Policy Exploration for JITDs (Java)

Team Datum

Recap

- Experimentation on current policies against Zipfian distribution
  - Cracking vs Adaptive Merge vs Swap
- Current target Workload : Zipfian Read Heavy Workload
  - Answer High-Probable queries quickly → Keep those Query bounds on the top levels of Tree.
  - Answer Low-probable queries in decent time → Balance the tree regularly.
- Implemented Policies
  - Constant Interval : Lower bound of range query
  - Constant Interval : Median Element
- Target Policy : Splay around most frequently accessed elements

Finding the Median Element from the Tree

- Median Element is found in 2 steps:
  - Step 1 : Get the count(‘n’) of the BTree Separators in the current structure.
  - Step 2 : Do the in-order traversal of the Cod and return the element at ‘n/2’ position
  - Time taken for finding the median = (3/2) n

- In order to find the median we tried to reduce the factor (3/2)n to (n/2), provided we knew the count. For this, we have implemented a small tweak where we can update the n during each BTree cog creation.

Median Key Value	Before	After
401	220783	68149
414	182273	61837
424	192975	66434
426	204051	79264

***Table**: Time taken (in nanoseconds) for getting the median key before and after we apply the tweak.*

Previous Policy

- Store the <Key, Value> pairs of <Separator, ReadCount> - For only the last R reads?
- Get 'm' number of high frequent values in such a way that there will be at least 'e' difference with it's next highest value.
- Perform splay on these 'm' separator values.
- If ReadCount is increasing, we can re-scale the count values, by using a weightage factor 'w' (for example, w = 0.02) that multiplies with each value in the map and updates its ReadCount.
- Repeat the above process and change the weightage factor 'w' variably if necessary.

Gist

- Maintain Most Frequently Accessed (MFA) reads separately using less space.
- Should take reasonably less time for maintenance.
- Use time decay to give more weightage to recently accessed separators.
- Should find MFA elements accurately.

How to find MFA Elements?

- It seems naïve to track the counts of all the accessed reads as given in previous policy. Then How?
  - Should be able to approximate the most frequent elements (frequently accessed bounds of range queries)

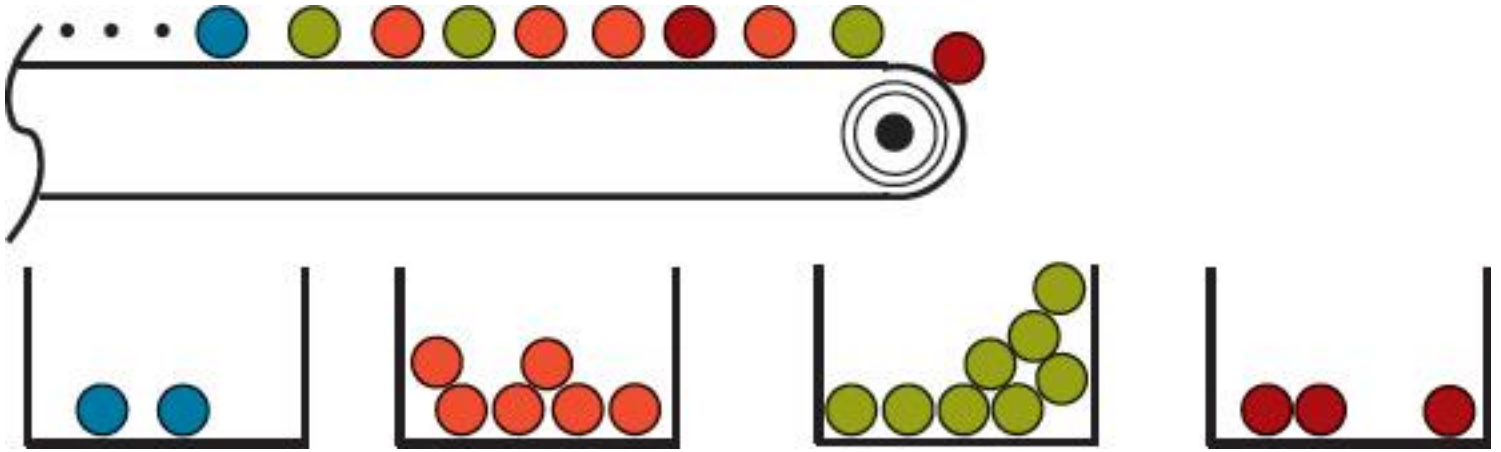
- Problem of finding frequent elements in data stream is one of the most discussed and experimented research topic.

- We can correlate our problem with the problem of finding frequent elements in data stream.
  - Data Stream <-----> Read Statistics of Users (lower bound of range queries (or) even the both query bounds)
  - Finding frequent elements <-----> Finding Most frequently accessed Separators in Cog structure.

Goal

- What is Frequent items problem?
- What are the considered solutions?
- What is the preferred algorithm of all? (For our need)
- Why is it preferred?

Frequent Items Problem



- A stream of items defines a frequency distribution over items.
- In this example, with a threshold of ϕ = 20% over the 19 items grouped in bins, the problem is to find all items with frequency at least 3.8 (ϕ . N = 0.2 * 19)
- In this case, the green and red items (middle two bins).

Definition : Given a stream S of n items t1 ...tn, the frequency of an item i is fi = |{j|tj = i}|. The exact φ-frequent items comprise the set {i|fi > ϕ*n*}.

But this is not possible to find accurately without storing all elements!!! (Requires More space)

Approximation Definition

Given a stream *S* of *n* items, the ε-approximate frequent items problem is to return a set of items *F* so that

- **Property - 1 :** For all items $fi > (\phi - \varepsilon)n$, *i Belongs To F, (False Positives Possible)*
- **Property - 2 :** There is no $i \notin F$ such that $fi > \phi n$. (No False Negatives)

Example :

Imagine a user who is interested in identifying all items whose frequency is at least 0.1% of the entire stream seen so far. Then $\phi = 0.1\%$. Let us assume we choose 'e' = 0.01% i.e. one-tenth of ϕ.

- As per property 1, no element with frequency below 0.09% will be output.
- As per property 2, all the elements with frequency exceeding $\phi = 0.1\%$ will be output.

Type of Algorithms

- **Counter Based Algorithms**
  - Counter-based algorithms decide for each new arrival whether to store this item or not, and if so, what counts to associate with it.
  - A common feature of these algorithms is that when given a new item, they test whether it is one of k being stored by the algorithm, and if so, increment its count.

- **Sketch Based Algorithms**
  - Term "sketch" to denote a data structure which can be thought of as a linear projection of the input.
  - It doesn't explicitly store counts for each given item.

The "Frequent" Algorithm : Frequent (k)

- The algorithm stores (item, count) pairs in a Set 'T'.
- When processing the ith item in the stream
  - if item is in Set 'T' then its count (ci) is increased by one;
  - else if there is space available to store in set, a new item is created with count initialized to '1'.
  - else decrement count of all elements and remove and replace the element with "0" count.
- Executing this algorithm with $k = 1/\varepsilon$ ensures that the count associated with each item on termination is at most εn below the true value.

$$
\begin{array}{l}
n \leftarrow 0; T \leftarrow \emptyset; \\
\textbf{for each } i: \\
\quad \textbf{do} \begin{cases}
n \leftarrow n+1; \\
\textbf{if } i \in T \\
\quad \textbf{then } c_i \leftarrow c_i + 1; \\
\quad \textbf{else if } |T| < k-1 \\
\quad \textbf{then} \begin{cases} T \leftarrow T \cup \{i\}; \\ c_i \leftarrow 1; \end{cases} \\
\quad \textbf{else for all } j \in T \\
\quad \textbf{do} \begin{cases} c_j \leftarrow c_j - 1; \\ \textbf{if } c_j = 0 \\ \quad \textbf{then } T \leftarrow T \backslash \{j\}; \end{cases}
\end{cases}
\end{array}
$$

Lossy Counting : LossyCounting (k)

- The algorithm stores tuples which comprise an item, a lower bound on its count, and a "delta" (Δ) value which records the difference between the upper bound and the lower bound.
- When processing the ith item in the stream
  - if information is currently stored about the item then its lower bound is increased by one;
  - else, a new tuple for the item is created with the lower bound set to one, and Δ set to n/k (Bucket Number - 1).
- Periodically, all tuples whose upper bound is less than n/k(Δ) are deleted.
- Worst case space required here is O(1/ε log (εn))

$$
\begin{array}{l}
n \leftarrow 0; \Delta \leftarrow 0; T \leftarrow \emptyset; \\
\textbf{for each } i: \\
\quad \textbf{do} \begin{cases}
n \leftarrow n+1; \\
\textbf{if } i \in T \\
\quad \textbf{then } c_i \leftarrow c_i + 1; \\
\quad \textbf{else} \begin{cases} T \leftarrow T \cup \{i\}; \\ c_j \leftarrow 1 + \Delta; \end{cases} \\
\textbf{if } \lfloor \frac{n}{k} \rfloor \neq \Delta \\
\quad \textbf{then} \begin{cases} \Delta \leftarrow n/k; \\ \textbf{for all } j \in T \\ \quad \textbf{do if } c_j < \Delta \\ \quad \textbf{then } T \leftarrow T \backslash \{j\} \end{cases}
\end{cases}
\end{array}
$$

Space Saving : SpaceSaving(k)

- *Here, k* (item, count) pairs are stored, initialized by the first *k* distinct items and their exact counts.
- As usual, when the next item in the sequence corresponds to a monitored item, its count is incremented; but when the next item does not match a monitored item, the (item, count) pair with the smallest count has its item value replaced with the new item, and the count incremented.
- Space required is *O*(*k*) nearly equal to O(1/ε).

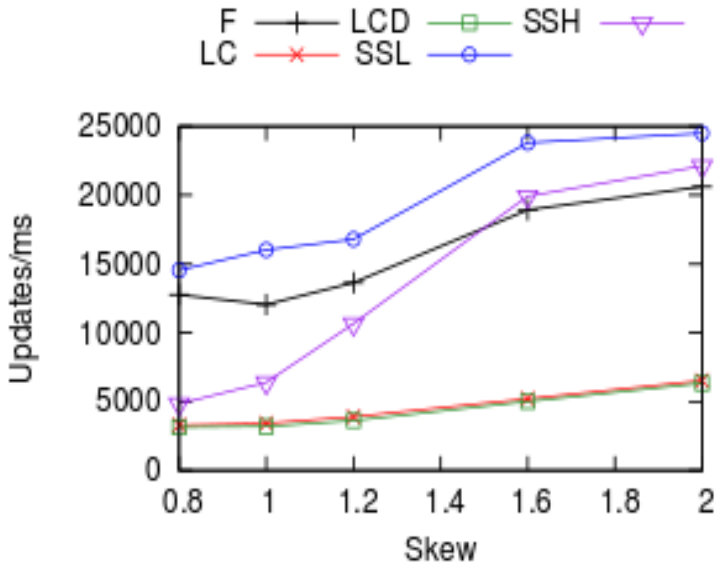
$n \leftarrow 0;$
$T \leftarrow \emptyset;$
for each i :

do
- $n \leftarrow n + 1;$
- **if** $i \in T$
  - **then** $c_i \leftarrow c_i + 1;$
  - **else if** $|T| < k$
    - **then**
      - $T \leftarrow T \cup \{i\};$
      - $c_i \leftarrow 1;$
    - **else**
      - $j \leftarrow \arg\min_{j \in T} c_j;$
      - $c_i \leftarrow c_j + 1;$
      - $T \leftarrow T \cup \{i\} \backslash \{j\};$

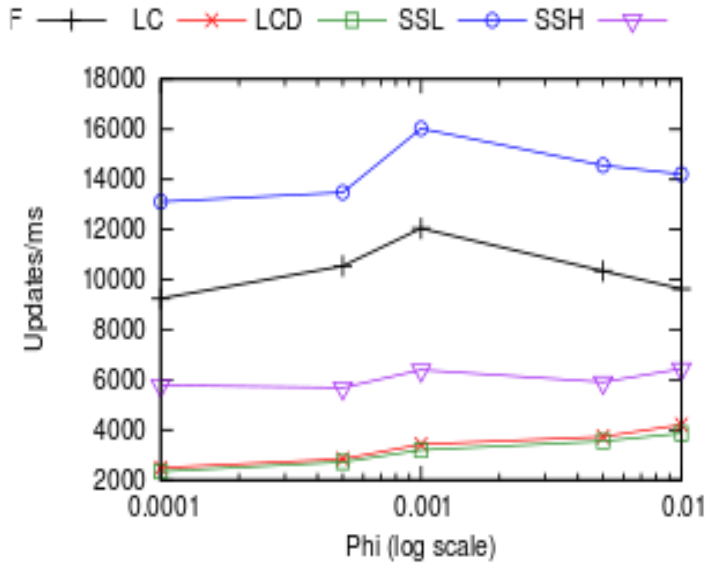
Which is Better?

F : Frequent Algorithm, LCD : Lossy Counting using Delta, LC : Lossy Counting without delta,
SSH : Space saving using Heap, SSL : Space saving using list.

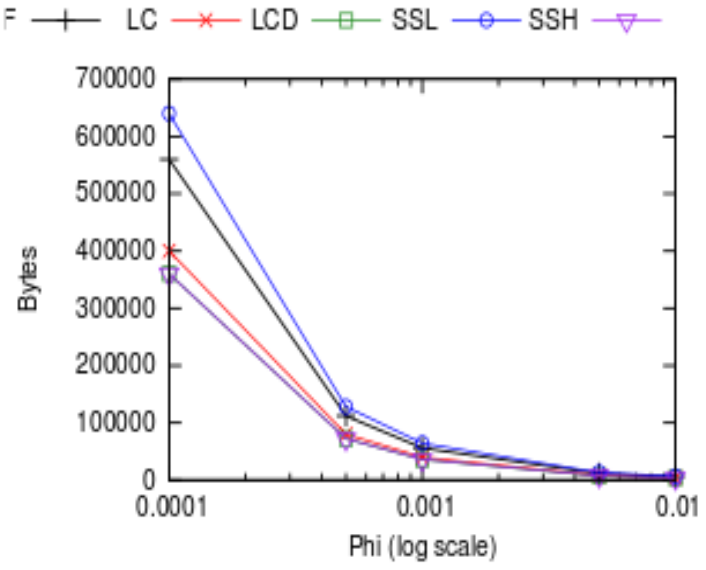
- All the experimental results have been from the journal Paper : Methods for finding frequent items in Data Streams.
- All the algorithms have been tested under similar parameters with skew factor = 1.0, phi = 0.001
- More Skewness -> less More frequent items -> easy to distinguish and less time to handle ->Increasing Precision(less false positives)
- More phi (frequency threshold) -> less More Frequent Items -> less space required and less time to handle ->Increasing Precision(less false positives)



(a) Zipf: Speed vs. Skew.



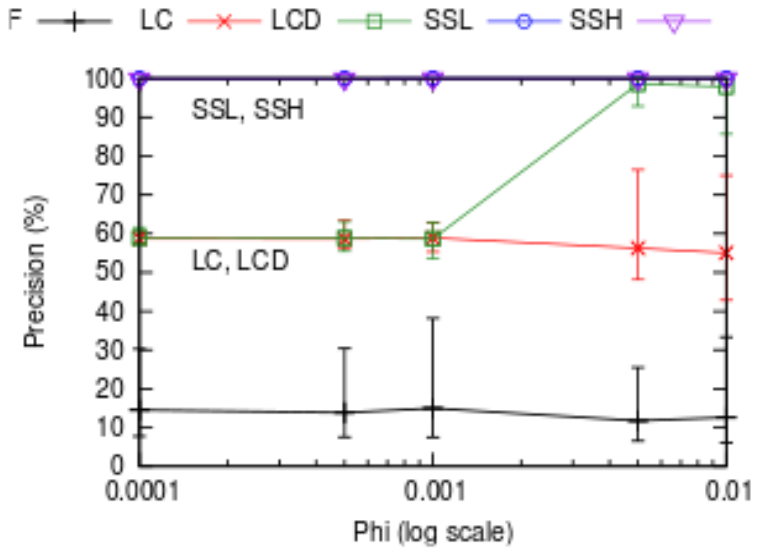
(b) Zipf: Speed vs. ϕ.



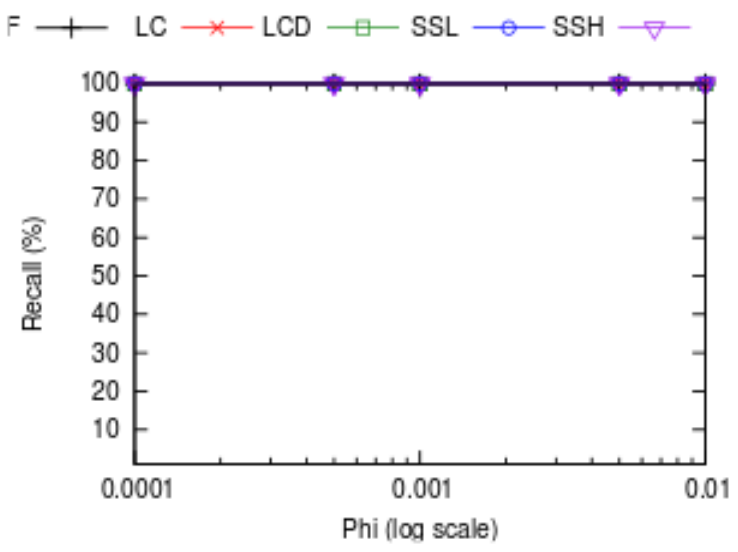
(c) Zipf: Size vs. ϕ.

Which is Better? (Accuracy?)

- SSH and SSL have full precision since they eliminate only the item with minimum count aggressively, it comes at the cost using ordered lists or Heap to find minimum frequent element.
- Thus, **SSH** exhibits very good performance in practice. It yields very good estimates, typically achieving 100% recall and precision, consumes very small space (than SSL), and is fairly fast to update (faster than LC and LCD)



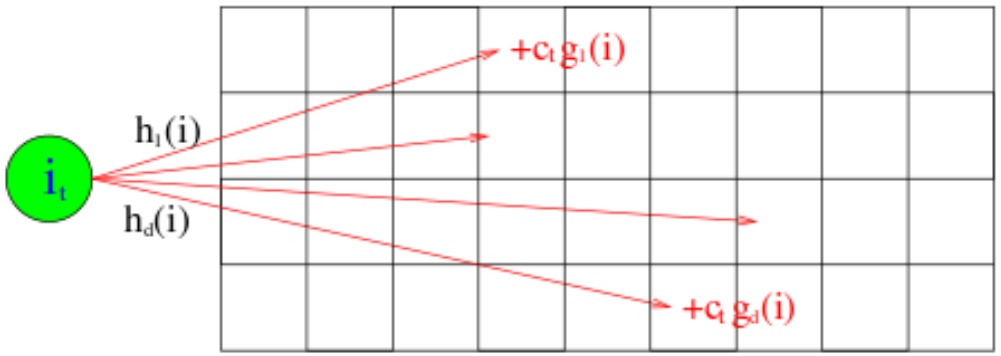
(g) Zipf: Precision vs. ϕ.



(e) Zipf: Recall vs. ϕ.

Count-Min Sketch

- Introduced in 2003 by Graham Cormode and S. Muthukrishnan.
- It is Data Structure which renders useful information about data streams with sublinear space.
- Data Structure's accuracy can be tuned by two parameters ε and δ.
- Data Stream is treated a vector and every element in vector is mapped to d cells in matrix of w x d dimensions using d pairwise-independent functions. For every element read Ct is added to d cell values which element maps to. In maintaining count perspective Ct = 1.
- To get count of any element x we look through all the d cell it maps to and find the minimum of all as minimum value will be more closer to actual count as many elements will be mapping to same cells.



Will it help to our need?

- To find frequently accessed elements, paper proposes to maintain small heap to maintain elements whose lowest count among all counts found from matrix is above threshold value.
- Threshold value can be tuned inorder to only have n(customized) number of elements in the elements.

Finally SSH Vs Count-Min Sketch ? Need to be found!!

Future Work

- Implementation of finalized Algorithm(SSH or Count-Min) to find frequent elements and using them to splay.
- Analyzing the results against current policies.
- Finding better way to splay at variable intervals.
  - Currently we have the idea of using Sedgewick's gap sequence formula to generate gaps using geometric progression in mind.

$4^k + 3 \cdot 2^{k-1} + 1$ generates $1, 8, 23, 77, 281, \ldots$

Thank you!