Delegation Sketch: a Parallel Design with Support for Fast and Accurate Concurrent Operations

Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren, Olaf Landsiedel
Marina Papatriantafilou
Chalmers University of Technology, Gothenburg, Sweden
{chasty,iwanw,magnus.almgren,olafl,ptrianta}@chalmers.se

Abstract

Sketches are data structures designed to answer approximate queries by trading memory overhead with accuracy guarantees. More specifically, sketches efficiently summarize large, high-rate streams of data and quickly answer queries on these summaries. In order to support such high throughput rates in modern architectures, parallelization and support for fast queries play a central role, especially when monitoring unpredictable data that can change rapidly as, e.g., in network monitoring for large-scale denial-of-service attacks. However, most existing parallel sketch designs have focused either on high insertion rate or on high query rate, and fail to support cases when these operations are concurrent.

In this work we examine the trade-off between query and insertion efficiency and we propose *Delegation Sketch*, a parallelization design for sketch-based data structures to efficiently support concurrent insertions and queries. *Delegation Sketch* introduces a domain splitting scheme that uses multiple, parallel sketches to ensure all occurrences of a key fall into the same sketch. We complement the design by proposing synchronization mechanisms that facilitate delegation of insertion and queries among threads, enabling it to process streams at higher rates, even in the presence of concurrent queries. We thoroughly evaluate *Delegation Sketch* across multiple dimensions (accuracy, scalability, query rate and input skew) on two massively parallel platforms (including a NUMA architecture) using both synthetic and real data. We show that *Delegation Sketch* achieves from 2.5X to 4X higher throughput, depending on the rate of concurrent queries, than the best performing alternative, while at the same time maintaining better accuracy at the same memory cost.

Olaf Landsiedel is also with Kiel University, Germany.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387542

ACM Reference Format:
Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren, Olaf Landsiedel and Marina Papatriantafilou. 2020. *Delegation Sketch*: a Parallel Design with Support for Fast and Accurate Concurrent Operations. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387542

1 Introduction

To process high-rate, high-volume data it is often necessary (in terms of space and processing time) to perform analytics not on the data itself, but rather on a succinct representation thereof. For this purpose, sketches have been proposed as a way to maintain data streams' state and answer queries on it (e.g. frequency of elements in the input or top-k most common elements) using limited memory, at the cost of giving approximate, rather than exact answers.

A representative example that shows the usefulness of sketches is network traffic monitoring. As traffic flows into a big network at high rates, e.g., at the ingress router of a university network, an administrator [10] or some system, e.g., a Network Intrusion Detection System or an SDN controller that does dynamic flow scheduling [33], might be interested to know at *any point in time* how many packets a given IP address has sent. Giving the exact answer to such a query requires storing all the incoming IP addresses and their counts, consuming memory proportional to the number of unique addresses. If, instead, an approximate answer is acceptable, a sketch can provide one with configurable error guarantees, using only a fixed amount of memory, without storing the IP addresses.

The literature on sketch-based algorithms offers a variety of ingenious techniques that mostly focus on the trade-off between memory consumption and accuracy [3, 5, 42]. Orthogonal to the need for small and accurate sketches is the need to process data at high rates. Thus, large research efforts focus on accelerating operations on the sketch, e.g., by using filters that process frequently found elements separately [32], which is important for many real-world input streams that are often highly skewed. As a result, high-throughput sketches are used in many applications, such as traffic monitoring [19, 24, 45] and data stream management tasks [7]. They are also used for communication reduction

in distributed monitoring algorithms [13] and help with dimensionality reduction in machine learning algorithms [21].

Over the last few years, there has been a significant interest in parallel architectures to achieve sufficient high-speed processing. Multi-core platforms are adopted in many settings, from high-end servers [20] to low-end embedded devices [2] on the edge. Sketches can benefit from parallelism: e.g., regarding network traffic monitoring, state-of-the-art single-thread approaches [25, 32] achieve several millions of operations per seconds, which is enough to process traffic from 10Gbps links, but as link capacities increase to more than 100Gps, the need for multi-core processing becomes apparent. However, most of the work proposed on sketches focuses on the single-thread case and not on parallel settings. For existing parallel designs, we identify that there are conflicting requirements when considering both insertions and queries: parallel designs that perform efficiently when there are only insertions fail to scale when there are concurrent queries, while designs that favor queries cannot handle concurrent insertions efficiently. With the exception of very recent work [30] which we discuss in the related work section, most papers do not address concurrency between insertions and queries. This research gap is important, as many applications have need of both operations concurrently, including the IP-frequency-counting example above and other monitoring applications. In cases such as intrusion detection, applications must be able to handle high-rate traffic and support frequent queries, since the traffic characteristics might change abruptly and unpredictably [24].

In this paper, we identify and provide means to balance trade-offs involved in parallelizing sketches with respect to the number of threads, the rate of concurrent queries and the input distribution. We propose *Delegation Sketch*, a generic parallelization scheme for sketches that is able to process high input rates and scale efficiently even in the presence of concurrent queries, while at the same time achieving high accuracy with the same or lower memory requirements compared to existing schemes. *Delegation Sketch* can be applied on various sketches that support insertions and point queries [3, 39, 44, 47] and aligns with the *regular* consistency specifications [22, 23, 29], i.e. a query takes into account all completed insert operations and possibly a subset of the overlapping ones. We make use of multiple parallel sketches and use hashing to ensure that the same key from different threads will end-up in the same sketch, allowing us to perform queries efficiently and more accurately. We also suggest a synchronization mechanism to delegate operations between threads, inspired by its uses in other concurrent data structure designs, e.g. in flat combining [17]. Our design: (a) allows threads to work on local data as much as possible, through the use of our proposed *Delegation Filters*, by aggregating multiple insertions on the same key locally without modifying any of the sketches; and (b) combines multiple queries on the same key and serves them quickly. In particular, we make the following contributions:

- We study trade-offs in parallelizing sketches, with respect to concurrent insertions and queries and show the gap in existing designs. We demonstrate that the choice of parallelization does not affect only throughput and scalability, but also the accuracy of the result.
- We propose a generic parallelization design, *Delegation Sketch*, that scales with the number of threads, handles millions of insertions per second and is able to gracefully support concurrent queries.
- We provide a synchronization scheme that minimizes communication between *Delegation Sketch* threads and efficiently delegates operations on the sketch to other threads. We also leverage this synchronization mechanism to combine operations on the sketch to significantly improve performance and scalability.
- We provide an extensive experimental evaluation of *Delegation Sketch* and study it in connection to known parallelization designs on two massively parallel platforms with up to 72 and 288 threads, using both synthetic and real data. We show that *Delegation Sketch* supports up to 4X higher processing throughput and performs queries with up to 2.25X lower latency than the next best performing alternative. At the same time, *Delegation Sketch* has the same accuracy as the most accurate alternative, using the same amount of memory.

The rest of the paper is organized as follows: Section 2 gives the required background on sketches and describes the system model we target in this work. In Section 3 we analyze existing parallelization designs and motivate the need for *Delegation Sketch*, whose overview is given in Section 4. In Sections 5 and 6 we describe our design in detail. In Section 7 we present and analyze the results of our experimental evaluation. We discuss related work in Section 8 and conclude in Section 9.

2 Preliminaries

In this section, we describe the Count-Min sketch, a simple and efficient sketch, widely applicable in practice. We also describe a known extension to it, the Augmented Sketch, which includes techniques that we also adopt in our design. We finish this section by describing our system model.

2.1 The Count-Min and Augmented Sketch

The *Count-Min Sketch* [5] is a series of counters arranged in a 2-D array, with w columns and d rows. Every row is associated with one of d pairwise-independent hash functions h_1,h_2,...,h_d, with h_i mapping keys from an input universe U to one of the w counters in row i. The sketch supports two operations: *insert*[1] and *point-query*. To insert a key K in the sketch, we increment the counter at position $h_i(K)$ at row

[1] Aka *update* in the literature. We use the term *insert* throughout the paper.

i, for each one of the d rows. To perform a point-query on a key K, we hash the key with the same hash functions and select the counter at position $h_i(K)$ at row i, for each one of the d rows. The answer to the query is simply the minimum value among the selected counters, since that counter is closest to the true frequency of K, i.e. contains less "noise" from colliding keys. The answer to point-queries on any key K is always equal or higher than K's true frequency $f(K)$ and is lower than $f(K) + \frac{e}{w}N$ with probability $1 - \frac{1}{e^d}$, where N is the number of keys in the sketch [5]. Thus, one can configure the number of rows and columns to achieve error guarantees appropriate to the application.

In *Augmented Sketch* [32], Roy et al. couple a sketch with a filter to increase insertion throughput, especially when the input is highly skewed. The purpose of the filter is to efficiently keep track of a small number of keys that are frequently found in the input. When a new key needs to be inserted, if it is in the filter, then its frequency is updated there, without involving the sketch. Similarly, when performing a query on a key, if we find it in the filter then we report its frequency without querying the sketch for that key. Performing an operation on the filter is much faster than performing it on the underlying sketch, e.g. compared to the Count-Min Sketch that requires hashing a key multiple times.

2.2 System Model

Here we introduce the assumptions and requirements we make on the hardware platform, the application requirements and the consistency requirements.

Hardware Requirements: We assume a multi-core system with a finite set of threads t_1,...,t_T where T can be larger than the number of physical processors, along with a typical memory hierarchy, i.e. a L1 cache per thread, L2 and L3 caches shared between threads and main memory (either uniform on non-uniform). We consider an asynchronous shared memory system supported by a coherent caching model, through which a thread can access a shared variable not in the memory of the core where the thread is running. We also consider that no thread will arbitrarily fail or stop making progress.

We adopt the cache-register stream processing model [12], where input keys are continuously processed as they arrive and their frequency is continuously updated in the sketch. We assume that each thread has its own input sub-stream of keys. These sub-streams can originate from different sources or may have been extracted from a single stream in a previous part of the processing pipeline, either in software or in hardware, e.g., considering processing of packets coming from the network, many networks cards distribute the stream of packets to different CPUs [15, 16].

Application Requirements: *At any point in time, the application might query the frequency of a specific key in the total stream, i.e. across all sub-streams.* We assume that queries are much less frequent than the rate at which keys enter the system, but a query must be served even as new keys are being inserted and not at a later point in time when there are no more keys to insert. We also assume that each thread is serving one operation at a time: either an insertion of a new key from the stream, or a query.

Consistency Requirements: A query for the frequency of a key, performed by any thread, returns an approximation of the true frequency of the key, within the bounds provided by the underlying sketch. In the case of the Count-Min Sketch this includes the invariant that the answer is an over-approximation of the true frequency. The result must take into account all previous insertions of a key, across all sub-streams, but might or might not include insertions that overlap with the query, i.e., those that take place after the query has been issued and before it returns the result. This is a common assumption for concurrent data structures and it aligns with similar consistency specifications in literature, e.g. the *regularity* consistency specification [22, 23, 29]. In the case of sketches, the effects of not counting overlapping insertions are overshadowed by the fact that the answer is already an approximation of the true frequency.

3 Problem analysis

In this section, we summarize the existing parallelization designs that serve as baselines and we analyze their trade-offs, in terms of the processing *throughput* of insertions and queries, the *accuracy* of the queries (i.e. the approximation error compared to the true frequency of a key) and the overall *scalability* of the design with the number of threads. We show that the existing designs have individual strengths but cannot efficiently handle the case of both insertions and queries, thus there is need for new parallelization designs.

3.1 Thread-local sketches

In the literature of sketch algorithms, most results focus on single thread performance and accuracy. When it comes to parallelization, most works [1, 32, 43] suggest the "*thread-local design*", where we have multiple sketches, one for each thread. Each thread inserts keys into its own sketch. To query a key, a thread queries every sketch and sums the results.

This design leads to very good scaling when there are only insertions, since each thread will work only on its own cache (sketches are usually small enough to fit L1 or L2 cache). However, the performance degrades significantly as soon as there are concurrent queries, since a querying thread needs to perform reads on all the sketches. The degradation worsens with the number of threads (since there are more sketches to read from) and becomes a major drawback in the highly parallel architectures we target in this work. Moreover, as shown in Section 5.1, this design leads to lower

Design name	Insertion Rate	Support for Queries	Scalability	Accuracy
Thread-local	high	low	high	low
Single-shared	low	high	low	high
Delegation Sketch	high	medium/high	high	high

Table 1. Comparison of parallelization designs.

accuracy relative its the memory requirements, as each subquery on a sketch introduces approximation errors that are then summed together.

3.2 Single-shared sketch

In work favoring queries over insertions [8, 37, 38] a single sketch is shared among all threads, a design henceforth referred to as the "*single-shared design*". Insert operations are slow, since threads require synchronization mechanisms, e.g. locks or (in the case of the Count-Min Sketch) atomic instructions and content on the memory of the sketch. Because of these reasons, in highly parallel environments targeted in this work, this design is not expected to scale with the number of threads when the input stream is inserted at high rates. However, queries are fast and accurate, since they do not involve collecting results from multiple sketches.

3.3 The need for a new design

Based on the discussion above, it is evident that the existing parallelization designs focus on two extreme use cases: they are effective either for applications that are only inserting keys at high rates with no queries (thread-local), or applications that will summarize a stream of keys once, and then only perform queries (single-shared).

In practice, many applications need to handle queries concurrently with insertions. Even though insertions are the most common operation for most applications (e.g., packet processing at high traffic rates), queries need to be handled concurrently as new keys are being inserted (e.g. IP counts must be queried at any time in traffic monitoring and flow scheduling). Moreover, support for high frequency queries, (e.g. one query every 1,000 insertions might mean one query every millisecond, depending on the input stream rate) is important for applications that need to react quickly to unpredictable changes [28, 36] or important events [24].

In order to serve such applications, we propose a new design, *Delegation Sketch*, that acts as a hybrid of the two designs mentioned earlier. We use multiple parallel sketches to allow our design to scale and perform insertions in parallel, but contrary to the thread-local design, a query needs to search for a key in only one of these sketches.

Table 1 summarizes the existing parallelization designs in comparison with *Delegation Sketch*. In the next section, we describe the main ideas and give an overview of our design.

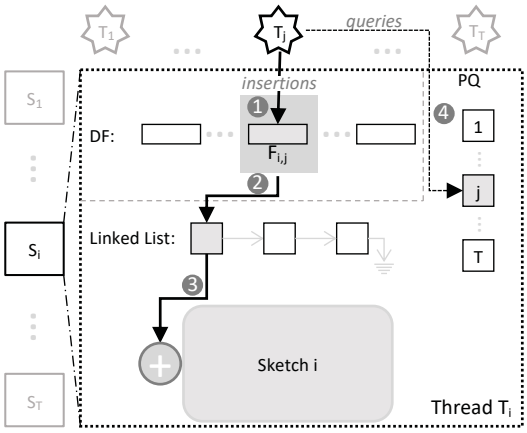


Figure 1. Outline of our design. DF stands for Delegation Filters and PQ stands for Pending Queries.

4 Overview of *Delegation Sketch*

Our design bases on two techniques: Domain Splitting and Operation Delegation. We outline both here and detail in the subsequent sections.

4.1 Domain Splitting

To make queries faster, the number of sketches that a query has to search must be limited. To this end, we logically distribute the input domain of possible keys to the T available sketches where each sketch is responsible for a set of keys. For every possible key K that can be found in the input stream of any thread t, we define as $Owner(K)$ the thread that is responsible for K. Finding the owner of a key can be as simple as $Owner(K) = K \ modulo \ T$. Every thread that wants to insert K will insert it into the sketch owned by $Owner(K)$. In this way, the same key (even if it is part of the input stream of different threads) will end up in a single sketch, making a query on that key a relatively cheap operation. In addition to making queries faster, splitting the domain of keys implies benefits on insertion speed, as well as on accuracy, for reasons we describe in Section 5.1.

4.2 Operation Delegation

As domain splitting requires a thread to insert into or query from an arbitrary sketch, we propose the use of filters (which we call *Delegation Filters*) that achieve this efficiently, minimizing inter-thread communication. An outline of our design, showing the series of Delegation Filters associated with Sketch i is shown in Figure 1. We give an overview of the purpose of filters here and explain the design and use of filters during insertion and query operations in detail in Section 6.

For every sketch, we keep a series of Delegation Filters, one for each thread. The first purpose of Delegation Filters is to allow each thread to combine multiple insertions of the

same key together, using only local updates without inter-thread communication. Instead of modifying the sketch every time there is an insertion operation on a key, threads aggregate the occurrence of the same keys in their stream (arrow 1 in Figure 1) and modify the sketch only when a sufficient number of keys have been aggregated. This is especially useful if the input is highly skewed: threads are doing insertions on the filters reserved for them most of the time, instead of modifying one of the sketches and causing contention.

The second purpose of Delegation Filters is to provide a unit of synchronization between a thread j that wants to insert keys to the sketch of thread i. The keys and their counts that thread j has aggregated in its filter will be inserted into a linked list of ready filters (arrow 2 in Figure 1) and eventually into the sketch by thread i (arrow 3).

Upon queries, a thread j will delegate a query operation on a key K and have it handled by another thread $i = Owner(K)$ (arrow 4). This design allows to optimize the number of times we have to search for the frequency of K in the sketch, by aggregating or "squashing" multiple pending queries on the same key to a single query operation on the sketch.

The use of delegation and the query "squashing" optimization are inspired by techniques used in concurrent data structures such as flat combining [17], where operations on a data structure are delegated for another thread that combines and performs them. Our design uses the Augmented Sketch (which we apply on top of the Count-Min sketch) as the underlying sketch, but different sketches that have the same interface (i.e. support insertions and point queries) can be used as well [3, 39, 44, 47]. In this work, we focus on point queries for frequency estimation, that are the basic type of queries supported by the Count-Min sketch. In the following section we detail the domain splitting technique and analyze its benefits, then describe the way we delegate operations.

5 Domain Splitting and Benefits

The idea of splitting the domain of keys has been proposed for different scenarios and goals; e.g. Dobra et al. [9] apply it for join-size estimation and leverage approximate knowledge of the stream distribution, Thomas et al. [39] use it to handle architecture-specific constraints of the Cell processor. Here we utilize it in order to handle queries accurately and efficiently, as explained in the following subsections. The algorithmic implementation and the synchronization of the *Delegation Sketch* operations are described in Section 6.

5.1 Influence on the overestimation error

Here we study the accuracy of the different designs. We show that *Delegation Sketch* is: (a) more accurate than the fastest parallelization design (thread-local) (b) as accurate as the most accurate (albeit slower) parallelization design while using the same amount of memory as those designs.

Due to the probabilistic nature of sketches, the result of querying for a key includes an amount of error. A query using the thread-local design involves querying all the sketches and summing the results. The intuition behind why domain splitting implies better accuracy than the thread-local design is that, by having all occurrences of the same key in a single sketch, it avoids aggregating the error from multiple sketches.

Reference sketch (single thread): Assume $f(i)$ is the frequency of key i, across the sub-streams of all threads. Based on [5], for a Count-Min sketch with w buckets and d rows, the estimate $\hat{f}(i)$ of key i is

$$f(i) \leq \hat{f}(i) \leq f(i) + \epsilon N \tag{1}$$

with probability $1 - \delta$, where $w = \frac{e}{\epsilon}$, $d = ln(1/\delta)$ and $N = \sum_{j \in U} f(j)$ where U is the universe of keys.

Thread-local: This design uses T sketches of size $w * d$ each and the estimate when querying each sketch t is

$$f_t(i) \leq \hat{f}_t(i) \leq f_t(i) + \epsilon N_t \tag{2}$$

with probability $1 - \delta$, where f_t denotes the frequencies of keys that are in the sub-stream of thread t, $N_t = \sum_{j \in U} f_t(j)$ and $\sum_{1 \leq T} f_t(i) = f(i)$. The total estimate $\hat{f}(i)$ is the sum of estimates from all the sketches so,

$$\sum_{1 \leq T} f_t(i) \leq \hat{f}(i) \leq \sum_{1 \leq T} f_t(i) + \epsilon \sum_{1 \leq T} N_t \tag{3}$$

or equivalently (by substitution)

$$f(i) \leq \hat{f}(i) \leq f(i) + \epsilon N \tag{4}$$

with probability at least $(1 - \delta)^T$.

This means that using the thread-local design (that uses T sketches with w buckets and d rows each) results to a similar bound as having one sketch with w buckets and d rows from Equation 1 and inserting all the elements in it.[2] For that reason, the thread-local design is far from optimal, considering the amount of memory it uses.

Single-shared: Using the same total memory as in the thread-local design (by using a single sketch with d rows and $T * w$ buckets), for the single-shared sketch we have:

$$f(i) \leq \hat{f}(i) \leq f(i) + \frac{\epsilon}{T} N \tag{5}$$

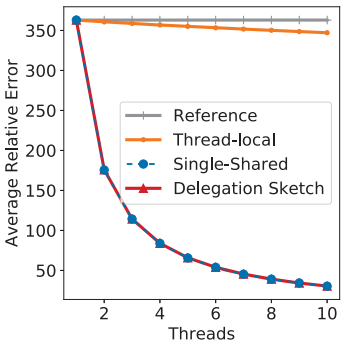
with probability $1 - \delta$.

Domain splitting: In our design, by splitting the domain based on the number of threads, we have

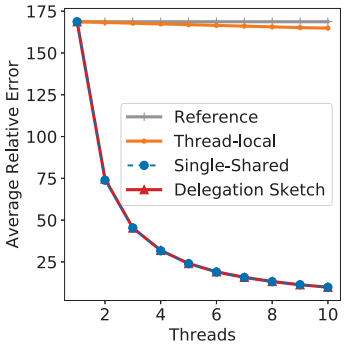
$$f(i) \leq \hat{f}(i) \leq f(i) + \epsilon N' \tag{6}$$

with probability $1 - \delta$, where N' is the total count of keys that hash to the same sketch as i. E.g. for a uniform distribution

[2] In practice it is slightly better than that, because in the thread-local design we take the estimate (i.e. the minimum count) from each sketch and sum them, rather than summing the individual cells in a single sketch and then taking the estimate.



(a) Uniform distribution.



(b) Zipf distribution (skew=1).

Parallelization Design	Memory
Reference (single thread)	$w * d$
Thread-local	$w * d * T$
Single-shared	$w * d * T$
Domain splitting (Delegation Sketch)	$w * d * T$

(c) Memory consumption of the different parallelization designs we consider in the analysis. w and d are constant.

Figure 2. Average relative error as the number of threads increases. We also include the memory consumption for each design. The single-shared version has the same average relative error as the domain-splitting one.

of keys, $N' = \frac{N}{T}$ and the bound is the same as in the single-shared design[3].

The aforementioned bounds for *Delegation Sketch* and thread-local design are in expectation and depend heavily on the input distribution. For this reason, we also examine the accuracy of those designs from an empirical point-of-view. In Figure 2 we show the difference in query error (in terms of the average relative error, also used in [32]) between the two approaches, using data from a uniform distribution (Figure 2a), as well as the Zipf distribution (Figure 2b). We have also included the "single-shared" sketch that uses a single sketch with the same total memory as the local-threads and domain-splitting designs, as well as the "reference" sketch that uses a single sketch with w buckets and d rows. For these experiments we used 600K keys taken from a universe of 100K distinct keys, then queried every key in that universe once. The memory footprint of each designs is shown in the table of Figure 2.

The results from Figure 2 align with the arguments above. The thread-local design has only slightly less error than the reference sketch, even though it uses T times more memory. Using domain splitting, the error decreases quickly based on the number of threads (equivalently, the number of sketches) in the system and its error is as low as that of a single-shared sketch that uses the same amount of memory.

5.2 Influence on query efficiency

Recall that in domain splitting, to perform a query on K, a thread will only have to query the sketch of $Owner(K)$, since all occurrences of K will have been inserted into that one (we explain how we perform query operations in detail in Section 6.2). For this reason, *domain splitting* leads to more efficient queries compared to the *thread-local* design where, as described earlier in Section 3, a querying thread will have to search for a key in multiple sketches. We support this claim experimentally in Section 7.4 where we present the latency of queries across different parallelization designs.

5.3 Influence on filter efficiency

Because with domain splitting the range of different keys that will be inserted in each sketch is smaller than U, the stream of keys that end-up on a sketch appears more skewed, which increases the effectiveness of any filters that may be used by the underlying sketch (e.g. the Augmented Sketch), both in terms of throughput and accuracy. This effect on accuracy is studied in detail in the experimental evaluation, Section 7.2.

6 Operation Delegation and Synchronization

The aforementioned benefits of domain splitting come with two challenges: (a) the fact that a thread will have to insert keys to another thread's sketch implies synchronization between threads, which needs to be done carefully in order to avoid bottlenecks and (b) if the input is highly skewed, some keys will be much more common than others, which, in turn, implies that some threads' sketches will be more busy handling a large part of the input keys. In this section we describe how we use filters, which we call *Delegation Filters* to address both of these challenges.

Delegation Filters: For every sketch, we keep a series of Delegation Filters, one for each thread. We want searching for a key and incrementing its count to be as fast as possible, so we choose to implement them in a very simple manner: a filter is a pair of two arrays of fixed, small size. The first array holds the keys and the second one holds the count of that key, at the same index. By keeping the filters small we can search the whole filter for a key using only a few SIMD instructions, similar to [32].

We now explain in detail the way we use these filters, along with the description of the algorithmic implementations of the *Insert* and *Query* operations.

[3]Later in Section 6 we introduce filters and their use in our design. We refine the bound of Equation 6 due to effects of filters in the Appendix.

Algorithm 1 Insert operation on thread j

```
1:  function Insert(key K)
2:      i ← Owner(K)
3:      Filter ← Sketches[i].DelegationFilters[j]
4:      (Filter is reserved exclusively for thread j)
5:      if K ∈ Filter then
6:          Increment count of K
7:      else
8:          Add K in Filter
9:          Set count of K to 1
10:     end if
11:     if Filter.size = MAX_SIZE then
12:         Sketches[i].LinkedList.push(pointer to Filter)
13:         while Filter.size = MAX_SIZE do
14:             process_pending_inserts()
15:         end while
16:     end if
17: end function
```

Algorithm 2 Processing pending inserts on thread i

```
1:  function process_pending_inserts
2:      while Sketches[i].LinkedList is not empty do
3:          Filter ← Sketches[i].LinkedList.pop()
4:          for each K in Filter do
5:              Insert K to the Sketches[i] (see Sec. 2)
6:          end for
7:          Flush Filter
8:          Filter.size ← 0
9:      end while
10: end function
```

6.1 Delegate Insertions

For a thread j to perform the *Insert* operation on a key K, it first tries to insert it in the Delegation Filter $F_{i,j}$, reserved for thread j at the sketch owned by $i = Owner(K)$. To do this, it first searches Filter $F_{i,j}$ for key K. If it is found, it increments the count at that location, otherwise it adds K to an empty slot in the filter and sets the count there to one (lines 4-9 in Algorithm 1). If the filter is full, thread j adds a pointer to the filter in a single-producer single-consumer concurrent linked list maintained for filters that are ready to be inserted in the sketch of thread i (line 11). Thread j will then wait until the filter is *consumed* (i.e. until the keys in the filter and their respective counts have been flushed into the sketch by thread i). Note that, until the filter becomes full, i.e. the number of distinct keys in the filter is equal to the size of the filter, thread j can keep updating the filter without any communication with any other thread. This is because, every thread j has its own reserved filter associated with the sketch of thread i, thus alleviating the need for synchronization. The high-level pseudo-code of *Insert* is shown in Algorithm 1.

Periodically, threads check the linked list of full filters associated with their own sketch. This check can be performed at different points, e.g. after a certain timeout, after a successful completion of an insert or query operation, or while the thread is waiting for another thread to consume its filter (line 12 of Algorithm 1). E.g., thread j checks its own list of filters, in parallel while waiting for its filter to be consumed at line 14 of Algorithm 1. A high level pseudo-code of how threads process pending inserts is shown in Algorithm 2. Thread i traverses the list of pointers to filters that are ready to be inserted into its sketch. For every such filter, the thread iterates over the keys in the filter and adds their counts to the sketch (line 4-6 of Algorithm 2), using the semantics of the underlying sketch. Then, the thread removes any keys and their counts from the filter and marks it as empty (lines 7-8).

Claim 1. *All keys and their counts inserted in Delegation Filter j of thread i will be eventually inserted in the sketch of thread i, assuming threads continue to make progress.*

This is ensured by requiring thread i to have *exclusive access* on the filter while it is in the process of consuming the filter and inserting its contents in the sketch. We achieve this in the following ways: (a) for thread i to be able to consume a filter, it must first find it in its concurrent link-list of ready filters; thread j only adds it in the list when it is full, at which point it stops inserting items in it; and (b) thread j will not start inserting keys in the filter unless it is marked as empty by thread i (line 8 of Algorithm 2).

6.2 Delegate Queries

Similarly to *Insert*, for a thread j to query the frequency of key K, it first finds the thread $i = Owner(K)$. In order to accurately answer the query, the thread must count all occurrences of K, that can be found in: (a) the sketch owned by i and (b) any of the T Delegation Filters associated with the sketch owned by i.

One option to achieve this is to have thread j search the sketch owned by i and its Delegation Filters. However, this would require synchronization between thread j and any of the T threads that might be concurrently accessing those Delegation Filters, as well as thread i that is inserting keys from the Delegation Filters into its sketch. Note that, allowing thread i to simply do this without any synchronization, might cause thread j to incorrectly "double count" occurrences of K: after thread j has counted X occurrences of K in a Delegation Filter, that filter might become full and get inserted into the sketch before thread j searches for K in the sketch, thus including X twice in the final answer.

Instead, we chose to delegate the query to thread i. Along with every sketch, we keep an array called *PendingQueries* of size T. Every item in the array holds a key, a counter (initially at zero) and a flag. Thread j adds key K at *PendingQueries[j]* and sets the flag there, to indicate that there is a pending query on key K (lines 4-6 of Algorithm 3). Thread j will then wait(checking its own list of filter and pending queries in the meantime at lines 8 and 9 of Algorithm 3) until the flag

Algorithm 3 Query operation on thread j and processing of pending queries on thread i

```
1:  function Query(key K)
2:      i ← Owner(K)
3:      PQ ← Sketches[i].PendingQueries
4:      PQ[j].key ← K
5:      PQ[j].count ← 0
6:      PQ[j].flag ← 1
7:      while PQ[j].flag = 1 do
8:          process_pending_inserts()
9:          process_pending_queries()
10:     end while
11:     return PQ[j].count
12: end function
13:
14: function process_pending_queries
15:     PQ ← Sketches[i].PendingQueries
16:     for t = 0; t < T; t++ do
17:         if PQ[t].flag = 1 then
18:             res ← 0
19:             K ← PQ[t].key
20:             for k = 0; k < T; k++ do
21:                 F ← Sketches[i].DelegationFilters[k]
22:                 res ← res + (count of K in F)
23:             end for
24:             res ← res + Sketches[i].get_estimate(K)
25:             PQ[t].count ← res
26:             PQ[t].flag ← 0
27:         end if
28:     end for
29: end function
```

is set back to zero by thread i and read the answer to the query from the counter.

Threads periodically loop over their *PendingQueries* array and check if there is a pending query in each item of the array. For every pending query, threads get the key from the array, search all Delegation Filters (line 20-22) and the sketch for this key (using the semantics of the underlying sketch), report the result at the counter for that key and set the flag to 0. Note that searching T Delegation Filters and one sketch, even though it becomes a costly operation as the number of threads increases, is faster than searching T sketches, which is required in the thread-local parallelization design.

High level pseudo-code for *Query*, as well as the process of serving pending queries is shown in Algorithm 3.

Claim 2. *The query operation of* Delegation Sketch *takes into account all previous, non-overlapping insertions by any thread.*

This is because the query operation takes into account all possible locations where a key K can be, i.e., both the sketch of thread $Owner(K)$, and the Delegation Filters associated with that sketch. This includes Delegation Filters that are not yet full. In this case, the query operations might miss some overlapping insertions of K that are happening concurrently at a filter, but will include completed insertions. If the occurrence of a key has been inserted in the filter, all later queries will take that occurrence into account, either when reading it from the filter, or from the sketch if it has been moved there. Due to the domain splitting technique described in Section 5, the query operation does not need to search for the key in any of the other sketches or filters.

Claim 3. *The query operation of* Delegation Sketch *does not "double-count" the occurrences of any key.*

This is ensured by the fact that only one thread is responsible for searching for a key in the filters and the sketch. During this time no other thread can insert keys in the sketch, which would result in "double-counting".

6.2.1 Optimization: Query Squashing. We now describe a simple optimization (not shown in Algorithm 3) that increases the performance of queries significantly, especially under conditions of high parallelism and input skew. When a thread i is done serving a delegated query on behalf of thread j, i.e. it has searched for key K in its sketch and the Delegation Filters associated with it, instead of just reporting the result to thread j, it iterates the array of pending queries to find other threads that have a pending query on the same key. Then, it reports the same result to those threads, without performing the actual search operations additional times, thus "squashing" the workload of multiple queries into one.

Note that this optimization does not report "stale" results and continues to respect the consistency specifications: the thread will only copy the same result to queries that are also pending, meaning that they are concurrent with the query of thread j. New queries that come after will trigger thread i to perform a new search of the sketch and the filters.

This optimization is made possible due to our design choice to delegate queries to other threads. In the next section, we evaluate its effects separately and show that it significantly increases the processing throughput, especially under highly skewed input.

7 Evaluation

We present a detailed evaluation of the performance of *Delegation Sketch* with respect to accuracy and processing throughput. First, we describe our experimental setup, followed by the experimental results.

7.1 Experiment Setup

Platform descriptions: We used two hardware platforms to evaluate our *Delegation Sketch*. *Platform A* is a dual socket NUMA server with 36 cores in total and 2-way hyper-threading at each core running at 2.1GHz, with 32KB L1 data cache, 256KB L2 cache and a 45MB shared L3 cache. It runs Ubuntu 16.04 and gcc v. 5.4. *Platform B* is a single socket, massively

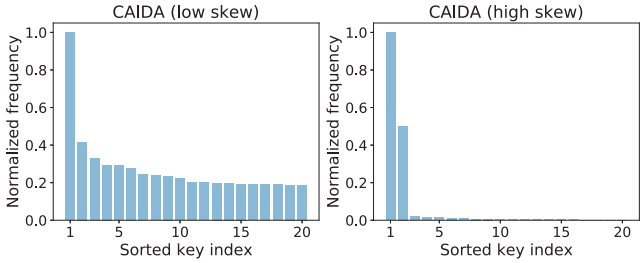


Figure 3. Normalized frequency of the 20 most frequent keys in the real world data sets used in the evaluation.

parallel Intel Xeon-Phi server with 72 cores, 4-way hyper-threading at each core running at 1.5GHz, using 32KB L1 data cache and 1MB L2 cache. It runs CentOS 7.4 and gcc v. 4.8.

Data sets: We used three sources of input data: *a) synthetic data* where the occurrence frequency of keys in the data set follow the Zipf distribution with a varying skew parameter. The Zipf distribution is widely used in the literature of sketch-based algorithms, as it captures the distribution of data related to many real world applications, such as packet counts, word count in a corpus of text, etc. *b)* two *real world data sets* taken from the CAIDA Anonymized Internet Traces 2018 Dataset [34]. From this trace, we use 22M packets that correspond to one minute of captured traffic from a high speed monitor. We extract the source IPs and source ports from the packet trace and use them as keys. This results in two input sets with very different characteristics: the frequencies at which IPs occur in the data set of IPs resemble a Zipf distribution with low skew, while the frequencies of ports in the data set of ports resemble a Zipf distribution with high skew. In Figure 3, we plot the normalized frequencies of the 20 most frequent keys for the two real world data sets.

As in [32], when we perform queries, we use the same distribution to determine on which keys we will perform them, i.e., we are more likely to perform queries for keys that are frequently found in the input stream.

Metrics: Our evaluation focuses on three metrics that are commonly used to characterize the performance of sketches: *accuracy*, *throughput* and *latency*. In Section 5, we already used the *average relative error* to evaluate the accuracy of different design choices. In this section, we additionally use the *absolute error per key* to indicate the over-approximation between the true frequency of a key in the stream and the frequency reported by the query. We report throughput as the number of operations (insertions or queries) per unit of time.

Parameters: In our experiments, we evaluate the effect of three main parameters: the *number of threads* in the system, the *skewness* of the input distribution and the *ratio of insertions vs queries* that each thread performs. Note that the *memory consumption* of each sketch is another important parameter that affects the performance of sketches in terms of accuracy and throughput. In order to have a fair comparison, we make sure that, for a given number of threads, *all versions use the same amount of memory. This includes all additional data structures involved, e.g., filters.* Since our delegation design needs memory for filters, we reduce the memory available to sketch accordingly, i.e., by using a smaller sketch, so that the total memory consumed is the same as the other designs we compare against. Similarly to [32], we achieve this by reducing the number of buckets at each row. Keeping the number of rows constant allows us to: (a) have the same δ probability bound for the estimate across all designs and (b) keep the number of hashes used (hence the cost of insertions/queries on the underlying sketches) the same across all designs. We quantify the effect this reduction of the number of buckets has on the overestimation error in the Appendix. For the case of the single-sketch parallelization, we increase the number of buckets as we add more threads, in order to have the same total size in memory as the other designs that use multiple sketches.

Baselines: We study the performance of *Delegation Sketch* in connection to the single-shared and thread-local sketches, described in Section 3. As described above, we keep the total amount of memory constant between different designs to ensure a fair comparison. We also include the Augmented Sketch using the thread-local design, i.e. we have one sketch and one filter per thread. In [32], the authors experiment with different filter sizes and evaluate the effectiveness of the filter. Based on that analysis, we use a filter size of 16 keys (and 16 counters) for all filters, including our Delegation Filters. In order to have a meaningful comparison with Augmented Sketch, we use Augmented sketch as the underlying sketch of *Delegation Sketch* i.e. every sketch in *Delegation Sketch* includes an additional 16 element filter. We also note that in our throughput evaluation (see later Section 7.3) we treat the Augmented Sketch baseline favourably: i.e., we do not attempt to enforce synchronization by making the filters thread-safe, i.e. the filters of Augmented Sketch can be accessed by any thread during queries. *Delegation Sketch* does not need special attention w.r.t. this, due to the synchronization mechanisms we describe in Section 6.

7.2 Comparing the accuracy of queries

In Section 5, we have already compared the accuracy of the different parallelization designs, in terms of Average Relative Error (ARE) and we have shown that, for the same total memory consumption, *Delegation Sketch* has very low ARE compared to the thread-local design. Moreover, *Delegation Sketch* is as accurate as the single-sketch design. We also showed that its accuracy increases with the number of threads.

Here we take a closer look at the accuracy of queries at each one of the input keys in our stream. For this experiment, we use a sketch with $d = 256$ and $w = 8$, use 4 threads and draw the input keys from the Zipf distribution with skew parameter 1. In Figure 4, we plot the error in the result of a query at every single key, using all the parallelization designs. For better presentation we have sorted the input keys based

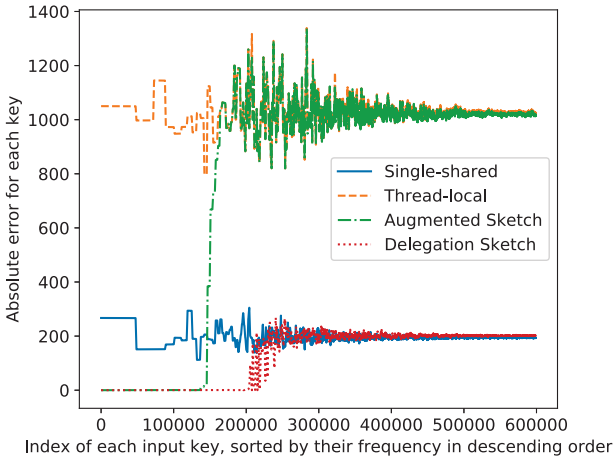


Figure 4. Error introduced for each key in the stream. The x-axis holds the indexes of each key in the stream, sorted by their frequency (descending order). The y-axis shows the absolute error added when performing a query on each key.

on their true frequency in descending order (e.g. the first 47K points in the x-axis correspond to the most frequent key, which has been seen 47K times in the input stream) and we plot the running mean of 1,000 keys.

Augmented Sketch and *Delegation Sketch* introduce no error on some of the most frequent keys in the stream, because of the filter used in the underlying sketch of both of those versions. Frequent keys are expected to be inserted in the filter and stay there most of the time. As a result, a query on those keys is more likely to report the true frequency of a key directly from the filter, rather than an approximation of it from the sketch. Note that this effect holds for more keys when using *Delegation Sketch* rather than Augmented Sketch. This is an effect of the domain splitting technique that reduces the range of keys that end-up at each sketch, thus making better use of the filter of the underlying sketch. *Delegation Sketch*, as expected according to the argumentation in Section 5.1, continues to be one of the most accurate ones even for low frequency keys, despite the fact that it uses a smaller sketch to accommodate space for the Delegation Filters.

7.3 Processing throughput

Here we evaluate the throughput of *Delegation Sketch* and compare it with the baselines, across the three following dimensions: (a) scalability with the number of threads, (b) query rate and (c) input skew. Finally, we evaluate the effect of the Query Squashing method (Section 6).

7.3.1 Overall scalability. Figure 5 shows the overall scalability of the different baselines for Platform A. For this experiment, we use input keys coming from the Zipf distribution with skew parameter 1.5. We gradually increase the number of threads, as well as the ratio of queries vs insertions. We report the average number of operations per second, out of 10 runs. We omitted standard deviation because it was insignificant in most cases.

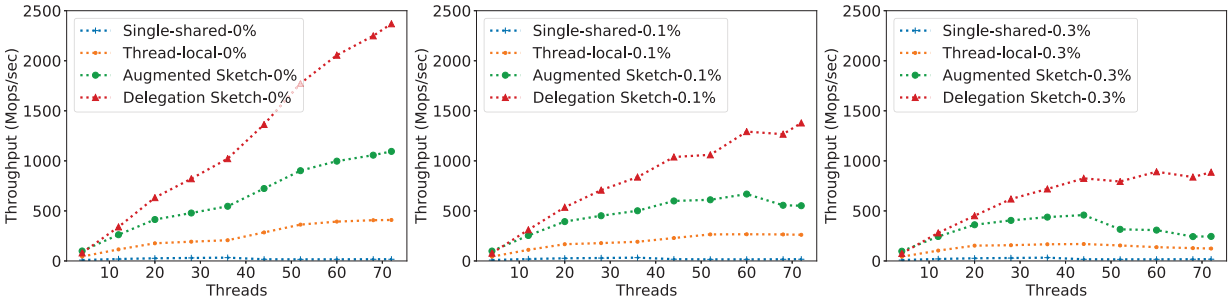
In Figure 5a, we present the results from the execution of a workload that contains only insertions. We see that the single-shared parallelization design cannot scale with the number of threads, while the thread-local designs (including parallel Augmented Sketch), as well as *Delegation Sketch* benefit from paralellization. This is in accordance with the trade-off analysis of Section 3. Even in the absence of queries, *Delegation Sketch* is up to 2X better than the next best baseline (Augmented Sketch), especially with more than 10 threads.

The introduction of even a small percentage of queries (Figures 5b and 5c) has a significant effect on processing throughput and scaling. With the exception of the single-shared design, the absolute throughput of all other designs is reduced. The thread-local design and the parallel Augmented Sketch stop scaling after approximately 40 threads in the case of the 0.3% query workload (Figure 5c) and actually perform worse with more threads. This is because increasing the number of threads introduces more sketches to search when serving a query. On the contrary, *Delegation Sketch* continues to benefit from parallelization, achieving up to 4 times higher throughput than the best performing baseline (Augmented Sketch). Also note that, on this platform, *Delegation Sketch* continuous to scale even under the effect of hyper-threading (that starts at 36 threads).

The same performance trend continues to hold on Platform B (Figure 6). The raw throughput achieved by each version is different, since this architecture has different characteristics (e.g. lower clock speed), but *Delegation Sketch* continues to outperform the baselines in all cases, especially with workloads that involve queries. While the performance of *Delegation Sketch* stops increasing when adding more than 150 threads in the 0.3% query workload, it is still more than 2 times faster compared to the baselines.

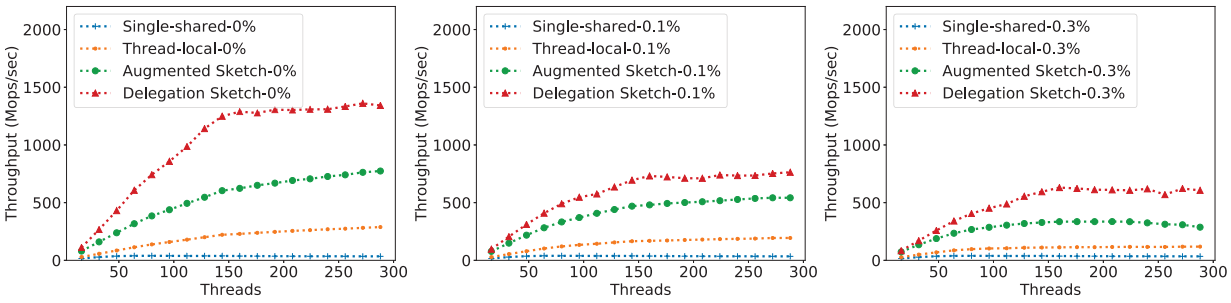
7.3.2 Evaluating the effects of query rates. We now turn our attention to query rates and evaluate how they affect performance. In this experiment, we use all the available parallelism on each platform and plot the achieved throughput in Figure 7. For both platforms, increasing the rate of queries in the workload has no effect on the relatively low throughput of the single-shared design. Contrary, all other parallelization designs suffer a performance hit, even at a low query rate (0.1%). In the case of *Delegation Sketch*, this is because increasing the number of threads increases the number of filters that must be searched during a query. However, *Delegation Sketch* sustains an overall higher throughput than the baselines, because it avoids searching multiple sketches.

7.3.3 Evaluating the effects of input skew. We now evaluate the effects of input skew on the performance of *Delegation Sketch*. In Figure 8, we present the throughput of all parallelization designs as we gradually increase the skew parameter of the distribution that generates the input keys,



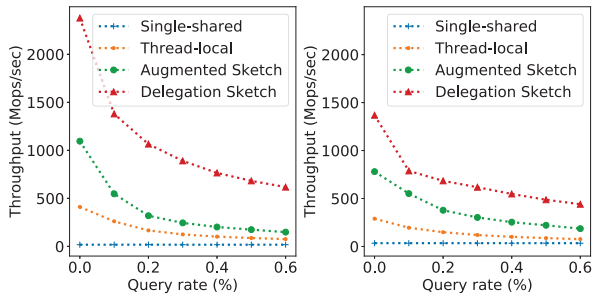
(a) 0% queries. **(b)** 0.1% queries. **(c)** 0.3% queries.

Figure 5. Platform A: Throughput and scalability comparison of all designs, using data from the Zipf distribution (skew=1.5).



(a) 0% queries. **(b)** 0.1% queries. **(c)** 0.3% queries.

Figure 6. Platform B: Throughput and scalability comparison of all designs, using data from the Zipf distribution (skew=1.5).



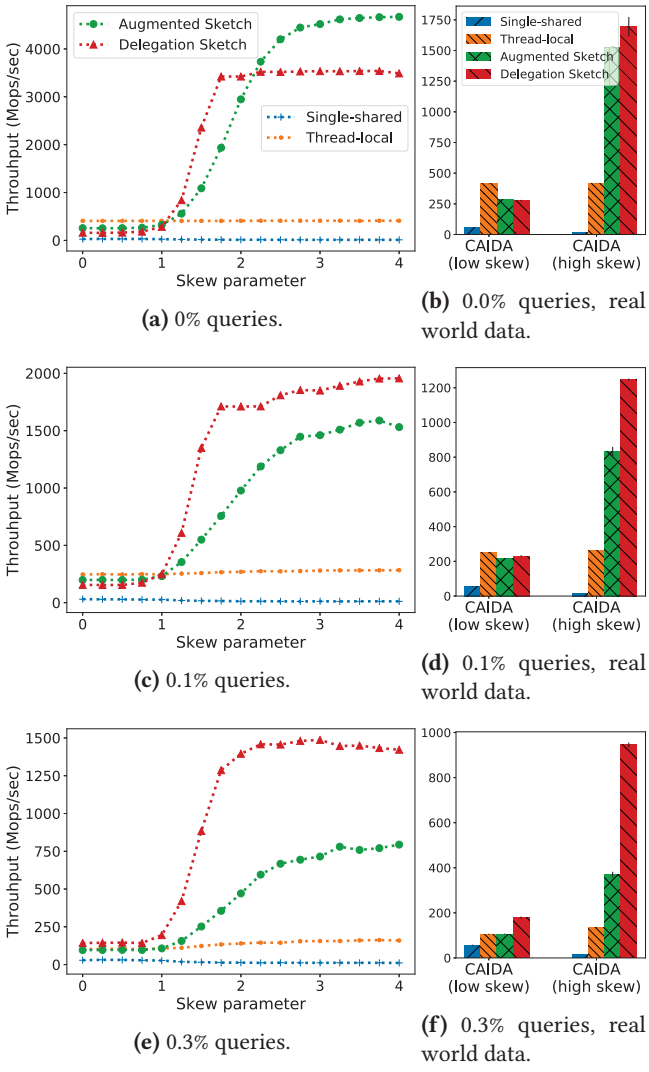
(a) Platform A. **(b)** Platform B.

Figure 7. The effect of queries on the performance of the different parallelization designs, across two platforms. Both sets of experiments use data from the Zipf distribution with a skew parameter of 1.5.

using three different query workloads. In the same figure, we also include the throughput achieved when using the two real world data sets we introduce in Section 7.1. We show the results of the execution in platform A and omit the results from platform B because they are equivalent.

In general, Augmented Sketch and *Delegation Sketch* gain a dramatic increase in throughput when the skew parameter is more than 1.0. This is because both versions rely heavily on filters, that accelerate the processing of keys that are frequently found in the input. This result is in accordance with the experimental evaluation of [32]. At low skew (parameter values 0-1) the thread-local design that does not use filters outperforms all others, since in this case the filters only add overhead. For medium skew (parameter values 1-2), *Delegation Sketch* outperforms Augmented Sketch even if there are no queries. This is due to: (a) the use of more filters (T delegation Filters per sketch) and (b) the domain splitting technique that reduces the range of keys that end up at each filter, making the input on that filter appear more skewed. At higher skew levels, most of the input stream is dominated by a few frequent elements. At this point, throughput stops increasing and Augmented Sketch outperforms *Delegation Sketch*. This is because, under such a high skewness, the per-key processing is so small that even the added overhead of computing $Owner(K)$ for *Delegation Sketch* becomes relatively significant. As expected, when we introduce queries in the workload, *Delegation Sketch* quickly outperforms the Augmented Sketch, even under high input skew (Figures 8c and 8e).

The same relative trends also hold with real-world data sets (Figures 8b, 8d and 8f). With the IP data set that exhibits low skew, the thread-local design outperforms the filter based ones in most cases, but *Delegation Sketch* performs better when using real-world data with high skew, especially at 0.3% query rates where it is more than 2 times faster than Augmented sketch and roughly 9 times faster than thread-local.



(a) 0% queries.

(b) 0.0% queries, real world data.

(c) 0.1% queries.

(d) 0.1% queries, real world data.

(e) 0.3% queries.

(f) 0.3% queries, real world data.

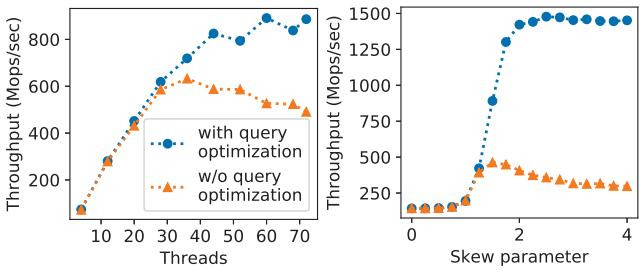
Figure 8. Platform A: Throughput comparison for different input skew and real data, using all the available threads (72). Note the different y-axis scale.

7.3.4 Evaluating the effects of query squashing.

We now evaluate the effect of Query Squashing separately. We compare the performance of *Delegation Sketch* to a modified version that does not include the optimization.

Figure 9a shows the scalability of both versions, in the same setting as the one used for Figure 5c. We see that, without the optimization, throughput starts to drop after 20 threads and cannot scale to more than 36 threads. This is because, after that point, a large number of threads attempt to perform queries and as a result the query operation becomes a bottleneck, especially on the thread that is responsible for the most frequent key in the stream. At 72 threads, our optimization brings roughly 1.8X speedup in throughput.

The same effect holds when we increase the input skew of the stream. Since the keys we perform queries on come from the same distribution as the keys we insert (see Section 7.1), when skew is high, most threads try to query the same key K and have to wait for the thread $i = Owner(K)$ to handle them. Our optimization manages to overcome that bottleneck: by "squashing" all those queries into one operation, thread i is able to handle them all without repeatedly searching the filters and the sketch for the same key. At high skew (parameter value of 3.0), Query Squashing brings up to 4.5 times speedup in throughput, without introducing any overhead when the skew is low.



(a) Scalability.

(b) Effect of input skew.

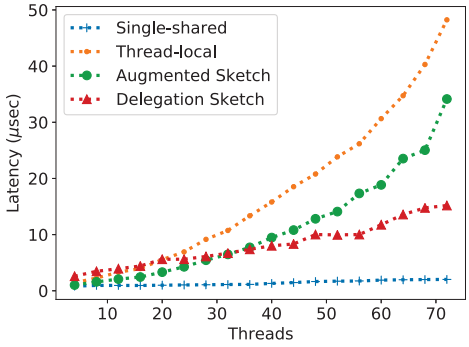
Figure 9. The effect of Query Squashing, compared to a modified version that does not include it. Left: scaling with the number of threads for fixed input skew. Right: the effects of input skew, using 72 threads. In both cases the workload contains 0.3% queries.

7.4 Query latency

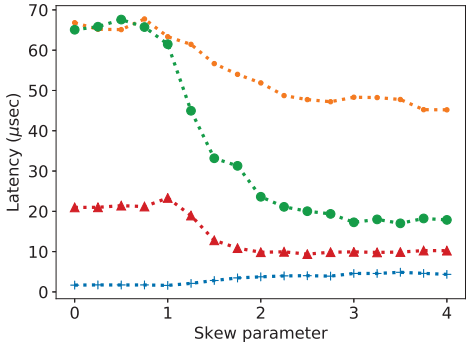
So far, we evaluated performance based on the throughput of operations. We now take a closer look at the latency of queries across different versions.

In Figure 10a, we present the average latency of query operations depending on the number of threads, using data taken from the Zipf distribution with skew parameter 1.2. Overall, the single-shared sketch has extremely low query latency, less than 2.5 μsec, which only rises slightly at more than 36 threads. As expected, queries in the single-shared approach are very efficient since they only need to search for the key in a single sketch, albeit at the cost of low insertion rate, as shown in Figure 5a. The latency of the thread-local design increases quickly with the number of threads, since the number of sketches that need to be searched increases. Augmented sketch has lower latency compared to the thread-local design, since some of the keys will be found in filters instead of sketches, but the overall latency remains high. *Delegation Sketch* manages to retain a lower latency than thread-local and Augmented Sketch under high parallelism (up to 2.25X and 3.18X times lower than the Augmented Sketch and thread-local respectively), since we search for a key in multiple filters but in at most one sketch.

Next, we fix the number of threads to 72 and vary the input skew. Again, the query latency of the single-shared approach is very low, but increases slightly under high skew,



(a) Latency of queries as the number of threads increases.



(b) Latency of queries for different input skew.

Figure 10. Platform A: Latency of query operations as the number of threads and the input skew increase.

since the parts of the sketch where the common keys hash into become contention points. When the skew is low, the query latency of *Delegation Sketch* is up to 3X lower than that of Augmented Sketch and thread-local, since we avoid searching in multiple sketches. As the skew increases, the latency of approaches that use filters (Augmented Sketch and *Delegation Sketch*) is reduced, but *Delegation Sketch* manages to outperform Augmented Sketch, through the use of our Query Squashing optimization.

In the above comparisons, also note the following: a) Queries in *Delegation Sketch* occasionally perform extra work before they are completed e.g. helping insertions by flushing filters into sketches. That extra work is still taken into account in the query latency. b) As already mentioned in Section 7.1, the queries of Augmented Sketch are treated favorably since we do not introduce any synchronization.

7.5 Summary of the evaluation

In summary, we study how *Delegation Sketch* fares with respect to the following dimensions: accuracy, scalability, support for concurrent queries and skewed input. We showed that *Delegation Sketch* is as accurate (and often better) than the most accurate baseline with the same amount of memory, due to the Domain Splitting technique. At the same time, *Delegation Sketch* is highly scalable on both a NUMA multi-core and a massively parallel architecture.

The benefits of *Delegation Sketch* are more pronounced in the presence of concurrent queries, where *Delegation Sketch* achieves a higher relative speedup than the best performing baseline (up to 4X) and continue to hold the presence of highly skewed input distributions.

8 Related work

In this section, we present related work on sketches, the parallelization of sketches, and on systems that use them.

Sketches: The literature contains numerous sketch designs. Cormode [4, 6] present an overview of synopsis and sketch base approaches. We summarize the most relevant here. Apart from the Count-Min Sketch (summarized in Section 2), there are more sketches that follow similar techniques. The Count-Sketch [3] uses the median (rather than the minimum) of the hashed counters to provide an unbiased approximation (rather than an over-approximation). FCM [39] dynamically adjusts the number of counters that are incremented to reduce the approximation error on low frequency items. HeavyGuardian [42] separates the counter for high frequency items, while instead ColdFilter [48] uses a filter to count the frequencies of low-frequency items. The aforementioned sketches provide ingenious designs to optimize the space-accuracy-performance trade-off. *Delegation Sketch* is a generic parallelization design that is orthogonal to the underlying sketch, so it can be used to improve the parallel performance of the above mentioned sketches.

Parallelization: As already mentioned, most work in sketches focuses on the single thread scenario. The few papers that discuss parallelization of sketches suggest either a single-shared sketch or a thread-local approach. ElasticSketch [43] proposes a new sketch that is parallelized for multi-cores using a thread-local design. Alipourfard et al. [1] evaluate different measurement algorithms and show that the thread-local design has lower latency compared to the single-shared one. In contrast to our work, both of these do not consider continuous concurrent queries. Mandal et al. [27] use multiple copies that can be periodically merged into a single sketch, which improves the accuracy but requires a lot of communication between threads. FCM [39] uses multiple copies and splits the domain of keys to buckets of threads, similar to our *Delegation Sketch*, but their design is specific to the Cell processor that has a single centralized unit that distributes the stream to several co-processors.

Tangwongsan et al. [37] follow a single-shared sketch design and include algorithms for many query types, however their work lacks an experimental evaluation and comparison of their results. Das et al. [8] also keep a single shared summary and combine multiple operations, similar to our design. However, their approach is specific to the Space-Saving summary. Taşyaran et al. [38] use a single sketch but parallelize the hash computations among threads, at the cost of requiring frequent synchronization points. Roy et al. [32] propose

a form of pipeline parallelism between two threads, where one thread is responsible for the filter while another handles the underlying sketch. Rinberg et al. [30, 31] is, to our knowledge, the only other work that considers concurrent queries and insertions. They reduce the need for synchronization by introducing more relaxed semantics for sketches and rigorously prove correctness and linearizability with respect to those semantics. As they also explain, this relaxation comes at the cost of accuracy. Observing that our evaluation here shows that *Delegation Sketch* is as accurate as the best performing alternative, while maintaining scalability, it is worthwhile to note the interest in cross-studying further concurrency-aware consistency formulations.

Applications and Systems: Sketches have found uses in many applications, especially in traffic monitoring. These have often lead to the creation of monitoring systems that use sketch designs tailored for such applications, e.g., in software or hardware switches. UniMov [26] and HashPipe [33] are implemented for smart network cards and heavily rely on sketches. NitroSketch [25] builds on top of a fast packet I/O framework in the user-plane (DPDK) and supports high insertion speeds that reach line rate on 40Gbps links. Apart from traffic monitoring, Garofalakis et al. [13] use sketches to reduce state transfer in distributed monitoring system. However, none of those those systems discuss the trade-offs related to concurrent queries when parallelizing sketches.

9 Conclusions and Future Work

In this paper, we introduce *Delegation Sketch*, a generic parallelization design for sketches achieving high processing rates and accuracy in the presence of concurrent queries. We analyze existing parallelization designs and find they cannot support both insertions and queries efficiently and have to prioritize one or the other. Contrary to this, *Delegation Sketch* reduces the cost of queries while at the same time maintains a high insertion rate through: (a) maintaining multiple parallel sketches, (b) splitting the domain of keys to different sketches and ensuring that all occurrences of the same key end-up in the same sketch, thus making queries faster and (c) using multiple filters that aggregate occurrences of the same key locally. By combining multiple concurrent queries of the same key, *Delegation Sketch* significantly reduces the cost of queries under high input skew.

We thoroughly evaluate *Delegation Sketch* across multiple dimensions, namely: accuracy, scalability, query rate and input skew. We use two massively parallel platforms and experiment with both real and synthetic data. Our evaluation shows that *Delegation Sketch* is able to scale up to more than 72 threads, especially in the presence of concurrent queries, achieving from 2.5X to 4X higher throughput than the best performing baseline, while being as accurate as the most accurate baseline. These results contribute to efficient and accurate stream processing, especially for applications that require frequent, responsive queries to dynamically changing data streams, e.g., software switches that schedule traffic dynamically or spatio-temporal monitoring applications and more [11, 14, 35, 46]. Extending the design with support for more types of queries, for even higher impact on such applications is an interesting future direction. It is also interesting to consider a co-design of *Delegation Sketch* together with efficient concurrent implementations of the underlying sketches (e.g. using concurrent counters [18, 40, 41]), as well as a distributed deployment across multiple nodes.

Acknowledgements

We would like to thank the anonymous EuroSys reviewers and our shepherd Aleksandar Prokopec for their valuable feedback that helped improve the paper. The research leading to these results has been partially supported by the Swedish Civil Contingencies Agency (MSB) through the projects RICS and RIOT, by the Swedish Foundation for Strategic Research (SSF) through the framework project FiC, by the Swedish Research Council (VR) through the project ChaosNet and the project AgreeOnIT, the Vinnova-funded project "KID-SAM", and from the European Community's Horizon 2020 Framework Programme under grant agreement 773717.

References

[1] Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. A comparison of performance and accuracy of measurement algorithms in software. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 18:1–18:14, New York, NY, USA, 2018. ACM.

[2] Arm. ODROID-XU3. https://developer.arm.com/graphics/development-platforms/odroid-xu3. Accessed: 2019-11-04.

[3] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 693–703, Berlin, Heidelberg, 2002. Springer-Verlag.

[4] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[5] Graham Cormode. *Count-Min Sketch*, pages 464–468. Springer New York, New York, NY, 2016.

[6] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

[7] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 35–46, New York, NY, USA, 2004. ACM.

[8] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *Proc. VLDB Endow.*, 2(1):217–228, August 2009.

[9] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 61–72, New York, NY, USA, 2002. ACM.

[10] Romaric Duvignau, Marina Papatriantafilou, Konstantinos Peratinos, Eric Nordström, and Patrik Nyman. Continuous distributed monitoring in the evolved packet core. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, pages 187–192,

2019.

[11] Zhang Fu, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Online temporal-spatial analysis for detection of critical events in cyber-physical systems. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 129–134. IEEE, 2014.

[12] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: A Brave New World*, pages 1–9. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[13] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. Sketch-based geometric monitoring of distributed stream queries. *Proc. VLDB Endow.*, 6(10):937–948, August 2013.

[14] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou, and Philippas Tsigas. Deterministic real-time analytics of geospatial data streams through scalegate objects. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 316–317, 2015.

[15] Red Hat. Receive Packet Steering (RPS). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rps, 2019. Accessed: 2019-10-22.

[16] Red Hat. Receive-Side Scaling (RSS). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rss, 2019. Accessed: 2019-10-22.

[17] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[18] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems (TOCS)*, 13(4):343–364, 1995.

[19] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 113–126, New York, NY, USA, 2017. ACM.

[20] Intel. Knights Landing. https://ark.intel.com/content/www/us/en/ark/products/codename/48999/knights-landing.html. Accessed: 2019-10-30.

[21] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. Sketchml: Accelerating distributed machine learning with data sketches. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1269–1284, New York, NY, USA, 2018. ACM.

[22] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77, 1985.

[23] Leslie Lamport. On interprocess communication, part ii: Algorithms. *Distributed Computing*, 1:86–101, 1986.

[24] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 311–324, Santa Clara, CA, March 2016. USENIX Association.

[25] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 334–350, New York, NY, USA, 2019. ACM.

[26] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.

[27] Ankush Mandal, He Jiang, Anshumali Shrivastava, and Vivek Sarkar. Topkapi: Parallel and fast sketches for finding top-k frequent elements. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 10921–10931, USA, 2018. Curran Associates Inc.

[28] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*, pages 69–82, 2013.

[29] Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 239–248. IEEE, 2015.

[30] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. Fast concurrent data sketches. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 117–129, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, and Hadar Serviansky. Fast concurrent data sketches. *CoRR*, abs/1902.10995, 2019.

[32] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1449–1463, New York, NY, USA, 2016. ACM.

[33] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA, 2017. ACM.

[34] The CAIDA UCSD anonymized internet traces - 2018. http://www.caida.org/data/passive/passive_dataset.xml, 2019. Accessed: 2019-11-03.

[35] Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Continuous monitoring meets synchronous transmissions and in-network aggregation. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 157–166. IEEE, 2019.

[36] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 285–294. IEEE, 2012.

[37] Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Parallel streaming frequency-based aggregates. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 236–245, New York, NY, USA, 2014. ACM.

[38] Fatih Tasyaran, Kerem Yildirir, Kamer Kaya, and Mustafa Kemal Tas. One table to count them all: Parallel frequency estimation on single-board computers. *CoRR*, abs/1903.00729, 2019.

[39] Dina Thomas, Rajesh Bordawekar, Charu C. Aggarwal, and Philip S. Yu. On efficient query processing of stream counts on the Cell processor. In *2009 IEEE 25th International Conference on Data Engineering*, pages 748–759, March 2009.

[40] Junchang Wang, Tao Li, and Xiong Fu. Accurate counting algorithm for high-speed parallel applications. *Concurrency and Computation: Practice and Experience*, 31(13):e5090, 2019.

[41] Roger Wattenhofer and Peter Widmayer. The counting pyramid: an adaptive distributed counting scheme. *Journal of Parallel and Distributed Computing*, 64(4):449–460, 2004.

[42] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 2584–2593, New York, NY, USA, 2018. ACM.

[43] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM

'18, pages 561–575, New York, NY, USA, 2018. ACM.

[44] Tong Yang, Lingtong Liu, Yibo Yan, Muhammad Shahzad, Yulong Shen, Xiaoming Li, Bin Cui, and Gaogang Xie. Sf-sketch: A fast, accurate, and memory efficient data structure to store frequencies of data items. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 103–106, April 2017.

[45] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 29–42, Lombard, IL, 2013. USENIX.

[46] Nikos Zacheilas, Vana Kalogeraki, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. Maximizing determinism in stream processing under latency constraints. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 112–123, 2017.

[47] Yang Zhou, Peng Liu, Hao Jin, Tong Yang, Shoujiang Dang, and Xiaoming Li. One memory access sketch: A more accurate and faster sketch for per-flow measurement. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, Dec 2017.

[48] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 741–756, New York, NY, USA, 2018. ACM.

A Influence of filters on the overestimation error

When we query a key in *Delegation Sketch*, we search for it in all the filters it may be located, so the fact that a key is not immediately inserted in the sketch does not mean that its impact is ignored. In fact, in cases of extremely high skew, some keys may never get inserted in the sketch, meaning that their *exact* frequency will be reported from the filters.

However, the use of filters affects the existing error bound derived for domain splitting (Equation 6) in two opposing ways. We update the bound of Equation 6 here, following the analysis of [32] adapted to *Delegation Sketch*.

First, because we want to keep the total memory consumption the same as that of the baselines to have a fair comparison with *Delegation Sketch*, we compensate for the memory of the filters by reducing the number of buckets at each row from the underlying sketch, as explained in Section 7.1. For a *Delegation Sketch* with w buckets, d rows, a filter size of f and T threads, we subtract $\frac{Tf}{d}$ buckets from the sketch. This increases the overestimation error of queries, but the effect is low for reasonably sized sketches and filters, as shown also experimentally in Section 7.2.

Second, when querying the frequency of a key, some occurrences of the key will be found in the filters. The filters hold the exact number of occurrences and not an approximation, contrary to the query result found in the sketch for that key. That means that: (a) the overestimation error of a query decreases because parts of the answer will be given by the filters that hold exact counts and (b) fewer items will be inserted in the sketch, which reduces the approximation error due to collisions.

Overall, the error bound from Equation 6, due to the use of filter is now

$$f(i) \leq \hat{f}(i) \leq f(i) + \tilde{\epsilon}(N' - \tilde{N}) \tag{7}$$

where

$$\tilde{\epsilon} = \frac{e}{w - \frac{Tf}{d}} = \epsilon \frac{1}{1 - \frac{Tf}{wd}} > \epsilon \tag{8}$$

and N', ϵ are the same as in Equation 6. $\tilde{N}$ is the number of the items held in the filters of sketch i and depends on the skewness of the distribution and the size of the filters.