Mining Data Streams

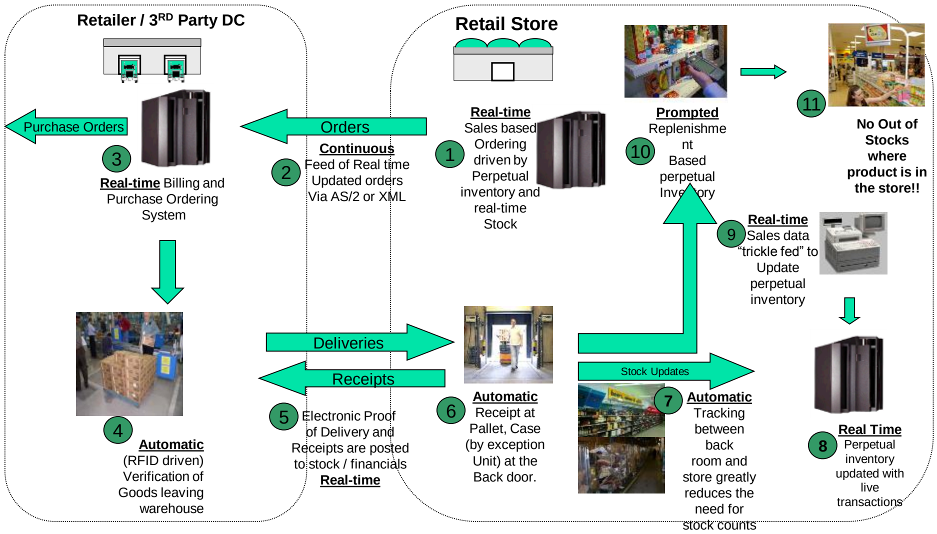
Panagiotis Karras

Aarhus U, Spring 2019

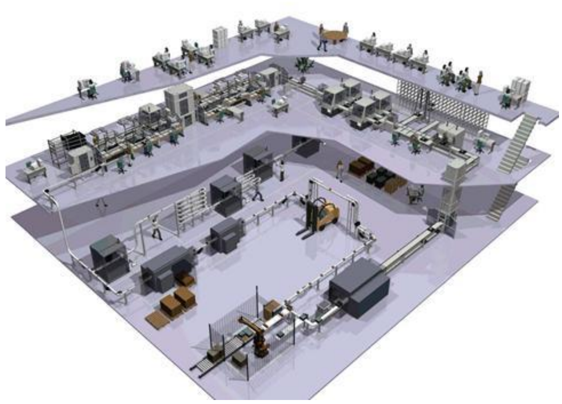
Today's Agenda

- Understand what a DSMS does.
- Queries on streams.
- Sliding Windows
- Counting on a stream:
  - The DGIM method.
  - How to derive an error bound.
- Counting distinct elements.
- Moments of a stream.

Motivating Example: Store Replenishment Process

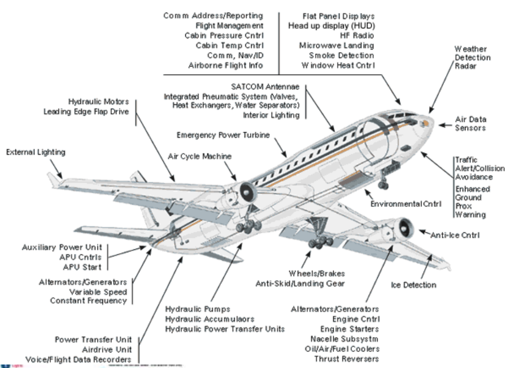


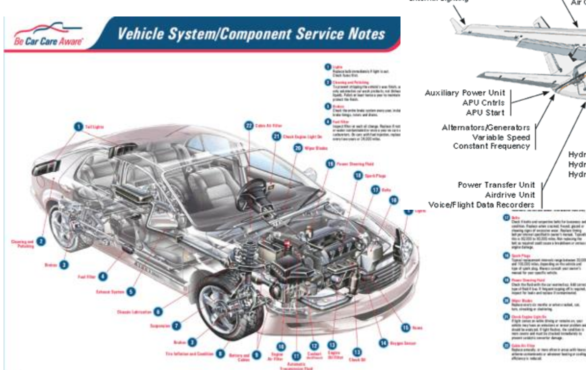
Motivating Example: Production Control System



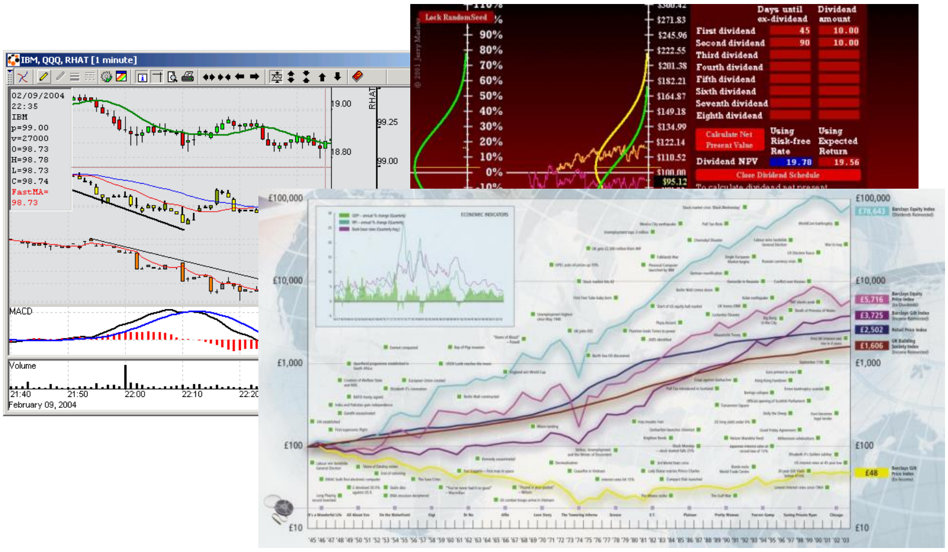
Motivating Example: Production Control System

Motivating Examples: Monitoring Vehicle Operation





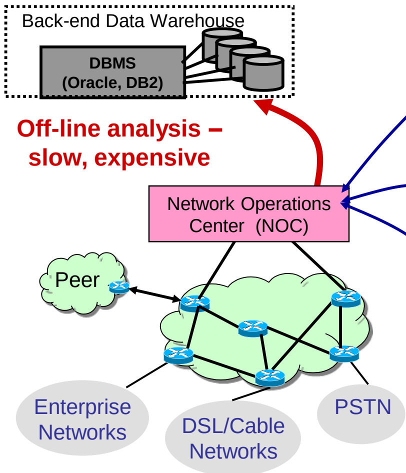
Motivating Example: Financial Applications



Motivating Examples: Web Data Streams

- 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.

Motivating Examples: Network Monitoring



What are the top (most frequent) 1000 (source, dest) pairs seen over the last month?

How many distinct (source, dest) pairs have been seen by both R1 and R2 but not R3?

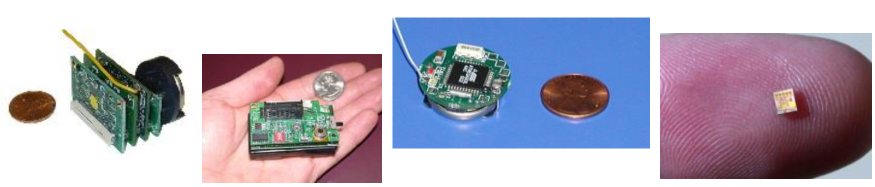
Set-Expression Query

```
SELECT COUNT (R1.source, R2.dest)
FROM  R1, R2
WHERE R1.dest = R2.source
```

SQL Join Query

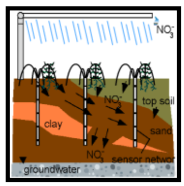
Motivating Examples: Sensor Networks

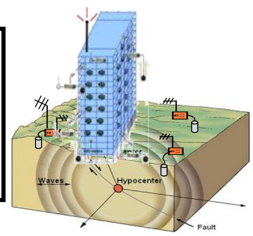
- the sensors era
  - ubiquitous, small, inexpensive sensors
  - bridging physical world and information technology



Motivating Examples: Sensor Networks

- the sensors era
  - ubiquitous, small, inexpensive sensors
  - bridging physical world to information technology
- unveil previously unobservable phenomena



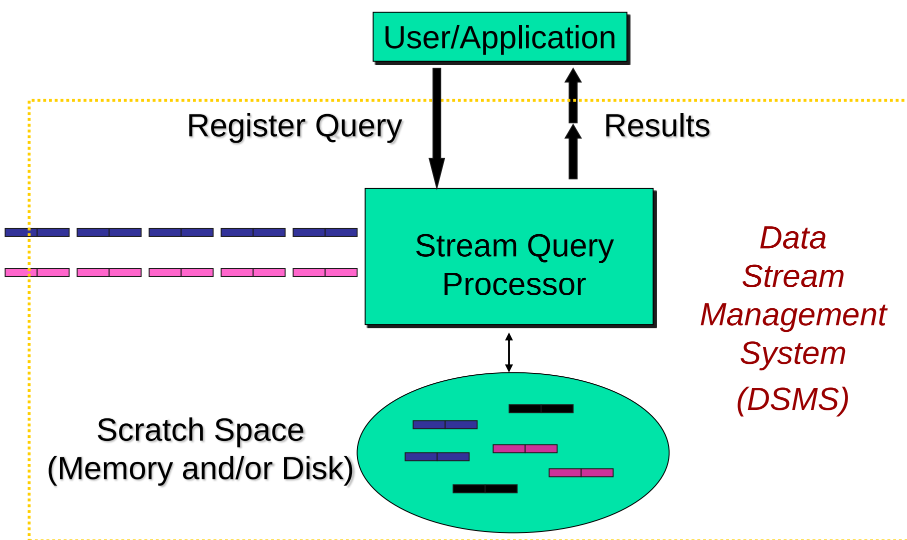
What all this requires

- Efficient streaming algorithms
  - process this data online
  - allow approximate answers
  - operate in a distributed fashion (network as distributed database)
  - also usable as one-pass algorithms for massive datasets
- new data mining algorithms
  - help in data analysis in the above setting

Data Stream Management System?

- Traditional DBMS – data stored in finite, persistent data sets
- New Applications – data input as continuous, ordered data streams
  - Network monitoring and traffic engineering
  - Telecom call records
  - Network security
  - Financial applications
  - Sensor networks
  - Manufacturing processes
  - Web logs and clickstreams
  - Massive data sets

Data Stream Management System!

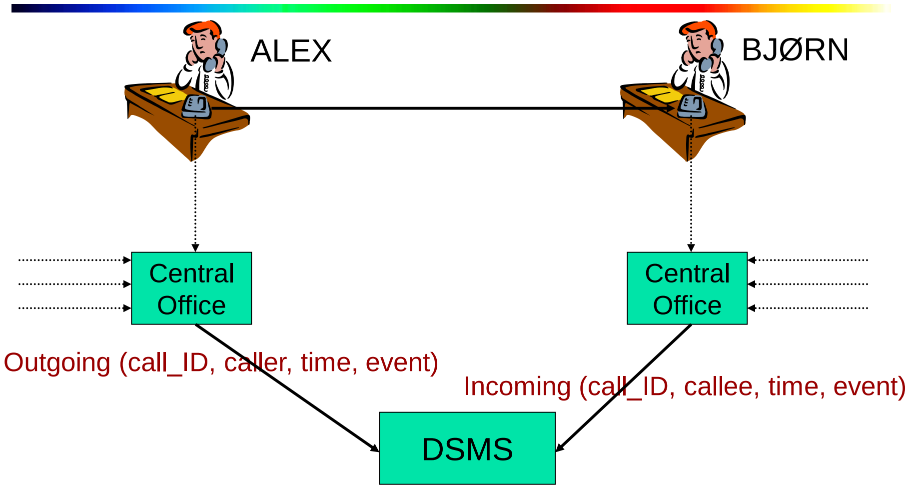


DBMS versus DSMS

- Persistent relations
- One-time queries
- Random access
- "Unbounded" disk store
- Only current state matters
- Passive repository
- Relatively low update rate
- No real-time services
- Precise answers
- Access plan determined by query processor, physical DB design

- Transient streams
- Continuous queries
- Sequential access
- Bounded main memory
- History/arrival-order is critical
- Active stores
- Possibly multi-GB arrival rate
- Real-time requirements
- Imprecise/approximate answers
- Access plan dependent on variable data arrival and data characteristics

Making Things Concrete



event = START or END

Query 1 (SELF-JOIN)

- Find all outgoing calls longer than 2 minutes

```
SELECT   O1.call_ID, O1.caller
FROM     Outgoing O1, Outgoing O2
WHERE    (O2.time – O1.time > 2
             AND   O1.call_ID = O2.call_ID
             AND   O1.event = START
             AND   O2.event = END)
```

- Result requires unbounded storage
- Can provide result as data stream
- Can output after 2 min, without seeing END

Query 2 (JOIN)

- Pair up callers and callees

```
SELECT   O.caller, I.callee
FROM     Outgoing O, Incoming I
WHERE    O.call_ID = I.call_ID
```

- Can still provide result as data stream
- Requires unbounded temporary storage …
- … unless streams are near-synchronized

Query 3 (group-by aggregation)

- **Total** connection time for each caller

```
SELECT      O1.caller, sum(O2.time – O1.time)
FROM        Outgoing O1, Outgoing O2
WHERE       (O1.call_ID = O2.call_ID
              AND O1.event = START
              AND  O2.event = END)
GROUP BY    O1.caller
```

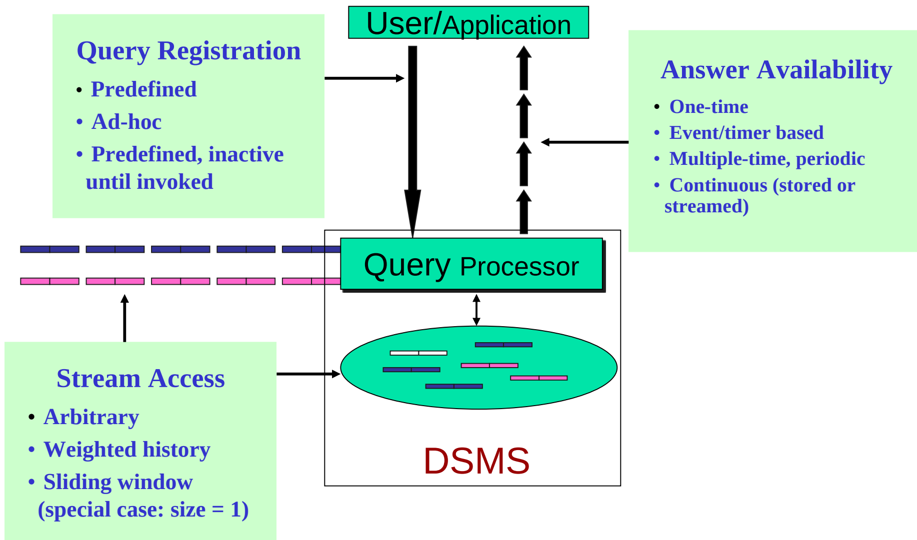
- Cannot provide result in (append-only) stream
  - Output updates?
  - Provide current value on demand?
  - Memory?

Data Model

- Append-only
  - Call records
- Updates
  - Stock tickers
- Deletes
  - Transactional data
- Meta-Data
  - Control signals, punctuations

System Internals – probably need all of the above

Query Model



Related Database Technology

- DSMS must use ideas, but none is substitute
  - Triggers, Materialized Views in Conventional DBMS
  - Main-Memory Databases
  - Distributed Databases
  - Pub/Sub Systems
  - Active Databases
  - Sequence/Temporal/Timeseries Databases
  - Realtime Databases
  - Adaptive, Online, Partial Results
- Novelty in DSMS
  - Semantics: input ordering, streaming output, …
  - State: cannot store unending streams, yet need history
  - Performance: rate, variability, imprecision, …

Unrestricted Window

- Queries refer to all data in a *window* that starts at the "beginning of time", extends up to the current time, and expands with time (potentially infinite length).

Unrestricted Window

$\boxed{q}$ 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

← Past Future →

Unrestricted Window

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

← Past Future →

Unrestricted Window

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

← Past Future →

Unrestricted Window

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

← Past Future →

Unrestricted Window

- Queries refer to all data in a *window* that starts at the "beginning of time", extends up to the current time, and expands with time (potentially infinite length).
- What happens when we try to compute joins in this model?
  - Join results involving some piece of data may appear at any time in the future
  - In order to correctly compute the result, we need to store all values that have appeared in the past!

Shifting Window

- Queries are about a *window* of length *N*, which advances by *N,* where *N* are the most recent elements received, or the most recent time units.

Shifting Window

q w e r t $\boxed{\text{y u i o p}}$ a s d f g h j k l z x c v b n m

← Past Future →

Shifting Window

q w e r t $\boxed{\text{y u i o p}}$ 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 $\boxed{\text{a s d f g}}$ h j k l z x c v b n m

← Past Future →

Shifting Window

q w e r t $\boxed{\text{y u i o p}}$ 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 $\boxed{\text{a s d f g}}$ 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 $\boxed{\text{h j k l z}}$ x c v b n m

← Past Future →

Shifting Window

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

← Past   Future →

Shifting Window

- Queries are about a *window* of length *N*, which advances by *N*, where *N* are the most recent elements received, or the most recent time units.
- Useful queries within this model:
  - average number of calls every day
  - std deviation of packet losses every 10 minutes
  - etc.

Sliding Window

- Queries are about a *window* of length *N*, covering the *N* most recent elements received, or most recent time units.

Sliding Window

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

← Past Future →

Sliding Window

- Queries are about a *window* of length N, covering the N most recent elements received, or most recent time units.
- Interesting case: N is so large it cannot be stored in memory, or even on disk.
  - Or, there are so many streams that we cannot store the values for all windows.

Counting Bits --- (1)

- 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.

Counting Bits --- (2)

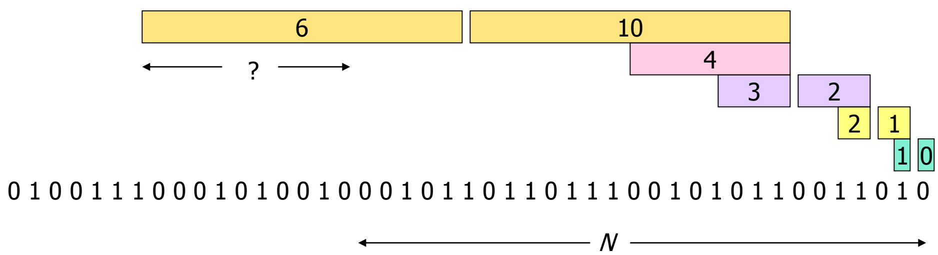
- You can't get an exact answer without storing the entire window.
- Real Problem: what if we cannot afford to store N bits?
  - E.g., we are processing 1 trillion streams and N = 1 trillion, but we're happy with an approximate answer.

Something 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 (included) region.

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.
  - $O(\log N)$ counts of $\log_2 N$ bits each.
- Easy update as more bits enter.
- Error in count no greater than the number of 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 (relative) error is unbounded! (we may miss them all and get 0 as an answer).

Fixup

- Instead of summarizing fixed-**length** blocks, summarize blocks with a fixed **number** of 1's.
  - Let the block "sizes" (number of 1's) increase exponentially.
- By controlling the number of 1's in the window, we ensure errors are small.

The 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.

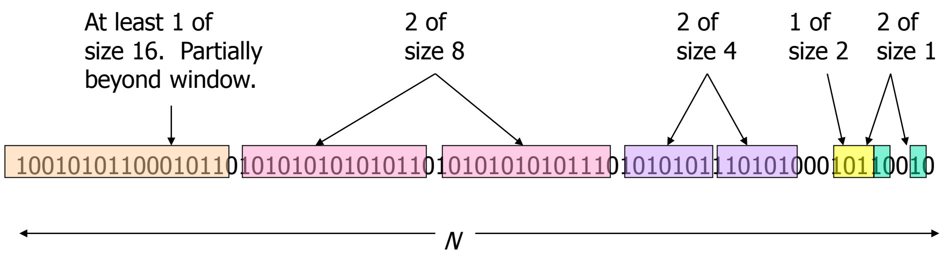
Buckets

- 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: number of 1's must be a power of 2.
  - That explains the log log N in (2). (**how?**)
  - It suffices to store its log only.

Representing a Stream by Buckets

- 1 or 2 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



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

100101011000101101010101010101101010101010111010101011101010001011100 10

Example

10010101100010110101010101010110101010101011110101010111010100010110010

00101011000101101010101010101101010101010111101010101110101000101100101

Example

10010101100010110101010101010110101010101011110101010111010100010110010

00101011000101101010101010101101010101010111101010101110101000101100101

Example

1001010110001011010101010101011010101010101110101010111010100010110010

0010101100010110101010101010110101010101011101010101110101000101100101

Example

1001010110001011010101010101011010101010101110101010111010100010110010

0010101100010110101010101010110101010101011101010101110101000101100101

0010101100010110101010101010110101010101011101010101110101000101100101

Example

1001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0 1

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1

010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1 1 0 1

Example

1001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0 1

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1

010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1 1 0 1

010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 11 0 1

Example

10010101100010110 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0

001010110001011 01 10101010101011 01 1010101010111 01 010101 1 110101 0 00 101 1 00 1 01

001010110001011 01 10101010101011 01 1010101010111 01 010101 1 110101 0 00 101 1 001 01

010110001011 01 10101010101011 01 1010101010111 01 010101 1 110101 000 101 1 001 0 1 1 01

010110001011 01 10101010101011 01 1010101010111 01 010101 1 110101 000 101 1 001 0 1 1 01

Example

10010101100010110 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0

001010110001011 01 10101010101011 01 1010101010111 01 010101 1 110101 0 00 101 1 00 1 01

001010110001011 01 10101010101011 01 1010101010111 01 010101 1 110101 0 00 101 1 001 01

010110001011 01 10101010101011 01 1010101010111 01 010101 1 110101 000 101 1 001 0 1 1 01

010110001011 01 10101010101011 01 1010101010111 01 010101 1 110101 000 101 1 001 0 1 1 01

Example

1001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0 1

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1

010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1 1 0 1

010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 11 0 1

Example

1001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0 1

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1

010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1 1 0 1

010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 11 0 1

010110001011 0 1010101010101101010101010111 0 1010101110101 000 1011001 0 11 0 1

Answering Queries

- 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, when we assume 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 for each size from 1 to 2^{k-1}, the true sum is **no less** than 2^k-1.
- Thus, the relative error is at most 50%.

Counting Distinct Elements

- Problem: a data stream consists 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.

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 different Web pages does each customer request in a week?

Using Small Storage

- Real Problem: what if we do not have space to store the complete set?
- Estimate the count in an unbiased way.
- Accept that the count may be in error, but limit the probability that the error is large.

Flajolet-Martin* Approach

- Pick a hash function h that maps each of the n elements to at least $\log_2 n$ bits.
- For each stream element a, let $r(a)$ be the number of trailing 0's in $h(a)$.
- Record R = the maximum $r(a)$ seen.
- Estimate = 2^R.

* Really based on a variant due to AMS (Alon, Matias, and Szegedy)

Why It Works

- The probability that a given $h(a)$ ends in **at least** r 0's is 2^{-r}.
- If there are m different elements, the probability that $R \geq r$ is:

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

Prob. all h(a)'s end in **fewer** than r 0's.

Prob. a given h(a) ends in **fewer** than r 0's.

Why It Works – (2)

- Since 2^{-r} is small, $1 - (1-2^{-r})^m \approx 1 - e^{-m2^{-r}}$.
- If $2^r >> m$, $1 - (1 - 2^{-r})^m \approx 1 - (1 - m2^{-r}) \approx m/2^r \approx 0$.

  First 2 terms of the Taylor expansion of e^x
- If $2^r << m$, $1 - (1 - 2^{-r})^m \approx 1 - e^{-m2^{-r}} \approx 1$.
- Thus, 2^R will almost always be around m.

Why It Does not Work

- $E(2^R)$ is actually infinite.
  - When R becomes $R+1$, probability halves, but value doubles.
- Workaround: use many hash functions, get many samples.
- How are samples combined?
  - Average? What if one very large value?
  - Median? All values are a power of 2.

Solution

- Partition your samples into small groups.
- Take the **average** of groups.
- Then take the **median** of the averages.

Generalization: Moments

- Suppose a stream has elements chosen from a set of n values.
- Let m_i be the number of times value i occurs.
- The k th *moment* is the sum of $(m_i)^k$ over all i.

Special Cases

- 0th moment = number of different elements in the stream.
  - The problem just considered.
- 1st moment = count of the numbers of elements = length of the stream.
  - Easy to compute.
- 2nd moment = *surprise number* = a measure of how uneven the distribution is.

Example: Surprise Number

- Stream of length 100; 11 values appear.
- Unsurprising: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9. Surprise # = 910.
- Surprising: 90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1. Surprise # = 8,110.

Thanks for slides to:

- Themis Palpanas
- Minos Garofalakis
- Divesh Srivastava
- Nick Koudas
- Jiawei Han
- Jeffrey Ullman
- Anand Rajaraman