World Wide Web
https://doi.org/10.1007/s11280-018-0546-5

FID-sketch: an accurate sketch to store frequencies in data streams

Tong Yang[1] · Haowei Zhang[1] · Hao Wang[1] · Muhammad Shahzad[2] · Xue Liu[3] · Qin Xin[4] · Xiaoming Li[1]

Received: 7 December 2017 / Revised: 21 February 2018 / Accepted: 12 March 2018

Abstract Sketches are being extensively used in a large number of real world applications to estimate frequencies of data items. Due to the unprecedented increase in the amount of Internet data and a relatively slower increase in the size of on-chip memories, existing sketches are becoming increasingly unable to keep the accuracy of the frequency estimates

Tong Yang and Haowei Zhang are the co-primary authors. Haowei Zhang and Hao Wang finished this work under the guidance of their supervisor: Tong Yang.

This article belongs to the Topical Collection: *Special Issue on Web and Big Data*
Guest Editors: Junjie Yao, Bin Cui, Christian S. Jensen, and Zhe Zhao

✉ Xue Liu
liux@dsp.ac.cn

Tong Yang
yangtongemail@gmail.com

Haowei Zhang
zhanghw.alpq@pku.edu.cn

Hao Wang
wanghao1996@pku.edu.cn

Muhammad Shahzad
mshahza@ncsu.edu

Qin Xin
xin11@mails.tsinghua.edu.cn

Xiaoming Li
lxm@pku.edu.cn

[1] Peking University, Haidian Qu, China

[2] North Carolina State University, Raleigh, USA

[3] Institute of Acoustics, Chinese Academy of Science, Beijing, China

[4] Tsinghua University, Haidian Qu, China

Published online: 11 April 2018

at an acceptable level. In this paper, we design a new sketch, called FID-sketch, that has a significantly higher accuracy and a much smaller on-chip memory footprint compared to the existing sketches. The key intuition behind the design of the FID-sketch is that before inserting an item, unlike prior sketches, it first estimates the current value of the frequency of that item stored in the sketch, and then increments as few counters as possible instead of incrementing a pre-determined fixed number of counters. We carried out extensive experiments to evaluate and compare the performance of FID-sketch with existing sketches on multi-core CPU and GPU platforms. Our experimental results show that our FID-sketch significantly outperforms the state-of-the-art with 36.7 *times* lower relative error. We have released the source code of our proposed sketch and other related sketches that we implemented at Github [21].

Keywords Sketch · Data streams · Accuracy · Speed · Measurement

1 Introduction

1.1 Background and Motivation

A *sketch* is a probabilistic data structure that stores frequencies of items in a multiset and returns an estimate of the frequency of any given item when requested. A multiset is a set in which one item may appear more than once. A sketch is associated with three operations: insertion, deletion, and query. *Insertion* of an item into the sketch updates the sketch such that the stored value of the frequency of that item in the sketch is increased by 1. The stored value of the frequency of an item that has never been inserted before is 0. *Deletion* of an item from the sketch updates the sketch such that the stored value of the frequency of that item in the sketch is decreased by 1. *Query*ing the frequency of an item in the sketch refers to the operation of processing the sketch and returning an estimate of the current frequency of that item. The accuracy of the frequency estimate of an item returned by the sketch is measured in terms of the *absolute relative error*.

The on-chip memory is fast memory which can be read and written very fast (such as L2 or L3 caches in CPU and GPU chips, and Block RAM in FPGA chips). The fast memory is expensive and limited in size, and therefore it is often used to store compact data structures with high speed requirements. On the contrary, the off-chip memory is slow memory whose speed of reading and writing is about ten times slower than that of the on-chip memory (such as Dynamic RAM, DRAM) [22]. Due to the advantages of large size and low cost, the off-chip memory is often used to store large data structures without high speed requirements.

Sketches are being extensively used in a broad spectrum of applications in big data scenarios. These applications include sharing web caches [15], tracking flows in network traffic [3, 7, 10], detecting attacks in routers [2], detecting network anomalies [4], finding frequent items (such as heavy hitters) [13, 25, 26], aggregating statistics in sensor networks[16, 28], and processing distributed data sets [23, 24, 27]. Large networking corporations such as AT&T [12] and Google [17] also use sketches for network monitoring and network data management. With more and more devices connecting to the Internet and causing an increase in the amount of Internet data at an unprecedented rate (total internet traffic will surpass 1ZB in 2016 and increase to 2.3ZB by 2020 [8]), existing sketches can hardly keep the accuracy of the frequency estimates at acceptable levels due to the lack of large enough fast memory. The reason is that with the increase in the number of items and their frequencies in a multiset, sketches need more memory to accurately store and estimate frequencies.

As sketches are often used in settings where they are queried at a high rate, they need to be implemented in the fast on-chip memory . Unfortunately, due to the the on-chip memory is very limited. Therefore, with the increasing amounts of data, existing sketches are not getting enough on-chip memory to store frequencies accurately. Motivated by this observation, in this paper, we design a new sketch that has a significantly smaller on-chip memory footprint and a much higher accuracy compared to existing sketches.

1.2 Insight from prior art

Charikar et al. proposed Count-sketch, which marks the pioneering work in sketches [6]. Estimates from Count-sketch incur two types of errors: over-estimation error, where the frequency estimate of an item is larger than the true frequency of that item, and under-estimation error, where the frequency estimate of an item is smaller than the true frequency of that item. To improve on the Count-sketch, Cormode and Muthukrishnan proposed Count-Min (CM) sketch [11]. The key improvement of the CM-sketch over the Count-sketch is that the estimates from CM-sketch suffer only from the over-estimation error and not the under-estimation error. Until recently, CM-sketch has been a popular choice due to its relatively small memory footprint and high accuracy.

A CM-sketch consists of d arrays and each array consists of w counters, as shown in Figure 1. We represent the i^{th} array of the CM-sketch with $\mathbf{A}_i$ and the j^{th} counter of this i^{th} array with $A_i[j]$, where $0 \leqslant i \leqslant d-1$ and $0 \leqslant j \leqslant w-1$. Each array $\mathbf{A}_i$ is associated with an independent hash function $h_i(.)$ that has a uniformly distributed output. Before the CM-sketch starts storing frequencies, all counters are first initialized to 0. To insert or delete an item e from the CM-sketch, *i.e.*, to increment or decrement the stored frequency of an item e, the sketch first computes the d hash functions $h_1(e), h_2(e), \ldots, h_d(e)$ and then increments or decrements, respectively, all counters $A_1[h_1(e)\%w], A_2[h_2(e)\%w], \ldots, A_d[h_d(e)\%w]$ by 1. To respond to a query requesting an estimate of the current stored frequency of an item e in the CM-sketch, the sketch first computes the d hash functions $h_1(e), h_2(e), \ldots, h_d(e)$ and then returns the value of the smallest counter(s) among $A_1[h_1(e)\%w], A_2[h_2(e)\%w], \ldots, A_d[h_d(e)\%w]$ as the frequency of the item. By carefully selecting the values of d and w based on the distribution of items in the data stream, the over-estimation error of CM-sketch can be bounded.

A key observation from our discussion of the CM-sketch is that for each insertion operation, the CM-sketch always increments d counters. In many cases, however, this is not only unnecessary but also detrimental to the accuracy of the CM-sketch. We explain this with an example. Suppose an item e has already been inserted $x-1$ times and needs to be inserted the x^{th} time into a CM-sketch. Suppose that the values of all the d counters that the d hash functions map the item e to are already greater than x (this happens due to the insertions of other items in the sketch). A CM-sketch would increment all these counters by one, which

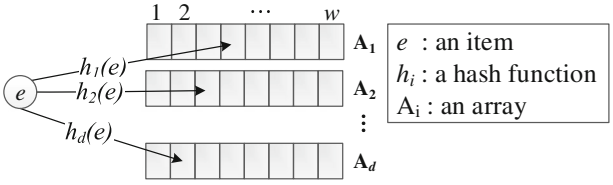

Figure 1 Count-Min sketch

will make the over-estimation error worse when querying this item e. If CM-sketch were not to increment the counters whose values are larger than the current actual frequency of the item being inserted, it would result in two advantages: 1) the accuracy of CM-sketch would improve; 2) as the sketch would be incrementing fewer counters, it would require counters with smaller size. An even better strategy is to increment only the smallest of all counters as long as the smallest counter(s) are less than the current actual frequency of the item being inserted. This will further reduce the over-estimation error and the memory footprint of the sketch without introducing any under-estimation errors.

Estan and Varghese proposed a variant of CM-sketch, called Conservative Update (CU) sketch, which is exactly the same as CM-sketch in architecture, *i.e.*, has d arrays of w counters each. CU-sketch, to some extent, incorporates our key insight of not incrementing counters greater than the current actual frequency of the item [14]. More specifically, to insert an item e in CU-sketch, the sketch increments only the smallest counter(s) among the d counters that the d hash functions map the item e to, but it does not take the current frequency of the item into account. The fundamental limitation of the CU-sketch is that it does not support deletions because to support deletions, one needs to keep track of the counters that are incremented at each insertion, but the CU-sketch does not perform such a tracking. If we apply the deletion process of CM-sketch on the CU-sketch, *i.e.*, first compute the d hash functions $h_1(e), h_2(e), \ldots, h_d(e)$ and then decrement the d counters $A_1[h_1(e)\%w], A_2[h_2(e)\%w], \ldots, A_d[h_d(e)\%w]$ by 1, subsequent query results from the resulting CU-sketch will be prone to having both under-estimation and over-estimation errors. As CU-sketch does not support deletions, it has not received as wide an acceptance in practice as the CM-sketch.

1.3 Proposed approach

In this paper, we propose a new sketch, called FID-sketch, which achieves significantly higher accuracy and smaller on-chip memory footprint compared to the existing sketches while at the same time supports all operations that the popular CM-sketch does, *i.e.,* insertions, deletions, and queries. FID-sketch does not suffer from under-estimation error. However, it does suffer from over-estimation error just like the CM-sketch, but its over-estimation error is much smaller compared to the CM-sketch. The key intuition behind the design of the FID-sketch is that before inserting an item, it first estimates the current frequency of that item, and then increments the smallest counter(s) only if the value of that smallest counter(s) is less than the current frequency estimate of that item. This significantly reduces the over-estimation error as well as the on-chip memory footprint.

Our proposed FID-sketch is comprised of 3 subsketches: one on-chip subsketch namely **F**ast-Query (FQ) subsketch and two off-chip subsketches namely **I**nsertion-Support (IS) subsketch and **D**eletion-Support (DS) subsketch; hence, the name FID-sketch. The motivation behind keeping the on-chip FQ-subsketch is to make the querying process of FID-sketch extremely fast. FID-sketch stores all information required to return the currently stored frequency of any given item in this FQ-subsketch. As the on-chip memory is limited, the memory footprint of the FQ-subsketch must be small. To achieve this, FID-sketch maintains the off-chip IS-subsketch, which helps determine whether the smallest counter(s) in the FQ-subsketch should be incremented or not when inserting a given item. As the number of counters updated in the FQ-subsketch per insertion is small (sometimes even none), the over-estimation error in the query result from the FQ-subsketch is also small. The motivation behind keeping the off-chip DS-subsketch is to enable deletions from the FQ-subsketch without causing any under-estimation errors in the subsequent query results. We would like

to clarify here that "FID-sketch" is the name we have given to our proposed scheme of using the three subsketches to store frequencies. The arrays and counters that are operated upon when inserting, deleting, and querying actually belong to the three subsketches.

The highlight of our proposed FID-sketch is that its querying operation is very fast and accurate. Unfortunately, it comes at the cost of slightly slower insertions and deletions. Our motivation behind trading-off the speeds of insertions and deletions for faster and more accurate querying is the fact that for a large number of modern applications, such as network monitoring, decision support systems, online data analysis, and e-commerce advertising engines, accurate and near real-time querying is critical because query speed directly affects the user experience and the performance of real-time applications [5, 19]. In these and many other modern applications, the insertion and deletion speeds are not as critical as the query speed because insertions and deletions are relatively few or can be processed in the background. Our proposed FID-sketch primarily targets such applications where high query speed and accuracy are the most critical factors.

1.4 Advantages over prior art and key contributions

FID-sketch is advantageous over CM-sketch because unlike CM-sketch, it increments fewer than d counters per insertion in the on-chip memory, and thus achieves significantly lower over-estimation error and significantly smaller on-chip memory footprint. FID-sketch is also advantageous over CU-sketch because it supports deletions and achieves better accuracy. In this paper, we make following three key contributions.

1. We propose a new sketch, namely the FID-sketch, which is based on the insights we developed from prior art, specifically the CM-sketch. FID-sketch achieves a much higher accuracy compared to the existing sketches and at the same time utilizes significantly less on-chip memory.
2. We carried out extensive experiments to evaluate and compare the performance of FID-sketch with existing schemes on multi-core CPU and GPU platforms. Our results show that FID-sketch significantly outperforms the state-of-the-art with *36.7 times* smaller error.
3. We have anonymously released the source code of FID and related sketches that we implemented at Github [21].

2 Related work

Charikar et al. did the pioneering work in sketches and proposed the Count-sketch [6] (C-sketch). The structure of the Count-sketch is exactly the same as CM-sketch except that each array $\mathbf{A}_i$ is associated with two hash functions $h_i(.)$ and $g_i(.)$. Each hash function $h_i(.)$ is uniformly distributed over the set of positive integers, whereas the hash functions $g_i(.)$ evaluates to -1 or +1 with equal probability. Any pair of hash functions $h_i(.)$ and $h_j(.)$, where $i \neq j$, are pairwise independent. Similarly, hash functions $g_i(.)$ and $g_j(.)$, where $i \neq j$, are also pairwise independent. The hash function $h_i(.)$ maps any given item to one of the w counters in the array $\mathbf{A}_i$ whereas the hash function $g_i(.)$ maps the item to either -1 or 1. To insert an item e, for all values of $i \in [0, w-1]$, Count-sketch calculates hash functions $h_i(e)$ and $g_i(e)$ and adds $g_i(e)$ to the counters $A_i[h_i(e)\%w]$. When querying the frequency of item e, Count-sketch reports the median of $\{A_1[h_1(e)] \times g_1(e), A_2[h_2(e)] \times g_2(e) \ldots A_d[h_d(e)] \times g_d(e)\}$ as an estimate of the frequency of the item e.

Unfortunately, Count-sketch suffers from both over-estimation and under-estimation errors. Therefore, several improvements have been proposed that do not suffer from the under-estimation errors but only suffer from the over-estimation errors, such as the CM-sketch [11] and CU-sketch [14] described earlier in Section 1.2. Count-Min-log (CML) sketch is a variant of the CU-sketch that uses logarithm-based approximate counters instead of linear counters [18]. It achieves better accuracy but suffers from both over-estimation error and under-estimation error, and cannot support deletion.

Another class of data structures that can be used to store frequencies of items is Bloom filters. Let $h_1^B(.), h_2^B(.), \cdots, h_k^B(.)$ be k independent hash functions with uniformly distributed outputs. Given a multiset of items, a Bloom filter (BF) first constructs an array B of m bits, where each bit is initialized to 0. To insert an item e of the multiset, the BF sets the k bits $B[h_1(e)\%m], \cdots, B[h_k(e)\%m]$ to 1. To process a query whether e is in the multiset, BF returns true if all corresponding k bits are 1 (*i.e.*, returns $\wedge_{i=1}^{k} B[h_i(e)\%m]$). A Bloom filter can identify whether an item is present in a multiset or not but ***cannot*** estimate its frequency.

Several updates to the conventional Bloom filters have been proposed, such as Spectral Bloom Filters (SBF) [9] and Dynamic Count Filters (DCF) [1], which can store frequencies of items. SBF replaces each bit in the conventional Bloom filter with a counter [9]. To insert an item e, SBF simply increments all the counters that the hash functions $h_1^B(e), \cdots, h_k^B(e)$ map it to. On querying for an item, SBF reports the value of the smallest counter(s) among all the counter to which the hash functions map the item to as the estimate of the frequency of that item in the multiset. DCF extends the concept of SBF while improving the memory efficiency of SBF by using two separate filters [1]. The first filter is comprised of fixed size counters while the size of counters in the second filter is dynamically adjusted. The use of two filters, unfortunately, increases the complexity of DCF, which degrades its query and update performance.

3 FID-sketch

In designing FID-sketch, we have three objectives: small on-chip memory footprint, small over-estimation error, and support of deletions. FID-sketch achieves these objectives by maintaining three subsketches: a small on-chip subsketch namely the FQ-subsketch, and two relatively larger off-chip subsketches namely the IS-subsketch and the DS-subsketch. The on-chip FQ-subsketch contains all information required to answer any query. Neither of the two off-chip subsketches are consulted in answering the queries. They are consulted only when inserting and deleting items. Next, we describe these three subsketches in detail along with the motivations behind their designs.

3.1 FQ-subsketch

Similar to a CM-sketch, an FQ-subsketch consists of d_{FQ} arrays and each array consists of w_{FQ} counters. We represent the i^{th} array of the FQ-subsketch with $\mathbf{F}_i$ and the j^{th} counter of this i^{th} array with $F_i[j]$, where $0 \leqslant i \leqslant d_{FQ} - 1$ and $0 \leqslant j \leqslant w_{FQ} - 1$. Each array $\mathbf{F}_i$ is associated with an independent hash function $h_i(.)$ that has a uniformly distributed output. Before the FQ-subsketch starts storing frequencies, all counters are initialized to 0. Next, we explain the insertion, deletion, and query operations of the FQ-subsketch.

Insertion To insert an item e in the FQ-subsketch, FID-sketch first computes the d_{FQ} hash functions $h_i(e)$ and identifies the smallest counter(s) among the d_{FQ} counters $F_i[h_i(e)\%w_{FQ}]$, where $0 \leqslant i \leqslant d_{FQ}-1$. It then inserts e into the IS-subsketch and obtains a threshold T_{ins} from the IS-subsketch. We will describe how T_{ins} is calculated from the IS-subsketch in detail in Section 3.2. For now, it suffices to state that this threshold is essentially an estimate of the current frequency of the item e in the IS-subsketch. If the smallest counter(s) among the d_{FQ} counters of the FQ-subsketch has(ve) a value that is not less than T_{ins}, these d_{FQ} counters should not be incremented because incrementing them will only increase the over-estimation error in subsequent queries. Therefore, when inserting e, FID-sketch increments the smallest counter(s) among the d_{FQ} counters of the FQ-subsketch only when their value is less than T_{ins}. Note that incrementing only the smallest counter(s) among these d_{FQ} counters (unlike CM-sketch, which would increment all d_{FQ} counters) leads to a smaller over-estimation error compared to CM-sketch. We append '(s)' with 'counter' in the sentences above because it is possible for more than one counter to attain the smallest value among the d_{FQ} counters.

Deletion To delete an item e from the FQ-subsketch, FID-sketch first computes the d_{FQ} hash functions $h_i(e)$ and identifies the d_{FQ} counters $F_i[h_i(e)\%w_{FQ}]$, where $0 \leqslant i \leqslant d_{FQ}-1$. After that, it consults the DS-subsketch to determine which of these d_{FQ} counters can be decremented without causing any under-estimation errors, and then decrements all such counters. In Section 3.3, we will explain how DS-subsketch helps in determining which of these d_{FQ} counters to decrement.

Query The query operation on the FQ-subsketch is exactly the same as that of the CM-sketch. To respond to a query requesting the current stored frequency of an item e in the FQ-subsketch, FID-sketch first computes the d hash functions $h_0(e), h_1(e), \ldots, h_{d_{FQ}-1}(e)$ and then returns the value of the smallest counter(s) among $F_0[h_0(e)\%w_{FQ}]$, $F_1[h_1(e)\%w_{FQ}]$, $\ldots$, $F_{d_{FQ}-1}[h_{d_{FQ}-1}(e)\%w_{FQ}]$ as the frequency of the item.

3.2 IS-subsketch

IS-subsketch helps in determining whether the smallest counter(s) of the FQ-subsketch should be incremented when inserting an item or not by providing the estimate of the current frequency of that item in the sketch. IS-subsketch is essentially just an ordinary CM-sketch with d_{IS} arrays and each array consists of w_{IS} counters. We represent the i^{th} array of the IS-subsketch with $\mathbf{I}_i$ and the j^{th} counter of this i^{th} array with $I_i[j]$, where $0 \leqslant i \leqslant d_{IS}-1$ and $0 \leqslant j \leqslant w_{IS}-1$. Each array $\mathbf{I}_i$ is associated with an independent hash function $h_i(.)$ that has a uniformly distributed output. As the IS-subsketch resides in the off-chip memory, which is available in abundant quantity, it can have significantly more number of counters per array compared to the FQ-sketch, *i.e.*, $w_{IS} >> w_{FQ}$. The consequence of having such a large CM-sketch in the off-chip memory is that it can estimate the frequency of any given item with very small over-estimation error due to its large memory footprint. Thus, whenever a new item e needs to be inserted into the FQ-subsketch, we first insert it in the IS-subsketch and then get an accurate estimate of the current frequency of that item e in the IS-subsketch. We use this estimate as the value of the threshold T_{ins}, which, in turn, is used in determining whether the smallest counter(s) in the FQ-subsketch need(s) to be incremented or not.

Note that the IS-subsketch is never used in answering queries from external applications. Those queries are answered entirely from the on-chip FQ-subsketch, and thus, the query operation still stays very fast. The overall insertion operation, however, has more overhead compared to the conventional CM-sketch because when inserting an item e, FID-sketch now has to perform three steps: 1) inserting the item into the IS-subsketch, 2) calculating the value of T_{ins}, and 3) using T_{ins} to determine whether or not to increment the smallest counter(s) of the FQ-subsketch. Fortunately, this larger overhead is not a problem because, as discussed at the end of Section 1.3, the insertion speed is not as critical as the query speed in many modern applications. Next, we formally describe the insertion, deletion, and query operations of the IS-subsketch.

Insertion To insert an item e in IS-subsketch, FID-sketch first compute the d_{IS} hash functions $h_i(e)$ and then increments all counters $I_i[h_i(e)\%w_{IS}]$ by 1, where $0 \leqslant i \leqslant d_{IS} - 1$.

Deletion To delete an item e from IS-subsketch, FID-sketch first computes the d_{IS} hash functions $h_i(e)$ and then decrements all counters $I_i[h_i(e)\%w_{IS}]$ by 1, where $0 \leqslant i \leqslant d_{IS} - 1$.

Query To execute the query requesting the current stored frequency of an item e in the IS-sketch, FID-sketch first computes the d_{IS} hash functions $h_0(e), h_1(e), \ldots, h_{d_{IS}-1}(e)$ and then returns the value of the smallest counter(s) among $I_0[h_0(e)\%w_{IS}]$, $I_1[h_1(e)\%w_{IS}]$, $\ldots$, $I_{d_{IS}-1}[h_{d_{IS}-1}(e)\%w_{IS}]$ as the frequency of the item e. We emphasize here again that the IS-subsketch is queried only to calculate the value of T_{ins} when an item needs to be inserted into the FQ-subsketch, and this query is generated by the FID-sketch itself, not by any external application. FID-sketch answers queries from external applications entirely from the FQ-subsketch.

3.3 DS-subsketch

DS-subsketch helps in determining which counters of the FQ-subsketch should be decremented when deleting an item without causing any under-estimation errors. It is challenging to delete items from the FQ-subsketch for two reasons. First, one needs to keep track of the exact counters that are incremented on each insertion. Second, to delete a given item e, one cannot simply decrement the counters that were incremented during the most recent insertion of e because the values of the counters that are incremented during the latest insertion of e determine which counters will be incremented during subsequent insertions of any other items. If we decrement any of the counters that were incremented during the most recent insertion of this item e, there is a risk of under-estimation error in subsequent queries. We demonstrate the existence of these two problems with the help of a simple example.

3.3.1 The deletion problem

Consider an FQ-subsketch with $d_{FQ} = 2$ and $w_{FQ} = 2$, *i.e.*, this FQ-subsketch consists of 2 arrays $\mathbf{F}_0$ and $\mathbf{F}_1$ with hash functions $h_0(.)$ and $h_1(.)$, respectively, and each array consists of two counters, which are all initialized to 0. We insert three items e_1, e_2, and e_3 into this FQ-subsketch in this sequence. Suppose $h_0(e_1) = 0$ and $h_1(e_1) = 0$, $h_0(e_2) = 0$ and $h_1(e_2) = 1$, and $h_0(e_3) = 1$ and $h_1(e_3) = 1$. Figure 2 shows snapshots of this FQ-subsketch before and after each insertion. To support the insertions of e_1, e_2, and e_3 into

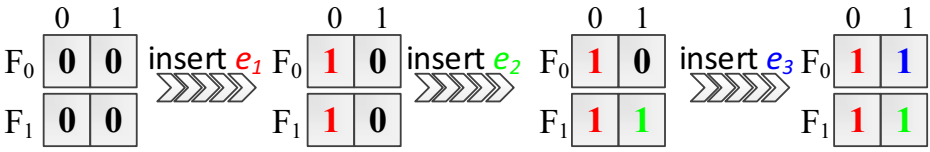

Figure 2 An example of the deletion problem

this FQ-subsketch, we also maintain an IS-subsketch. However, to keep the discussion of this example concise, we do not explicitly describe the operations on the IS-subsketch when inserting these three items; those operations will be self-evident from the text.

To insert e_1, we first insert it into the IS-subsketch, and obtain the value of the threshold T_{ins} from it for this item. We then check the values of the two counters $F_0[h_0(e_1)\%2]$ and $F_1[h_1(e_1)\%2]$, *i.e.*, $F_0[0]$ and $F_1[0]$, which are both currently 0 and thus, certainly less than T_{ins} because T_{ins} is always greater than 0 (T_{ins} is calculated after inserting e in the IS-subsketch). As both counters are smallest and also $< T_{ins}$, we increment them both by 1. To insert e_2, we first insert it into the IS-subsketch, and obtain the value of the threshold T_{ins} from it for this item. We then check the values of the two counters $F_0[h_0(e_2)\%2]$ and $F_1[h_1(e_2)\%2]$, *i.e.*, $F_0[0]$ and $F_1[1]$ and pick the one with the smallest value, which in this case is $F_1[1]$. As the current value of $F_1[1]$ is 0, it is also certainly $< T_{ins}$, and thus we increment it by 1. To insert e_3, we first insert it into the IS-subsketch, and obtain the value of the threshold T_{ins} from it for this item. We then check the values of the two counters $F_0[h_0(e_3)\%2]$ and $F_1[h_1(e_3)\%2]$, *i.e.*, $F_0[1]$ and $F_1[1]$ and pick the one with the smallest value, which in this case is $F_0[1]$. As the current value of $F_0[1]$ is 0, it is also certainly $< T_{ins}$, and thus we increment it by 1.

At this point, let us look at what would happen if we were to delete e_2 from the FQ-subsketch, *i.e.*, reduce the value of its stored frequency in the FQ-subsketch by 1. The hash functions $h_0(e_2)$ and $h_1(e_2)$ map e_2 to the counters $F_0[0]$ and $F_1[1]$. Recall that when inserting e_2, we only incremented $F_1[1]$ and not $F_0[0]$. Therefore in deleting e_2, we must not decrement $F_0[0]$ because we never incremented it when inserting e_2. This shows that to delete any item, one needs to keep track of exactly which counters were incremented when inserting an item. Furthermore, we cannot decrement $F_1[1]$ either because its value after inserting e_2 determined which counters in the FQ-subsketch would be incremented in inserting e_3. If we decrement $F_1[1]$ when deleting e_2, a subsequent query for the frequency of e_3 would result in a returned value of 0 (*i.e.*, $\min\{F_0[h_0(e_3)\%2], F_1[h_1(e_3)\%2]\} = \min\{F_0[1], F_1[1]\} = 0$), whereas the actual frequency of e_3 currently is 1. This means that if we decrement $F_1[1]$ when deleting e_2, the subsequent queries will have under-estimation error, which we do not desire. This shows that even if we keep track of exactly which counters were incremented when inserting an item, we may not be able to decrement them when deleting that item due to the risk of under-estimation error in subsequent queries. Therefore, to delete e, one needs to delete all items in reverse order (from both FQ-subsketch and IS-subsketch) that were inserted after the latest insertion of e, decrement the counters that were incremented on the latest insertion of e, and then reinsert all subsequent items again (both in FQ-subsketch and IS-subsketch), which is a very expensive and slow process.

Solution Direction To enable deletions from the FQ-subsketch, our solution is to maintain another sketch, which we call the DS-subsketch. The DS-subsketch enables us to determine which of the d_{FQ} counters, $F_i[h_i(e)\%w_{FQ}]$ $(0 \leqslant i \leqslant d_{FQ} - 1)$, of the

FQ-subsketch can be decremented when deleting the item e without causing any under-estimation errors. Furthermore, it enables this without requiring us to maintain any information about the exact counters that were incremented when inserting the item e. This, however, comes at a small cost: the DS-subsketch does not support perfect deletions from the FQ-subsketch, *i.e.*, after decrementing some counters when deleting the x^{th} insertion of an item e from the FQ-subsketch, the resultant FQ-subsketch may not resemble an FQ-subsketch that would result if the item e was never inserted the x^{th} time. Nonetheless, the DS-subsketch assisted deletions still serve both primary objectives behind deletions, *i.e.*, reduce the over-estimation error and reduce the memory footprint of the sketch. The DS-subsketch can be of two types: 1) a standard CM-sketch, or 2) a Subtraction-sketch, which is obtained by *subtracting* the counters of FQ-subsketch from the corresponding counters of the standard CM-sketch. Next, we first describe the DS-subsketch of type CM-sketch and discuss the room for improvement in it. After that we describe the final version, DS-subsketch of type Subtraction-sketch, which achieves those improvements.

3.3.2 DS-subsketch of type CM-sketch

This type of DS-subsketch is essentially just an ordinary CM-sketch with d_{FQ} arrays and w_{FQ} counters per array, which equal those in the FQ-subsketch. We represent the i^{th} array of this DS-subsketch with $\mathbf{D}_i^{CM}$ and the j^{th} counter of this i^{th} array with $D_i^{CM}[j]$, where $0 \leqslant i \leqslant d_{FQ}-1$ and $0 \leqslant j \leqslant w_{FQ}-1$. Each array $\mathbf{D}_i^{CM}$ is associated with an independent hash function $h_i(.)$ that has a uniformly distributed output. The design of the DS-subsketch of type CM-sketch is based on the observation that given a multiset, if we build an FQ-subsketch and a DS-subsketch using equal number of arrays and counters per array, then every counter of the FQ-subsketch will always be smaller than or equal to the corresponding counter of the DS-subsketch, *i.e.*, $F_i[h_i(e)\%w_{FQ}] \leqslant D_i^{CM}[h_i(e)\%w_{FQ}]$, where $0 \leqslant i \leqslant d_{FQ}-1$. The reason is that whenever an item is inserted in both FQ-subsketch and the DS-subsketch, all counters $D_i^{CM}[h_i(e)\%w_{FQ}]$, where $0 \leqslant i \leqslant d_{FQ}-1$, are always incremented by 1, while not all of the corresponding counters $F_i[h_i(e)\%w_{FQ}]$ are always incremented. Next, we first describe the insertion and deletion operations of the DS-subsketch of this type, and then explain how it helps in identifying counters of FQ-subsketch that can be decremented when deleting an item. We never need to perform a query operation on DS-subsketch to estimate frequency of any item.

Insertion To insert an item e in the DS-subsketch, we first compute the d_{FQ} hash functions $h_i(e)$ and then increment all counters $D_i^{CM}[h_i(e)\%w_{FQ}]$ by 1, where $0 \leqslant i \leqslant d_{FQ}-1$.

Deletion To delete an item e from the DS-subsketch, we compute the d_{FQ} hash functions $h_i(e)$ and decrement all counters $D_i^{CM}[h_i(e)\%w_{FQ}]$ by 1, where $0 \leqslant i \leqslant d_{FQ}-1$.

Whenever an item e is inserted, the FID-sketch not only inserts it into the IS-subsketch and FQ-subsketch but also into the DS-subsketch. Whenever an item e needs to be deleted, the FID-sketch first deletes it from both the IS-subsketch and the DS-subsketch. After that, it compares the d_{FQ} counters $F_i[h_i(e)\%w_{FQ}]$ of the FQ-subsketch with the corresponding d_{FQ} counters $D_i^{CM}[h_i(e)\%w_{FQ}]$ of the DS-subsketch, and decrements all those counters of the FQ-subsketch for which $F_i[h_i(e)\%w_{FQ}] > D_i^{CM}[h_i(e)\%w_{FQ}]$. This steps ensures that every counter of the FQ-subsketch sketch is always smaller than or equal to the corresponding counter of the DS-subsketch, as discussed earlier.

Analysis When inserting an item, the FID-sketch increments d_{IS} counters of the IS-subsketch and d_{FQ} counters of the DS-subsketch of type CM-sketch. In the FQ-subsketch, in the worst case, it increments d_{FQ} counters, and in the best case, it increments no counters. When deleting an item, the FID-sketch decrements d_{IS} counters of the IS-subsketch and d_{FQ} counters of the DS-subsketch of type CM-sketch. In the FQ-subsketch, in the best case, it decrements d_{FQ} counters, and in the worst case, it decrements no counters. There is room for improvement to reduce the number of counter updates per insertion and deletion, as we will describe next.

3.3.3 DS-subsketch of type Subtraction-sketch

To reduce the number of counter updates per insertion and deletion, we introduce DS-subsketch of type Subtraction-sketch, which also contains d_{FQ} arrays and w_{FQ} counters per array, just like the FQ-subsketch. We represent the i^{th} array of this type of DS-subsketch with $\mathbf{D}_i^S$ and the j^{th} counter of this i^{th} array with $D_i^S[j]$, where $0 \leqslant i \leqslant d_{FQ} - 1$ and $0 \leqslant j \leqslant w_{FQ} - 1$. Each counter of the DS-subsketch of type Subtraction-sketch equals the difference between the corresponding counters of the DS-subsketch of type CM-sketch and the FQ-subsketch, *i.e.*, each counter $D_i^S[j]$ satisfies the equation $D_i^S[j] = D_i^{CM}[j] - F_i[j]$, where $0 \leqslant i \leqslant d_{FQ} - 1$ and $0 \leqslant j \leqslant w_{FQ} - 1$. Note that in order to maintain the DS-subsketch of type Subtraction-sketch, we do not need to first maintain a DS-subsketch of type CM-sketch and then subtract the FQ-subsketch from it. Instead, we modify the insertion and deletion operations of FID-sketch to directly maintain the DS-subsketch of type Subtraction-sketch. Next, we first describe the insertion and deletion operations of this type of DS-subsketch and then explain how it reduces the number of counter updates per insertion and deletion.

Insertion Whenever an item e is to be inserted, the FID-sketch first inserts it into the IS-subsketch and obtains the threshold T_{ins}, as described in Section 3.1. After that, the FID-sketch computes the d_{FQ} hash functions $h_i(e)$ and increments the smallest counter(s) among the d_{FQ} counters $F_i[h_i(e)\%w_{FQ}]$ if the smallest counter(s) are smaller than T_{ins}, as described in Section 3.2. Finally, the FID-sketch increments all corresponding counters in this DS-subsketch of type Subtraction-sketch that were not incremented in the FQ-subsketch, *i.e.*, if it did not increment $F_i[h_i(e)\%w_{FQ}]$, then it increments $D_i^S[h_i(e)\%w_{FQ}]$; otherwise, it leaves $D_i^S[h_i(e)\%w_{FQ}]$ as it is, where $0 \leqslant i \leqslant d_{FQ} - 1$. This last step ensures that each counter $D_i^S[j]$ of the DS-subsketch of type Subtraction-sketch satisfies the equation $D_i^S[j] = D_i^{CM}[j] - F_i[j]$ for all values of i and j.

Deletion Whenever an item e is to be deleted, the FID-sketch first deletes it from the IS-subsketch, as described in Section 3.1. After that, it computes the d_{FQ} hash functions $h_i(e)$ and decrements all non-zero counters among the d_{FQ} counters $D_i^S[h_i(e)\%w_{FQ}]$ of the DS-subsketch of type Subtraction-sketch. Finally, the FID-sketch decrements all corresponding counters in the FQ-subsketch that were not decremented in this DS-subsketch of type Subtraction-sketch, *i.e.*, if it did not decrement $D_i^S[h_i(e)\%w_{FQ}]$, then it decrements $F_i[h_i(e)\%w_{FQ}]$; otherwise, it leaves $F_i[h_i(e)\%w_{FQ}]$ as it is, where $0 \leqslant i \leqslant d_{FQ} - 1$. This last step again ensures that each counter $D_i^S[j]$ satisfies the equation $D_i^S[j] = D_i^{CM}[j] - F_i[j]$ for all values of i and j.

Analysis When inserting an item, the FID-sketch increments d_{FQ} counters in total among both the FQ-subsketch and the DS-subsketch of type Subtraction-sketch and d_{IS} counters of the IS-subsketch. Similarly, when deleting an item, the FID-sketch decrements d_{FQ} counters in total among both the FQ-subsketch and the DS-subsketch of type Subtraction-sketch and d_{IS} counters of the IS-subsketch. Thus, we have made the number of counter updates when inserting or deleting an item using the DS-subsketch of type Subtraction-sketch equal to the best case scenario when using the DS-subsketch of type CM-sketch. Note that this is achieved by simply modifying the insertion and deletion operations and without changing any other parameters of the subsketches such as the number of arrays, the number of counters per array, or the number of hash functions. Note that the DS-subsketch of type Subtraction-sketch can be used to enable deletions in other sketches as well, which otherwise do not support deletions, such as the CU-sketch and CML-sketch.

4 Experimental results

In this section, we evaluate our FID-sketch and present its side-by-side comparison with four prior state-of-the-art sketches, namely Count-sketch [6], CM-sketch [11], CU-sketch [14], and CML-sketch [18] on CPU and GPU platforms. As we will see shortly, among existing sketches, CML-sketch has the highest accuracy for multisets with zipfian distribution. Unfortunately, CML-sketch is unsuitable for most applications due to three reasons. First, it does not support deletions. Second, it suffers from both over-estimation and under-estimation errors. Last, it has slow query speed due to its complex manipulation of random numbers. Consequently, CM-sketch is still the most popular sketch. Thus, we primarily compare the performance of FID-sketch with it.

4.1 Experimental setup

Compute Platforms We implemented the sketches on two different types of platforms: CPUs and GPUs. We did our implementation for CPU platform on a Thinkstation D30 server with 2 Intel CPUs (Xeon E5-2620, 2.00 GHz, 6 physical cores). We did our implementation for GPU platform on an NVIDIA GPU (Quardro 4000, 950 MHz, 2047 MB device memory, 256 CUDA cores).

Parameter Settings Following are the four parameters that affect sketch performance: 1) number of arrays, represented by d_{FQ} for the FQ-subsketch and the DS-subsketch, by d_{IS} for the IS-subsketch, and by d for all other sketches; 2) number of counters per array, represented similarly by w_{FQ}, w_{IS}, and w; 3) maximum number of insertions, represented by U for all sketches; 4) number of inserted distinct items, represented by V for all sketches; and Unless stated otherwise, we used these parameters: 1) $d_{FQ} = d_{IS} = d = 4$; 2) $w_{FQ} = w = 30K$, $w_{IS} = 300K$; 3) U = 10M; 4) V = 100K.

Workloads We conducted our experiments on two types of workloads: *uniform* workloads, in which the item frequencies follow a uniform distribution and *zipfian* workloads, in which the item frequencies follow a zipfian [20] distribution.

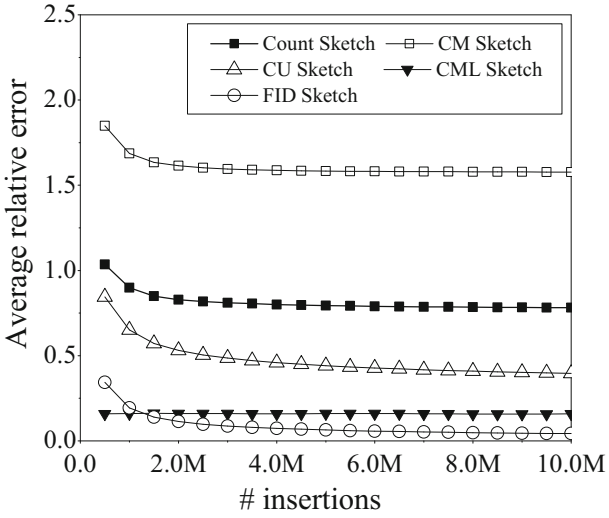

Figure 3 Uniform distributed insertions

4.2 Evaluation of absolute relative error

The accuracy of the frequency estimate of an item returned by a sketch is defined as the average absolute relative error (ARE) in the frequency estimate compared to the actual frequency of the item. Let the actual frequency of an item e be $f(e)$ and the frequency estimate returned by the sketch be $\hat{f}(e)$. The ARE of this estimate is given by $|\hat{f}(e) - f(e)|/f(e)$. Next, we present results for the impact of insertions on the ARE of the sketches

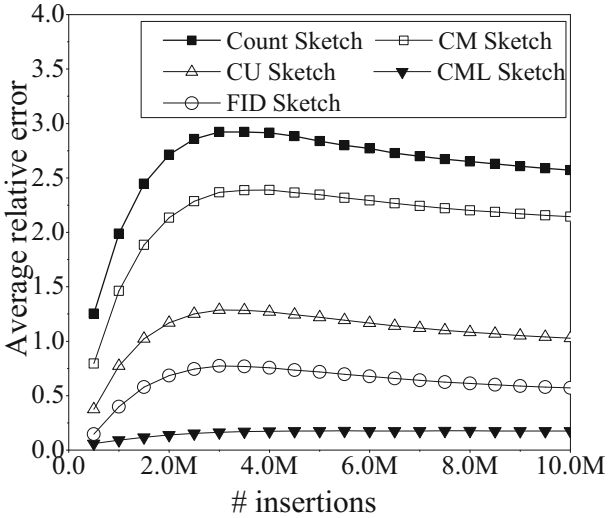

Figure 4 Zipfian distributed insertions

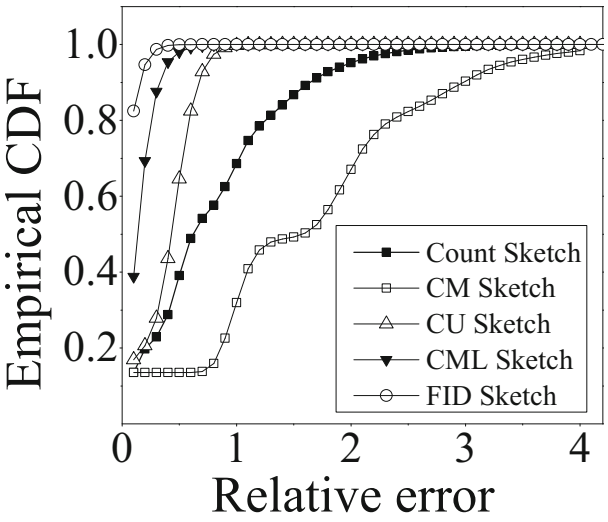

Figure 5 CDF of ARE (uniform dist)

followed by the impact of deletions. After that we measure the impact of the size of the on-chip sketch and the number of distinct items on the AREs of the sketches.

4.2.1 Effect of insertions

Our experimental results show that with the same space consumed, after 10M insertions, the FID-sketch achieves an ARE of only 0.043 and 0.57 with uniform and zipfian workloads, respectively, which is 36.7 times and 3.75 times less than the ARE of CM-sketch,

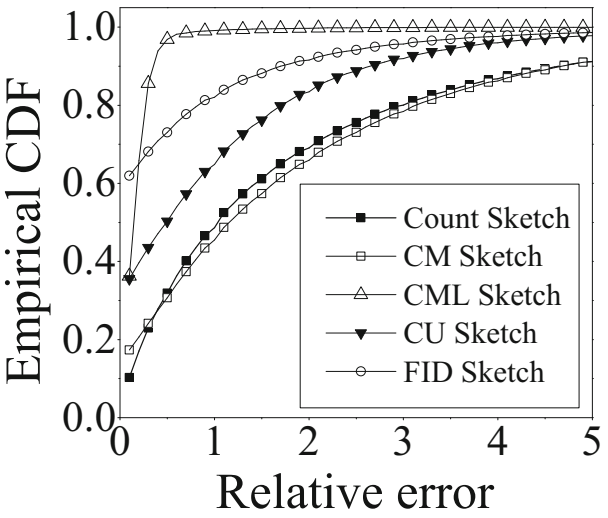

Figure 6 CDF of ARE (zipfian dist)

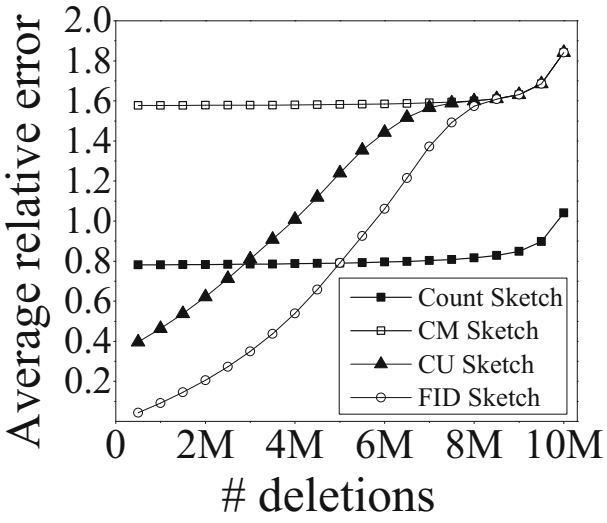

Figure 7 Uniform distributed deletions

respectively. Figures 3 and 4 show the AREs of the 5 sketches for uniform and zipfian distributed insertions, respectively, when the number of insertions is varied from 0 to 10M.

Our experimental results also show that with uniform workloads, for 82.5% of items stored in the FID-sketch, ARE is less than 0.1. This percentage is 6.1 times higher than the percentage of items stored in the CM-sketch for which ARE is less than 0.1. With zipfian workloads, for 61.9% of items, the ARE of FID-sketch is less than 0.1. This percentage is 3.6 times higher than the corresponding percentage for the CM-sketch. Figures 5 and 6 show the CDF of the relative error for all sketches when all 100K distinct items were inserted with the total insertions of 10M.

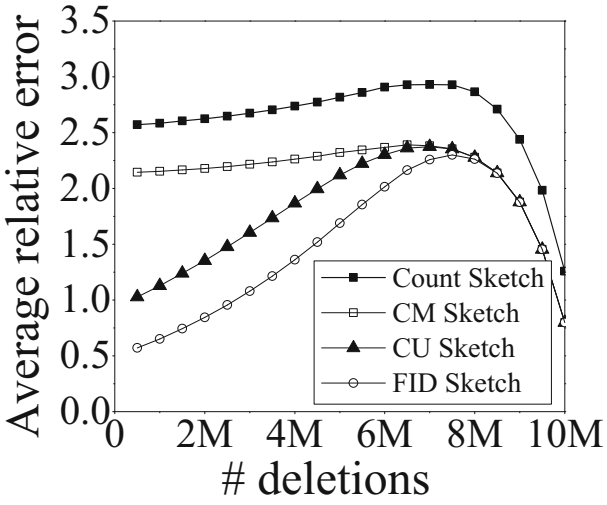

Figure 8 Zipfian distributed deletions

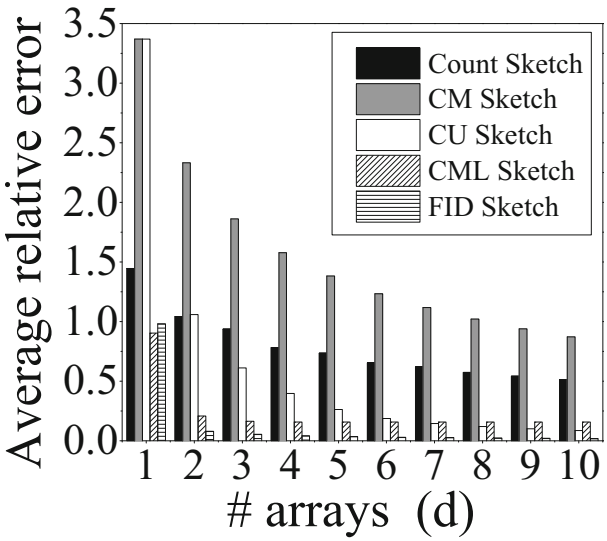

Figure 9 ARE vs. # of arrays

4.2.2 Effect of deletions

Our experimental results show that during deletions, the ARE of FID-sketch is always smaller than the AREs of CU-sketch and CM-sketch. Figures 7 and 8 show the AREs after different number of deletions for the five sketches. Note that the original CU-sketch does

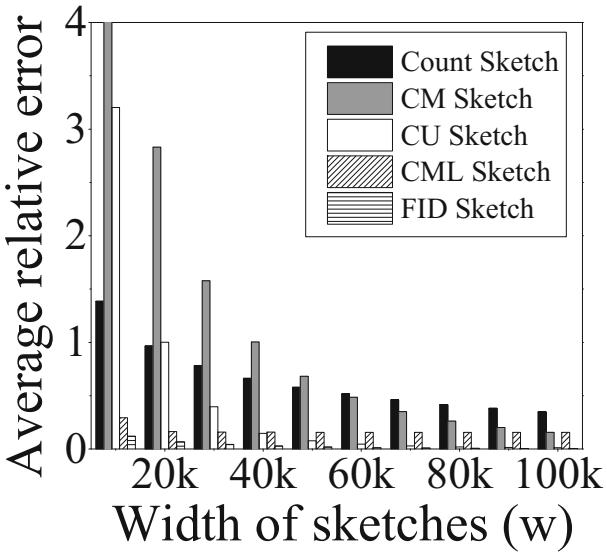

Figure 10 ARE vs. counters per array

NOT support deletions. We enable deletions in it using our DS-subsketch. We start deletions after inserting all 100K items 10M times.

4.2.3 Effect of on-chip sketch size

Our experimental results show that the ARE is almost inversely proportional to the number of arrays and the number of counters per array in the on-chip sketch. Increasing the number of counters per array is a more efficient way to improve the accuracy compared to increasing the number of arrays. We make these observations from Figures 9 and 10, which show the results for uniform workloads. We have not included figures for zipfian workloads due to the lack of space and because they show similar trends as Figures 9 and 10. To conduct these experiments, we varied the number of arrays in the sketches and the number of counters per array and measured the AREs using 100K distinct items and 10M total insertions.

4.2.4 Effect of the number of distinct items

Our experimental results show that the AREs of all sketches increase almost linearly with increase in the number of distinct items in the sketch, however, the rate of increase is smallest for the FID-sketch. This is shown in Figure 11 which plots the AREs of the five sketches. With a fixed sketch size, the more distinct items inserted into a sketch, the more items map to the same counters, resulting in higher error in frequency estimates.

4.3 Evaluation of speed

As the query processes of the CM-sketch, CU-sketch, and our FID-sketch are exactly the same and we observed that the results for query speed are similar, we only present the results for our FID-sketch using 3 different data sets.

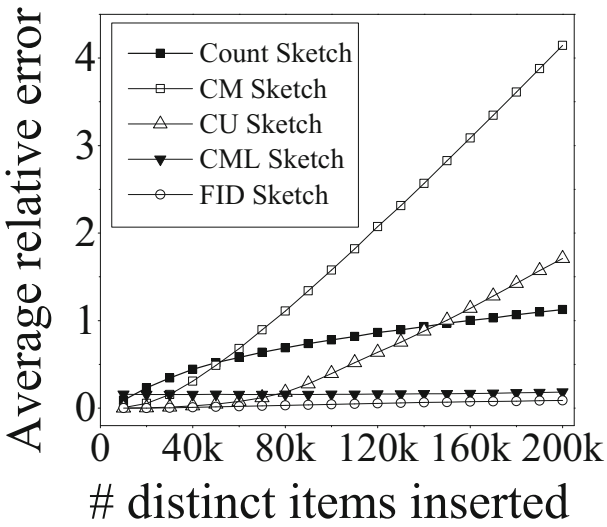

Figure 11 ARE vs. # of distinct items

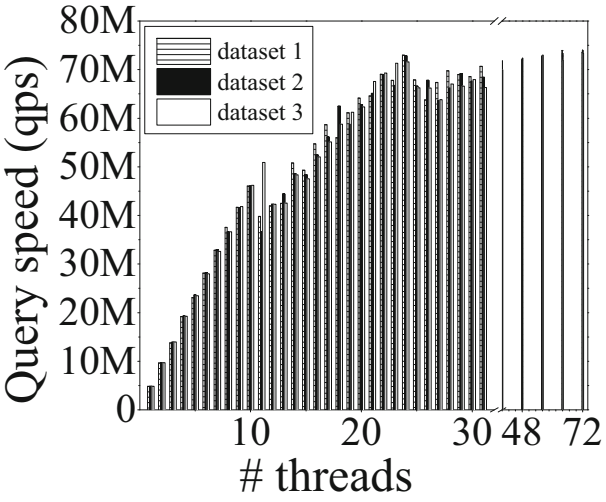

Figure 12 Query speed vs. # of threads

4.3.1 On multi-core CPU platform

Our results show that the query speed of FID-sketch increases almost linearly as the number of threads increases. The query speed reaches almost 70 Mqps when using 24 threads. This is shown by Figure 12, which plots the query speed of our FID-sketch by varying the number of threads. We also observe that when the number of threads exceeds 24, the query speed does not increase further. This is because our CPU has 2×6 cores with Hyper-Threading, which can handle $2 \times 6 \times 2 = 24$ simultaneous threads.

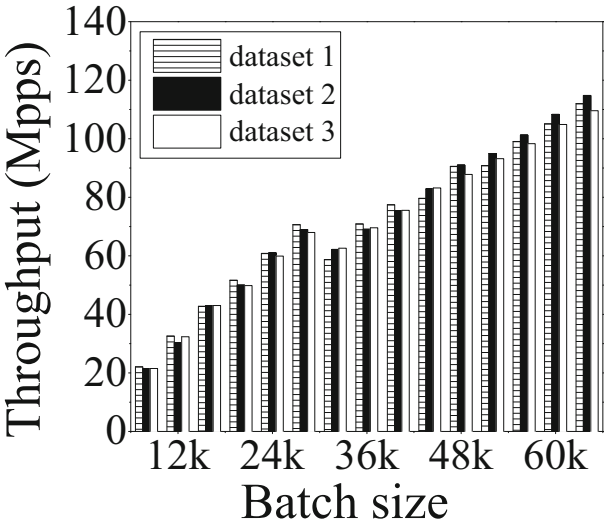

Figure 13 Throughput vs. batch size

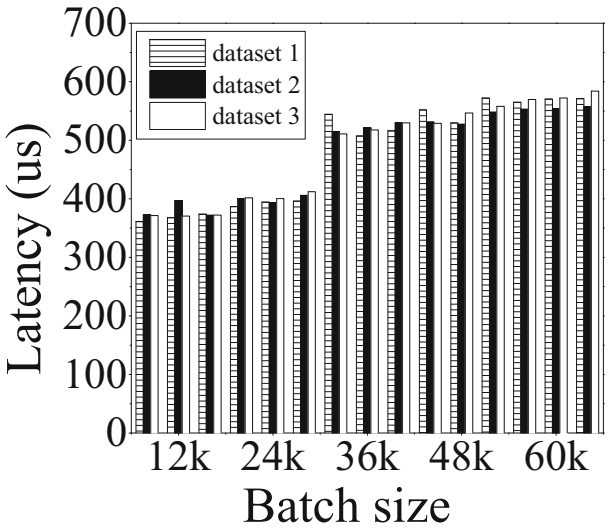

Figure 14 Latency vs. batch size

4.3.2 On GPU platform

For GPU platform, we focus on two metrics, namely *throughput* and *latency* for query requests.

GPU Throughput/Latency vs. Batch Size Our experimental results show that GPU processing throughput and latency grow as the batch size increases; the effect of batch size is more prominent on the throughput than on latency. The more query requests a batch

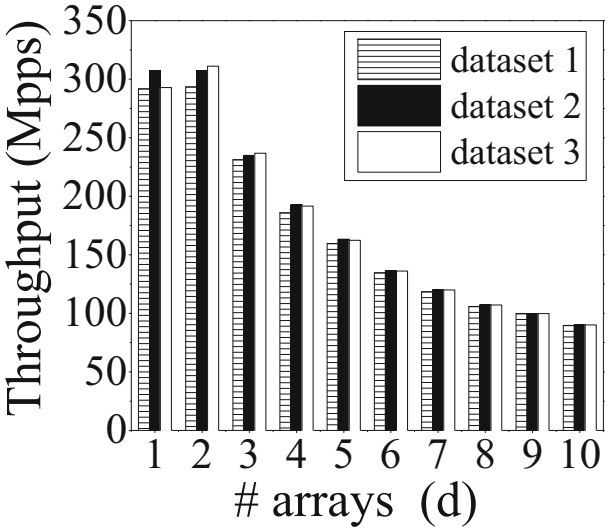

Figure 15 Throughput vs. # of arrays

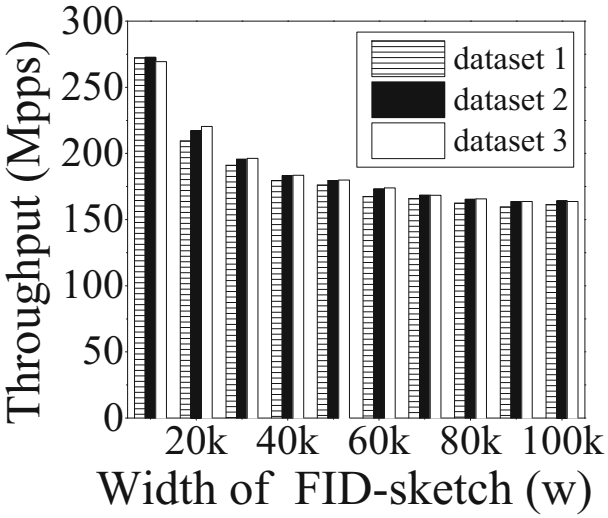

Figure 16 Throughput vs. counters/array

sends, the more parallelization the GPU employs, and the higher processing throughput it achieves. However, increasing the batch size increases the processing latency. Thus, a tradeoff between throughput and latency needs to be made. Figures 13 and 14 plot the processing throughput and processing latency, respectively, for different batch sizes.

GPU Throughput vs Sketch Size Our experimental results show that the GPU batch processing throughput decreases with increasing number of arrays and number of counters per array of a sketch. The effect of the number of arrays is more prominent on the throughput compared to the number of counters per array. Smaller sketch sizes mean better spatial locality; thus, smaller sketches make use of GPU's cache more efficiently and achieve better processing throughput. However, smaller sketches also mean poorer accuracy. Figures 15 and 16 plot the throughput for queries with varying number of arrays and number of counters per array, respectively.

5 Conclusion

In this paper, we propose a new sketch called FID-sketch. The key advantage of our FID-sketch is that it achieves a much higher accuracy compared to the state-of-the-art and at the same time, achieves the smallest on-chip memory footprint while maintaining the same query speed as prior sketches. Therefore, it is very well suited for the emerging data intensive network applications. FID-sketch achieves this by maintaining three subsketches: an on-chip FQ-subsketch that contains all information to process queries; an off-chip IS-subsketch that enables fewest counter increments in FQ-subsketch for insertions and thus reducing the error and memory footprint of the FQ-subsketch; and an off-chip DS-subsketch that enables deletions from the FQ-subsketch. We carried out extensive experiments on multi-core CPU and GPU to evaluate the accuracy of the FID-sketch and to compare it with prior sketches. Our experimental results show that our proposed sketch significantly

outperforms the state-of-the-art. Our proposed on-chip/off-chip hybrid sketch method is a generic model and can also be applied to other sketches.

Acknowledgments This work is partially supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (2014CB340405), NSFC (61672061), the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, and National Science Foundation (CNS 1616317, CNS 1616273).

References

1. Aguilar-Saborit, J., Trancoso, P., Muntes-Mulero, V., Larriba-Pey, J.-L.: Dynamic count filters. ACM SIGMOD Record, pp/ 26–32 (2006)
2. Barman, D., Satapathy, P., Ciardo, G.: Detecting attacks in routers using sketches. In: Proceedings of the High Performance Switching and Routing (2007)
3. Bu, T., Cao, J., Chen, A., Lee, P.P.: A fast and compact method for unveiling significant patterns in high speed networks. In: Proceedings of the IEEE INFOCOM, pp. 1893–1901 (2007)
4. Callegari, C., Cyprus, N.: Statistical approaches for network anomaly detection. In: Proceedings of the ICIMP (2009)
5. Chakrabarti, K., Garofalakis, M., Rastogi, R., Shim, K.: Approximate query processing using wavelets. VLDB **10**(2-3), 199–223 (2000)
6. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. In: Automata, Languages and Programming (2002)
7. Chen, A., Jin, Y., Cao, J., Li, L.E.: Tracking long duration flows in network traffic. In: Proceedings of the IEEE INFOCOM (2010)
8. Cisco visual networking index: Forecast and methodology, 2015–2020. CISCO White paper
9. Cohen, S., Matias, Y.: Spectral bloom filters. In: Proceedings of the ACM SIGMOD, pp. 241–252 (2003)
10. Cormode, G., Garofalakis, M.: Sketching streams through the net: Distributed approximate query tracking. In: Proceedings of the VLDB (2005)
11. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithm. **55**(1), 58–75 (2005)
12. Cormode, G., Johnson, T., et al.: Holistic udafs at streaming speeds. In: Proceedings of the SIGMOD (2004)
13. Cormode, G., Hadjieleftheriou, M.: Finding frequent items in data streams. Proc. VLDB **1**(2), 1530–1541 (2008)
14. Estan, C., Varghese, G.: New directions in traffic measurement and accounting. Proc. ACM SIGMCOMM **32**(4), 323–338 (2002)
15. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: A scalable wide-area web cache sharing protocol. In: Proceedings of the ACM SIGCOMM (1998)
16. Kollios, G., Byers, J.W., Considine, J., Hadjieleftheriou, M., Li, F.: Robust aggregation in sensor networks. IEEE Data Eng. Bull. **28**(1), 26–32 (2005)
17. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data Parallel analysis with sawzall. Dyn. Grids Worldw. Comput. **13**(4), 277–298 (2005)
18. Pitel, G., Fouquier, G.: Count-min-log sketch: Approximately counting with approximate counters. arXiv:1502.04885 (2015)
19. Potti, N., Patel, J.M.: Daq: a new paradigm for approximate query processing. In: Proceedings of the VLDB (2015)
20. Powers, D.M.: Applications and explanations of Zipf's law. In: Proceedings of the EMNLP-CoNLL. Association for Computational Linguistics (1998)
21. Source code of FID sketches with CUDA implementation. https://github.com/papers2016/FID-sketch.git
22. Yang, T., Xie, G., Li, Y., et al.: Guarantee IP lookup performance with FIB explosion. In: Proceedings of the SIGCOMM (2014)
23. Yang, T., Liu, A.X., Shahzad, M., Zhong, Y., Fu, Q., Li, Z., Xie, G., Li, X.: A shifting bloom Filter Framework for Set Queries. In: Proceedings of the VLDB (2016)
24. Yang, T., Liu, A.X., Shahzad, M., Yang, D., Fu, Q., Xie, G., Li, X.: A Shifting Framework for Set Queries. In: Proceedings of the IEEE/ACM Transaction on Networking (ToN) (2017)

25. Yang, T., Zhou, Y., Jin, H., Chen, S., Li, X.: Pyramid Sketch: a Sketch Framework for Frequency Estimation of Data Streams. In: Proceedings of the VLDB (2017)
26. Zhang, Y., Singh, S., Sen, S., Duffield, N., Lund, C.: Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In: Proceedings of the ACM IMC (2004)
27. Zhao, Q.G., Ogihara, M., Wang, H., Xu, J.J.: Finding global icebergs over distributed data sets. In: Proceedigs of the ACM PODS. ACM (2006)
28. Zhou, Y., Yang, T., Jiang, J., Cui, B., Yu, M., Li, X., Uhlig, S.: Cold Filter: A Meta-Framework for Faster and More Accurate Stream Processing. In: Proceedings of the SIGMOD (2018)