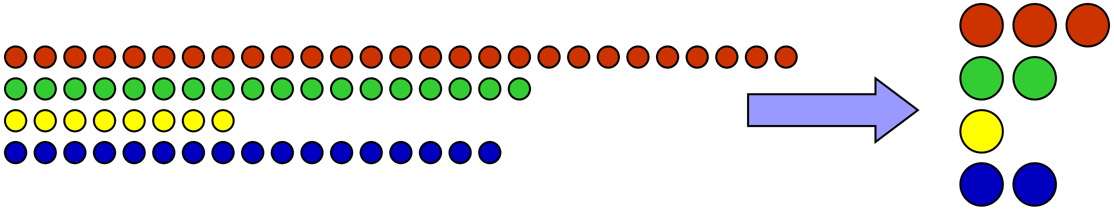
Engineering Streaming Algorithms



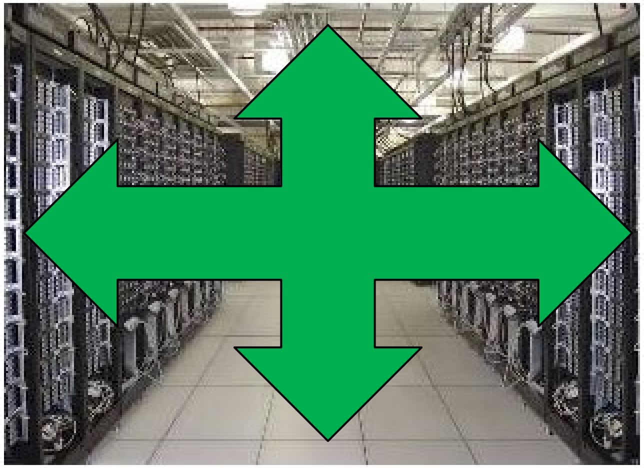
Graham Cormode

University of Warwick

G.Cormode@Warwick.ac.uk

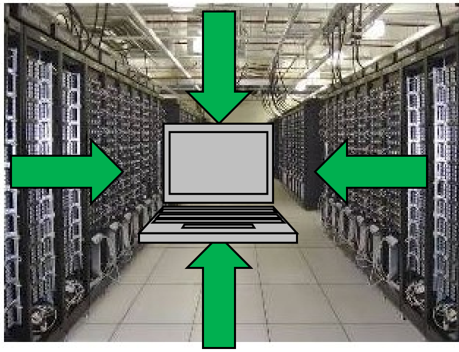
Computational scalability and "big" data

- Most work on massive data tries to **scale up the computation**
- Many great technical ideas:
  - Use many cheap commodity devices
  - Accept and tolerate failure
  - Move code to data, not vice-versa
  - MapReduce: BSP for programmers
  - Break problem into many small pieces
  - Add layers of abstraction to build massive DBMSs and warehouses
  - Decide which constraints to drop: noSQL, BASE systems



- Scaling up comes with its disadvantages:
  - Expensive (hardware, equipment, **energy**), still not always fast
- This talk is not about this approach!

Downsizing data



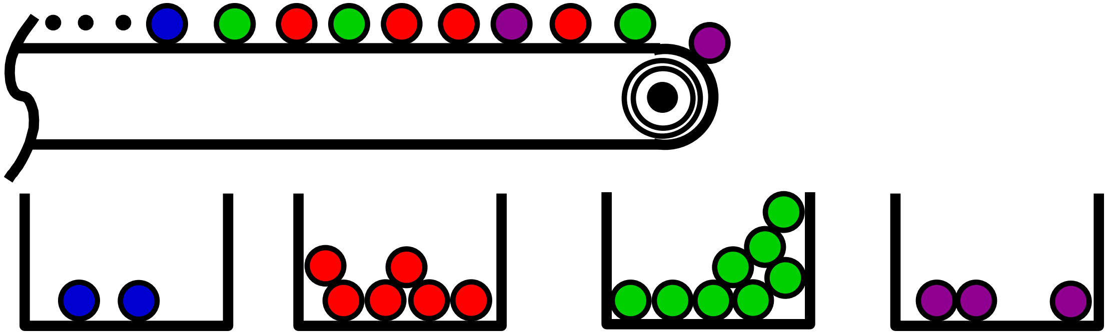
- A second approach to computational scalability: **scale down the data** as it is seen!
  - A compact representation of a large data set
  - Capable of being analyzed on a single machine
  - What we finally want is small: human readable analysis / decisions
  - Necessarily gives up some accuracy: approximate answers
  - Often randomized (small constant probability of error)
  - Much relevant work: samples, histograms, wavelet transforms
- Complementary to the first approach: not a case of either-or
- Some drawbacks:
  - Not a general purpose approach: need to fit the problem
  - Some computations don't allow any useful summary

Outline for the talk

- The frequent items problem
- Engineering streaming algorithms for frequent items
  - From algorithms to prototype code
  - From prototype code to deployed code
- Next steps: robust code, other hardware targets
- Bulk of the talk is on two (actually, one) very simple algorithms
  - Experience and reflections on a ‘simple’ implementation task

The Frequent Items Problem

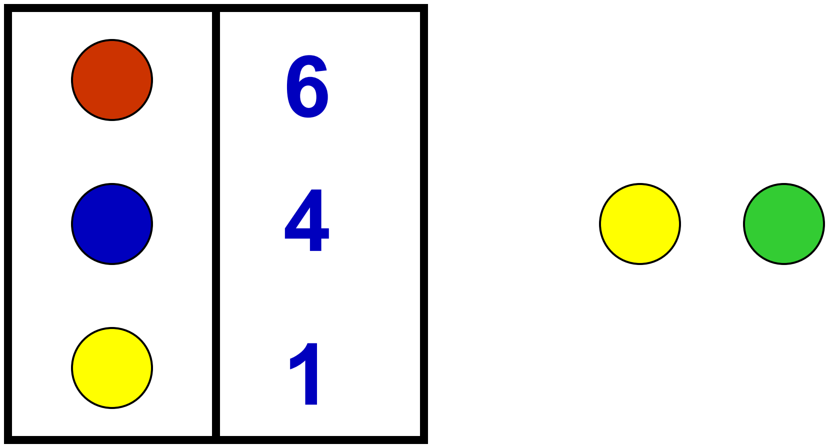
- The Frequent Items Problem (aka Heavy Hitters): given stream of N items, find those that occur most frequently
  - E.g. Find all items occurring more than 1% of the time
- Formally "hard" in small space, so allow approximation
- Find all items with count $\geq \phi N$, none with count $< (\phi - \varepsilon) N$
  - Error $0 < \varepsilon < 1$, e.g. $\varepsilon = 1/1000$
  - Related problem: estimate each frequency with error $\pm \varepsilon N$



Why Frequent Items?

- A natural question on streaming data
  – Track bandwidth hogs, popular destinations etc.
- The subject of much streaming research
  – Scores of papers on the subject
- A core streaming problem
  – Many streaming problems connected to frequent items (itemset mining, entropy estimation, compressed sensing)
- Many practical applications deployed
  – In search log mining, network data analysis, DBMS optimization

Misra-Gries Summary (1982)



- Misra-Gries (MG) algorithm finds up to k items that occur more than 1/k fraction of the time in the input
- Update: Keep k different candidates in hand. For each item:
  - If item is monitored, increase its counter
  - Else, if < k items monitored, add new item with count 1
  - Else, decrease all counts by 1

Frequent Analysis

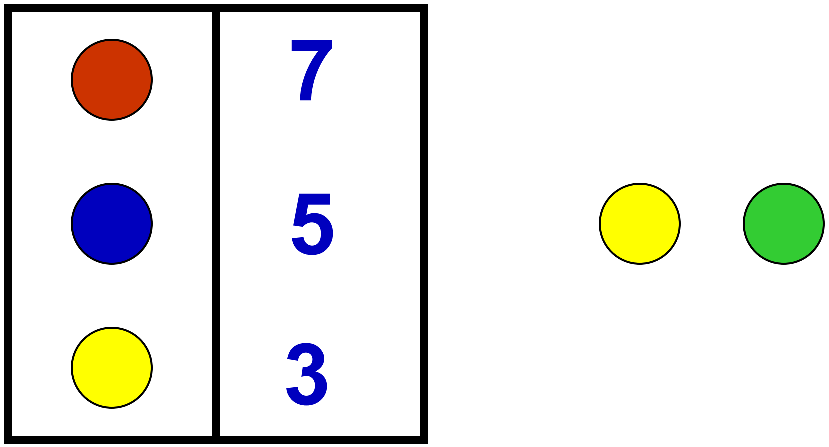
- Analysis: each decrease can be charged against k arrivals of different items, so no item with frequency N/k is missed
- Moreover, $k=1/\varepsilon$ counters estimate frequency with error εN
  - Not explicitly stated until later [Bose et al., 2003]

- Some history: First proposed in 1982 by Misra and Gries, rediscovered twice in 2002
  - Later papers discussed how to make fast implementations

Merging two MG Summaries [ACHPWY '12]

- Merge algorithm:
  - Merge the counter sets in the obvious way
  - Take the $(k+1)$th largest counter $= C_{k+1}$, and subtract from all
  - Delete non-positive counters
  - Sum of remaining counters is M_{12}
- This keeps the same guarantee as Update:
  - Merge subtracts at least $(k+1)C_{k+1}$ from counter sums
  - So $(k+1)C_{k+1} \leq (M_1 + M_2 - M_{12})$
  - By induction, error is
    $((N_1-M_1) + (N_2-M_2) + (M_1+M_2-M_{12}))/(k+1)=((N_1+N_2) - M_{12})/(k+1)$
    (prior error) (from merge) (as claimed)

SpaceSaving Algorithm



- "SpaceSaving" (SS) algorithm [Metwally, Agrawal, El Abaddi 05] is similar in outline
- Keep $k = 1/\varepsilon$ item names and counts, initially zero
  Count first k distinct items exactly
- On seeing new item:
  - If it has a counter, increment counter
  - If not, replace item with least count, increment count

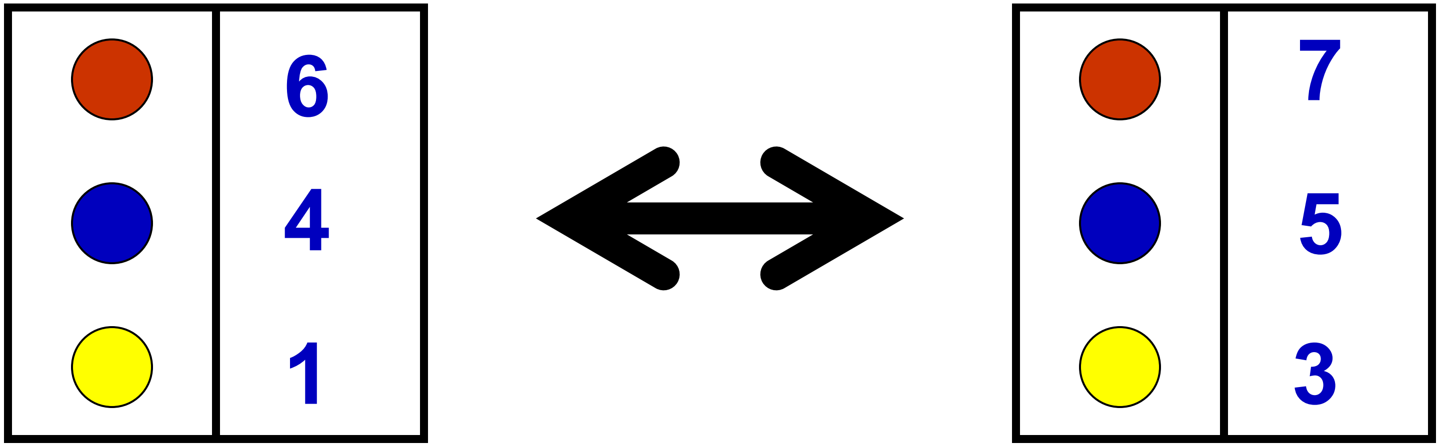
SpaceSaving Analysis

- Smallest counter value, min, is at most εn
  - Counters sum to n by induction
  - $1/\varepsilon$ counters, so average is εn: smallest cannot be bigger
- True count of an uncounted item is between 0 and min
  - Proof by induction, true initially, min increases monotonically
  - Hence, the count of any item stored is off by at most εn
- Any item x whose true count $> \varepsilon n$ is stored
  - By contradiction: x was evicted in past, with count $\leq min_t$
  - Every count is an overestimate, using above observation
  - So est. count of $x > \varepsilon n \geq min \geq min_t$, and would not be evicted

So: Find all items with count $> \varepsilon n$, error in counts $\leq \varepsilon n$

Two algorithms, or one?

- A belated realization: SS and MG are the same algorithm!
  - Can make an isomorphism between the memory state
- Intuition: “overwrite the min” is conceptually equivalent to delete elements with (decremented) zero count
- The two perspectives on the same algorithm lead to different implementation choices

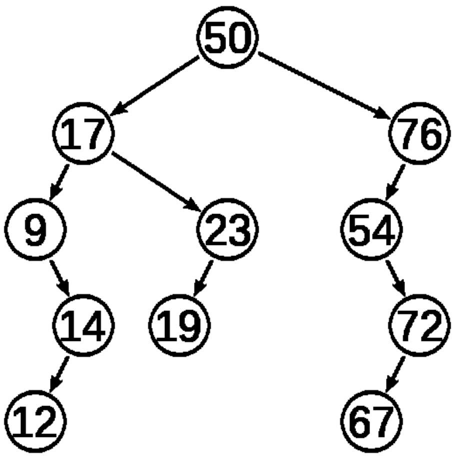


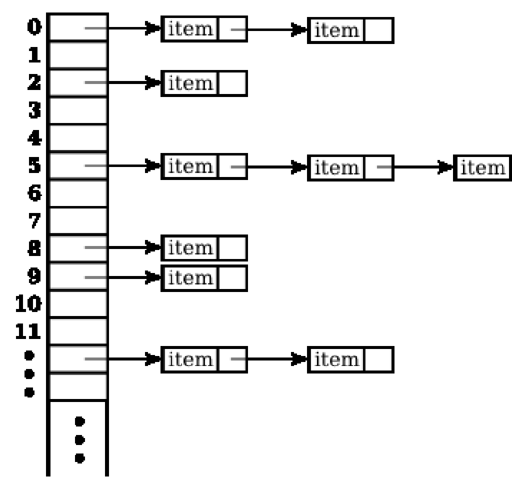
Implementation Issues

- These algorithms are really simple, so should be easy… right?
- There is surprising subtlety in implementing them
- Basic steps:
  - Lookup is current item stored? If so, update count
  - If not:
    - Find min weight item and overwrite it (SS)
    - Decrement counts and delete zero weights (MG)
- Several implementation choices for each step
  - Optimization goals: speed (throughput, latency) and space
  - I discuss my implementation experience and current thoughts

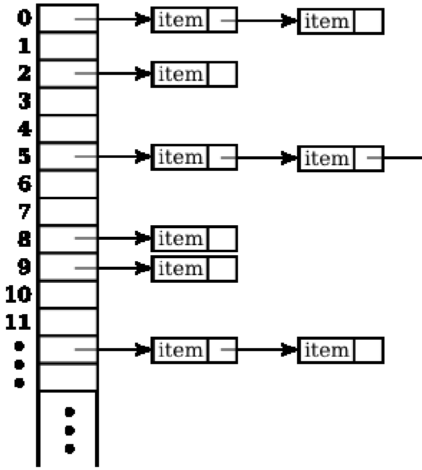
Lookup Item

- Lookup: is current item stored
  - The canonical dictionary data structure problem
- Misra Gries paper: use balanced search tree
  - O(log k) worst case time to search
- Hash table: hash to O(k) buckets
  - O(1) expected time, but now alg is randomized
    - May have bad worst case performance?
  - How to handle collisions and deletions?
    - (My implementations used chaining)
  - Could surely be further optimized…
    - Use cuckoo hashing or other options?
    - Can we use fact that table occupancy is guaranteed at most k?





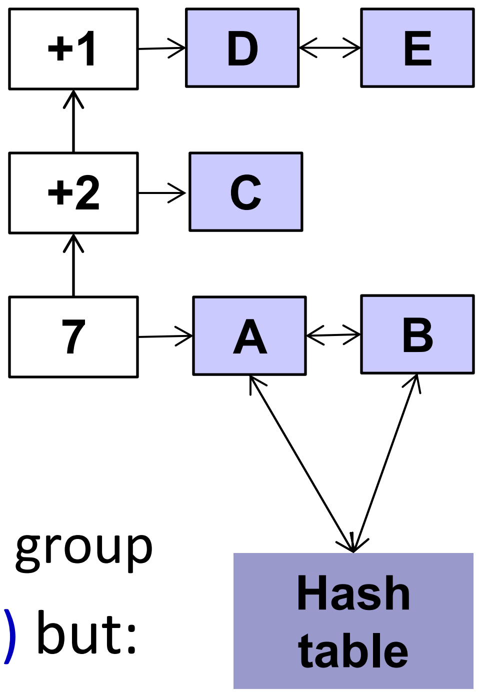
Decrement Counts



- Decrement counts could be done simply
  - Iterate through all counts, subtract by one
  - A blocking operation, O(k) time
- Proof of correctness means it happens < n/k times
  - So would be O(1) cost amortized…
  - (considered too fiddly to deamortize when I implemented)
    - Multithreaded/double buffered approach could simplify

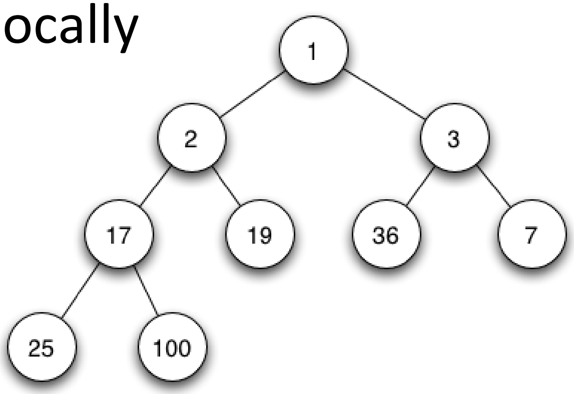
Decrement Counts: linked list approach

- Linked list approach (Demaine et al. 02):
  - Keep elements in a list sorted by frequency
  - Store the difference between successive items
  - Decrement now only affects the first item
- But increments are more complicated:
  - Keep elements with same frequency in a group
  - Since we only increase count by 1, move to next group
- Increments and decrements now take time O(1) but:
  - Non-standard, lots of cases (housekeeping) to handle
  - Forward and backward pointers in circular linked lists
  - Significant space overhead (about 6 pointers per item)



Overwrite min

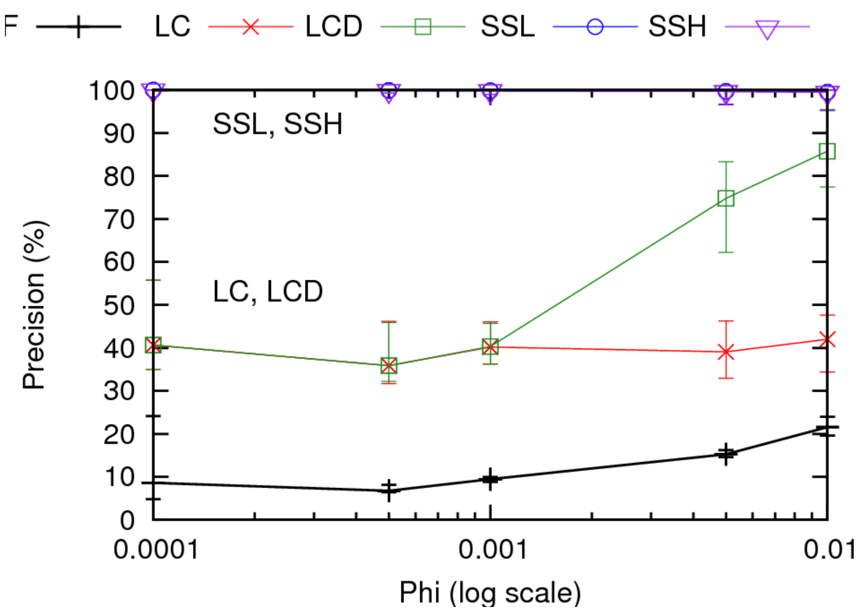
- Could also adapt the linked list approach
  - Keep items in sorted order, overwrite current min
- Findmin is a more standard data structure problem
  - Could use a minheap (binary, binomial, fibonacci…)
  - Increments easy: update and reheapify O(log k)
    - Probably faster, since only adding one to the count
  - All operations O(log k) worst case, but may be faster "typically":
    - Heap property can often be restored locally
    - Head of heap likely to be in cache
    - Access pattern non-uniform?

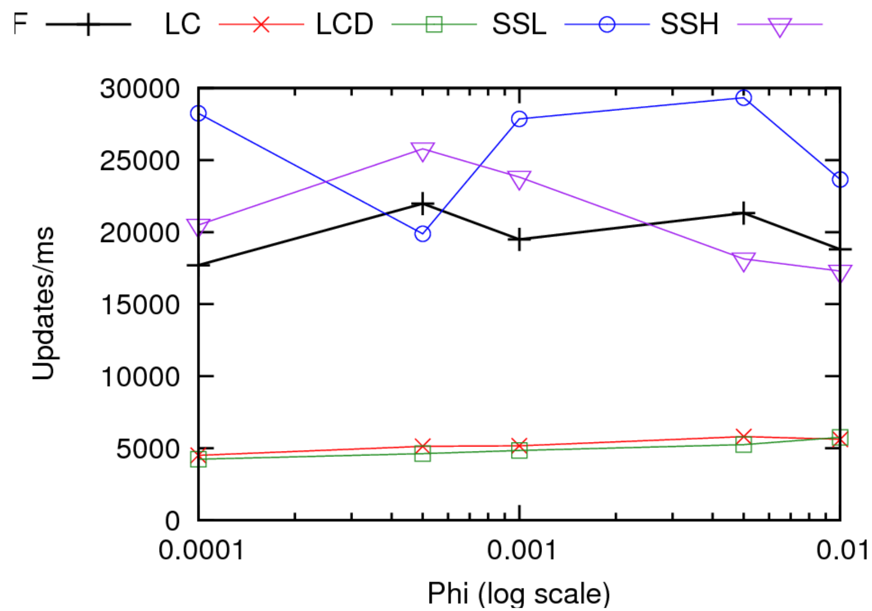


Experimental Comparison

- Implementation study (several years old now)
  - Best effort implementations in C (use a different language now?)
  - All low-level data structures manually implemented (using manual memory management)
  - `http://hadjieleftheriou.com/frequent-items/index.html`
- Experimental comparison highlights some differences not apparent from analytic study
  - E.g. algorithms are often more accurate than worst-case analysis
  - Perhaps because real inputs are not worst-case
- Compared on a variety of web, network and synthetic data

Frequent Algorithms Experiments



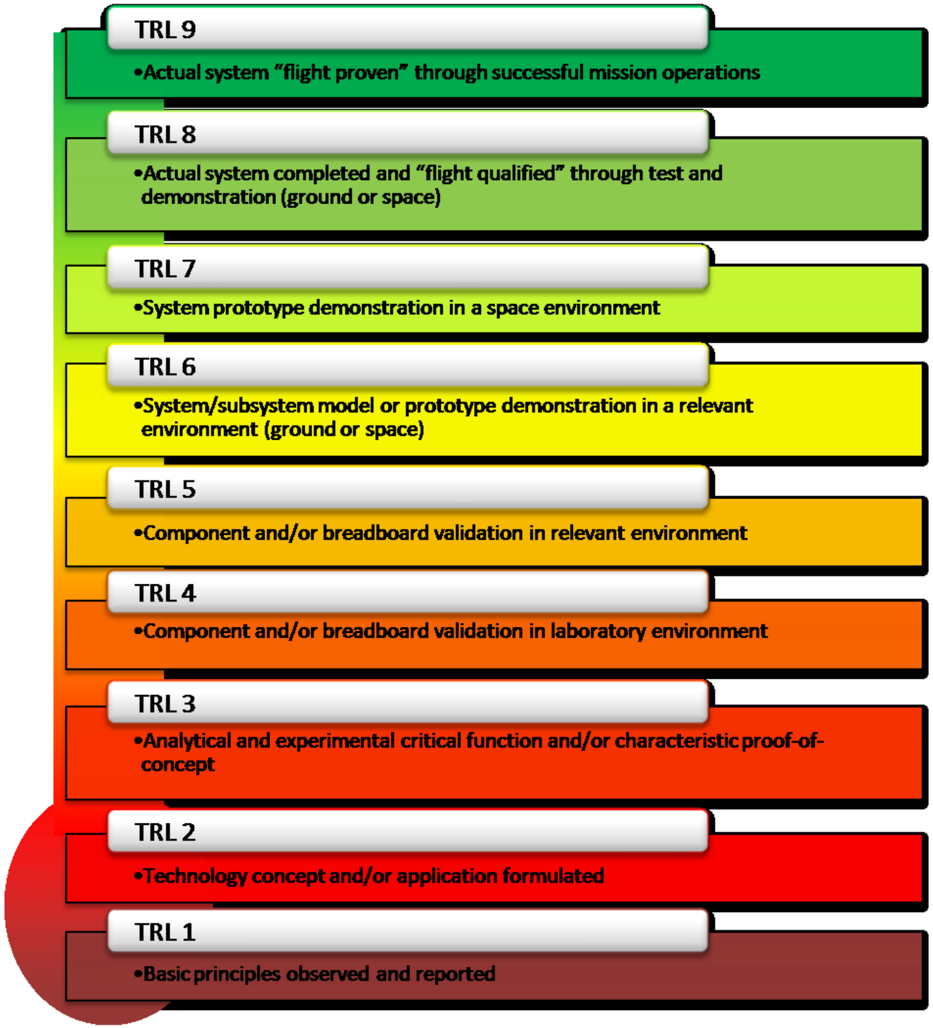


- Two implementations of SpaceSaving (SSL, SSH) achieve perfect accuracy in small space (10KB – 1MB)
- Misra Gries (F) has worse accuracy: different estimator used
- Very fast: 20M – 30M updates *per second*
  - Heap seems faster than linked list approach

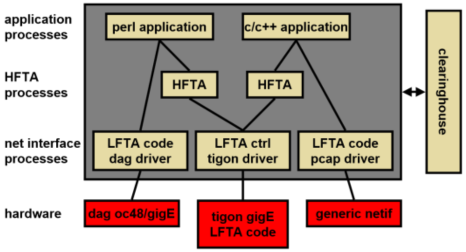
Frequent Algorithms Summary

- These algorithms very efficient for arrivals-only case
  - Use $O(1/\varepsilon)$ space, guarantee εN accuracy
  - Very fast in practice (many millions of updates per second)
- Similar algorithms, but a surprisingly clear “winner”
  - Over many data sets, parameter settings, SpaceSaving algorithm gives appreciably better results
- Many implementation details even for simple algorithms
  - “Find if next item is monitored”: search tree, hash table…?
  - “Find item with smallest count”: heap, linked lists…?
- Not much room left for improvement in core algorithm?
  - Maybe more explicitly model input distributions (skewed)?

Ready for prime time?



Streaming in practice: Packet stream analysis

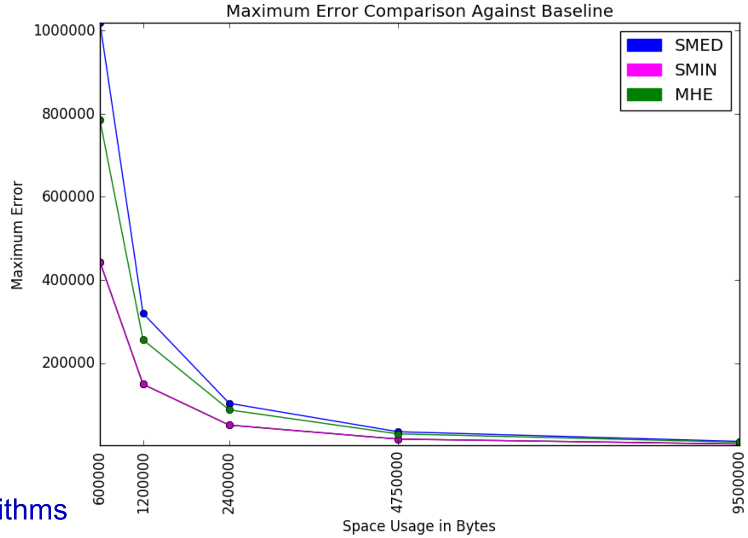


- AT&T Gigascope / GS tool: stream data analysis
  - Developed since early 2000s
  - Based on commodity hardware + Endace packet capture cards
- High-level (SQL like) language to express continuous queries
  - Allows "User Defined Aggregate Functions" (UDAFs) plugins
  - Sketches in gigascope since 2003 at network line speeds (Gbps)
  - Flexible use of streaming algs to summarize behaviour in groups
  - Rolled into standard query set for network monitoring
  - Software-based approach to attack, anomaly detection
- Current status: latest generation of GS in production use at AT&T
  Also in Twitter analytics, Yahoo, other query log analysis tools

More Recent Progress

[Anderson et al ’17] report their experience at Yahoo!

- Delete min operation can be amortized over multiple steps
- Instead of deleting based on min of k, used median of 2k counts
- Estimate median by sampling rather than quickselect
- May be seen as similar to a merge and prune approach
- Several times faster again than heap-based method
- Moderately increased error compared to delete min
- Java sketch library: https://datasketches.github.io/



Conclusions

- Finding the frequent items is one of the most studied problems in data streams
  - Continues to intrigue researchers (for better or worse)
  - Many variations proposed (for weighted or negative updates)
  - Algorithms have been deployed in Google, AT&T, elsewhere…
  - New variants continue to be suggested
- Other streaming primitives have been similarly engineered
  - E.g. Bloom Filters, Hyperloglog (Heule et al '13), Quantiles
  - More general sketches that can handle deletions and insertions
- Areas for more work:
  - Allow easier composition of algorithms
  - Adapt to new models (parallel, distributed, FPGA/GPU)