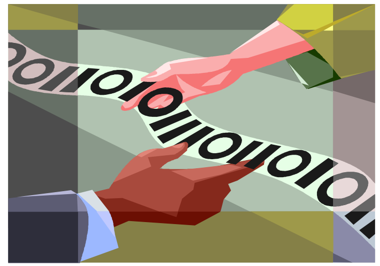
Mining Streaming Data

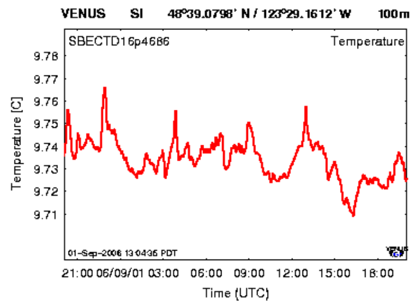
Goal

To understand the **importance** of mining methods for streaming data, and learn about existing **methods handling** streaming data

Characteristics/Description



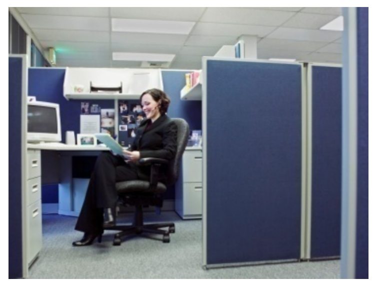
- Stream data sets are…
  - Continuous
  - Massive
  - Unbounded
  - Possibly infinite



- Fast changing and requires fast, real-time response
  - Examples:

    Life threatening: collision avoidance

    Lost revenue/transactions: hung-up networks



Sizing the challenge

- WalMart Records >20 Million Transactions
- Google Handles >100 Million Searches
- AT&T produces >275 million call records
- Earth sensing satellite produces GBs of data
- Twitter handles millions of tweets in a second

This just in a day!

The Stream Model

- Input tuples enter at a rapid rate, at one or more input ports.
- Impractical to store the whole streaming data
- How do you make critical calculations about the stream using a limited amount of (primary or secondary) memory?
- Simple calculation per data due to time and space constraints

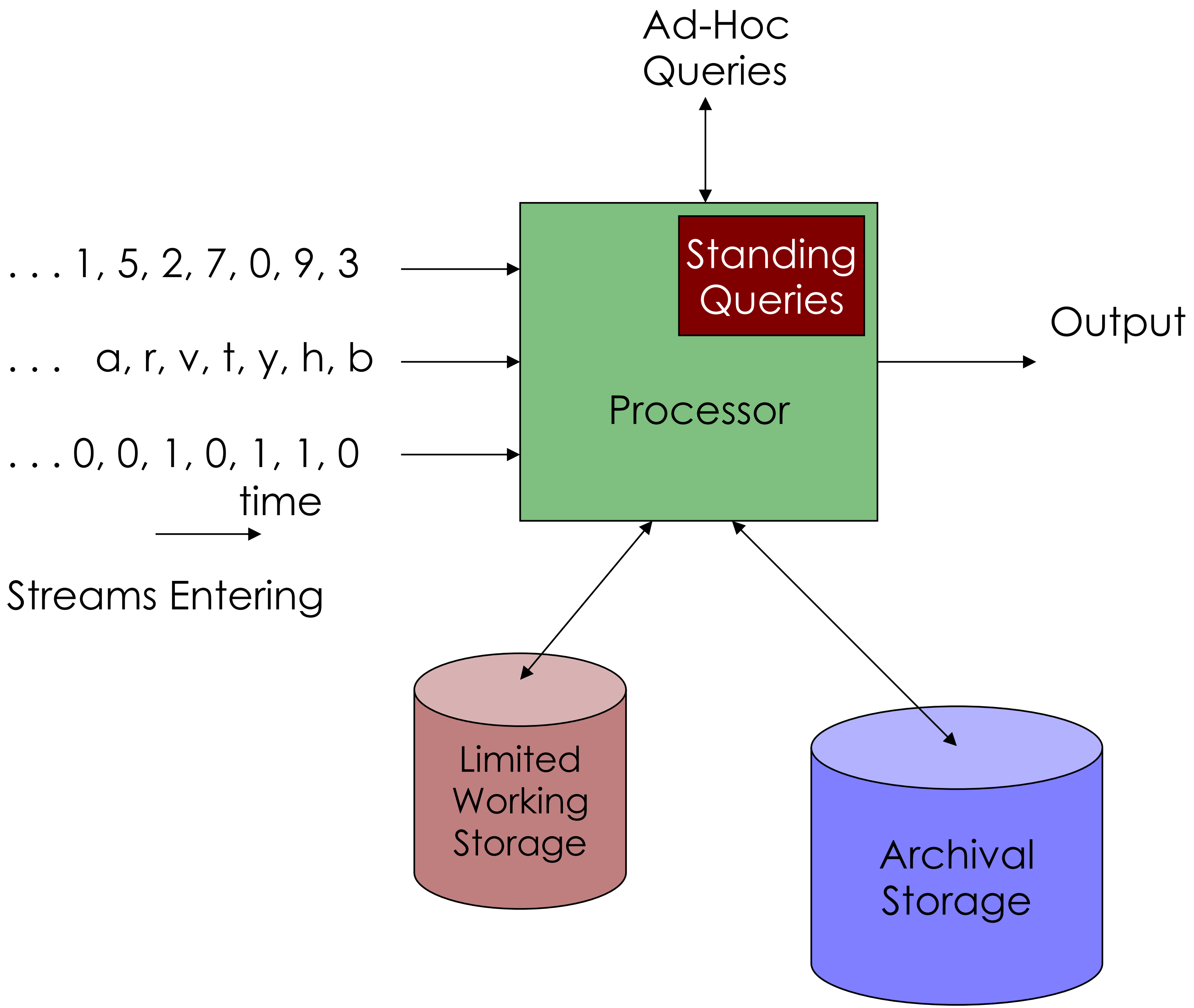
Time/Space constrained

- Not enough memory
- Can't afford storing/revisiting the data
  - Single pass computation
- External memory algorithms for handling data sets larger than main memory cannot be used.
  - Do not support continuous queries
  - Too slow real-time response

Two Forms of Query

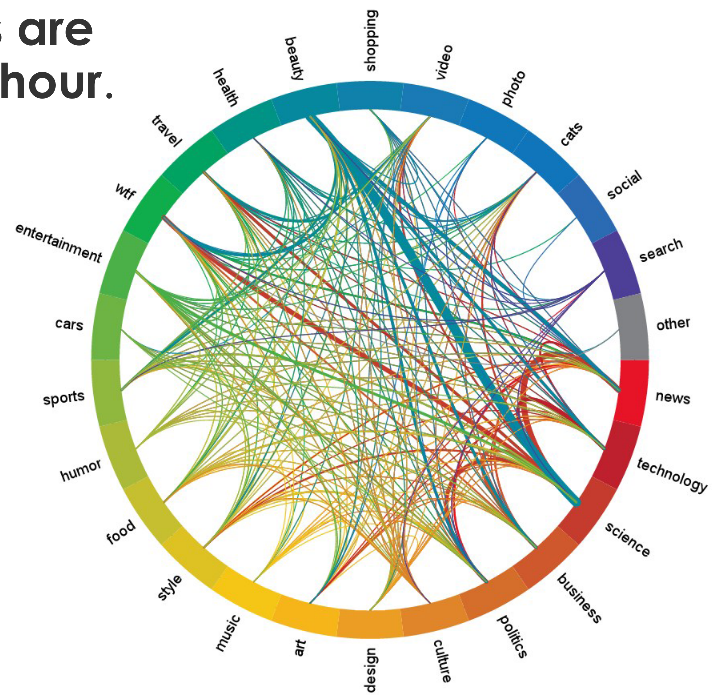
1. ***Ad-hoc queries***: Normal queries asked one time about streams.
   - Example: What is the maximum value seen so far in stream *S*?

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*.


Ad-Hoc Queries
Standing Queries
Processor
Output
. . . 1, 5, 2, 7, 0, 9, 3
. . . a, r, v, t, y, h, b
. . . 0, 0, 1, 0, 1, 1, 0
time
Streams Entering
Limited Working Storage
Archival Storage

Example Applications

- Mining query streams.
  - E.g.: Google wants to know what queries are **more frequent today than yesterday**.
- Mining click streams.
  - E.g.: 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.
  - E.g.:
    - Gather information for optimal routing.
    - Detect denial-of-service attacks.



Sliding Windows

- A useful model of stream processing
  - 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 too large*** to be stored in main memory
  - Or, ***too many streams*** that windows for all do not fit in main memory.

Sliding Windows

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 →

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.

Question:

- What do we do now for the next input *i*?

Example: Averages

Answer:

- 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

- Can show that an exact sum or count of the elements in a window:
  - Cannot use less space than the window itself.
- If willing to accept an ***approximation***:
  - Can use much less space.
- Consider the simple case of counting bits
- Sums are a fairly straightforward extension.

Counting Bits

- **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 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:
  - **D**atar, **G**ionis, **I**ndyk, and **M**otwani.
- Store only $O(\log_2 N)$ bits per stream.
  - N = 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.

Timestamps

- Each bit in the stream has a *timestamp*, starting 0, 1, …
- Record timestamps modulo N (the window size), so we can represent any *relevant* timestamp in $O(\log_2 N)$ bits.

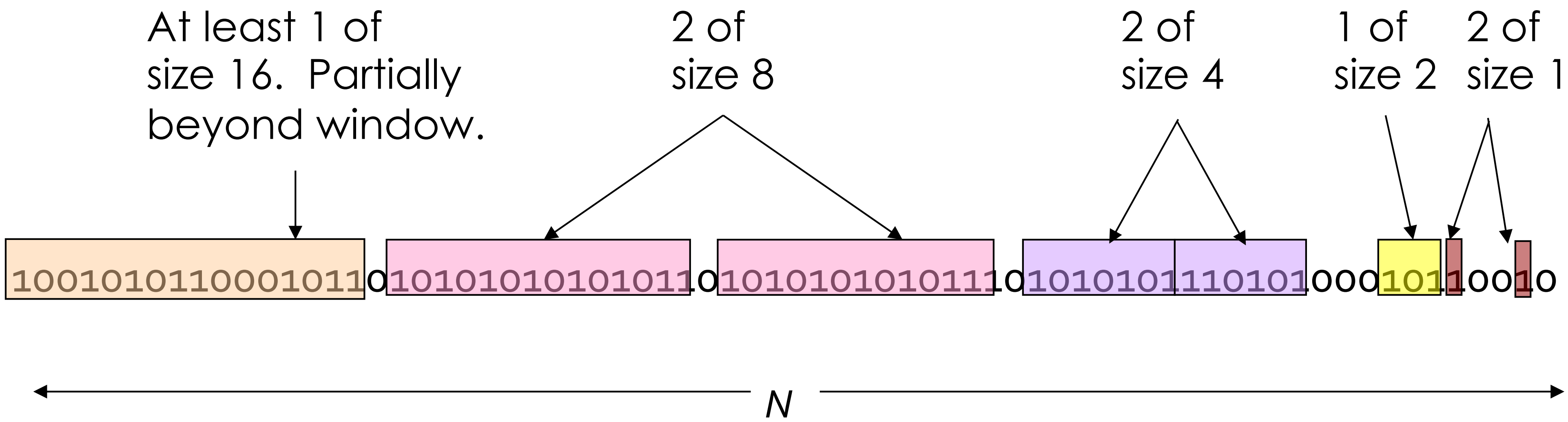
Buckets

- A segment of the window, represented by a record consisting of:
  - The **timestamp** of its end [O(log N) bits].
  - 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.

Representing a Stream by Buckets

- Either one or two buckets with the same power-of-2 number of 1's.
- Buckets do not overlap.
- Buckets sorted by size.
  - Older buckets not smaller than newer buckets.
- Buckets disappear when their end-time is > N time units in the past.

Example: Bucketized Stream



Updating Buckets

- New bit 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:
  - Create a new bucket of size 1, for just this bit.
    - End timestamp = current time.
  - If there are now three buckets of size 1, combine the oldest two into a bucket of size 2.
  - If there are now three buckets of size 2, combine the oldest two into a bucket of size 4.
  - And so on …

Example: Managing Buckets

Initial

10010101100010110101010101010101101010101010111010101011101010001011 0010

1 arrives; makes third block of size 1.

001010110001011010101010101010110101010101011101010101110101000101100101

Combine oldest two 1's into a 2.

001010110001011010101010101010110101010101011101010101110101000101100101

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

0101100010110101010101010101101010101010111010101011101010001011001011101

Combine two 1's into a 2.

0101100010110101010101010101101010101010111010101011101010001011001011101

The effect ripples all the way to a 16.

0101100010110101010101010101101010101010111010101011101010001011001011101

Querying

- To estimate the number of 1's in the most recent $k < N$ bits:
  - Restrict your attention to only those buckets whose end time stamp is at most k bits in the past.
  - Sum the sizes of all these buckets but the oldest.
  - 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.

Error Bound

- Suppose the oldest bucket within range has size *2i*.
- Then by assuming *2i -1* of its 1's are still within the window, we make an error of at most *2i -1*.
- Since there is at least one bucket of each of the sizes less than 2i, and at least 1 from the oldest bucket, the true sum is no less than 2i.
- Thus, error at most 50%.

Space Requirements

- 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 *2log N* buckets.

- Bloom Filters
- Sampling Streams
- Counting Distinct Items

Filtering Stream Content

- To motivate the **Bloom-filter** idea, consider a web crawler.
- It keeps, centrally, a list of all the URL's it has found so far.
- It assigns these URL's to any of a number of parallel tasks
  - these tasks stream back the URL's they find in the links they discover on a page.
- It needs to filter out those URL's it has seen before.

Role of the Bloom Filter

- A Bloom filter placed on the stream of URL's will declare that **certain URL's have been seen before**.
- Others will be declared new, and will be added to the list of URL's that need to be crawled.
- Unfortunately, the Bloom filter can have false positives.
  - It can declare a **URL seen before when it hasn't**.
- But if it says "never seen," then it is truly new.
- Solution: **restart the filter periodically**.

Example: Filtering Chunks

- Suppose we have a database relation stored in a DFS, spread over many chunks.
- We want to find a particular value v, looking at as few chunks as possible.
- A Bloom filter on each chunk will tell us certain values are there, and others aren't.
  - As before, **false positives possible**
- But now things are exactly right: if the filter says v is not at the chunk, it surely isn't.
  - Occasionally, we retrieve a chunk we don't need, but can't miss an occurrence of value v.

How a Bloom Filter Works

- A *Bloom filter* is an array of bits, together with a number of hash functions.
- The argument of each hash function is a stream element, and it returns a position in the array.
- Initially, all bits are 0.
- When input x arrives, we **set to 1 the bits *h(x)*, for each hash function *h***.

Example: Bloom Filter

- Use $N = 11$ bits for our filter.
- Stream elements = integers.
- Use two hash functions:
  - $h1(x)$ =
    - Take **odd-numbered bits from the right** in the binary representation of x.
    - Treat it as an integer i.
    - Result is i modulo 11.
  - $h2(x)$ = same, but **take even-numbered bits**.

Example – Continued

Stream element	h_1	h_2	Filter contents
			00000000000
25 = 11001	5	2	00100100000
159 = 10011111	7	0	10100101000
585 = 1001001001	9	7	10100101010

Note: bit 7 was already 1.

Bloom Filter Lookup

- Suppose element *y* appears in the stream, and we want to know if we have seen y before.
- Compute *h(y)* for each hash function *y*.
- If all the resulting bit positions are 1, say we have seen *y* before.
  - We could be wrong.
    - Different inputs could have set each of these bits.
- If at least one of these positions is 0, say we have not seen *y* before.
  - We are certainly right.

Example: Lookup

- Suppose we have the same Bloom filter as before, and we have set the filter to 10100101010.
- Lookup element y = 118 = 1110110 (binary).
- $h_1(y)$ = 14 modulo 11 = 3.
- $h_2(y)$ = 5 modulo 11 = 5.
- Bit 5 is 1, but bit 3 is 0, so we are sure y is not in the set.

Performance of Bloom Filters

- Probability of a false positive depends on the density of 1's in the array and the number of hash functions.
  - = (fraction of 1's)$^{\text{\# of hash functions}}$.
- The number of 1's is approximately the number of elements inserted times the number of hash functions.
  - But collisions lower that number slightly.

Sampling a Stream

When Sampling Doesn't Work

- Suppose Google would like to examine its stream of **search queries** for the past month to find out what fraction of them were **unique** – asked only once.
- But to save time, we are only going to sample 1/10th of the stream.
- The fraction of unique queries in the sample != the fraction for the stream as a whole.
  - In fact, we can't even adjust the sample's fraction to give the correct answer.

Example: Unique Search Queries

- The length of the sample is 10% of the length of the whole stream.
- Suppose a query is unique.
  - It has a 10% chance of being in the sample.
- Suppose a query occurs exactly twice in the stream.
  - It has an 18% chance of appearing exactly once in the sample.
- And so on … The fraction of unique queries in the stream is unpredictably large.

Sampling by Value

- My mistake: I sampled based on the position in the stream, rather than the value of the stream element.
- The right way: hash search queries to 10 buckets 0, 1,…, 9.
- Sample = all search queries that **hash to bucket 0**.
  - All or none of the instances of a query are selected.
  - Therefore the fraction of unique queries in the sample is the same as for the stream as a whole.

Controlling the Sample Size

- Problem: What if the total sample size is limited?
- Solution: Hash to a large number of buckets.
- Adjust the set of buckets accepted for the sample, so your sample size stays within bounds.

Example: Fixed Sample Size

- Suppose we start our search-query sample at 10%, but we want to limit the size.
- Hash to (say) 100 buckets, 0, 1,…, 99.
  - Take for the sample those elements hashing to buckets 0 through 9.
- If the sample gets too big, get rid of bucket 9.
- Still too big, get rid of 8, and so on.

Sampling Key-Value Pairs

- This technique generalizes to any form of data that we can see as tuples (K, V), where K is the "key" and V is a "value."
- Distinction: We want our sample to be based on picking some set of keys only, not pairs.
  - In the search-query example, the data was "all key."
- Hash keys to some number of buckets.
- Sample consists of all key-value pairs with a key that goes into one of the selected buckets.

Example: Salary Ranges

- Data = tuples of the form (EmpID, Dept, Salary).
- Query: What is the average range of salaries within departments?
- Key = Dept.
- Value = (EmpID, Salary).
- Sample picks some departments, has salaries for all employees of that department, including its min and max salaries.
- Result will be an unbiased estimate of the average salary range.

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.

Application Examples

- 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 **unique users** visited Facebook this month?
- How many **different pages** link to each of the pages we have crawled.
  - Useful for estimating the PageRank of these pages.
    - Which in turn can tell a crawler which pages are most worth visiting.

Estimating Counts

- Real Problem: what if we do not have space to store the complete set?
  - Or we are trying to count lots of sets at the same time.
- 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

- A probabilistic counting algorithm
- Used to estimate number of **distinct** elements in a large file originally
- Use little memory
- Single pass only
- Based on statistical observation made on bits of hashed values

Flajolet-Martin Approach (2)

Estimating the cardinality of a multiset *M*:

for *i* :=0 **to** *N* - 1 **do** *BITMAP[i] :=0;*

for all x in *M* do

begin

index := ***p****(hash*(x));

if *BITMAP[index] = 0* **then** *BITMAP[index]* := 1;

end;

R := the largest *index* in *BITMAP* whose value equals to 1

Estimate* := 2*R

Flajolet-Martin Approach (3)

- If the final BITMAP looks like this: 0000,0000,1100,1111,1111,1111
- The left most 1 at position 15
- Thus, have around 2^{15} distinct elements in the stream