Mining Data Streams

The Stream Model
Sliding Windows
Counting 1’s

The Stream Model

◆ Data enters at a rapid rate from one or more input ports.
◆ The system cannot store the entire stream.
◆ How do you make critical calculations about the stream using a limited amount of (secondary) memory?
Applications --- (1)

In general, stream processing is important for applications where
- New data arrives frequently.
- Important queries tend to ask about the most recent data, or summaries of data.

Applications --- (2)

- 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.

Applications --- (3)

- Sensors of all kinds need monitoring, especially when there are many sensors of the same type, feeding into a central controller, most of which are not sensing anything important at the moment.
- Telephone call records summarized into customer bills.
Applications --- (4)

- Intelligence-gathering.
  - Like “evil-doers visit hotels” at beginning of course, but much more data at a much faster rate.
    - Who calls whom?
    - Who accesses which Web pages?
    - Who buys what where?

Sliding Windows

- 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 still so large that it cannot be stored on disk.
  - Or, there are so many streams that windows for all cannot be stored.
Problem: given a stream of 0’s and 1’s, be prepared to answer queries of the form “how many 1’s 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.

You can’t get an exact answer without storing the entire window.

Real Problem: what if we cannot afford to store \( N \) bits?

- E.g., we are processing 1 billion streams and \( N = 1 \) billion, but we’re happy with an approximate answer.

Summarize exponentially increasing regions of the stream, looking backward.

Drop small regions when they are covered by completed larger regions.
Example

We can construct the count of the last \( N \) bits, except we're not sure how many of the last 6 are included.

What's Good?

- Stores only \( O(\log^2 N) \) bits.
- Easy update as more bits enter.
- Error in count no greater than the number of 1's in the "unknown" area.

What's Not So Good?

- As long as the 1's are fairly evenly distributed, the error due to the unknown region is small --- no more than 50%.
- But it could be that all the 1's are in the unknown area at the end.
- In that case, the error is unbounded.
Fixup

◆ Instead of summarizing fixed-length blocks, summarize blocks with specific numbers of 1’s.
  ◆ Let the block "sizes" (number of 1’s) increase exponentially.
  ◆ When there are few 1’s in the window, block sizes stay small, so errors are small.

DGIM* Method

◆ Store \(O(\log^2 N)\) bits per stream.
◆ 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.

* Datar, Gionis, Indyk, and Motwani

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 \(O(\log_2 N)\) bits.
Buckets

- A bucket in the DGIM method is a record consisting of:
  1. The timestamp of its end \(O(\log N)\) bits.
  2. The number of 1's between its beginning and end \(O(\log \log N)\) bits.
- **Constraint on buckets:** number of 1's must be a power of 2.
  - That explains the \(\log \log N\) in (2).

Representing a Stream by Buckets

- Either one or two buckets with the same power-of-2 number of 1's.
- Buckets do not overlap in timestamps.
- Buckets are sorted by size (\# of 1's).
  - Earlier buckets are not smaller than later buckets.
- Buckets disappear when their end-time is \(> N\) time units in the past.

Example

- 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
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.
- 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
Querying

- To estimate the number of 1's in the most recent $N$ bits:
  1. Sum the sizes of all buckets but the last.
  2. Add in half the size of the last bucket.
- Remember, we don’t know how many 1’s of the last bucket are still within the window.

Error Bound

- Suppose the last bucket has size $2^k$.
- Then by assuming $2^{k-1}$ of its 1’s are still within the window, we make an error of at most $2^{k-1}$.
- Since there is at least one bucket of each of the sizes less than $2^k$, the true sum is no less than $2^k-1$.
- Thus, error at most 50%.

Extensions (For Thinking)

- How to improve the precision of the result?
- Can we use the same trick to answer queries “How many 1’s in the last $k$?” where $k < N$?
- Can we handle the case where the stream is not bits, but integers, and we want the sum of the last $k$?
Application: Counting Items

- **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.

Pros and Cons

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