Mining Data Streams

The Stream Model
Sliding Windows
Counting 1’s

Jeffrey D. Ullmam
Stanford University
Data Management Vs. Stream Management

- In a DBMS, input is under the control of the programming staff.
  - SQL INSERT commands or bulk loaders.
- Stream management is important when the input rate is controlled externally.
  - Example: Google search queries.
The Stream Model

- Input tuples enter at a rapid rate, at one or more input ports.
- The system cannot store the entire stream accessibly.
- How do you make critical calculations about the stream using a limited amount of (primary or secondary) memory?
Two Forms of Query

1. **Ad-hoc queries**: Normal queries asked one time about streams.
   - **Example**: What is the maximum value seen so far in stream $S$?

2. **Standing queries**: Queries that are, in principle, asked about the stream at all times.
   - **Example**: Report each new maximum value ever seen in stream $S$. 

Streams Entering

Processor

Limited Working Storage

Archival Storage

Output

Ad-Hoc Queries

Standing Queries

... 1, 5, 2, 7, 0, 9, 3

... a, r, v, t, y, h, b

... 0, 0, 1, 0, 1, 1, 0

time
Applications

- 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.
    - Often caused by annoyed users clicking on a broken page.
- IP packets can be monitored at a switch.
  - Gather information for optimal routing.
  - Detect denial-of-service attacks.
A useful model of stream processing is that queries are about a window of length $N$ – the $N$ most recent elements received.

- **Alternative**: elements received within a time interval $T$.

- **Interesting case**: $N$ is so large it cannot be stored in main memory.
  - Or, there are so many streams that windows for all do not fit in main memory.
Example: Averages

- Stream of integers, window of size $N$.
- **Standing query**: what is the average of the integers in the window?
- For the first $N$ inputs, sum and count to get the average.
- Afterward, when a new input $i$ arrives, change the average by adding $(i - j)/N$, where $j$ is the oldest integer in the window before $i$ arrived.
- **Good**: $O(1)$ time per input.
- **Bad**: Requires the entire window in main memory.
Counting 1’s

Approximating Counts
Exponentially Growing Blocks
DGIM Algorithm
Approximate Counting

- You can show that if you insist on an exact sum or count of the elements in a window, you cannot use less space than the window itself.
- But if you are willing to accept an approximation, you can use much less space.
- We’ll consider the simple case of counting bits, which includes counting elements of a certain type as a special case.
- Sums are a fairly straightforward extension.
Problem: given a stream of 0’s and 1’s, be prepared to answer queries of the form “how many 1’s in the most recent $k$ bits?” where $k \leq N$.

Obvious solution: store the most recent $N$ bits.

But answering the query will take $O(k)$ time.
- Very possibly too much time.

And the space requirements can be too great.
- Especially if there are many streams to be managed in main memory at once, or $N$ is huge.
Example: Bit Counting

- Count recent hits on URL’s belonging to a site.
- Stream is a sequence of URL’s.
- Window size $N = 1$ billion.
- Think of the data as many streams – one for each URL.
- Bit on the stream for URL $x$ is 0 unless the actual stream has $x$. 
DGIM Method

- Name refers to the inventors:
  - Datar, Gionis, Indyk, and Motwani.
- Store only $O(\log^2 N)$ bits per stream.
  - $N = \text{window size}$.
- Gives approximate answer, never off by more than 50%.
  - Error factor can be reduced to any $\varepsilon > 0$, with more complicated algorithm and proportionally more stored bits.
Each bit in the stream has a \textit{timestamp}, starting 0, 1, ...

Record timestamps modulo $N$ (the window size), so we can represent any \textit{relevant} timestamp in $O(\log_2 N)$ bits.
A **bucket** is a segment of the window; it is represented by a record consisting of:

1. The timestamp of its end \([O(\log N) \text{ bits}].\)
2. The number of 1’s between its beginning and end.
   - Number of 1’s = *size* of the bucket.

**Constraint on bucket sizes:** number of 1’s must be a power of 2.

- Thus, only \(O(\log \log N)\) bits are required for this count.
Either one or two buckets with the same power-of-2 number of 1’s.

- Buckets do not overlap.
- Buckets are sorted by size.
  - Older buckets are not smaller than newer 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.

---

2 of size 8

2 of size 4

1 of size 2

2 of size 1

---

N
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.

If the current bit is 0, no other changes are needed.
Updating Buckets: Input = 1

- 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: Managing Buckets

Initial

1 arrives; makes third block of size 1.

Combine oldest two 1’s into a 2.

Later, 1, 0, 1 arrive. Now we have 3 1’s again.

Combine two 1’s into a 2.

The effect ripples all the way to a 16.
To estimate the number of 1’s in the most recent $k \leq N$ bits:

1. Restrict your attention to only those buckets whose end time stamp is at most $k$ bits in the past.
2. Sum the sizes of all these buckets but the oldest.
3. Add half the size of the oldest bucket.

Remember: we don’t know how many 1’s of the last bucket are still within the window.
Suppose the oldest bucket within range has size \(2^i\).

Then by assuming \(2^{i-1}\) of its 1’s are still within the window, we make an error of at most \(2^{i-1}\).

Since there is at least one bucket of each of the sizes less than \(2^i\), and at least 1 from the oldest bucket, the true sum is no less than \(2^i\).

Thus, error at most 50%.
We can represent one bucket in $O(\log N)$ bits.

- It’s just a timestamp needing $\log N$ bits and a size, needing $\log \log N$ bits.
- No bucket can be of size greater than $N$.
- There are at most two buckets of each size 1, 2, 4, 8,...
- That’s at most $\log N$ different sizes, and at most 2 of each size, so at most $2\log N$ buckets.
Exponentially Decaying Windows

Efficient Maintenance of E.D.W.’s Application to Frequent Itemsets
Viewpoint: what is important in a stream is not just a finite window of most recent elements.

- But all elements are not equally important; “old” elements less important than recent ones.
- Pick a constant $c << 1$ and let the “value” of the $i$-th most recent element to arrive be proportional to $(1-c)^i$. 
Numerical Streams

- **Common case**: elements are numerical, with \( a_i \) arriving at time \( i \).
- The stream has a value at time \( t \): \( \sum_{i \leq t} a_i(1-c)^{t-i} \).

- **Example**: are we in a rainy period?
  - \( a_i = 1 \) if it rained on day \( i \); 0 if not.
  - \( c = 0.1 \).
  - If it rains every day, the value of the sum is \( 1+.9+(.9)^2+... = 1/c = 10. \)
  - Value will be higher if the recent days have been rainy than if it rained long ago.
Exponentially decaying windows make it easy to maintain this sum.

When a new element x arrives:
1. Multiply the previous value by 1-c.
2. Add x.
Imagine many streams, each Boolean, each representing the occurrence of one element.

**Example**: sales of items.

- One stream for each item.
  - Stream has a 1 when an instance of that item is sold.

Want the most “frequent” sets of items.

- Frequency can be represented by the “value” of the stream in the decaying-window sense.

But there are too many itemsets to maintain the value for every stream.
A-Priori-Like Approach

- Take the support threshold \( s \) to be \( 1/2 \).
  - I.e., count a set only when the value of its stream is at least \( 1/2 \).
  - *Aside*: \( s \) cannot be greater than \( 1 \), because then we could never start counting any set.
- Start by counting only the singleton items that are above threshold.
- Then, start counting a set when it occurs at time \( t \), *provided* all of its immediate subsets were already being counted (before time \( t \)).
1. Suppose set of items $S$ are all the items sold at time $t$.

2. Multiply the value for each itemset being counted by $(1-c)$.

3. Add 1 to the values for every set $T \subseteq S$, such that either:
   - $T$ is a singleton, or
   - Every immediate subset of $T$ was being counted at time $t-1$.

4. Drop any values $< 1/2$. 