In many data mining situations, we 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 tuples enter at a rapid rate, at one or more input ports (i.e., streams)
- The system cannot store the entire stream accessibly
- How do you make critical calculations about the stream using a limited amount of (secondary) memory?
Ad-Hoc Queries

Processor

Standing Queries

Output

Streams Entering

Limited Working Storage

Archival Storage

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

Time
Problems on Data Streams

- Types of queries one wants on answer on a stream:
  - 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

- We will examine these two problems today
Problems on Data Streams

- Types of queries one wants on answer on a stream: (we’ll do these on Wed)
  - 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 from a Data Stream

- Since we can not store the entire stream, one obvious approach is to store a **sample**

- **Two different problems:**
  - Sample a **fixed proportion** of elements in the stream (say 1 in 10)
  - Maintain a **random sample of fixed size** over a potentially infinite stream
    - At any “time” $n$ we would like a random sample of $s$ elements
      - For all $k$, each of $n$ elements seen so far has equal prob. of being sampled
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 on two different days?
- Have space to store $\frac{1}{10}$th 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
Simple question: What fraction of queries by an average user are duplicates?

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

- Correct answer: $\frac{d}{s+d}$
- Sample will contain $s/10$ of the singleton queries and $2d/10$ of the duplicate queries at least once
- But only $d/100$ pairs of duplicates
  - $d/100 = \frac{1}{10} \times \frac{1}{10} \times d$
- Of $d$ “duplicates” $18d/100$ appear once
  - $18d/100 = (\frac{1}{10} \times \frac{9}{10}) + (\frac{9}{10} \times \frac{1}{10}) \times d$
- So the sample-based answer is: $\frac{d}{10s+19d}$
Solution: Sample Users

- Pick $1/10^{th}$ 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 size \(a/b\):**
  - Hash each tuple’s key uniformly into \(b\) buckets
  - Pick the tuple if its hash value is at most \(a\)
Problem 2: Fixed-size sample

Suppose we need to maintain a sample $S$ of size exactly $s$

- E.g., main memory size constraint

Why? Don’t know length of stream in advance

- In fact, stream could be infinite

Suppose at time $t$ we have seen $n$ items

- Ensure each item is in the sample $S$ with equal probability $s/n$
Algorithm:

- 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 $s/n$, pick 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
Proof: By Induction

- We prove this by induction:
  - Assume that after \( n \) elements, the sample contains each element seen so far with probability \( s/n \)
  - We need to show that after seeing element \( n+1 \) the sample maintains the property
    - Sample contains each element seen so far with probability \( s/(n+1) \)
  - Obviously, after we see \( n=s \) elements the sample has the wanted property
    - Each out of \( n=s \) elements is in the sample with probability \( s/s = 1 \)
Proof: By Induction

- After $n$ elements, the sample $S$ contains each element seen so far with probability $s/n$
- Now element $n+1$ arrives
- For elements already in $S$, probability of remaining in $S$ is:
  \[
  (1 - \frac{s}{n+1}) + \left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right) = \frac{n}{n+1}
  \]
  - Element $n+1$ discarded
  - Element $n+1$ not discarded
  - Element in the sample not picked
- 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
**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 so large it cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
Sliding Window: 1 Stream

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

\[ q w e r t y u i o p a s d f g h j k l z x c v b n m \]

\[ q w e r t y u i o p a s d f g h j k l z x c v b n m \]

\[ q w e r t y u i o p a s d f g h j k l z x c v b n m \]

\[ q w e r t y u i o p a s d f g h j k l z x c v b n m \]

\[ N = 6 \]
Problem:

- Given a stream of 0s and 1s
- Be prepared to answer queries of the form
  \[ \text{How many 1s are in the last } k \text{ bits?} \] 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 not 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

- How many 1s are in the last $N$ bits?
- Simple solution that does not really solve our problem: **Uniformity assumption**

Maintain 2 counters:
- $S$: number of 1s
- $Z$: number of 0s so far

How many 1s are in the last $N$ bits? $N \cdot S / (S + Z)$

But, what if stream is non-uniform?
- What if distribution changes over time?
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
Solution that doesn’t (quite) work:

- Summarize exponentially increasing regions of the stream, looking backward
- Drop small regions if they begin at the same point as a larger region

We can construct the count of the last $N$ bits, except we are not sure how many of the last 6 are included.
What’s Good?

- Stores only $O(\log^2 N)$ bits
  - $O(\log N)$ 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
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

- 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

```
1001010110001011010101010101011010101010101110101010111010100010110010
```

\[ N \]
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 $O(\log_2 N)$ bits
DGIM: Buckets

- A *bucket* in the DGIM method is a record consisting of:
  1. The timestamp of its end \([O(\log N) \text{ bits}]\)
  2. 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 (2)
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

Properties we maintain:
- 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

At least 1 of size 16. Partially beyond window.

2 of size 16
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

- 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 ...
Example: Updating Buckets

0101010110001011010101010101011010101010101011010101010101011101010101110101000101100101
0010101100010110101010101010110101010101011101010101110101000101100101
0010101100010110101010101010110101010101011101010101110101000101100101
0101100010110101010101010110101010101011101010101110101000101100101
0101100010110101010101010110101010101011101010101110101000101100101
0101100010110101010101010110101010101011101010101110101000101100101

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

Thus, error at most 50%
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 + $\frac{1}{2}$ 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?
Reducing the Error

- Instead of maintaining 1 or 2 of each size bucket, we allow either \( r - 1 \) or \( r \) for \( r > 2 \)
  - Except for the largest size buckets; we can have any number between 1 and \( r \) of those

- Error is at most \( 1/(r) \)

- By picking \( r \) appropriately, we can tradeoff between number of bits and the error