly $s$ items
  - E.g., main memory size constraint
- Why? Don’t know the 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 reason about the problem: say $s = 2$

Stream: $a x c y z k g e g…$

At $n = 5$, each of the first 5 items is included in the sample $S$ with equal prob.
At $n = 7$, each of the first 7 items is included in the sample $S$ with equal prob.

Impractical solution would be to store all the $n$ items 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 $s$ items of the stream to $S$
  - Suppose we have seen $n$ items, and now the $(n+1)^{th}$ item arrives ($n > s$)
    - With probability $s/n + 1$, keep the $(n+1)^{th}$ element, else discard it
    - If we picked the $(n+1)^{th}$ 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 $n$ items, $S$ contains each item seen so far with prob. $s/n$
  - Let $S_n = \text{sample set after } n \text{ arrivals}$

New item: selection probability

$\text{Prob}[n \in S_{n+1}] = p_{n+1} \coloneqq s/n + 1$

Previously sampled item $m$: induction

$m \in S_n \text{ w.p. } p_n \Rightarrow m \in S_{n+1} \text{ w.p. } p_{n+1} = p_n \times n/(n+1) = s/n + 1$
Proof: By Induction

- We prove this by induction:
  - Assume that after $n$ items, the sample contains each item seen so far with probability $s/n$
  - We need to show that after seeing item $n+1$ the sample maintains the property
    - Sample contains each item seen so far with probability $s/(n+1)$

- Base case:
  - After we see $n=s$ items the sample $S$ has the desired property
    - Each out of $n=s$ items is in the sample with probability $s/s = 1$

Proof: By Induction

- Inductive hypothesis: After $n$ items, the sample $S$ contains each item seen so far with prob. $s/n$
- Now item $n+1$ arrives
- Inductive step: For items already in $S$, probability that the algorithm keeps them in $S$ is:

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

- So, at time $n$, items in $S$ were there with prob. $s/n$
- Time $n \rightarrow n+1$, item stayed in $S$ with prob. $n/(n+1)$
- So prob. That item is in $S$ at time $n+1 = \frac{s}{n+1}$
Some applications rely on **ALL** historical data.

But for several applications, **OLD** data is considered less important or relevant and could skew results from **NEW** trends or conditions.

- Trending new articles on Twitter
- Trending sales on Amazon
- …

**Common approaches** addressing Old data:

- **Aging Model**: elements are associated with “weights” that decrease over time
  - may use some *linear* or *exponential decay* formulas
- **Stream Data Windows**: new elements are incorporated as they appear
Stream Data Window Types

- **Windows based on ordering attributes** (e.g., time)
  - Let T be the window length (size) expressed in units of the ordering attribute (e.g., T may be a time window)

- **Windows based on tuple counts**
  - Let N be the window length (size) expressed in tuples (sliding, tumbling) over the stream

- **Windows based on explicit markers** (e.g., punctuations)
  - Enables data item-dependent variable length windows e.g., a stream of auctions
    - Application inserted “end-of-processing” markers
    - Each data item identifies “beginning-of-processing”

- **Variants** (e.g., partitioning tuples in a window) size

---

**Stream Data Windows: Sliding vs. Tumbling**

A useful model of stream processing is that queries are about a *window* of length *N* – the *N* most recent elements received

---

**Sliding Window**

- Give me how many times have we sold X in the last *k* sales, updating results *every new sale*

**Tumbling Window**

- Give me how many times have we sold X in the last *k* sales, updating results *every 100 sales*
Sliding Window on a Single Stream

Window size $N = 6$

$qwertyuiopasdfghjklzxcvbnm$
$qwertyuiopasdfghjklzxcvbnm$
$qwertyuiopasdfghjklzxcvbnm$
$qwertyuiopasdfghjklzxcvbnm$
$qwertyuiopasdfghjklzxcvbnm$

---

Let’s Start Simple: Counting Bits

- **Problem:**
  - Given a stream of 0s and 1s
    - **Amazon example:** For every product $X$ we keep 0/1 stream of whether that product was sold in the $n$-th transaction
  - 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 $N+1^{st}$ bit

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0

---

Suppose $N = 6$
The Cost of Exact Counts

- You cannot get an exact answer without storing the entire window.
  - In fact, we need N bits, even if the only query we can ask is “how many 1’s are in the entire window of length N?”

- Real Problem: N is still so large that it cannot be stored on disk, or there are so many streams that windows for all cannot be stored.
  - E.g., we’re processing 1 billion streams and \( N = 1 \) billion thus we cannot afford to store N bits.

- We need to summarize even the window and be happy with an approximate answer.

An Attempt: Simple solution

- A simple solution that does not really solve our problem:
  - Uniformity assumption

- Maintain 2 counters:
  - \( S \): number of 1’s from the beginning of the stream
  - \( Z \): number of 0’s from the beginning of the stream

- How many 1’s are in the last N bits? \( N \cdot \frac{S}{S+Z} \)

- But, what if stream is non-uniform?
  - What if distribution changes over time?
DGIM [Datar, Gionis, Indyk, Motwani] Method

- DGIM solution [SODA’02] that does not assume uniformity
- Each bit in the stream has a timestamp, the position in which it arrives
  - The first bit has timestamp 1, the second has timestamp 2, and so on
- Record timestamps modulo \( N \) (the window size), so we can represent any relevant timestamp in \( O(\log_2 N) \) bits
  - If \( N = 1M \), then we need 20 bits
- We use \( O(\log_2^2 N) \) bits to represent a window of \( N \) bits per stream (more latter)
- Solution gives approximate answer, never off by more than 50%
  - Error factor can be reduced to any fraction \( \varepsilon > 0 \), with more complicated algorithm and proportionally more stored bits (with a constant factor that grows as \( \varepsilon \) shrinks)

Idea: Exponential Windows

- Solution that doesn’t (quite) work:
  - Summarize exponentially increasing regions (blocks) of the stream, looking backward. Regions consisting of
    a) the timestamp of its right (most recent) end
    b) the number of 1’s in between its beginning and end
  - Drop small regions if they begin at the same point as a larger region

We can reconstruct the count of the last \( N \) bits, except we are not sure how many of the last 6 1’s are included in the \( N \)
What’s Good?

- Stores only $O(\log_2^2 N)$ bits (<< $N$ bits)
  - $O(\log_2 N)$ 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 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
  - Notice that it is OK for some 0’s to lie between the blocks
- A *bucket* in the DGIM method is a record consisting of:
  1. The *timestamp of its end* \([O(\log_2 N)]\) bits
  2. The *number of 1s* between its beginning and end \([O(\log_2 \log_2 N)]\) bits
- **Constraint on buckets:** Number of 1s must be a power of 2
  - That explains the \([O(\log_2 \log_2 N)]\) in 2 above

Representing a Stream by Buckets

- There are 5 rules that must be followed when representing a stream by buckets:
  - The right end of a bucket is always a position with a 1
  - Buckets do not overlap in timestamps
    - no position is in more than one bucket
  - There are one or two buckets of any given size, up to some max. size
  - All sizes must be a power of 2 of 1s
  - Buckets are sorted by size
    - Earlier buckets (as we move to the left) are not smaller than later buckets
Example: Bucketized Stream

At least 1 of size 16. Partially beyond window.

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 (right) 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 …
- Update time complexity: \( O(\log N) \)
Example: Updating Buckets

Current state of the stream:

```
0010101100010110101010101010110101010101011101010101110101000101100101
```

Bit of value 1 arrives

```
0010101100010110101010101010110101010101011101010101110101000101100101
```

Two oldest purple buckets get merged into a yellow bucket

```
0010101100010110101010101010110101010101011101010101110101000101100101
```

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

```
10101100010110101010101010110101010101011101010101110101000101100101101
```

Buckets get merged…

```
10101100010110101010101010110101010101011101010101110101000101100101101
```

State of the buckets after merging

```
10101100010110101010101010110101010101011101010101110101000101100101101
```

How to Query?

- To estimate the number of 1s in the most recent N bits:
  1. Determine bucket B with the earliest timestamp that includes at least some of the N most recent bits
  2. Sum the sizes of all buckets to the right of B
     - recall that "size" means the number of 1s in the bucket
  3. Final estimate: Add half the size of bucket B

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

Bucket b

N

• How many 1’s?
  ◆ Estimated: 1 + 1 + 2 + 4 + 4 + 8 + 8 + ½*16 = 36
  ◆ Exact: 1 + 1 + 2 + 4 + 4 + 8 + 8 + 7 = 35

Error Bound: Proof

• Why is error 50%? Let’s prove it!
• Suppose the last bucket B has size $2^r$
• Case 1: estimate is less than exact answer:
  ◆ Worst case: all 1’s of B are in range of the query but only half are included (by def. of the estimate)
  ◆ Thus, estimate is at least 50% of the exact answer
• Case 2: estimate is greater than exact answer:
  ◆ Worst case: only the rightmost bit of B ($2^r$) is in range of the query and only one bucket of each size smaller than B exists
    • Exact: $1 + 2^{r-1} + 2^{r-2} + \ldots + 1 = 2^r$
    • Estimated: $2^{r+1} + 2^{r-1} + 2^{r-2} + \ldots + 1 = 2^r + 2^{r-1} - 1$
  ◆ Thus, estimate is at most 50% greater than the exact answer
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 \( k \) bits? 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?
Counting on Streams 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 (i.e. integer value up-to \( 2^m \))
     - Treat \( m \) bits of each integer as a separate stream
     - Use DGIM to count 1s in each integer
     - The sum is \( \sum_{i=0}^{m-1} c_i 2^i \) \( c_i \) ...estimated count for \( i \)-th bit (least sign. is \( c_0 \))
  2. Use buckets to keep partial sums
     - Sum of elements in size \( b \) bucket is at most \( 2^b \) (unless bucket has only 1 integer)

Example

\[
\begin{array}{ccccc}
\text{most recent} & 7 & 6 & 5 & 4 & 3 \\
\text{depends on largest number} & 2 & 1 & 23 & 5 & 4 \\
\end{array}
\]

\[
\begin{array}{cccccc}
& - & - & 1 & 1 & 1 \\
& - & - & - & 1 & 0 \\
& - & - & - & - & 1 \\
& 1 & 0 & 1 & 1 & 1 \\
& - & - & 1 & 0 & 1 \\
& - & - & 1 & 0 & 0 \\
\end{array}
\]

\( k = 4 \) window size \( 7 + 2 + 1 + 23 = 33 \)

\[
\begin{align*}
3 \times 2^0 + 3 \times 2^1 + 2 \times 2^2 + 1 \times 2^4 &= 33 \\
in \text{practice, the 1's are estimated via DGIM}
\end{align*}
\]
So far …

- 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

Filtering Data Streams
Motivating Applications

- 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 info items (news articles)
  - people express interest in certain sets of keywords
  - determine whether each info items matches user’s interest
- Web crawling
  - a web crawler keeps, centrally, a list of all the URLs it has found so far
  - it assigns these URLs to any of a number of parallel tasks
  - these tasks stream back the URLs they find in the links they discover on a page
  - it needs to filter out those URLs it has seen before

Filtering Data Streams

- Recall: 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
First Cut Solution

- Given a set of keys $S$ that we want to filter
- Create a bit array $B$ of $n$ bits
  - Initialized to all 0s
- Choose a hash function $h$ with range $[0,n)$
  - MD5, SHA256, Murmur
- 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

- Creates false positives but no false negatives
- Will invalid item passed through the filter (false positives)?
  - If the item is NOT in $S$, it may still be output
  - Why? Collision
- Will valid items be filtered out (false positives)?
  - If item is in $S$, we surely output it
Analysis

- \(|S| = 1\) billion email addresses
  \(|B| = 1\)GB = 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

- 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:
  - \(n\) Targets = bits/buckets
  - \(m\) Darts = hash values of items

\[
1 - (1 - 1/n) = 1 - e^{-m/n}
\]
Analysis: Throwing Darts

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

- Example: $10^8$ 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

- Randomized data structure introduced by Burton Bloom [CACM 1970]
  - It represents a set for membership queries, with false positives
  - Probability of false positive can be controlled by design parameters

- Consider: $|S| = m$, $|B| = n$
  - 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 $i = 1, \ldots, k$)

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

- \( S = \{S_1, S_2, S_3\} \)
- \( H = \{h_1, h_2\} \)

\[
\begin{align*}
    h_1(S_1) &= 7 & h_2(S_1) &= 2 & h_1(S_2) &= 0 \\
    h_2(S_2) &= 1 & h_1(S_3) &= 2 & h_2(S_3) &= 5
\end{align*}
\]

Bit array \( B \):

\[
\begin{align*}
    11100101000
\end{align*}
\]

- \( h_1(I_1) = 1 \)
- \( h_2(I_1) = 3 \)
  
  Since bit at position 3 is 0, drop \( I_1 \).

- \( h_1(I_2) = 7 \)
- \( h_2(I_2) = 5 \)
  
  Since bits at positions 5 and 7 are both 1, output \( I_2 \) (potentially a false positive).

Bloom Filter – Analysis

- What fraction of the bit vector \( B \) are 1s?
  - Throwing \( k \times m \) 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 \( k = 6 \): \( (1 - e^{-6/8})^6 = 0.0235 \)

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

- Cons:
  - Small false positive probability
  - No deletions
  - Can not store associated objects

- Is it better to have 1 big B or \( k \) small Bs?
  - It is the same: \( (1 - e^{km/n})^k \) vs. \( (1 - e^{m/(nk)})^k \)
  - But keeping 1 big B is simpler
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

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 distinct products have we sold in the last week?
- How many unique users have accessed a website per month?
- How many unique IP addresses have passed packages through a router?
Distinct IP Addresses that Contacted a Router

Addresses seen:
- 18.9.22.69
- 69.172.200.24
- 106.10.165.51

Exact Algorithm

- Estimate the count in an unbiased way
  - Keep an array, \( a[1..N] \), initially all zero
  - Also keep a counter \( C \)
  - Every time an item \( i \) arrives, look at \( a[i] \)
  - If it is zero, increment \( C \) and set \( a[i]=1 \)
  - Return \( C \) as the number of distinct items
- Time: \( O(1) \) per update and per query
- But space is \( O(N) \)

- Real problem: What if we do not have space to maintain the set of elements seen so far?
  - Accept that the count may have a little error, but limit the probability that the error is large
Approach: hash data stream elements uniformly to $\log N$ bit values, i.e.:

- $f: x_i \rightarrow \{0, 1\}^{\log N}, x_i = x_j \Rightarrow f(x_i) = f(x_j)$
- $P(f(x_i) = 1) = p$ identically, independently distributed for all $x_i$

Assumption: the larger the number of distinct elements in the stream, the more distinct the occurring hash values, and the more likely one with an “unusual” property appears

- assume we encountered $m$ distinct $x_i$ the probability $P_m$ that $f(x_i) = 0$ for all $i$ depends on the number of unique $x_i$, not on the total number of elements

Probabilistic Counting:

- Keep a bit array $a[1..\log N]$, initially 0
- Pick a hash function $h$ that maps each of the $N$ elements to at least $\log N$ bits
- Compute $f(x_i)$ for every element $x_i$ in the stream, and set $a[f(x_i)]$ to 1
- Somehow extract from this the approximate number of distinct elements

One possibility of interpreting “unusual” is the hash tail: the number of 0’s a binary hash value ends in

- $100110101110100110101100010110000000$

Let $f$ be drawn from a family of strongly 2-Universal hash functions mapping onto $\{0..U-1\}$

For each stream element $x_i$, let $r(x_i)$ be the number of trailing 0s in $f(x_i)$

- E.g., say $h(i) = 12$, then 12 is 1100 in binary, so $r(i) = 2$

For each stream element $x_i$, set $a[r(h(x_i))] = 1$

Let $R$ be the maximum $j$ such that $a[j] = 1$

- $R = \max_j r(x_j)$, over all the elements $x_j$ seen so far

Estimated number of distinct elements $= 2^R$
Example

- Suppose the stream is 1, 0, 2, 1, 2, 0, 4, 0, 1, 2, 0, 1...

- Let \( h(x) = (3x + 1) \mod 5 \)

- So the transformed stream (\( h \) applied to each element) is 4, 1, 2, 4, 2, 1, 3, 1, 4, 2, 1, 4

- We compute \( r \) of each element in the stream:
  2, 0, 1, 2, 1, 0, 0, 0, 2, 1, 0, 2

  - So \( R = 2 \). Output \( 2^2 = 4 \)

Why It Works: Intuition

- Very very rough and heuristic intuition why Flajolet-Martin works:
  - \( f(a) \) hashes \( a \) with equal prob. to any of \( N \) values
  - Then \( f(a) \) is a sequence of \( \log_2 N \) bits, where \( 2^r \) fraction of all \( a \)s have a tail of \( r \) zeros
    - \( P(f(a) \text{ has tail length of at least } r) = 1 / (2 \times 2 \times \ldots \times 2) = 1 / 2^r \)
  - About 50% of \( a \)s hash to **00**
    - About 25% of \( a \)s hash to ***0
    - So, if we saw the longest tail of \( r=2 \) (i.e., element hash ending *100)
      then we have probably seen about 4 distinct items so far
  - So, it takes to hash about \( 2^r \) elements before we see one with zero-suffix of length \( r \)

\[
\frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{8} \quad \frac{1}{16} \quad \frac{1}{32} \quad \frac{1}{64} \quad \ldots
\]
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, the proposed estimate is neither too low nor too high
  - $2^R$ will almost always be around $m!$

- What is the probability that a given $f(a)$ ends in at least $r$?
  - $f(a)$ 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 $m$ distinct elements:

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

  Prob. all end in fewer than $r$ zeros
  Prob. that a given $f(a)$ ends in fewer than $r$ zeros

- 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 $m/2^r \to 0$
    - 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 $m/2^r \to \infty$
    - So, the probability of finding a tail of length $r$ tends to 1

- Thus, $2^R$ will almost always be around $m!$
Computing Moments

Generalization: Moments

- Suppose a stream has elements chosen from a set $A$ of $N$ values
- Let $m_i$ be the number of times value $i$ occurs in the stream
- The $k^{th}$ moment is

$$\sum_{i \in A} (m_i)^k$$
Special Cases

\[ \sum_{i \in A} (m_i)^k \]

- **0th moment** = number of distinct elements
  - The problem just considered

- **1st moment** = count of the numbers of elements = length of the stream
  - Easy to compute

- **2nd moment** = surprise number \( S \) = a measure of how uneven the distribution is

Example: Surprise Number

- Stream of length 100
  - 11 distinct values

- Item counts: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9
  - Surprise \( S = 910 \)

- Item counts: 90, 1, 1, 1, 1, 1, 1, 1, 1, 1
  - Surprise \( S = 8,110 \)
AMS [Alon, Matias, and Szegedy] Method

- AMS method works for all moments
  - Gives an unbiased estimate

- We will just concentrate on the 2nd moment \( S \)

- We pick and keep track of many variables \( X \):
  - For each variable \( X \) we store \( X.el \) and \( X.val \)
    - \( X.el \) corresponds to the item \( i \)
    - \( X.val \) corresponds to the count of item \( i \)
  - Note this requires a count in main memory, so number of \( X \)s is limited

- Our goal is to compute \( S = \sum_i m_i^2 \)

One Random Variable (\( X \))

- How to set \( X.val \) and \( X.el \)?
  - Assume stream has length \( n \) (we relax this later)
  - Pick some random time \( t (t < n) \) to start, so that any time is equally likely
  - Let at time \( t \) the stream have item \( i \). We set \( X.el = i \)
  - Then we maintain count \( c (X.val = c) \) of the number of \( i \)s in the stream starting from the chosen time \( t \)

- Then the estimate of the 2nd moment (\( \sum_i m_i^2 \)) is:
  \[
  S = f(X) = n (2 \cdot c - 1)
  \]
  - Note, we will keep track of multiple \( X \)s, \((X_1, X_2, \ldots, X_k)\) and our final estimate will be \( S = 1/k \sum_i^k f(X_i) \)
2nd moment is $S = \sum_i m_i^2$

- $c_i$ ... number of times item at time $t$ appears from time $t$ onwards
  $(c_1 = m_a, c_2 = m_a - 1, c_3 = m_b)$

- $E[f(X)] = \frac{1}{n} \sum_{t=1}^n n(2c_t - 1)$
  
  $= \frac{1}{n} \sum_i n(1 + 3 + 5 + \cdots + 2m_i - 1)$

- Group times by the value seen
- Time $t$ when the last $i$ is seen ($c_t = 1$)
- Time $t$ when the penultimate $i$ is seen ($c_t = 2$)
- Time $t$ when the first $i$ is seen ($c_t = m_i$)

Little side calculation:

$(1 + 3 + 5 + \cdots + 2m_i - 1) = \sum_{i=1}^{m_i} (2i - 1) = \frac{2m_i(m_i + 1)}{2} - m_i = (m_i)^2$

Then $E[f(X)] = \frac{1}{n} \sum_i n (m_i)^2$

So, $E[f(X)] = \sum_i (m_i)^2 = S$

We have the second moment (in expectation)!
Higher-Order Moments

- For estimating $k^{th}$ moment we essentially use the same algorithm but change the estimate:
  - For $k=2$ we used $n (2 \cdot c - 1)$
  - For $k=3$ we use: $n (3 \cdot c^2 - 3c + 1)$ (where $c=X.val$)

- Why?
  - For $k=2$: Remember we had $1 + 3 + 5 + \cdots + 2m_i - 1$ and we showed terms $2c-1$ (for $c=1,\ldots,m$) sum to $m^2$
    - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c - 1)^2 = m^2$
    - So: $2c - 1 = c^2 - (c - 1)^2$
  - For $k=3$: $c^3 - (c-1)^2 = 3c^2 - 3c + 1$

- Generally: Estimate $= n (c^k - (c - 1)^k)$

Combining Samples

- In practice:
  - Compute $f(X) = n(2 \cdot c - 1)$ for as many variables $X$ as you can fit in memory
  - Average them in groups
  - Take median of averages

- Problem: Streams never end
  - We assumed there was a number $n$, the number of positions in the stream
  - But real streams go on forever, so $n$ is a variable – the number of inputs seen so far
Streams Never End: Fixups

1. The variables $X$ have $n$ as a factor – keep $n$ separately; just hold the count in $X$

2. Suppose we can only store $k$ counts. We must throw some $X$s out as time goes on:
   - **Objective:** Each starting time $t$ is selected with probability $k/n$
   - **Solution:** (fixed-size sampling!)
     - Choose the first $k$ times for $k$ variables
     - When the $n^{th}$ element arrives ($n > k$), choose it with probability $k/n$
     - If you choose it, throw one of the previously stored variables $X$ out, with equal probability

Counting Itemsets
Counting Itemsets

- **New Problem:** Given a stream, which items appear more than $s$ times in the window?
- **Possible solution:** Think of the stream of baskets as one binary stream per item
  - $1$ = item present; $0$ = not present
  - Use **DGIM** to estimate counts of $1$s for all items

Extensions

- In principle, you could count frequent pairs or even larger sets the same way
  - One stream per itemset
- **Drawbacks:**
  - Only approximate
  - Number of itemsets is way too big
Exponentially Decaying Windows

- Exponentially decaying windows: A heuristic for selecting likely frequent item(sets)
  - What are “currently” most popular movies?
    - Instead of computing the raw count in last $N$ elements
    - Compute a smooth aggregation over the whole stream
  - If stream is $a_1, a_2, \ldots$ and we are taking the sum of the stream, take the answer at time $t$ to be:
    \[
    = \sum_{i=1}^{t} a_i (1 - c)^{t-i}
    \]
    - $c$ is a constant, presumably tiny, like $10^{-6}$ or $10^{-9}$
  - When new $a_{t+1}$ arrives:
    Multiply current sum by $(1-c)$ and add $a_{t+1}$

Example: Counting Items

- If each $a_i$ is an “item” we can compute the characteristic function of each possible item $x$ as an Exponentially Decaying Window
  - That is: $\sum_{i=1}^{t} \delta_i \cdot (1 - c)^{t-i}$
    where $\delta_i=1$ if $a_i=x$, and 0 otherwise
  - Imagine that for each item $x$ we have a binary stream ($1$ if $x$ appears, 0 if $x$ does not appear)
  - New item $x$ arrives:
    - Multiply all counts by $(1-c)$
    - Add $+1$ to count for element $x$
  - Call this sum the “weight” of item $x$
Sliding Versus Decaying Windows

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

Example: Counting Items

- What are “currently” most popular movies?

- Suppose we want to find movies of weight $> 1/2$
  - Important property: Sum over all weights $\sum_t (1 - c)^t$ is $1/[1 - (1 - c)] = 1/c$

- Thus:
  - There cannot be more than $2/c$ movies with weight of $1/2$ or more

- So, $2/c$ is a limit on the number of movies being counted at any time
Extension to Itemsets

- Count (some) itemsets in an E.D.W.
  - 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 (1-\( c \))
  - For uncounted items in B, create new count
  - Add 1 to count of any item in B and to any itemset contained in B that is already being counted
  - **Drop counts** < \( \frac{1}{2} \)
  - 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 $< (2/c) \cdot (\text{avg. number of items in a basket})$

- 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

References

- CS246: Mining Massive Datasets Jure Leskovec, Stanford University, 2014
- CS5344: Big Data Analytics Technology, TAN Kian-Lee, National University of Singapore 2014
- Sampling for Big Data Graham Cormode, Nick Duffield, KDD’14 Tutorial
- 198:671 Processing Massive Data Sets Graham Cormode S. Muthukrishnan Rutgers University 2012
- Streaming in a Connected World: Querying and Tracking Distributed Data Streams Graham Cormode, Minos Garofalakis EDBT’08 Tutorial