

Data Stream Methods

Graham Cormode

graham@dimacs.rutgers.edu

S. Muthukrishnan

muthu@cs.rutgers.edu

Plan of attack

- **Frequent Items / Heavy Hitters**
- Counting Distinct Elements
- Clustering items in Streams

What are we going to do today?

Frequent items

- Classical algorithms
- Lower bounds
- New approaches for this problem and its extensions.

Some questions

- We see a large number of individual transactions (such as Amazon book sales)
 - What are the top sellers today?
- We are monitoring network traffic
 - Which hosts/subnets are responsible for most of the traffic? (data collection may be distributed)
- We have a network of satellites monitoring events over large areas
 - Which areas are experiencing the most activity over a week / day / hour?

The problem

- We will imagine that we are observing a stream of data
- This stream consists of a sequence of integers in the range $1 \dots U$ for some upper bound, U
- We will be interested in which integers occur most frequently within the stream

Example:

1, 5, 8, 9, 10, 3, 23542341234, 15, 289, 31516, 23, 8,
3571, 1251, 8, 5, 124, 289, 17, 15, 23542341234, 126,
1251, 5, ...

Who cares?

- Does anyone really care about which integers occur frequently?

Well, does anyone really care about sorting integers?

NO!

But...

- People care about sorting records indexed by integers
- People care about sorting records with keys that can be compared (names, SS-ids, Phone #s, etc.)

Will study the problem with integers, but really we are interested in IP addresses, book titles, geography, whatever

What if we don't have integers?

- We can usually map items to integers
 - (732) 445-4580 → 7,324,454,580
 - 128.6.75.111 →
 $111 + 256(75 + 256(6 + 256 * 128))$
 $= 2147896175$ (32 bits)
 - "Graham" → 6 bytes (48 bits)
- What about long names, like "Shanmugavelayutham"?
- Or addresses, words, images?
-

Hash functions

- We can find some function that maps large items to integers
- These may be exact (one-to-one), or they may map multiple items to the same number
- We would like that the chances of this are small
 - Example 1: take just first few characters of name
 - Example 2: add up the ASCII value of characters
 - Example 3: use MD5 checksum
- We will encounter this kind of function later, in more detail...

Back to Frequent Items

- We want to find which items occur most frequently, so...
- Let's set up an array, initialized to zero
- And add one to entry i when we see i in the stream
- Then find which items have the highest count

OK, we're done, let's go home...

Hold on a minute

This won't work...

If we are tracking IP addresses, then our array needs to have 2^{32} entries = 4Gb

If we are looking at pairs of IP addresses, then we need 16,384 Pb (1Pb = 2^{20} Gb)

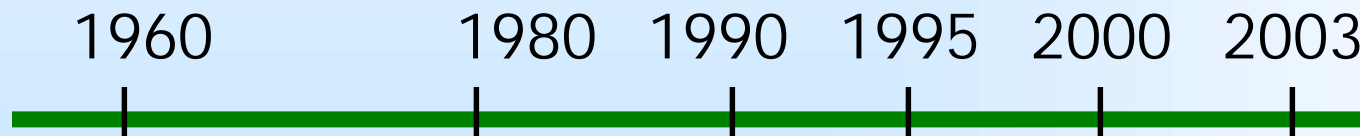
Even if we can spare the memory, scanning the array to find the frequent items will take a long time.

Eg Scanning Amazon's transactions for a day to find popular items

May be important to answer these questions quickly, with small space, even if not a pure stream scenario

Timeline of CS

- We will describe a classical algorithm for this problem



2000-2003 Recent

1995-2000 Last Century

1990-1995 Civil War Era

1980-1990 Middle Ages

1960-1980 Classical Times

Before 1960 Prehistorical?

MJRTY

- Boyer, Moore, 1982 & Fischer, Salzberg 1982
- Find a Majority Item, if one exists
 - Initialize Counter to 0
 - For each item in the stream
 - If the counter is zero, put the item in the bucket, set counter to 1
 - Else, if the next item = item in bucket, add 1 to count
 - Else subtract 1 from count

3 Count: 2

Examples of MJRTY

A Majority item is one that occurs more than half the time

Stream	1	1	2	2	2	2	3	1	1	3	1	1	3	1	1	3	1	1	1	1	3	1	
Bucket	1	1	1	1	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Counter	1	2	1	0	1	2	1	0	1	0	1	2	1	2	3	2	3	4	5	6	5	6	

What about the following stream?

1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 1 2 ...

There is no majority, but the method will return some item as a candidate

The Majority is always right

- If there is no Majority element, then the algorithm will return some item with no guarantees about its frequency.
- Put the other way, the algorithm will find any item which occurs more than half of the time.
- How can we be sure?
- We will prove the properties of the algorithm

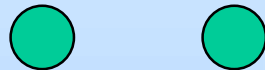
Proving the properties



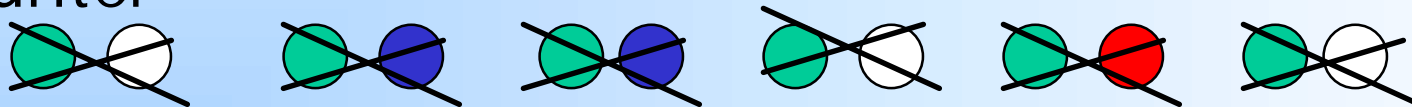
Imagine pairing up items: every non-majority item is paired up with a non-majority item.



Then there will be some majority items left over



Each pair cancels out – we add and subtract to the counter



Overall, we end must end up with a positive count for the majority item no matter how we arrange the items

Generalizing the method

The majority guarantees to find any item that occurs more than $\frac{1}{2}$ of the time

We generalize it to find any item that occurs with frequency greater than $\frac{1}{k+1}$ by using k counters and k buckets:

Frequent

For each item in the stream

- If the item is in one of the buckets, increase its count by one
- Else, if there is a bucket with count zero, put it in that bucket and set its count to 1
- Else, reduce the counts of every bucket by 1

Proving the claims

Claim: if there are k items which occur with frequency greater than $1/(k+1)$, then we will find them all

How likely is it that this will occur for typical data sets?

Can there be data sets with more than k items with frequency greater than $1/(k+1)$?

Stronger claim: every item with frequency greater than $1/(k+1)$ will be included in a bucket

HW: Prove both these claims. How long does it take to process each item in the stream? How long does it take to return the candidate frequent items?

Recent Work

- There has been a lot of activity on this problem in the last year:
 - Manku and Motwani, VLDB 2002
 - Charikar, Chen, Farach-Colton, ICALP 2002
 - Demaine, Lopez-Ortiz, Munro, ESA 2002
 - Karp, Papadimitriou, Shenker, ACM TODS 2003
 - C, Muthukrishnan, PODS 2003
 - Several others in progress...
- Papers on frequent items are a frequent item!

Extensions to the Question

There are many variations of the problem to consider:

- What if you see one stream, and I see another stream. How can we combine our observations to find items that are frequent in the union of the streams?
- What if items arrive and depart?
- What if we want to know (sub)sets that occurred together ("People who bought X also bought Y")
- What if we want to know what are the biggest changes in frequency.
 - If a_i counts the frequency of i yesterday, and b_i counts its frequency today, what is $|a_i - b_i|$? Or (a_i/b_i) ?

Pushing the Limits

Some things will not be possible to evaluate.

We should know what is not possible before we proceed.

For example, suppose our solution could tell us the count of every item from 1 to U .

Then we must use storage proportional to U .

Otherwise, we could make a fortune with a data compressor that breaks the laws of information theory...

Why?

- Suppose the memory contents of the proposed algorithm was less than U bits.
- Then we can use it to store a bitstring B with U bits in it.
- From B , compute a data stream: if bit i is set to 1 in B , then include i in the data stream (else, don't)

B: 1001001010110...

↓
1,4,7,9,11,12 →

Frequent
Items
Algorithm

→ ?

Storing a bitstring of U bits requires U bits of memory...

Stronger Bounds

We will use a problem from communication complexity to help us prove some stronger bounds

Disjointness

Suppose **Alice** and **Bob** both have bitstrings of length U , and they want to collaborate to decide whether there is some location i where both bitstrings are 1.

Computing the answer requires an amount of communication linear in U , even if we are allowed to use randomization, and get the answer wrong with some constant probability p .

Using Disjointness

- We can use the hardness of the disjointness problem in a smart way to show that certain frequent items problems will be hard
- This is the same kind of idea as with reductions between NP-Complete problems
- We will take an instance of disjointness, and show how an algorithm to find the k items with highest frequency would solve disjointness.
- From this, we can conclude that no such method can exist without using space linear in U .

Transforming Disjointness

Instance of disjointness

Alice has $a[1..U]$, Bob has $b[1..U]$ where $a[i], b[i] = 0$ or 1 for all i

We want to know if $a \cdot b = 0$ or > 0

Transformation

Alice creates a stream of the locations where a is not zero and passes it to the algorithm

She sends every index **twice**

Then she sends

$U+1, U+1, U+1, U+2, U+2, U+2 \dots U+k, U+k, U+k$

She then communicates the whole contents of the working space of the algorithm to Bob...

Transforming Disjointness 2

Bob then does the same for his bitstring: he creates a stream of indexes where the bitstring is non-zero, and passes it to the algorithm. Then he asks for the frequent items.

Claim: if the algorithm outputs $U+1 \dots U+k$ as the frequent items, then the bitstrings are disjoint

If the algorithm outputs any other value, then the bitstrings are not disjoint.

Why? Because if the strings are not disjoint, then some index occurs in both a and b , so it will appear four times in the stream. Otherwise, any index will appear at most twice.

The items $U+1 \dots U+k$ appear exactly three times each.

Conclusions about the bounds

This means we cannot hope to find the top k items (even just the most frequent item) without using a lot of space.

But, in many common cases, the most frequent items are a lot more frequent than the others (here, the difference in counts were relatively small)

The approach of finding items that occur more than some fraction, ϵ , of the time seems a better one.

Eg, Find all data flows that are consuming more than 1% of the total bandwidth

Disjointness and Compressibility are two powerful ways to show what can't be done in small space

New results

Now, let's consider what happens when items arrive and depart ("Grand Central" or "Turnstile" model)

The count of an item is the number of arrivals, less the number of departures.

(are there situations where the number of departures might be higher than the number of arrivals?)

This is important when considering dynamic situations (eg people begin and end wireless phone calls, network flows start and stop, and so on).

Let's initially look at the case where we just care about a majority item.

Dynamic Majority

Will the previous approach still work? (or, can it be tweaked so that it works?)

Example. $+i$ means i arrives, $-i$ means i departs

$+1 +1 +1 +2 +3 +4 +4 +4 +4 +4 +4 +4$

4 is the majority item, until

$-4 -4 -4 -4 -4 -4 -4 -4$

Now 1 should be found as the majority item.

But how would we remember that?

Any ideas how to fix the MJRTY method to work with dynamic streams like this?

A New Approach

Let's try to find some other way of finding the majority element.

We can keep counts of groups of items.

If the count of a group is more than $\frac{1}{2}$ the total, then we know that if there is a majority item then it must be in that group

More importantly, it rules out anything not in group

How can we choose groups to allow us to pick out the majority item?

Also, we need to make sure the groups we choose have a small description (since if we store the description of the group, we want small space).

Binary Search

How would we do this with multiple passes?

Binary search – first, figure out if the majority item was in $1..U/2$ or $U/2..U$

Then recurse on the halves in subsequent passes

What if we did this all in one pass?

Naive approach: try every possible split we might want.

But there are U different splits!

Parallel Binary Search

Observation: can do several searches at the same time.
The first group is $1..U/2$. Now think about the second split.

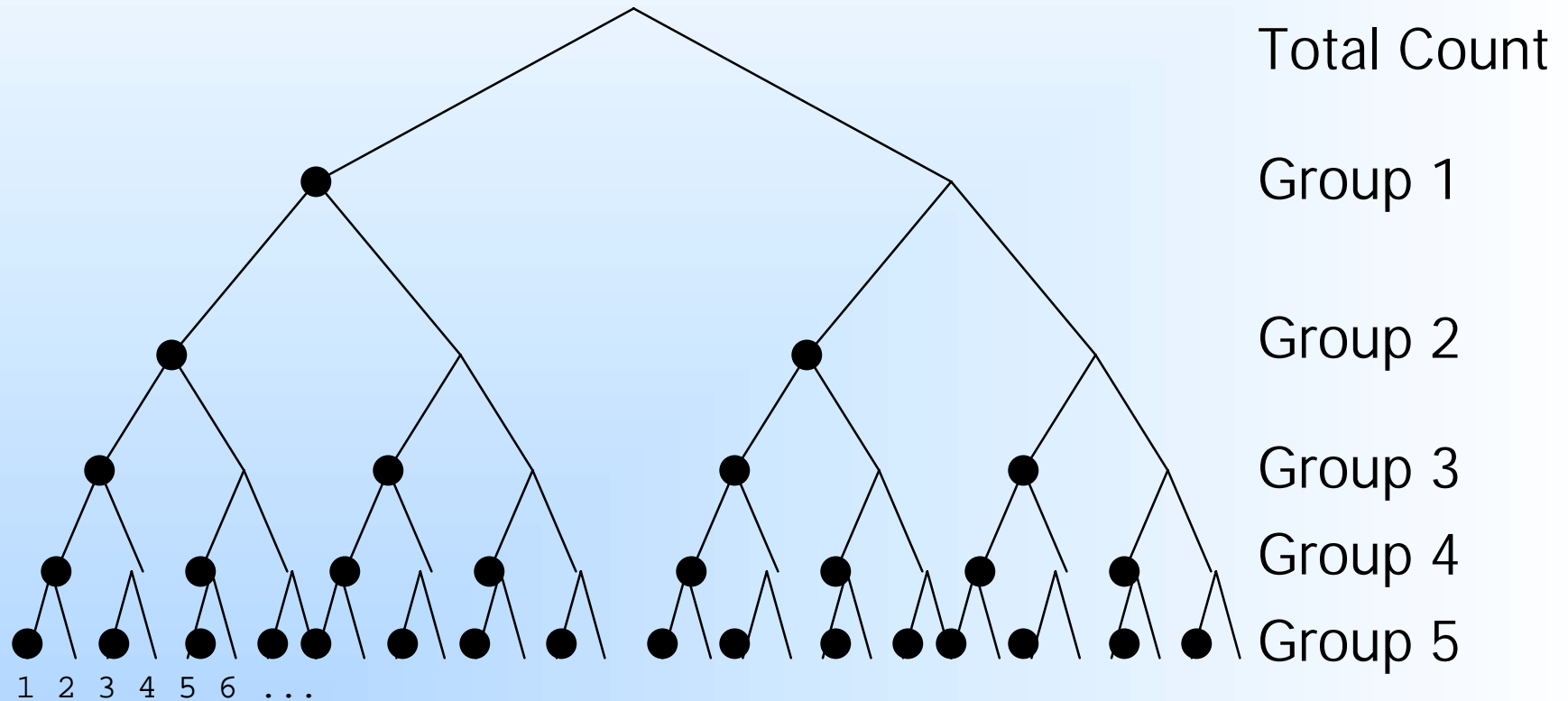
Either we want to know whether the majority is in
 $(1..U/4 \text{ or } U/4..U/2)$ or $(U/2..3U/4 \text{ or } 3U/4.. U)$

Suppose we make a group that has
 $1..U/4$ and $U/2..3U/4$

The first group tells us which half the majority lies in.
Together the second tells us which quarter the majority lies in.

We can continue splitting...

Group Structure



We require $\log_2 U$ groups in total

How do we figure out which group an item falls in?

Which Group?

Here is a neat trick* ... suppose current item is i

Write i in binary

If the j 'th bit of i is 1, then include i in group j

For example... $13 = 00001101$

So it is included in groups 1,3 and 4.

When we see an arrival of item i , add 1 to all its groups; for a departure, subtract 1

Claim: if there is a majority item, we can find it.

* Borrowed from coding theory... see Hamming codes

Proof

Consider the binary representation of the majority item.

If the j 'th bit is one, then group j will have more than half the count.

So by looking at the counts we can read off the index of the majority item.

As for the dynamic part, departures exactly cancel out with earlier arrivals, so the counts are always correct.

Double Check

Let's return to our previous example...

+1 +1 +1 +2 +3 +4 +4 +4 +4 +4 +4 +4

Total=12 Group1=4 Group2=2 Group3=7

So the majority index is $100_2 = 4$

-4 -4 -4 -4 -4 -4 -4 -4

Total=5 Group1=4 Group2=2 Group3=0

So the new majority index is $001_2 = 1$

It works!

A useful property

Suppose several people are monitoring different bunches of items to find the majority

Example: AT&T, AOL, MSN...

Can they combine their observations?

Using MJRTY, no, using the grouping method, yes...

Compute sum of totals, sum of group1, sum of group2 and so on.

The result is exactly the same as if one person had observed everything from all the different streams.

So we can distribute the monitoring process...

Group Testing

The idea is based on group testing – testing a number of items at once, to see if they contain a “positive” item.

Originated with testing batches of blood together for diseases – if the disease is reasonably rare, it is more efficient than testing all items individually

Here, a frequent item is considered “positive”

Two kinds: adaptive and non-adaptive

We are interested in non-adaptive procedures

Can we come up with a process that will work for multiple frequent items, drawing inspiration from group testing?

Generalizing

The log-levels idea works well for finding a single frequent item.

We will need to do something extra for multiple frequent items

Suppose that we pick a random subset of the universe

If there is only one frequent item in it, then using log-groups to divide the subset, we will be able to find it.

Outline procedure

Procedure to find k frequent items:

- Pick S subsets of the universe (subsets of $1..U$)
- For each item in the stream i :
 - For each subset s_j that contains i
 - Add (Subtract) 1 to the appropriate log-groups of s_j for an arrival (departure)
- To find the frequent items:
 - For each subset, find if there is a frequent item and read its index from the log-groups structure
 -

A long way to go yet

There are many details to fix up for this procedure:

- How big to make S (number of subsets)?
- How do we store each subset efficiently?
- What guarantee do we have that this will work?
- What happens if there isn't exactly one frequent item in a subset – can we detect this?
-

Picking the Subsets

If we choose elements of the universe at random, then we will need to store all these.

If the subset is large, it will require a large amount of space to store

Let's do something different

Suppose we use a function $f: U \times N \rightarrow \{0..M\}$

If $f(i) = j$, we include i in set j

If we can store f in some efficient way, then the cost of storing the subsets will be low.

Hash Functions

We will use **Pairwise Independent Hash Functions**.

A family of functions, $F: \{0..U-1\} \rightarrow \{0..n-1\}$ is **Pairwise Independent** if for all $x_1 \neq x_2$, and f chosen uniformly at random from F , then

$$\Pr[f(x_1) = f(x_2)] = 1/n$$

Essentially, the effect on x_1 and x_2 is independent.

The definition may seem complicated, but simple pairwise independent hash functions exist.

A Family of Strongly 2-Universal Functions

Let p be some prime number $> n$

$$f_{a,b}(x) = ((ax + b) \bmod p) \bmod n$$

Defines a family of strongly 2-Universal Functions for choices of a, b from $\{0..p-1\}$

If interested, find more details in a textbook, eg
[Motwani, Raghavan "Randomized Algorithms"](#)

We will not prove the properties here.

The important fact is that $f_{a,b}$ can be represented compactly: we need to store a, b : each is a number with $\log p$ bits. p is going to be $< 2n$ (why?)

Using Hash Fns to define sets

We can use the hash functions to define the subsets of the universe

Suppose we are looking for k frequent items which occur with frequency greater than $1/(k+1)$

Then we set the parameters $n = 2k$ (this is a choice we will justify over the next few slides)

We pick set s_j by choosing a_j and b_j

Item i is included in set j if $f(i)=j$, that is if

$$((a_j x + b_j) \bmod p) \bmod n = j$$

Probability Results

We want to choose subsets so that there is exactly one frequent item in them

What are the chances of this happening?

If we can bound the chance of **not** having exactly one frequent item in the subset, then this will do.

- There might be two (or more) frequent items in the same subset
- There might be no frequent items in a subset
-

Two or more frequent items

Each frequent item is placed in some bucket.

The expected frequency of other items in the bucket is just

$$\sum_i \text{freq}(i)/2k \leq k/2(k+1)k = \frac{1}{2}(k+1)$$

By the Markov inequality, the chance that the total frequency of other items in the bucket is more than $1/(k+1)$ is $< \frac{1}{2}$

Repetitions

If we do this repeatedly, then chances are, every frequent item will eventually land in a bucket with no other frequent items.

If we repeat the above procedure $\log k/\delta$ times (using different hash functions), then the probability of failing on all of them is

$$2^{-\log k/\delta} = \delta/k$$

Over at most k frequent items, using union bound, total probability is at most δ

Finally...

We now have the whole algorithm...

- Pick $S = \log k/\delta$ values a_j, b_j in range $\{0..k-1\}$ uniformly at random
- For each item in the stream i :
 - For $j=1$ to S :
 - If $((a_j x + b_j) \bmod p) \bmod 2^k = m$ then
 - Add (Subtract) 1 to the appropriate log-groups of $[j,m]$ for an arrival (departure)
- To find the frequent items:
 - For $j=1$ to S , for $m=1$ to 2^k find if there is a frequent item and read its index from the log-groups of $[j,m]$

Last details

How do we tell if there is a frequent item in a group?

We are looking for items with frequency $> 1/(k+1)$

So, if there is a frequent item in some set, then its count must be $> \text{total}/(k+1)$

If the count for a subset $< \text{total}/(k+1)$, then there is no frequent item in that group.

If we are looking through the log groups of a set, and we find some `groupM` such that

$\text{count}(\text{groupM}) > \text{total}/(k+1)$ and

$\text{count}(\text{set}) - \text{count}(\text{groupM}) > \text{total}/(k+1)$

then there are two or more frequent items in the set

Can we always spot 2 frequent items in the same set?

Finishing up

Running time

To process an item, compute $O(\ln k/\delta)$ hash functions.

Use $\log U$ time to process the groups for these

Total time is $O(\ln k \log U)$

One minor issue... the algorithm can get fooled if the total frequency of other items in the set add up to more than $1/(k+1)$

This rarely happens in practice... and if there really are k frequent items, then this can't happen

Practical Issues

Implemented this algorithm and tried it out on some real telephone call data streams (calls begin and end)

Compared 'Group Testing' to a couple of other algorithms, tweaked so that they could handle departures.

Tried the Frequent algorithm from earlier, and Lossy Counting (Manku & Motwani '02)

These algorithms are faster (can be close to constant time to process each item), but don't naturally generalize to arrival and departures

Test set up

Wanted to test the recall and precision of the different methods

Recall = % of frequent items that were found

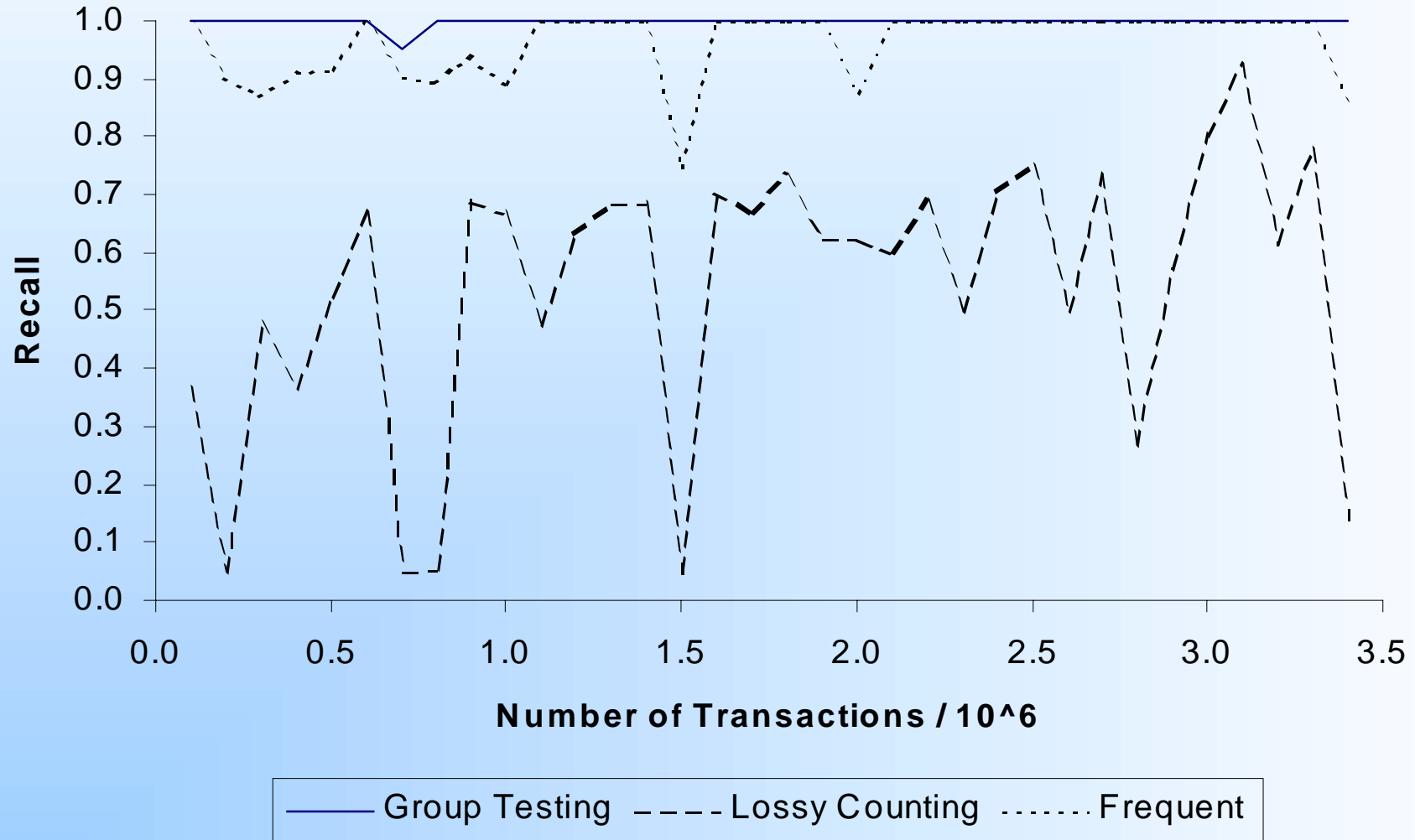
Precision = % of found items that were frequent

A relatively small experiment... processed a few million phone calls (from one day)

Can make the code available if you want to experiment with it (1000 lines of C for all 3 methods)

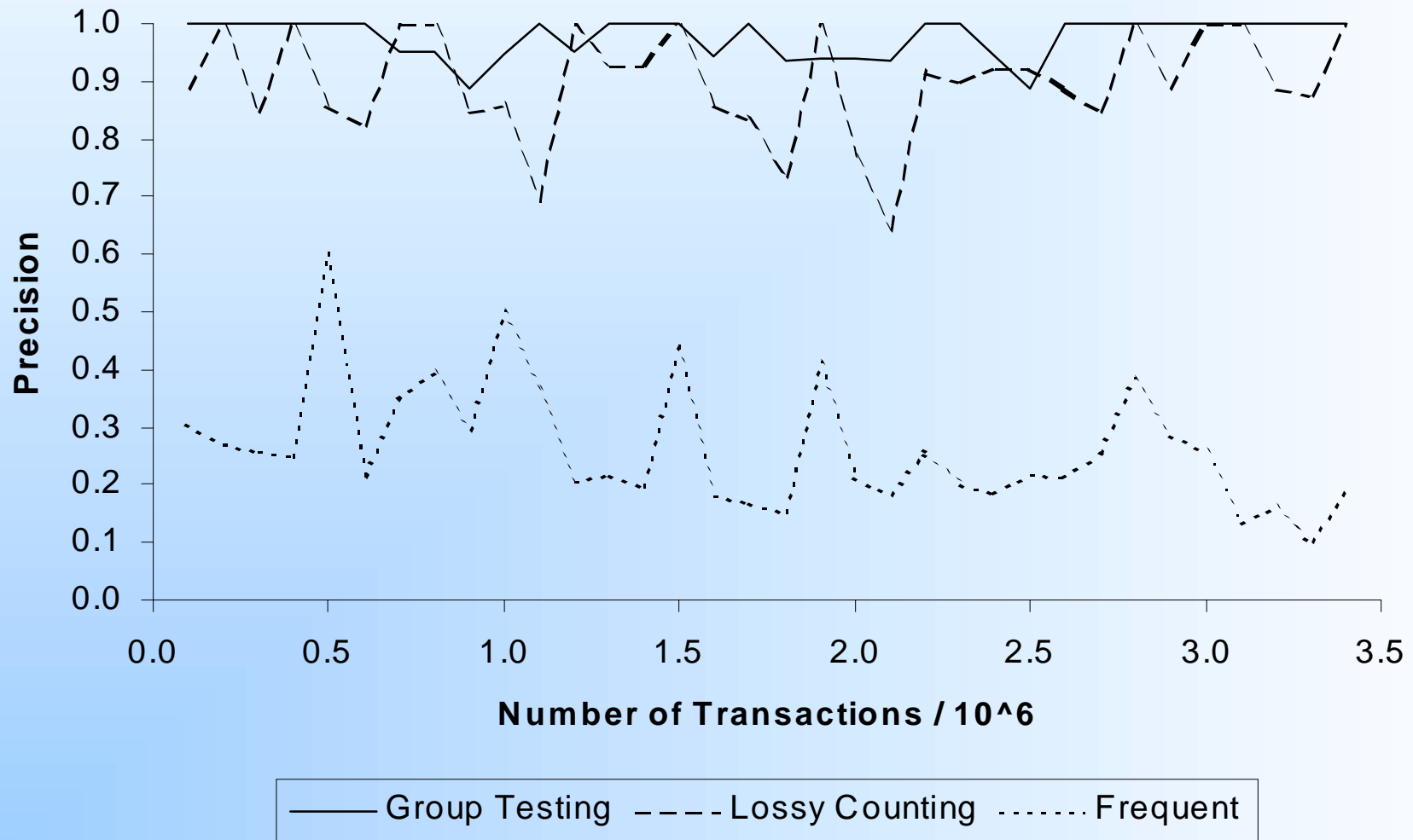
Recall

Recall on Real Data



Precision

Precision on Real Data



Observations

Lossy Counting is good at precision, not at recall

Frequent is good at recall, not at precision

Group Testing is better at both

This stuff checks out in practice

If you have a real problem, theory can get you so far,
but trying it out is also helpful

- Worst cases might not happen
- $O(\log n)$ can beat $O(1)$ if n is not big enough

The Moral of the Story

- We took a very general problem, of finding items which occur frequently
- Abstracted this as a problem on integers
- What seems initially trivial becomes more challenging with memory limitations
- Made use of techniques from around CS:
 - Group Testing from Combinatorics
 - Compactly Representable Hash functions
 - Some probability for the analysis
- Experiments verify theory (esp for database people)

References

- MJRTY - A Fast Majority Vote Algorithm
R. Boyer and S. Moore, U. Texas Tech report, 1982
- Finding Repeated Elements, J. Misra and D. Gries,
Science of Computer Programming, 1982
- Approximate Frequency Counts over Data Streams,
G. Manku and R. Motwani, VLDB 2002
- What's Hot and What's Not: Tracking Most
Frequent Items Dynamically, G. Cormode and S.
Muthukrishnan, PODS 2003