[Technical Report] Benchmarking Sketch-Based Data Stream Algorithms

[Experiments and Analyses]

Chenxingyu Zhao[1], Tong Yang[1], Yucheng Huang[1], Yinda Zhang[1]
Qianhui Liu[1], Bin Cui[1], Tilman Wolf[2]
[1]Department of Computer Science, Peking University
[2]Department of Electrical and Computer Engineering, University of Massachusetts Amherst
{dkzcxy, lqianhui, bin.cui}@pku.edu.cn, wolf@umass.edu
{yangtongemail, huangyc96, hgdkgszyd}@gmail.com

1. PROGRAMMING MODEL

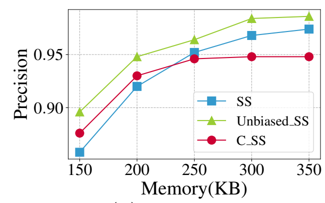
We describe the programming model using c++ style *class* shown as the code block *Class Sketch*. The class *Sketch()* has a member *Sketches* which is an probabilistic data structure (PDS) used by the algorithm. And the class has one *Update* function to update *Sketches* when a new item arrives, and a set of task-related *Query* functions to answer queries over data streams.

```
class Sketch()
{
   //probabilistic data structures
   PDS sketches;
   //customized parameters
   parameter phi,epsilon,...;
   //other customized members
   hash_function Hash();
   hash_table HT;
   auxiliary_data_structure heap;
   ......
public:
   //To update sketches
   void Update(string Key,int V);
   //To report estimated frequency
   int frequencyQuery(string Key);
   //To report topk items
   vector< KV pair > topkQuery(int k);
   ...
   //other query functions
}
```

***DSAB* Programming Model for Users**

2. COLDFILTER-SPACESAVING

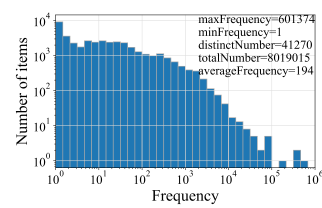
For the comparison between M-P algorithms and A-F algorithms, we compare SS and C-SS.



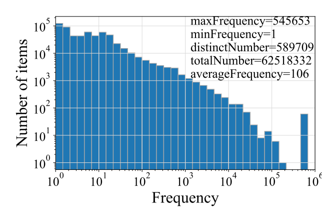
(a) IPtrace

Figure 11: SS augmented by Cold-Filter.

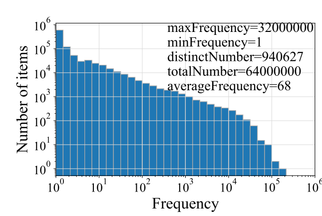
3. DATASET DISTRIBUTION



(a) kosarak



(b) IPtrace



(c) webdocs

Figure 11: Distribution of Realworld Datasets

4. PSUEDOCODE

Algorithm 1: CM

```
array of counters C[d][w]; //initially 0
d : the number of hash functions;
w : the number of per-row counters;
Function Update(Key, V):
    for i ← 1 to d do
        C[i][h_i(Key)] ← C[i][h_i(Key)] + V;
    if have heap to maintain top-k then
        update heap
```

Algorithm 2: CM-CU

```
array of counters C[d][w]; //initially 0
d : the number of hash functions;
w : the number of per-row counters;
Function Update(Key, V):
    minFreq ← min_{1≤j≤d}{C[j][h_j(Key)]};
    for i ← 1 to d do
        if C[i][h_i(Key)] == minFreq then
            C[i][h_i(Key)] ← C[i][h_i(Key)] + V;
    if have heap to maintain top-k then
        update heap
```

Algorithm 3: Count

```
array of counters C[d][w]; //initially 0
w : the number of per-row counters;
d hash functions : h_1, h_2, ..., h_d;//h_i : Key → [1 : w]
d hash functions : g_1, g_2, ..., g_d;//g_i : Key → +1, −1
Function Update(Key, V):
    for i ← 1 to d do
        C[i][h_i(Key)] ← C[i][h_i(Key)] + V × g_i(Key);
    if have heap to maintain top-k then
        update heap
```

Algorithm 4: Space Saving and Variants

```
Stream-Summary SS; // list of (item, counter) pairs
c : the capacity of SS;
p : the probability of replacing minimal item;
Function Update(Key, V):
    if item Key in SS then
        SS[Key].counter ← SS[Key].counter + V;
    else if SS.size < c then
        SS.add((Key, V));
    else
        (Key_min, U_min) ←
          the pair with the minimal counter;
        Increment U_min ← U_min + V;
        Replace key_min ← Key with the probability p ;
```

Algorithm 5: Lossy-Counting

```
set of entries E; // entry form (item, counter, Δ)
w : width of window;
n : number of items (not distinct);//initially 0
Function Update(Key, V):
    n ← n + 1;
    if item Key in E then
        E[Key].counter ← E[Key].counter + V;
    else
        E ← E + (Key, 1, ⌊n/w⌋);
    if n ≡ 0 mod w then
        for each entry e in E do
            if e.counter ≤ n/w then
                E ← E − e;
```

Algorithm 6: HeavyGuardian

```
hash table HT;
hash function h(.), g(.);
HT[h(.)].heavypart : array of KV pair (item, counter);
HT[h(.)].lightpart : array of counters;
Function Update(Key, V):
    if item Key in HT[h(Key)].heavypart then
        update the KV pair of Key
    else if HT[h(Key)].heavypart has empty cell then
        HT[h(Key)] ← HT[h(Key)] + (Key, V);
    else
        Exponential Decay;
        if HT[h(Key)].heavypart has zero-counter cell then
            HT[h(Key)] ← HT[h(Key)] + (Key, V);
        else
            HT[h(Key)][g(Key)]_l ←
              HT[h(Key)][g(Key)]_l + V;
```

Algorithm 7: ASketch

```
filter F;//array of (item, old_count, new_count)
classical sketch S_c;
Function Update(Key, V):
    if item Key in F then
        F[Key].new_count ← F[Key].new_count + V;
    else if F not full then
        F ← F + (Key, 0, V);
    else
        S_c.insert(Key, V);
        V_{S_c} ← S_c.FrequencyQuery(Key) ;
        if V_{S_c} > minF.new_count then
            K_min ← minimum new_count item in F;
            if F[K_min].new_count − F[K_min].old_count > 0
              then
                S_c.insert(K_min, F[K_min].new_count −
                  F[K_min].old_count);
                F ← F − (K_min, old_count, new_count);
            F ← F + (Key, V_{S_c}, V_{S_c});
```

Algorithm 8: UNIVMON

d hash functions : $h_1(.), h_2(.), ..., h_d(,)$;
$//h_i(.) : item \rightarrow \{0, 1\}$
classical sketches $S_c[d]$;
Function `Update`$(Key,\ V)$**:**
  $S_c[i].insert(Key, V)$;
  for $i \leftarrow 2$ *to* d **do**
    if $h_i(Key) == 1$ **then**
      $S_c[i].insert(Key, V)$;
    else
      break;

Algorithm 9: COLD-FILTER

classical sketch S_c;
d_1 hash functions : $h_i()$; d_2 hash functions : $g_j()$;
two layers of the filter: $L_1[\omega_1]$, $L_2[\omega_2]$;
ω_1, ω_2 : number of counters;
Function `Update`$(Key,\ V)$**:**
  for $i \leftarrow 1$ *to* d_1 **do**
    update the corresponding counters $L_1[h_i(Key)]]$
  if $L_1[h_i(Key)]$ *concurrently overflow* **then**
    for $j \leftarrow 1$ *to* d_2 **do**
      update the corresponding counters $L_2[g_j(Key)]$
    if $L_2[g_j(Key)]$ *concurrently overflow* **then**
      S_c.`Update`(Key, V);

Algorithm 10: PYRAMID

classical sketch S_c;
pyramid layers of counters $L[n_L]$;
n_L : number of layers in pyramid ;
Function `Update`$(Key,\ V)$**:**
  $S_c.insert(Key, V)$;
  $Boolean\ carryin \leftarrow False$;
  if *counter overflow happens during* S_c.`Update`(Key, V) **then**
    $carryin \leftarrow True$;
  $i \leftarrow 1$;
  while *carryyin* **do**
    $L[i].increment$;
    if $counter\ overflow\ happens\ during\ L[i].increment$ **then**
      $carryin \leftarrow Ture$;
    else
      $carryin \leftarrow False$