Data Streams

- A data stream is a continuous sequence of data records (often, tuples) that arrive at a rapid pace
- Examples:
  - Search queries
  - Streams of tweets
  - Click streams on media sites; “likes” on FB
  - Credit card transactions
  - Sensor data
- Note that in the examples above, a distributed source (users, sensors) is generating the stream
Processing Data Streams

The data keeps on coming, and volume dictates that:

- Processing often happens through a single pass over the data
  - It is sometimes possible to perform a small number of passes over the data
- Only a constant amount (hence negligible portion) of the stream can be kept in memory
  - Gives rise to “sliding window” processing

At even greater scales:

- Distributed stream processing ingests and processes the stream across multiple nodes
  - Requires adaptations of many algorithms, often not easy (next slide)
- Sampling: sub-sample the stream & work on a smaller sample of it
  - Many application and data specific pitfalls in sampling correctly
- If latency is allowed: piecemeal incremental processing
Reservoir Sampling – Definition

- Problem definition: select a subset of $k$ elements, uniformly at random, in a stream of unknown length
  - Can’t select the element indices in advance
  - Can’t store the entire stream in memory
  - Can’t perform a second pass over the data
- Alternative formulation: after ingesting any $j$ elements, $j \geq k$, have ready a subset of $k$ elements chosen uniformly at random from elements $1, \ldots, j$

Reservoir Sampling – Inductive Reasoning

- Base: for $j=k$, the first $k$ elements in the stream form the sampled subset
- Assume we have a subset of $k$ items chosen u.a.r. from elements $1, \ldots, j$ and consider element $j+1$
- Element $j+1$ should belong to the subset w.p. $k/(j+1)$
- If $j+1$ is chosen to join the subset, all existing subset elements equally deserve to be displaced
  - Each with probability $1/k$
Reservoir Sampling – Algorithm

- Input: k (sample size), stream S (S[1], S[2], ...)
- Let R (the reservoir) be an array of size k, R[1],...,R[k]

1. Initialize R with the first k elements in the stream: R[j]=S[j] for all j=1,...,k
2. For (j=k+1; S[j] exists; j++) {
   1. r = random_integer[1,j]
   2. if (r ≤ k) // element j is chosen
   3. }
3. }
4. Return R

Frequent Itemset Mining

- Let there be a (perhaps implicit) universe U of items
- Definition: a transaction is a subset of U
  - Typically, transactions are very small subsets of U
- Problem definition: given a set of transactions, find subsets of U that are contained in many transactions
  - "Many": beyond absolute or relative thresholds, top-k, etc.
  - "Subsets": of/above a certain given size, or any size
- Prototypical application: market-basket analysis

{ A, C, E }
Sub-problem: Frequent Item Mining

- Find all items appearing more than $\theta |S|$ times in stream $S$, for any $0 < \theta < 1$
- For convenience, assume $1/\theta$ is an integer
- Let PF ("potentially frequent") be an initially empty map of key-value pairs – keys are elements, values are counters
- Algorithm (Karp-Papadimitriou-Shanker, 2002):
  - Collect elements into PF, counting their appearances
  - Whenever $1/\theta$ distinct elements are encountered, decrease all counters by 1 and remove from PF those whose counters are 0
  - Output the elements in PF that survive this process

Frequent Item Mining Algorithm

1. $PF = \emptyset$
2. foreach element $e \in S$
3.  if $PF$.hasKey($e$)
   4.    $PF$.value($e$)++
   5.  }
3.  Else
5.    $PF$.insert($e$, 1)
8.  If $|PF| = 1/\theta$
9.     Foreach key $k \in PF$
10.    $PF$.value($k$)--
11.       if $PF$.value($k$) == 0
12.          $PF$.remove($k$
13.     }
14. Output $PF$
Frequent Item Mining: Analysis

- Algorithm’s main loop:
  1. Collect elements into PF, counting their appearances
  2. Whenever $1/\theta$ distinct elements are encountered, decrease all counters by 1 and remove those whose counters are zero

- Observations:
  - Every time (2) happens, $1/\theta$ non-zero counters are decreased
  - Hence, this can happen no more than $\theta|S|$ times
  - Hence, any element’s counter is decreased no more than $\theta|S|$ times
  - Hence, any element that appears in $S$ more than $\theta|S|$ times must have a non-zero counter when the algorithm ends, i.e. appear in PF

- However, PF may also hold “false positives” – infrequent items

Frequent Item Mining: Example

Let $\theta=1/3$, $1/\theta=3$; $S = A B A C B D A B A E B F A$
### Different Perspective: Approximate Counting of Stream Elements

- **Recap of observations:**
  - Every time \(2\) happens, \(1/\theta\) non-zero counters are decreased
  - Hence, this can happen no more than \(\theta|S|\) times
  - Hence, any element’s counter is decreased no more than \(\theta|S|\) times

- The algorithm thus approximates the count of all elements in \(S\) within an additive factor of \(\theta|S|\) given memory of \(O(1/\theta)\):

  \[
  \text{Define: } \text{Approximate\_count\_in\_S}(e) = \begin{cases} 
  0 & e \notin \text{PF} \\ 
  \text{count}(e) & e \in \text{PF} 
  \end{cases}
  \]

  Then, \(\forall e \in S\),
  \[
  \text{True\_count\_in\_S}(e) - \text{Approximate\_count\_in\_S}(e) \leq \theta|S|
  \]

### Sliding Window Aggregates

- **Input:**
  - A stream of numbers \(s_0, s_1, s_2, \ldots\)
  - A window size parameter \(N\)

- **Output:** at any time \(t\), aggregates over the last \(N\) stream elements \(s_{t-N+1}, \ldots, s_t\)

- **The aggregates:**
  - Arithmetic aggregates: sum, product, etc.
  - Moments: average, variance, etc.
  - Order statistics: median, etc.
  - Other: number of unique elements, most frequent elements, etc.
This Class: Sliding Window Summation of a Binary Stream

- Input:
  - A binary stream $b_0, b_1, b_2, ...$
  - A window size parameter $N$
- Output: at any time $t$, the sum of the last $N$ stream elements $b_{t-N+1}, ..., b_t$
- For precise results, must keep the entire window in RAM
  - Prohibitive for large $N$ and perhaps unnecessary
- Can we trade off accuracy and memory?
- The next slides follow M. Datar and R. Motwani in showing a scheme that approximates the sum to within a $1/k$ relative error in $O(k \log N \log N/k)$ memory

Sliding Window Summation of a Binary Stream: Definitions, Notations

- A $(q,i)$-bucket is a subsequence of the stream $b_{q-a}, ..., b_q$ such that:
  - $b_q$, the most recent element of the bucket, equals 1
  - $b_{q-a-1}$, the element beyond the least recent element of the bucket, also equals 1
  - The sum of the bucket is $2^i$: $\sum_{j=0,...,a} b_{q-j} = 2^i$
- Example: 0 1 1 0 0 0 1 0 1 0 0 1 0 1 1 0 1 0 1 1
  - (8,1) bucket
  - (23,3) bucket
Sliding Window Summation of a Binary Stream: Definitions, Notations

- An *m-tiling* of the window \(b_{t-N+1},...,b_t\) at time \(t\) is a set of buckets \((q_1,i_1),...,(q_m,i_m)\) such that:
  - \(b_{q_1}\) is the most recent 1 in the window \((t-N+1 \leq q_1 \leq t)\)
  - Every element in \(b_{t-N+1},...,b_{q_1}\) belongs to exactly one bucket*
  - \(q_1 > q_2 > ... > q_m \geq t-N+1\) (i.e. all buckets start in the window, and they are numbered by the recency of their start index)
  - \(i_1 \leq i_2 \leq ... \leq i_m\), i.e. the buckets grow exponentially in sum as they age (recall that the sum of each bucket is 2^\(i\))

- Example for \(N=15\) (window elements are *blue*):

```
010010110011110000010
```

Intuition for \(1/2\) Relative Error

- Imagine the window is covered by an m-tiling of buckets \((q_1,i_1),...,(q_m,i_m)\) such that:
  - \(i_1, i_2 = 0\) (i.e. first two buckets contain a single 1)
  - For all \(j=2,...,m\): \(i_j = 1 + i_{j-1}\) (i.e. subsequent buckets’ sum doubles per bucket)
  - In other words, the sum of bucket \(j\) \((j>1)\) = \(2^{i_j}\)

- Example: 
```
010010110011110000010
```
### Intuition for $\frac{1}{2}$ Relative Error (cont.)

- **Example:** 010010110011110000010

- **Observation 1:** For all $j > 1$, $\sum_{k<j} 2^i k = 2^i j$
  - In other words, the number of 1s in any non-first bucket is equal to the number of 1s in all previous buckets combined.
  - Since $\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$

- **Observation 2:** $m < 2 + \log N$
  - Since a $(q, i)$-bucket is at least $2^i$ elements large, and the first $m-1$ buckets together cover at least $2^{m-2}$ elements but fail to cover the entire window, we get $2^{m-2} < N$.

- **Observation 3:** $1 + \sum_{j<m} 2^i j \leq \text{sum of window} \leq \sum_{j\leq m} 2^i j$

28 May 2017

---

### Intuition for $\frac{1}{2}$ Relative Error (cont.)

- **Example:** 010010110011110000010

- **Observation 1:** For all $j > 1$, $\sum_{k<j} 2^i k = 2^i j$

- **Observation 3:** $1 + \sum_{j<m} 2^i j \leq \text{sum of window} \leq \sum_{j\leq m} 2^i j$

- **Recall:** That whenever $m > 1$, bucket $m$ contains $2^{m-2}$ 1s, as do all previous buckets combined.

- **Reinterpreting obs. 3:** $1 + 2^{m-2} \leq \text{sum of window} \leq 2^{m-1}$

- **Whenever $m > 2$, estimate the sum to be $2^{m-2} + 2^{m-3}$ and:**
  \[
  \frac{|\text{True sum} - \text{estimation}|}{\text{True sum}} \leq \frac{2^{m-3}}{1 + 2^{m-2}} < \frac{1}{2}
  \]

- **By obs. 2, we require $O(\log N)$ buckets**

28 May 2017
From Intuition to Construction

- The tiling shown cannot be maintained as the stream is processed (why?)
- However, the ideas can be generalized to a maintainable tiling
- Assume we want relative error $1/2k$
  - We will roughly create $k$ copies of the previous construction
  - Actually, we will maintain between $k$ and $k+1$ copies of it

Construction for $1/(2k)$ Error

- At all times $t$, purge any $(t',j)$ bucket where $t-t' \geq N$
- Initially, grow up to $2k+1$ buckets covering a single 1
  - Whenever the $2k+2$ such bucket is created, merge the oldest two into a bucket covering two 1s
- As long as there exists a $(t',i)$ bucket for $i > 1$, then for any $j$ such that $0 < j < i$, there must be either $k$ or $k+1$ buckets covering $2^j$ 1s
  - Whenever the $k+2$ such bucket is created, merge the oldest two into a bucket covering $2^{j+1}$ 1s
- This construction is both sustainable and easily maintainable
Visualizing the Construction

- K=1, N=10
- \[\begin{array}{cccccc}
1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}!\]

Estimation of the Sum of the Window

- If all buckets cover a single 1, the window’s sum is simply the number of buckets in the tiling.
- From now on, assume not all buckets cover a single 1.
- Let the tiling be \( (q_1, i_1), ..., (q_m, i_m) \); denote \( r = i_m \).
- Estimate the sum of the window to be \( 2^{r-1} + \sum_{j<m} 2^{i_j} \)
  - Half the number of 1s in the last bucket, plus all the 1s in the previous buckets.
- Claim: \( |\text{True sum} - \text{estimation}| / \text{True sum} \leq 1/2^k \)
- Trivially, \( |\text{True sum} - \text{estimation}| \leq 2^{r-1} \)
**Visualizing the Construction**

K=1, N=10

<table>
<thead>
<tr>
<th>Est:6</th>
<th>T:7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 1 0</td>
<td>1 1 0 1 1 0 1 0</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

**Correctness**

- There are 2k or 2k+1 buckets covering a single 1
- If r=i_m =1, our estimate is off by at most 1 and the claim holds
- Otherwise, assume r>1
- For any 0<j<r, by construction there are k or k+1 buckets covering 2^j 1s
- The sum of all but the last bucket is thus at least k2^r
  - At least k copies of the “intuitive” construction
- The sum of the window is at least 1+k2^r
- Thus, |True sum – estimation| / True sum ≤ 2r-1/(1+k2^r) < 1/2k
Memory Requirements

- Bounding m, the number of required buckets:
  - We know $k^2 < N$, and hence $r < \log(N/k)$
  - We know $m \leq (k+1)r + 2k+1 < (k+1)(r+2)$
  - Hence $m < (k+1)(2 + \log N/k)$
- Per bucket, we need to encode offset and number of 1s covered
  - Since we purge offsets more than N in the past, the relative offset of every non-purged bucket requires $\log(N)$ bits
  - The log of the number of 1s covered requires $\log\log(N)$ bits
- Overall memory requirements: $O(k \log N \log N/k)$