Streaming Algorithms for Straggler Detection

David Eppstein and Michael T. Goodrich
Univ. of California, Irvine
Computer Science Department

Streaming Algorithms

Compute some function of a large data set

View each data item once in sequential order

Use very small auxiliary storage (much less than data size)

Why?

Scale efficiently to huge data sets too large for main memory

Handle network data streams without random access

Allow computation on machines with very small capacity
due to size, cost, or power consumption constraints

Standard toy problem:
Find the missing element

Input: stream of 99 different numbers in the range 1 – 100

Output: the missing number

Problem: solve this using only storage for a single number

Standard toy problem: Find the missing element

Input: stream of 99 different numbers in the range 1 – 100

Output: the missing number

Solution:

Store $S = \sum x_i$

Missing number is $5050 - S$

Standard toy problem: Find the missing element

Input: stream of 99 different numbers in the range 1 – 100

Output: the missing number

Solution:

Store $S = \sum x_i$

Missing number is 5050 - S

Work modulo 128: need only 7 bits of storage

Straggler detection

Intuitive definition:

You are manning the security desk of a large building

Everyone who walks into or out of the building
checks in or checks out with their id

At the end of the day, there should be nobody left inside

Your task:

Verify that nobody is left, and, if not,
identify the few stragglers left in the building

Straggler detection

Formal definition:

Input: sequence of events

insert x

delete x

Guaranteed to be the case that:

— each id x is inserted and deleted at most once

— each deleted x has previously been inserted

— at most K insertions have no matching deletion

Output:

A list of the insertions without matching deletion

Straggler detection

Two potential applications:

High bandwidth multicasting

- when packet x is multicast, insert (x, receiver) pairs
- when receiver acknowledges the packet, delete pair
- straggler = dropped packet to be retransmitted

Distributed grid computations

- when sending subtask to processor, insert task id
- when processor completes subtask, delete task id
- straggler = missing subtask to be recomputed

Known results

Ganguly and Majumder, PODS 2006

Introduce the problem

Solve space-optimally: $O(K \log n)$ bits, n = range of possible ids

Slow decoding time (cubic or quartic in K, ignoring logarithmic factors)

Cormode and Muthukrishnan, ACM Trans. DB 2005

Solve more general problem: high frequency items in data stream

Randomized, higher space bounds: $O(K \log^2 n \log(1/p))$, p = mistake probability

New results

I. Deterministic space-optimal algorithm with fast decoding

O(K log n) bits, better constant than Ganguly and Majumder

Insert, delete linear in K
Decoding time quadratic in K

Algebraic approach: Store sums of powers of inserted items

Use Newton's identities to solve system of symmetric polynomials

Careful choice of Galois field allows inversion of Newton's identities
while allowing the application of fast root-finding algorithms

New results

II. Generalization of problem allowing spurious deletions

What if an adversary can try to trick the algorithm
by including a small number of deletions not matching previous insertions?
Algorithm must find all stragglers and all spurious deletions

Lower bound:
No deterministic algorithm can solve the problem
in space sublinear in the range of ids

Proof idea:
Show that any streaming algorithm has a group structure
Use pigeonhole principle to find two inputs mapped to same group element
Subtract them to find a single input that confuses the algorithm

New results

III. Randomized solution of generalized problem

New data structure: "invertible Bloom filter"

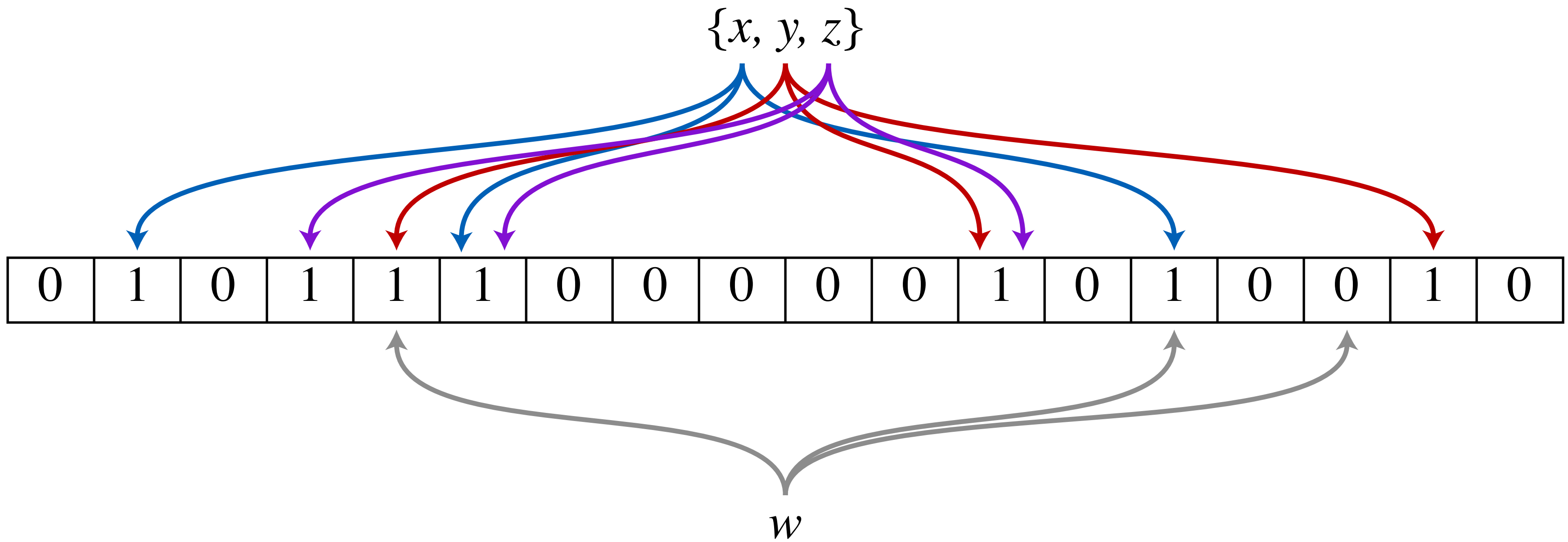
hashing technique related to Bloom filter [Bloom, CACM 1970]
and counting Bloom filter
[Bonomi et al., ESA 2006; Fan et al., ACM/IEEE Trans. Networking 2000]

Space: $O(K \log n \log(1/p))$, nearly optimal

Time per insert or delete: $O(\log(1/p))$
Time per decode: linear in K

Bloom filter

Hashed representation of a set of n items
Store array of 2nq bits, initially all zero
Map each item x to q bits hash(x,1), hash(x,2), ..., hash(x,k)
For each item in the set, store a one in all of its mapped bits



If an item belongs to the set, all of its bits will be one
If an item does not belong to the set, w/prob ≥ $1 - 2^{-q}$, some bit will be zero

Invertible Bloom filter

$hash_1(x,i)$ mapping each item to $q \sim \log(1/p)$ hash table locations as before
$hash_2(x)$ providing a "signature" for each item

Each hash table location stores counter, sum(x), and sum($hash_2(x)$)
where both sums are over the identities of the inserted items
that $hash_1$ maps to that location

Use modular arithmetic to save bits in each sum

Invertible Bloom filter: how to update

$hash_1(x,i)$ mapping each item to $q \sim \log(1/p)$ hash table locations as before
$hash_2(x)$ providing a "signature" for each item

Each hash table location stores counter, sum(x), and sum($hash_2(x)$)
where both sums are over the identities of the inserted items
that $hash_1$ maps to that location

To insert x:

```
for i in 1, 2, ..., q:
    table[hash1(x,i)].count += 1
    table[hash1(x,i)].sumx += x
    table[hash1(x,i)].sumhash += hash2(x)
```

Deletion similar (subtract instead of add)

Invertible Bloom filter: how to decode

$hash_1(x,i)$ mapping each item to $q \sim \log(1/p)$ hash table locations as before
$hash_2(x)$ providing a "signature" for each item

Each hash table location stores counter, sum(x), and $sum(hash_2(x))$
where both sums are over the identities of the inserted items
that $hash_1$ maps to that location

Hash location is "pure" if only one straggler maps to it

If location j is pure, $hash_2(sum(x)/count) = sum(hash_2(x))/count$

With high probability:
All but a small fraction of stragglers are mapped to a pure location
No seemingly-pure location is actually impure

Invertible Bloom filter: cleanup

What to do about the small number of stragglers
that are not mapped to a pure cell?

Could remove the pure stragglers, continue decoding the remaining array

Probably works, hard to analyze

Instead, use a separate table with q=2

Repeatedly remove pure stragglers

Forms very sparse random graph with vertex=hashed location, edge=straggler
Repeated removal works if graph has no cycle, true with high probability

Conclusions

Two new algorithms for straggler detection

Deterministic, optimal space, fast operations

Randomized, near-optimal space, faster, allows spurious deletions

Likely applications

Impossibility result for spurious deletions

Interesting example of provable difference between
randomized and deterministic streaming algorithms