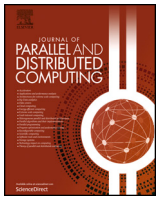
Journal of Parallel and Distributed Computing 204 (2025) 105134

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



QPOPSS: *Query and Parallelism Optimized Space-Saving* for finding frequent stream elements

Victor Jarlow [a,b,*], Charalampos Stylianopoulos [a,c], Marina Papatriantafilou [a]

[a] *Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden*
[b] *R&D Department, AstaZero, Gothenburg, Sweden*
[c] *Emnify, Berlin, Germany*

ABSTRACT

The frequent elements problem, a key component in demanding stream-data analytics, involves selecting elements whose occurrence exceeds a user-specified threshold. Fast, memory-efficient ϵ-approximate synopsis algorithms select all frequent elements but may overestimate them depending on ϵ (user-defined parameter). Evolving applications demand performance only achievable by parallelization. However, algorithmic guarantees concerning concurrent updates and queries have been overlooked. We propose Query and Parallelism Optimized Space-Saving (QPOPSS), providing concurrency guarantees. A cornerstone of the design is a new approach for the main data structure for the *Space-Saving* algorithm, enabling support of very fast queries. QPOPSS combines minimal overlap with concurrent updates, distributing work and using fine-grained thread synchronization to achieve high throughput, accuracy, and low memory use. Our analysis shows space and approximation bounds under various concurrency and data distribution conditions. Our empirical evaluation relative to representative state-of-the-art methods reveals that QPOPSS's multithreaded throughput scales linearly while maintaining the highest accuracy, with orders of magnitude smaller memory footprint.

1. Introduction

Efficient data synopses are a core component of many applications, including online processing of events, click-streams, web log analysis, natural language processing, heavy flow detection in computer networks, dimensionality reduction in machine learning (ML), and more [10,12,24,32,33]. In essence, data synopses can be used to answer queries pertaining to continuous data streams, and they give (often very close to accurate) answers with low memory usage and fast processing. For example, on a stream of web page clicks from a vast number of users, one may estimate unique users clicking a specific link, the most active users, quantiles describing the time spent on a web page, and more.

In the literature, there is an established volume of knowledge on data synopses [14,10,19,41,30,33]. As the rate at which streaming data is produced increases, new approaches that incorporate computational parallelism are required to provide continuous and efficient processing [11,43,37,38,47]. However, most prior work has not considered the impact of concurrent update and query operations on query correctness. This gap in the literature means that the way such algorithms perform (in terms of processing timeliness and accuracy) in a parallel execution is largely unclear.

In this work, we target the problem of identifying the *frequent elements* of a stream, i.e., those stream elements whose occurrence exceeds a user-specified threshold. The problem has applications in continuous monitoring, data processing, and ML pipelines, as discussed in related literature [9,16,21,29,40,42,44]. One intuitive application is in network traffic monitoring. An IP network flow identifies a connection and is represented by a 5-tuple of source and destination IP addresses, source and destination port numbers, and protocol. A small number of distinct flows, dubbed *elephant flows*, tend to make up a large share of the bandwidth consumption; tracking them is useful for accounting and statistics [18], detecting anomalous patterns such as DDoS attacks [22], and for dynamically scheduling network traffic in software-defined networks [3]. An exact answer requires tracking each unique flow. Due to the massive number of bit-combinations possible, this can consume memory space on the order of exabytes, making it an impractical approach. If a small and controllable error is acceptable, the memory consumed can be drastically reduced by using a synopsis data structure. Moreover, high-rate streams place demands on the per-packet processing time (PPT). Backbone routers, which process vast amounts of data, e.g., the optical carrier bandwidth specifications OC-192 and OC-768, can require a PPT of less than 100 ns and 25 ns, respectively [28].

* Corresponding author at: Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden.
E-mail addresses: victor.jarlow@ri.se, victor.jarlow@outlook.com (V. Jarlow), charalampos.stylianopoulos@gmail.com (C. Stylianopoulos), ptrianta@chalmers.se (M. Papatriantafilou).

https://doi.org/10.1016/j.jpdc.2025.105134
Received 24 July 2024; Received in revised form 13 March 2025; Accepted 12 June 2025
Available online 17 June 2025

Achieving such low PPT, in the presence of a high incoming packet rate, is in some cases only possible by parallelism.

Much of the previous work on *parallelizing frequent elements algorithms* [17,9,29] addresses performance on only updates, without evaluating the effects of concurrent queries. Understanding performance under concurrent updates and queries is crucial, since streaming queries often require real-time answers without interrupting high-rate processing, implying a trade-off between timeliness and accuracy. Considering this need, we make the following contributions:

- We introduce *Query and Parallelism Optimized Space-Saving* (QPOPSS), a novel extension of the Space-Saving algorithm [33], designed to support concurrent updates and queries while maintaining high approximation accuracy. QPOPSS balances between efficient memory use and real-time processing, a key advancement in handling high-rate streams with parallelism.
- A major insight of our work is the design of the *Query Optimized Space-Saving* (QOSS), which enables low-latency bulk queries in the Space-Saving algorithm. This, in turn, minimizes update overlaps, ensuring higher accuracy under concurrent operations, an issue often overlooked in previous work.
- We present a comprehensive analysis of the QPOPSS's properties, examining its performance under various concurrency and data distribution conditions. We establish novel space and approximation bounds and provide insights into the efficiency and accuracy trade-offs inherent in parallel data stream processing, an area that has been largely unexplored in existing research.
- Our extensive evaluation of the open-source implementation [26] uses real-world and synthetic high-rate data to explore trade-offs in parallelism, accuracy, memory consumption, and throughput. Through empirical comparison with state-of-the-art methods (e.g., Topkapi [29] and PRIF [44]), we demonstrate that QPOPSS achieves orders of magnitude lower memory usage, sometimes requiring 100x fewer bytes than other methods, while offering superior accuracy and linear speedup.

The rest of the paper is structured as follows: Section 2 covers the background, a description of the system model, and the basic metrics of interest. Section 3 analyzes the problem at hand, motivating a novel balanced approach, and outlines the associated challenges relative to the state of the art. For QPOPSS, we give an overview and its algorithmic implementation in Section 4 while Sections 5 and 6 cover its analysis and empirical evaluation. We discuss other related work and present our conclusions in Sections 7 and 8, respectively.

2. Preliminaries

In an unbounded stream of elements $S := \tau_1, \tau_2, ..., \tau_N, ...$ for any N, we say that an element $e \in U$ has a frequency count $f_N(e) = |\{j | \tau_j = e\}|$ after N elements have been processed, where U is the universe of possible elements. From this point on, we assume that S contains only elements of positive unit weight, also known as the cash-register model [20].

The *ϕ-frequent elements problem* is concerned with selecting the elements of a stream with a frequency count above ϕN, where $\phi \in [0, 1]$. To find the ϕ-frequent elements of an arbitrary stream using a deterministic algorithm, $O(|U|)$ space is necessary [27,30]. For applications where an approximation is acceptable, we can relax the exact problem to the *ϵ-approximate ϕ-frequent elements problem*,[1] defined in [31] as follows.

[1] From this point on, we sometimes refer to the *ϵ-approximate ϕ-frequent elements problem* as the frequent elements problem.

Definition 1. Given a stream S of N elements, the *ϵ-approximate ϕ-frequent elements* problem is to *report* a set F containing all elements $e \in U$ with $f_N(e) > N\phi$ and no element $e \in U$ with $f_N(e) < N(\phi - \epsilon)$, where $0 < \epsilon \leq \phi < 1$.

Accordingly, F must contain all elements with frequency of occurrence higher than $N\phi$ but may also include elements that occur at least $N(\phi - \epsilon)$ times due to approximation-induced errors.

Also specified in [31], a closely related definition concerns the estimated frequency count of an individual element.

Definition 2. Given a stream S of N elements, an *ϵ-approximate frequency estimation* denoted $\hat{f}_N(e)$, of the true frequency count of $e \in U$ is bounded: $f_N(e) \leq \hat{f}_N(e) \leq f_N(e) + N\epsilon$.

In other words, the estimated count of an element is bounded from above by the sum of the elements' actual frequency count and a fraction of the stream length, as decided by the ϵ-factor.

Algorithms that report the ϵ-approximate ϕ-frequent elements described in Definitions 1 and 2 generally provide at least two operations:

- ***Update(e,w)***: element e is processed, registering its occurrence in the stream. If *weighted* updates are supported, then w is the number of simultaneous arrivals of e to update the data structure with.
- ***Query(N, ϕ)***: Returns F from Definition 1 with estimated frequency count of each individual element $e \in F$ adhering to the bounds in Definition 2.

These algorithms can be distinguished into two classes:

Counter-based Algorithms: Typically deterministic and keep a fixed-size set that contains tuples of element and their estimated occurrences in the stream. When an individual element is observed, its associated estimated occurrence is incremented. The sets' fixed size demands a tactic for managing the occurrences of an element not in the set while the set is full. The tactic often leads to an incurred error but can be chosen to minimize the error depending on the area of use. Keeping more counters reduces the error and vice versa, highlighting the *memory space/error trade-off*, present in all synopsis data structures. Prominent such algorithms are the *Frequent* (also known as the *Misra-Gries*) Algorithm [34,18,27], *Lossy Counting* [31], and *Space-Saving* [33].

Sketch-based Algorithms: Utilize randomized hashing to compress the stream state into a 1- or 2-dimensional array of counters. Updating the sketch involves computing a hash value for the incoming element. A counter corresponding to the hash value is then, e.g., incremented or decremented, concluding the update operation. Since the same hash value can be computed for multiple different elements (i.e., hash collisions), the compressed stream state is encoded in the counters. Element queries are carried out by computing e.g. the minimum or median, on these counters. Heap data structures can be used to supplement sketches and track frequent elements. Similar to counter-based algorithms, sketch-based ones imply a memory space/error trade-off. Influential sketch-based algorithms that can form the components of a solution to the frequent elements problem include the *Count-Min Sketch* [15] and *Count Sketch* [10].

Comparison: Counter-based algorithms demonstrate superior accuracy per memory byte when processing a continuous stream of positive updates compared to sketch-based algorithms [25]. Moreover, Space-Saving has been shown to guarantee better accuracy than both Frequent and Lossy Counting while having better or equivalent throughput compared to them [14].

Space-Saving For self-containment, we summarize the basics of Space-Saving [33] for the ϵ-approximate frequent elements problem: a set of m tuples of the form $(e, \hat{f}_N(e))$ are kept. If an element e that is in the set arrives, the associated estimated count $\hat{f}_N(e)$ is incremented. If e is not in the set, and the set contains fewer than m tuples, then $(e, 1)$ is added to the set; if the set contains m tuples, though, the one with the least

counter, $(e_{min}, \hat{f}_N(e)_{\{min\}})$ is identified, and the tuple $(e_{new}, \hat{f}_N(e)_{min}+1)$ takes its place, replacing e_{min} and incrementing the estimated count by 1. When queried, the tuples with estimated frequency $> N\phi$ are output. Setting the number of counters to $m = \frac{1}{\epsilon}$ ensures that the set of ϵ-approximate ϕ-frequent elements is returned [33].

Concurrency model: We consider a multicore system, with executions threads $t_1, ..., t_T$, that do not arbitrarily fail or halt, and are capable of communicating using asynchronous shared memory, 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. Each thread can perform one of two *operations* at a time, i.e., do an *update*, by consuming an item from the input stream, or respond to a *frequent elements query*.

Performance metrics: These are about both *time/space efficiency* and *error* [14,30,33]. Key time/space metrics are *latency* (duration of operations), *throughput* (number of updates or queries carried out per unit of time), *scalability* (the ability to utilize efficiently multiple threads), and *consumed memory*. Regarding error metrics, the aforementioned ϵ factor can be tuned to impact the *precision* (fraction of relevant elements reported out of all reported elements), *recall* (fraction of relevant elements reported out of all relevant elements), and *average relative error* (the average of all per-element absolute estimation errors divided by each actual in-stream occurrence). Further, we need to consider the effects of concurrency on estimation error, which we discuss in the following section, analyzing key trade-offs.

3. Problem analysis

Several works present designs that leverage parallelism for the frequent elements problem [44,29,9,16,40]. The previous designs can be distinguished into those based on *Global Data Structure* and *Thread-local Data Structures*. We discuss their associated trade-offs and describe the problem of query accuracy as it relates to concurrency. Lastly, we motivate the need for a new approach.

3.1. Global data structure

This design uses a shared data structure that all threads access for querying and updating. As a result, efficient synchronization methods are crucial to the algorithm's design. The memory footprint is similar to sequential frequent elements algorithms, with any excess memory being attributed to inter-thread synchronization. The main benefit is that a query can be answered by accessing the single synopsis tracking the ϵ-approximate ϕ-frequent elements.

The Cooperative Thread Scheduling Framework (CoTS) [16] and its multi-stream extension [17] belong to this category. The design allows threads to delegate their updates to the single thread currently accessing the shared data structure. The works lack discussion of the effect of overlapping updates and queries on accuracy and rely on the theoretical bounds of the sequential Space-Saving algorithm.

3.2. Thread-local data structures

These designs utilize one synopsis data structure per thread. Since each thread only updates its local data structure, the update rate scales well with the number of threads and can be carried out without synchronization. Due to data structure duplication, these designs sacrifice accuracy guarantees and use memory proportional to the number of threads, exceeding that of a sequential solution. Moreover, to perform a query, a thread must merge multiple synopses, which can become a major source of latency with a high number of threads.

The Topkapi sketch [29] and the parallel algorithm in [9] utilizing Space-Saving, follow this approach; worth noting is that neither of them supports overlapping queries and updates. The PRIF algorithm in [44] allows overlapping queries and updates. Its memory-intensive design features several thread-local data structures that periodically update a single large data structure that serves as the final synopsis. While the memory/accuracy trade-off is rigorously examined, query throughput and concurrency guarantees are not discussed.

3.3. Accuracy and consistency

Since synopsis queries target *approximate* output, it can be observed that strict consistency requirements and associated synchronization can induce an overly excessive, partly unnecessary overhead in the concurrent setting. Works on concurrency-aware semantics discuss notions such as regularity, intermediate value linearizability, k-out-of-order relaxation and more [23,35,37,38]. These specifications model operations on objects that return values complying with "observing" associated subsets of updates, relative to the sets implied by linearizability or sequential consistency. Such concurrency models inspire further analysis of accuracy and consistency for queries overlapping updates.

3.4. Need for a balancing approach

The aforementioned categories represent two contrasting perspectives. First, the global-data structure can be scanned quickly, enabling high query throughput. However, that single data structure can become detrimental under high-rate streams processing, due to synchronization overhead. Secondly, the thread-local approach allows for high update throughput and scalability since multiple threads can process stream elements in parallel. However, the frequent elements are scattered across several distinct data structures that occupy precious bytes of memory and require latency-inducing assembly upon querying.

Clearly, there is a need for a concurrent approach to the frequent elements problem that balances and preferably combines, to a large extent, the properties of both categories favorably: a high update/query throughput, low memory space, high accuracy, and low query latency solution to the highest possible extent. Moreover, an essential missing part in the literature is a thorough analysis (both theoretical and empirical) of the effects of overlapping updates and queries. This analysis can serve as a tool for further exploration of the involved trade-offs. To this end, we identify a *set of challenges* in designing a balanced approach, namely to enable:

- **[C1]** high query and update throughput without impacting query latency;
- **[C2]** parallelism without an inherent impact on memory and accuracy;
- **[C3]** reasoning about accuracy guarantees in the presence of overlapping updates and queries.

We discuss how we address these challenges in the following section, where we present our method. The properties of our proposed method regarding challenge [C3] are further discussed in Section 5.

4. Query and Parallelism Optimized Space-Saving

In this section, we present our proposed method, an accuracy-preserving multithreaded design for finding the frequent elements of a stream, supporting concurrent updates and queries. Due to the properties of Space-Saving as mentioned in Section 2, we chose it as a component in our design, hereafter referred to as Query and Parallelism Optimized Space-Saving (QPOPSS).

We begin with an overview of the QPOPSS design and its components along with some auxiliary concepts, followed by our sequential Space-Saving algorithmic implementation, with latency optimizations as motivated in Section 3.4. Finally, we describe both the update and query procedure of QPOPSS in detail.

4.1. Design overview

Addressing [C1]: We present our proposal to reduce query latency and improve query throughput drastically, namely the *Query Optimized*

Space-Saving (QOSS) algorithmic implementation, that diminishes overlaps of queries with concurrent updates. A central element of Query Optimized Space-Saving is the min-max heap data structure [5], through which we can easily find both the element with the least count, which is essential to the Space-Saving update procedure, and the elements with the largest counts, promoting high query throughput. This is achieved as the min-max heap groups elements with similar counts together, allowing for their swift selection during a query.

Addressing [C2]: For maintaining accuracy and space guarantees in a concurrent setting, we identified that domain splitting [43] and delegating responsibility for a subset of the domain of possible elements to each thread through fixed-size, bounded filters can benefit Space-Saving's accuracy and memory efficiency, as we show in Section 5. This implies a new challenge regarding performing a global query over parts of the data structure maintained by different threads; we explain how we address this through efficient synchronization in section 4.5.

Addressing [C3] – in conjunction with [C1]: Regarding *concurrent updating*, we build on a thread-cooperation technique, shown in [43], that promotes thread-local updates to a large extent. We extend it to support *global queries*, which we describe with proper context in the upcoming subsection 4.4. Regarding *concurrent querying*, a lightweight query procedure leads to minimal contention on the QOSS algorithm data structures in memory, in conjunction with buffered updates. Furthermore, the risk of overlapping thread access is reduced, promoting high throughput and low latency. We analyze the consistency-related implications of our query method in Section 5.

4.2. Auxiliary concepts

Besides query optimization that targets parallelism-aware accuracy, QPOPSS builds on two concepts from [43], which we reiterate here, for self-containment.

Domain Splitting partitions U, the input domain, and distributes ownership to each of the T threads, using the function $owner : U \to \{1..T\}$, with $U_i = \{e \in U \mid owner(e) = i\}$ denoting the subdomains. This can be implemented using a simple hash function, such as modulo: $owner(e) = e \bmod T$.

Delegation Filters facilitate efficient inter-thread communication by buffering elements owned by other threads. Full filters are handed to the thread owning the contained elements. For each of the T dispatched threads, a series of T Delegation Filters are reserved for each thread, arranged in a $T \times T$ matrix, bounding the consumed space. Delegation filters are small and fixed in size, ensuring a low memory footprint.

At this point, we depart from the approach taken in [43] and describe the foundations of our method in the following subsections, starting with our query-optimized Space-Saving algorithmic implementation in the following subsection.

4.3. Query Optimized Space-Saving

Emphasizing improved query processing timeliness, we propose our Query Optimized Space-Saving (QOSS) algorithmic implementation of Space-Saving. QOSS retains the accuracy guarantees and memory requirements of Space-Saving while using optimized underlying data structures and query procedures.

Efficient implementations of Space-Saving include the *Space-Saving Linked List* (SSL) and *Space-Saving Heap* (SSH), as evaluated in [14]. SSL, similar to the *Stream-Summary* in [33], maintains a linked list sorted by estimated frequency. SSH, on the other hand, stores elements in a *min-heap* of size m, enabling retrieval of the least frequent element in $O(1)$ time. While SSH is slightly slower than SSL, it requires significantly less memory and supports *weighted updates*, which SSL does not. Weighted updates allow multiple occurrences of an element to be processed at the same cost as a single update, making SSH the preferred choice in weighted settings [4], as required in our work.

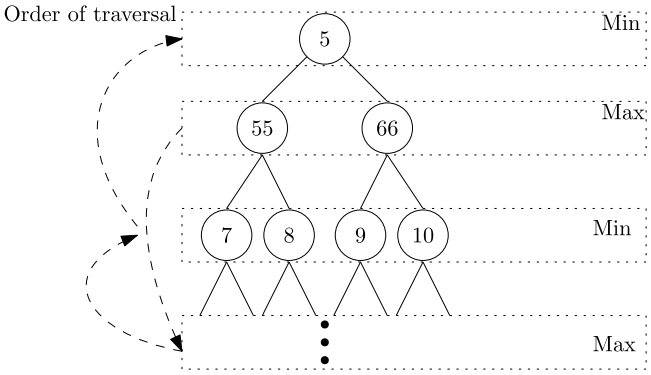


Fig. 1. A binary min-max tree with alternating levels. The dashed arrows depict the traversal order during a QOSS query.

However, a notable *shortcoming* of SSH is that answering a query requires traversing an array of size m. During this traversal, each element's count is compared to a threshold to determine if it belongs to the output set, resulting in a time complexity linear in m.

To alleviate this shortcoming and address [C1] of Section 3.4, the *Query Optimized Space-Saving* (QOSS) keeps a *min-max heap* [5] data structure of size m with the counter count satisfying that:

- at an even level (min-level) it is less than all of its descendants;
- at an odd level (max-level) it is greater than all of its descendants.

The min-max heap allows: a) finding the least element in $O(1)$, which is essential for performing update operations quickly, and b) performing a query in $O(|F|)$ time, where $|F|$ is the number of frequent elements from Definition 1. The number of elements in F is commonly significantly less than m, especially when the input stream is skewed. This modification introduces a slight per-element processing overhead. This overhead is overshadowed by the overall throughput gain when queries are repeatedly carried out while high-rate streams are processed.

Algorithm 1 shows the QOSS pseudocode and a description of it follows here.

Initialization: As seen in Algorithm 1 lines 1 through 5, the number of counters, m, is determined using the desired ϵ-factor. In line 3, steps ensure that each node has either 3 or 0 grandchildren. A counter is a tuple consisting of an element identifier e and an estimated count $\hat{f}_N(e)$. The m counters are organized in a bounded-size tree structure as a min-max heap denoted H (see line 4). Counters can be accessed through a fixed-size hash map M, initialized at line 5.

Updates: Lines 6 through 12 detail the update procedure. If e is found through hash-table lookup, $\hat{f}_N(e)$ is incremented by w, the weight of the update. Otherwise, e takes the place of the counter with the least count, $\hat{f}_N(e)_{min}$, which is incremented by w. If the min-max heap property of H is violated, it will be restored through at most $O(log(m))$ swaps involving $(e, \hat{f}_N(e))$.

Queries: The query procedure leverages the min-max heap to minimize unnecessary counter-threshold comparisons. It prioritizes max-level counters first, as they have higher counts and are more likely to belong to F (see Fig. 1). A stack is used for traversal, while a set V tracks visited parent and grandparent counters when traversing min-levels (lines 15-16). Traversal begins with the children of the root counter, which hold the largest counts (line 17). If a max-level counter exceeds $N\phi$, it is added to the output (lines 21-23), and traversal continues. Otherwise, since all descendant counters must be below $N\phi$, traversal stops. When reaching the lowest max-level, traversal proceeds upwards through min-levels until returning to the root counter (lines 31-32, 34-36).

Query Time Complexity: With a binary min-max heap, a query requires at most $5|F|$ counter comparisons. This follows from selecting any subtree rooted at $r \in F$. As stated in line 26, each subtree incurs at most

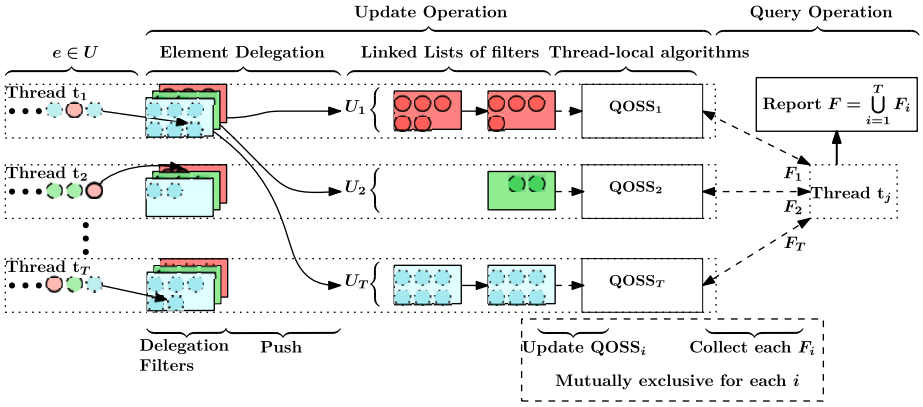


Fig. 2. Overview of the update and query operations. Thread t_1 transfers full filters to the owner-threads for subsequent insertion into the reserved thread-local QOSS data structures. Queries are mutually exclusive with insertions and gather the subset of frequent elements tracked by each thread into F.

Algorithm 1 Algorithmic implementation of the Query Optimized Space-Saving (QOSS) algorithm using a binary min-max heap.

1: **function** $InitializeQOSS(\epsilon \in [0,1])$
2: $m \leftarrow 4\lfloor \frac{\lceil \epsilon^{-1} \rceil}{4} \rfloor + 3$ ▷ All nodes have 3 or 0 grandchildren
3: let H be a bounded min-max heap of m counters initialized to $(\emptyset, 0)$
4: let M be a bounded hash map of $O(m)$ pointers to counters
5: **function** $UpdateQOSS(e \in U,\ w \in \mathbb{N})$
6: **if** $(e, \hat{f}_N(e)) \in M$ **then**
7: $i \leftarrow M.Find(e)$
8: $H[i] \leftarrow (e, \hat{f}_N(e) + w)$
9: **else**
10: $(\_, \hat{f}_N(e)_{min}) \leftarrow H[1]$
11: $H[1] \leftarrow (e, \hat{f}_N(e)_{min} + w)$
12: Ensure min-max heap property of H is maintained
13: **function** $QueryQOSS(\phi \in [0,1],\ N \in \mathbb{N})$
14: initialize empty stack
15: V $\leftarrow \emptyset$
16: Begin by pushing the two max counters $\{2, 3\}$ to the stack
17: **while** stack is not empty **do**
18: $i \leftarrow$ stack.pop()
19: $(e, \hat{f}_N(e)) \leftarrow H[i]$
20: **if** $\hat{f}_N(e) >= \phi N$ **then**
21: Output $(e, \hat{f}_N(e))$
22: traverse_next_level()
23: **function** *traverse_next_level*
24: **if** $\lfloor \log_2(i) \rfloor \equiv 1 \mod 2$ **then** ▷ i is on a max-level
25: **if** $4i + 3 <= m$ **then** ▷ i has grandchildren
26: push $4i + j, j \in \{0, 1, 2, 3\}$ to stack
27: **else**
28: **if** $2i + 1 <= m$ **then** ▷ i only has children
29: push $2i + j, j \in \{0, 1\}$ to stack
30: **else** ▷ i has no children or grandchildren
31: push $\lfloor \frac{i}{2} \rfloor$ to stack if $\lfloor \frac{i}{2} \rfloor \notin V$
32: $V \leftarrow V \cup \{\lfloor \frac{i}{2} \rfloor\}$
33: **else** ▷ i is on a min-level
34: **if** $\lfloor \frac{i}{4} \rfloor > 0$ **then** ▷ i has a grandparent
35: push $\lfloor \frac{i}{4} \rfloor$ to stack if $\lfloor \frac{i}{4} \rfloor \notin V$
36: $V \leftarrow V \cup \{\lfloor \frac{i}{4} \rfloor\}$

four counter comparisons. Summing over all $r \in F$ results in a total of $|F| + 4|F| = 5|F|$ comparisons. This optimization significantly reduces the number of comparisons, especially in practice, where input streams often exhibit skewed distributions

The effects of the QOSS improvements are studied in Section 6, comparing the algorithmic implementation to a baseline and evaluating the query latency and throughput. A better query latency implies less overlapping with concurrent updates, facilitating improved accuracy.

Algorithm 2 Update operation on thread j. Constants D and E are user-defined and describe the filter size and the maximum number of processed elements per thread before handover.

1: **function** $UpdateQPOPSS$(Element e)
2: $i \leftarrow$ *Owner* (e)
3: $Filter \leftarrow Threads[i].DelegationFilters[j]$
4: (this $Filter$ is reserved for thread j)
5: **if** $e \in Filter$ **then**
6: Increment count of e
7: **else**
8: Add e in Filter
9: Set count of e to 1
10: Increment $N[j]$ by 1
11: Increment c by 1
12: (c counts updates since last handover)
13: **if** $Filter.size = D$ or $c = E$ **then**
14: Push the reserved filters of j to the MPSC-stack of the respective owners
15: **while** There are unflushed filters reserved for j **do**
16: *process_pending_updates*()
17: $c \leftarrow 0$

4.4. Concurrent updates

This section provides a detailed discussion of the update procedure in the presence of concurrency, which consists of multiple algorithmic components arranged in a pipeline (see Fig. 2). An arbitrary thread j that processes an input-element e, owned by a thread $i = owner(e)$ (line 2, Algorithm 2) inserts e in the Delegation Filter of thread i that is reserved for thread j (lines 3-9).

If the element is not in the filter, it is added with a count of one; otherwise, its count is incremented. The counters $N[j]$ (total insertions by thread j) and c (insertions since thread j last handed over its filter) are both incremented (lines 10–11). The filters may be implemented using two arrays, one for elements and one for counts, where corresponding entries share the same index (similar to Content-Addressable Memory [36]). The length of the arrays is D. If D is kept small, the count of an element can be found efficiently by a simple linear search.

To improve scalability and prevent filter staleness (discussed further in the next subsection), elements are inserted into Delegation Filters until either (1) a filter reaches its capacity of D elements or (2) the thread has processed E elements since the last handover. When either condition is met (line 11), the thread hands over all filters to their respective owners (line 12). Handing over and flushing all filters periodically, especially in a skewed input distribution, enhances query accuracy since filter element counts are excluded from the query result (see Section 4.5). A filter is handed over to the owner by pushing it to a concurrent multiple-producer single-consumer (MPSC) stack re-

Algorithm 3 Processing pending updates on thread i.

```
1:  function process_pending_updates
2:      if Threads[i].MPSC-Stack empty then
3:          return
4:      if Try-lock of Threads[i] taken then
5:          return
6:      while Threads[i].MPSC-Stack is not empty do
7:          Filter ← Threads[i].MPSC-Stack.pop()
8:          for each (Element e, Weight w) ∈ Filter do
9:              Threads[i].UpdateQOSS(e,w)
10:         Empty Filter
11:         Filter.size ← 0
12:     Threads[i].mutex ← 0
```

Algorithm 4 Query operation on thread j.

```
1:  function QueryQPOPSS(ϕ ∈ [0,1])
2:      N ← sum(N_i), i ∈ {1..T}
3:      while there exists QOSS_i, i ∈ {1..T} not yet queried do
4:          if Try-lock of Threads[i] taken then
5:              Try another thread
6:          else
7:              Threads[i].QueryQOSS(ϕ,N)
8:              Release try-lock of Threads[i]
9:          process_pending_updates()
10:     Output frequent elements
```

served for each thread.[2] Thread i processes its own *pending updates* until all its filters are marked empty (lines 15-16) before it resets c to zero (line 17) and processes the next input.

Processing pending updates: Each thread periodically checks for ready filters to process in its MPSC stack, the absence of which causes immediate function termination (lines 2-3 in Algorithm 3). Line 2 exhibits behavior typical of fine-grained synchronization, where thread i may need to retry processing pending updates. However, this is not an issue, as threads only process pending updates when no other useful work is available. Specifically, when thread i checks if the MPSC stack is empty while thread j has just pushed an update, two outcomes are possible: (1) if the change has propagated, thread i detects the update and processes it immediately without delay; (2) if the change has not yet propagated, thread i does not detect the update and does not proceed, but will detect the update next time it processes its pending updates.

Inversely, ready filters are processed given the successful acquisition of a thread-specific try-lock mutex (lines 4-5), preventing possible data races due to concurrent updates and queries (see Section 4.5) operations that target the thread-local QOSS instance. Contention on the lock is low, as it is only accessed when the MPSC stack contains a ready filter or when a query is carried out, and can be implemented with a simple test-and-set lock. The elements of each filter are fed as weighted updates to the QOSS instance (lines 8-9), and the filter is marked as empty (lines 10-11). Finally, the lock is released (line 12), allowing a querying thread to read the QOSS data structure.

4.5. Concurrent frequent elements queries

Any of the T dispatched threads may answer a frequent elements query, while other threads may concurrently perform updates or queries. The implications of this are discussed in Section 5. A query aims to report the set of frequent elements from Definition 1. To calculate the threshold value $N\phi$, needed to efficiently select the frequent elements, the query procedure begins by estimating the stream length, N. This value is computed as the sum of the elements processed by each thread, $N[i]$ (line 2 in Algorithm 4). To collect the subset of frequent elements tracked by each thread-local QOSS algorithm, the querying thread tries to acquire the test-and-set lock associated with each thread (line 3). Once the lock of a thread has been acquired, a query is issued to the corresponding QOSS algorithm (line 7).

Recall that the try-lock acquisition is a non-blocking action; if a thread cannot acquire it immediately, it will simply retry later. Due to the aggregation of elements in Delegation Filters, contention on these accesses is low. Meanwhile, the thread can do other useful work, namely processing its pending updates if any (line 9). Furthermore, in the QOSS algorithmic implementation (section 4.3), a query is processed in $O(|F|)$ time, where $|F|$ is the number of frequent elements, further reducing the contention.

Query scalability enhancement: We identify a performance tradeoff in the design of Delegation Filters: During a query, buffered element occurrences in the Delegation Filters can be ignored to improve query speed and overall throughput. However, this introduces a slack between when an element is first observed and subsequently reported in a query. By bounding the maximum sum of element counts in a Delegation Filter by a constant E, we can guarantee a bounded handover delay, which can be kept low concerning the usual inaccuracy inherent to data synopses. At the same time, we aim to maximize the portion of elements in the QOSS data structures. We achieve this by introducing the aforementioned mechanisms concerning D and E for promptly handing over filters to owner threads at a fixed rate. Indeed, initial experiments suggested that exploring this tradeoff resulted in up to 1.73x higher throughput and 0.5x lower query latency (with 24 threads, $\phi = 10\epsilon = 0.0001$, E = 1000, D = 32, querying on average once per 10 updates using real IP-packet data), compared to the non-enhanced approach, while keeping the reporting delay below E element occurrences multiplied by the number of threads. The side effect on the approximate output introduced by the enhancement diminishes rapidly with the length of the execution, as we show in Section 5, where we define the prevailing consistency guarantees.

5. Analysis

Having described our method for estimating the frequent elements, we now focus on the space requirements and estimation guarantees under concurrent updates and queries. To aid us in this task, we define a set of symbols common to our analysis in Table 1.

We initiate the discussion using a meta-lemma containing useful lemmas and theorems from [33] that apply to the QOSS algorithm.

Lemma 1. QOSS preserves the following properties (implied from the respective lemmas and theorems in [33])

1. (From Lemma 3.3 in [33]) If the number of counters m is chosen such that $m = \frac{1}{\epsilon}$, then the minimum counter value of QOSS, denoted as $F_{\min}$, is less than or equal to $\lfloor N\epsilon \rfloor$, where N is the length of the stream.
2. (From Theorem 3.5 in [33]) Any element that occurs more than $F_{\min}$ times in S is guaranteed to be tracked by QOSS.
3. (From Lemma 3.4 in [33]) Elements tracked by QOSS are overestimated by at most $F_{\min}$. In other words: $f_N(e) \leq \hat{f}_N(e) \leq f_N(e) + \epsilon N$.
4. (From Lemma 4.3 in [33]) Any element that occurs in S more frequently than the maximum possible value of $F_{\min}$ is guaranteed to be reported, regardless of stream order.

The rest of this section adheres to the following structure: First, we show that maintaining accuracy requires fewer counters when the QOSS algorithm observes a stream of elements belonging to a subset of the original domain of possible elements. Second, we describe the counter requirements of QPOPSS, composed of multiple QOSS. Lastly, we focus on the consistency guarantees of the frequent elements in the face of

[2] Note that the number of dispatched threads T and the filter size D bound the memory allocated to the MPSC stack associated with a thread to be $O(TD)$.

Table 1
Descriptions of symbols used.

Symbol	Description
ϵ	User-specified approximation factor.
ϕ	User-specified frequent element threshold.
S	The stream of elements.
N	The stream length of S.
U	Domain of S.
e	An element of a stream.
$r(e)$	The rank of a stream element.
$f_N(e)$	The number of occurrences of e in S.
$\hat{f}_N(e)$	The estimated number of occurrences of e in S.
m	The number of counters in QOSS.
F	Set of elements and estimated occurrence in S tracked by QOSS.
$F_{\min}$	Least estimated occurrence of an element in S tracked by QOSS.
T	Number of dispatched threads.
$\mathbb{D}(e)$	The number of counts of e in a Delegation Filter.
E	Parameter controlling the number of elements in delegation filters.
D	Number of slots in a delegation filter.
ζ	The Euler–Riemann function.
a	Skew parameter for Zipf distribution.
N_S	Stream length at the start of a query.
N_E	Stream length at the end of a query.

concurrently overlapping queries and updates, and provide consistency-implied accuracy bounds.

5.1. Domain splitting and space requirements

We begin by analyzing the number of counters required by QOSS to accurately report the frequent elements defined in Section 2 when processing elements from a split-domain stream, i.e., a stream where elements not in a specific subset of the universe of possible elements are omitted.

To this end, we introduce $S_{1..x}$ as the x-prefix of an unbounded stream S, containing the first x elements. Each symbol in the sequence, called S_i for each $i \in \{1..x\}$, can be found in $U \cup \{\varnothing\}$, the universe of possible elements in union with the null symbol. We include $\varnothing$ to denote the absence of an element, used for highlighting element-wise differences between streams. Furthermore, we introduce a function to transform a stream to a *stream block* (analogous to a set block [8]), containing only stream elements from a specific set block B of a partition of U. The following are three function definitions for constructing a stream block, counting the number of deleted elements in a stream, and finding the length of a bounded stream.

$$\text{block}(S,B) = \begin{cases} S_1 \text{ if } S_1 \in B \text{ else } \varnothing & \text{if } x = 1 \\ (\varnothing, \text{block}(S_{2...x}, B)) & \text{if } S_1 \notin B \\ (S_1, \text{block}(S_{2...x}, B)) & \text{if } S_1 \in B \end{cases}$$

$$\#\text{del}(S) = \begin{cases} 1 \text{ if } S_1 = \varnothing \text{ else } 0 & \text{if } x = 1 \\ \#\text{del}(S_{2...x}) & \text{if } S_1 \neq \varnothing \\ 1 + \#\text{del}(S_{2...x}) & \text{if } S_1 = \varnothing \end{cases}$$

$$\text{len}(S) = \begin{cases} 0 \text{ if } S_1 = \varnothing \text{ else } 1 & \text{if } x = 1 \\ 1 + \text{len}(S_{2...x}) & \text{if } S_1 \neq \varnothing \\ \text{len}(S_{2...x}) & \text{if } S_1 = \varnothing \end{cases}$$

Using these definitions, a stream block can be constructed as in the following example: if $U = \{a,b,c,d\}$, $S_{1..5} := a,a,b,d,c$, and $B = \{a,c\}$, then $\mathcal{B}_{1..5} := block(S,B) = a,a,\varnothing,\varnothing,c$, $\#\text{del}(\mathcal{B}) = 2$, and $\text{len}(\mathcal{B}) = 3$.

From this point, to bound the number of counters needed by QPOPSS to produce the frequent elements from Definition 1, we observe the relationship between the stream's domain size and the minimum counter in QOSS.

Lemma 2. When QOSS observes the stream block $\mathcal{B} := \text{block}(S,B)$ and maintains $m = \frac{1}{T\epsilon}$ counters, the minimum counter is at most $\lfloor \frac{N}{\epsilon} \rfloor$, if $|B| = \lceil \frac{|U|}{T} \rceil$.

Proof. Let j and L be arbitrary positive integers. Consider a stream S with length $len(S) = L(m+1+j)$, containing $m+1+j$ distinct elements, each repeated L times. Let these elements belong to the set U, such that $|U| = m+1+j$. Given $|B| = \lceil \frac{|U|}{T} \rceil$, the stream $\mathcal{B} := block(S,B)$ has a length of $len(\mathcal{B}) = L\frac{m+1+j}{T}$. We utilize claim 1 in Lemma 1 to determine the minimum counter of QOSS while observing $\mathcal{B}$:

$$F_{\min} \leq \lfloor L \frac{m+1+j-\frac{(m+1+j)(T-1)}{T}}{m} \rfloor = \lfloor \epsilon L\big(m+1+j\big) \rfloor = \lfloor N\epsilon \rfloor \quad \square$$

This follows naturally from the linear relationship between consumed space and the accuracy of the Space-Saving algorithm, and has implications for the required number of counters of QOSS under domain splitting in the following.

Lemma 3. When QOSS observes the stream block $\mathcal{B} := \text{block}(S,B)$ and maintains $m = \frac{1}{T\epsilon}$ counters, it tracks every element occurring more than $\lfloor N\epsilon \rfloor$ times with an estimation error of at most $\lfloor N\epsilon \rfloor$ if $|B| = \lceil \frac{|U|}{T} \rceil$.

Proof. From Lemma 2, we establish that QOSS maintains a minimum counter value $F_{\min} \leq \lfloor N\epsilon \rfloor$. Leveraging claims 2 and 3 in Lemma 1, elements occurring more than $\lfloor N\epsilon \rfloor$ times are guaranteed to be tracked, with an overestimation error of at most $\lfloor N\epsilon \rfloor$. $\square$

Since QPOPSS consists of QOSS instances, their space requirements and accuracy guarantees are interlinked. QPOPSS dispatches T QOSS instances, one for each thread. According to Lemma 3, each requires $\frac{1}{T\epsilon}$ counters to track the frequent elements. Therefore, the total number of counters needed to track and report the ϵ-approximate frequent elements of S is $T\frac{1}{T\epsilon} = \frac{1}{\epsilon}$.

Corollary 1. QPOPSS requires $\frac{1}{\epsilon}$ counters to track every element in S occurring more than $\lfloor N\phi \rfloor$ times, with an estimation error at most $\lfloor N\epsilon \rfloor$.

We now investigate the number of required counters, assuming that the input data stream conforms to the Zipf distribution.

Theorem 1. QOSS with $m = \left(\frac{1}{\epsilon T}\right)^{\frac{1}{a}}$ counters, fed with stream $\mathcal{B} := \text{block}(S,B)$, tracks every element occurring more than $N\phi$ times and reports occurrences with an error of at most $\lfloor N\epsilon \rfloor$, provided S is constructed from a noiseless Zipf distribution with $a > 1$, regardless of stream permutation.

Proof. According to claim 4 in Lemma 1, QOSS reports frequent elements occurring more often than the maximum possible value of $F_{\min}$. For a Zipf-distributed input stream, the maximum value of $F_{\min}$ is less than or equal to the cumulative occurrences of the elements ranked between $m+1$ and $|U|$, divided equally over the number of counters: $F_{min}^{zipf} \leq \frac{N}{m}\frac{\sum_{i=m+1}^{|U|}\frac{1}{i^a}}{\sum_{i=1}^{|U|}\frac{1}{i^a}}$. Similarly, the number of occurrences of an element of a particular rank is described as $\frac{N}{r(e)^a}\frac{1}{\sum_{i=1}^{|U|}\frac{1}{i^a}}$. The following proof obligation describes the ranks of elements whose occurrences exceed the maximum value of F_{min}^{zipf}:

$$\frac{1}{r(e)^a} > \frac{1}{m}\sum_{i=m+1}^{|U|}\frac{1}{i^a} \tag{1}$$

The right-hand side of the inequality can be simplified:

$$\frac{1}{r(e)^a} > \frac{1}{m^a} \sum_{i=2}^{\frac{|U|}{m}} \frac{1}{i^a}$$

Since $\sum_{i=2}^{\frac{|U|}{m}} \frac{1}{i^a}$ has no closed-form expression, $\zeta(a) - 1$ is used as a substitute, imposing a greater constraint on m:

$$\frac{1}{r(e)^a} > \frac{1}{m^a}(\zeta(a) - 1)$$

Given that B is formed by randomly selecting $\frac{|U|}{T}$ elements from U with uniformity (approximately), the cumulative elements are denoted by $(\zeta(a) - 1)$ can be assumed to be evenly distributed among threads[3]:

$$\frac{1}{r(e)^a} > \frac{1}{m^a} \frac{(\zeta(a) - 1)}{T}$$

This simplifies to:

$$m > r(e) \left(\frac{\zeta(a) - 1}{T} \right)^{\frac{1}{a}} \tag{2}$$

Now, considering the inequality $\frac{N}{r(e)^a \zeta(a)} < N\epsilon$, satisfied for element ranks that occur more often than the threshold. The inequality can be solved for $r(e)$ to obtain the least element rank satisfying the above inequality:

$$r(e) \geq \left(\frac{1}{\epsilon \zeta(a)} \right)^{\frac{1}{a}}$$

We can now substitute $r(e)$ in inequality 2:

$$m > \left(\frac{\zeta(a) - 1}{T \epsilon \zeta(a)} \right)^{\frac{1}{a}} = \left(\frac{1}{T\epsilon} - \frac{1}{T\epsilon\zeta(a)} \right)^{\frac{1}{a}}$$

Since the residual $\frac{1}{T\epsilon\zeta(a)}$ is negligible for large $\zeta(a)$, and $\left(\frac{1}{T\epsilon} - \frac{1}{T\epsilon\zeta(a)}\right)^{\frac{1}{a}}$ remains sufficiently small as $\zeta(a)$ decreases, setting $m = \left(\frac{1}{T\epsilon}\right)^{\frac{1}{a}}$ guarantees that the proof obligation in inequality (1) is satisfied, i.e., elements with rank less than $\left(\frac{1}{\epsilon\zeta(a)}\right)^{\frac{1}{a}}$ exceed the maximum minimum counter value, and will therefore be reported by QOSS. □

Having determined the space requirements of QOSS when observing a stream with Zipfian distribution, we can discuss the space requirements of QPOPSS. QPOPSS dispatches T threads, each with its own QOSS algorithm instance requiring $\left(\frac{1}{T\epsilon}\right)^{\frac{1}{a}}$ counters according to Theorem 1.

Corollary 2. QPOPSS requires $T\left(\frac{1}{T\epsilon}\right)^{\frac{1}{a}}$ counters to track every element of a stream S occurring more than $\lfloor N\epsilon \rfloor$ times and reports the number of occurrences of elements with an error of at most $\lfloor N\epsilon \rfloor$, given that S is constructed from a noiseless Zipf distribution with $a > 1$, regardless of stream permutation.

Ultimately, Corollaries 1 and 2 describe the required number of counters needed by QPOPSS to track the ϵ-approximate ϕ-frequent elements across various distributions. Additionally, the Delegation Filters described in 4.4 contain counters equal to the number of threads squared times the number of counters kept by each filter ($T^2 D$). Having established space requirements, we now explore the query consistency guarantees of QPOPSS.

[3] This assumption is motivated by the fact that the $|U| - (m+1)$ least frequent elements contribute little to the total count and greatly outnumber m.

5.2. Query consistency guarantees

In this section, we provide an invariant for the approximation guarantees of the frequent elements and their occurrence reported by QPOPSS as they relate to challenge [C3]. We use a similar reasoning and method as Rinberg and Keidar [37], who defined bounds for the estimated count of an element on a concurrent Count-Min Sketch, given that its counters are monotonically increasing, as is the case with counters in QOSS. Before we delve into the consistency analysis, we first discuss the implications of the algorithm design.

Query Scalability Enhancement: As implied by the *query scalability enhancement* of QPOPSS described in Section 4.4, the parameter E represents the maximum number of elements present in a delegation filter at any point in time. Since there are T delegation filters that can contain an element, the maximum number of element occurrences that can be missing from a reported element count is $T \cdot E$, which is put more concisely as the following lemma.

Lemma 4. When S consists of elements drawn from an arbitrary distribution, $\mathbb{D}(e)$, the number of counts of e inside delegation filters, is less than $T \cdot E$.

Note that the lemma is a rather substantial overestimation. In common executions, Delegation Filters contain various elements. Being fully occupied by a single element is unlikely in each Delegation Filter.

Suppose the input distribution is noiseless Zipf with skew parameter $a > 1$. In that case, we can give a tighter bound on the number of counts of a particular element e inside Delegation Filters.

Lemma 5. When S consists of elements drawn from a noiseless Zipf distribution with infinite domain and skew parameter $a > 1$, $\mathbb{D}(e)$ is at most $\frac{T \cdot E}{\zeta(a) r(e)^a}$.

Query consistency: To capture the notion of queries that are concurrent with updates, we introduce N_S and N_E, which represent the stream lengths at the start and end of a query. These values are ordered such that $N_S \leq N \leq N_E$. We update claim 3 in Lemma 1 to capture potential concurrent update operations during a query as follows.

Lemma 6. Given a stream S of N elements, QPOPSS estimates the occurrence of an element $e \in U$ such that $f_{N_S}(e) - \mathbb{D}(e) \leq \hat{f}_N(e) \leq f_{N_E}(e) + \epsilon N_E$.

Proof. We have that $f_{N_S}(e) \leq f_{N_E}(e)$. There are at most $\mathbb{D}(e)$ counts of element e that have not yet been inserted into the QOSS instance of the owner of e, therefore, $f_{N_S}(e) - \mathbb{D}(e) \leq \hat{f}_N(e)$ is the minimum value a counter can assume. The maximum overestimation of QOSS is ϵN, which is maximized at the end of a query when N_E elements have been processed. Therefore, the estimated count of an element is at most $\hat{f}_N(e) \leq f_{N_E}(e) + \epsilon N_E$. □

We provide a consistency guarantee for the set of elements reported by QPOPSS, aligning with Definition 1 for queries spanning more than 0 updates.

Theorem 2. Given a stream S of N elements, QPOPSS is guaranteed to report the set F containing all elements $e \in U$ with $f_{N_S}(e) > \phi N_S + \mathbb{D}(e)$, and no elements $e \in U$ with $f_{N_E}(e) < (\phi - \epsilon)N_S - \epsilon(N_E - N_S)$, where $0 < \epsilon \leq \phi < 1$.

Proof. The QPOPSS algorithm reports all elements with an estimated count

$$\hat{f}(e)_{N_S} > \phi N_S \tag{3}$$

This is true since N_S is calculated at the start of a query; all elements with $\hat{f}_N(e) > \phi N_S$ are reported, and QOSS counters increase monotonically.

As shown in Lemma 6, $\hat{f}_N(e)$ is at least (a) $f_{N_S}(e) - \mathbb{D}(e)$, and at most (b) $f_{N_E}(e) + N_E\epsilon$. Substituting $\hat{f}_N(e)$ for (a) in inequality (3) yields:

$$f_{N_S}(e) > \phi N_S + \mathbb{D}(e) \tag{4}$$

Substituting $\hat{f}_N(e)$ for (b) in inequality (3) yields:

$$f_{N_E}(e) > \phi N_S - \epsilon N_E$$

Expression (a) and the negation of (b) together give that all elements $f_{N_S}(e) > \phi N_S$ and no elements $f_{N_E}(e) < \phi N_S - \epsilon N_E$ are reported by QPOPSS. □

According to Theorem 2 QPOPSS will report all elements occurring more than ϕN_S times after processing N_S elements, given that delegation filters are empty. With fixed-size delegation filters, QPOPSS tends to report all elements more frequently than ϕN_S as the stream length grows in relation to the delegation filter size. This notion is formalized in Theorem 3 below.

Theorem 3. As the length of the stream N tends to infinity, QPOPSS reports *all* ϵ-approximate frequent elements.

Proof. We start by normalizing inequality (4):

$$\frac{f_{N_S}(e)}{N_S} > \phi + \frac{\mathbb{D}(e)}{N_S}$$

The element e occurs a fraction $P(e) = \frac{f_{N_S}(e)}{N_S}$ of the time in the stream. We then have that:

$$\lim_{N_S \to \infty} P(e) > \phi + \frac{\mathbb{D}(e)}{N_S} \to P(e) > \phi$$

Thus, once enough elements have been processed, all elements with $P(e) > \phi$ are reported. □

Theorem 4. As N tends to infinity, QPOPSS achieves perfect recall for streams constructed by drawing elements from a Zipf distribution with $a > 1$.

Proof. Rewriting inequality (4) in its Zipfian form:

$$\frac{1}{\zeta(a) r(e)^a} > \phi + \frac{\mathbb{D}(e)}{\zeta(a) r(e)^a N_S}$$

Solving for r(e) gives:

$$\left(\frac{1}{\zeta(a)\phi} - \frac{\mathbb{D}(e)}{\zeta(a)\phi N_S}\right)^{\frac{1}{a}} > r(e)$$

Simplifying yields:

$$\left(\frac{1}{\zeta(a)\phi}\right)^{\frac{1}{a}} \left(1 - \frac{\mathbb{D}(e)}{N_S}\right)^{\frac{1}{a}} > r(e)$$

We then have that:

$$\lim_{N_S \to \infty} \left(\frac{1}{\zeta(a)\phi}\right)^{\frac{1}{a}} \left(1 - \frac{\mathbb{D}(e)}{N_S}\right)^{\frac{1}{a}} > r(e) \to \left(\frac{1}{\zeta(a)\phi}\right)^{\frac{1}{a}} > r(e)$$

This means that all elements with a rank lower than $\left(\frac{1}{\zeta(a)\phi}\right)^{\frac{1}{a}}$ are guaranteed to be reported given that enough elements have been processed. □

Table 2
The number of frequent elements for different threshold values of ϕ in the CAIDA and selected Zipf data sets.

Data set \ ϕ	10^{-3}	10^{-4}	10^{-5}
CAIDA	44	1555	10463
Zipf a = 1.25	74	467	2952
Zipf a = 2	24	77	246
Zipf a = 3	9	20	43

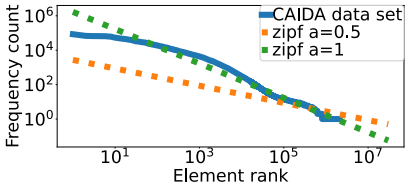


Fig. 3. Rank and count of each unique element in the CAIDA data set. Zipf distributions with skew 0.5 and 1 are plotted as a guide. Note the logarithmic scale on x- and y-axes.

To summarize, the analysis underscores the memory and space benefits of operating on a subset of the original domain (Corollaries 1 and 2). It also highlights the impact of excluding elements from Delegation Filters on query consistency and accuracy, offering bounds for the latter (Lemmas 4 and 5). Moreover, it addresses the consistency guarantees of frequent elements amid concurrent queries and updates (Theorems 2, 3, and 4).

6. Evaluation

Considering the analytical properties of QPOPSS, we proceed with in-depth empirical evaluation. We begin by describing the experimental setup in detail. Then, we investigate the performance of QOSS compared to Space-Saving. We also compare QPOPSS and the representative works [29,44] on throughput and scalability, accuracy and memory requirements, as well as query latency.

6.1. Experimental setup

Computing platforms: The experiments were carried out on two platforms: *Platform A*, a dual-socket NUMA server with 2 Intel Xeon X5675 processors, each with 12 cores and 2-way hyperthreading. Each core runs at 3.07 GHz, with cache sizes L1 32 KB, L2 256 KB, and 12 MB shared L3 cache. It runs Debian 10.9 and gcc v.8.3. *Platform B* is a dual-socket Intel(R) Xeon(R) CPU E5-2695 v4 NUMA server with 36 cores (2.1 GHz) and 2-way hyper-threading. Each core runs at 2.1 GHz and uses 32 KB L1 data cache and 256 KB L2 cache, and 45 MB shared L3 cache. It runs openSUSE Tumbleweed and gcc v.14.2.1.

We run experiments on both platforms. In particular, we use Platform B when the higher parallelism it enables is needed for adequate evaluation.

Data sets: Both synthetic and real data were used in the evaluation of QPOPSS. Unless otherwise stated, the synthetic data sets contain $100M$ elements, sampled from a universe of $|U| = 100M$ elements according to the probability mass function of the Zipf distribution such that $f_N(e) = \frac{N}{H_{|U|,a} r(e)^a}$ [46], where $H_{|U|,a} = \sum_{i=1}^{|U|} \frac{1}{i^a}$. In total, 11 synthetic data sets were created from Zipf distributions with skew parameter a ranging from 0.5 to 3 in increments of 0.25. Hereinafter, we use the term skew to mean a in $f_N(e) = \frac{N}{H_{|U|,a} r(e)^a}$.

A real-world data set was extracted from the CAIDA Anonymized Internet Traces 2019 data set [1] by selecting an arbitrary 60-minute window of IP packet traffic in an arbitrary direction of a backbone interface. Each packet contains a 5-tuple (flow) of source and destination IP addresses, source and destination ports, and transport layer protocol

used. The set contains roughly $21M$ packets belonging to around $2.1M$ unique flows. As shown in Fig. 3, the distribution of the flows is similar to that of synthetic data generated with a skew parameter of 1. The number of frequent elements in the data set is detailed in Table 2 for different values of ϕ. For this parameter, we use values similar to established works in the field [27,14,33], to facilitate comparison with other approaches. As for the synthetic data, the expression $\left(\frac{1}{\zeta(a)\phi}\right)^{\frac{1}{a}}$ describes the least element rank for a certain threshold value ϕ and Zipf distribution skew parameter a.

Metrics: Evaluation metrics include query and update *throughput* (millions of operations per second), *accuracy*, *memory consumption* (megabytes reserved), and the *latency* (the time between the start and end of a query in microseconds). More specifically, accuracy is measured as *average relative error (ARE)* (ratio of estimated element occurrences to actual occurrences), *precision* (ratio of actual positive frequent elements to the number of reported frequent elements), and *recall* (ratio of actual positive frequent elements to the number of actual frequent elements).

Measurement Methodology: Our experiments measured throughput as the number of operations (updates and queries) per time unit. Throughput experiments were executed for 10 seconds while processing a stream of 100 million elements repeatedly for the duration. The number of counters of each baseline was set according to the respective theoretical bound. The query latency was calculated by measuring processor clock cycles between a query's start and end. This was done for multiple queries whose mean value was calculated. The accuracy metrics were measured by processing a stream and issuing a single query at the end. The memory consumption was calculated by selecting an accuracy level and computing the memory consumption for the different baselines according to their theoretical space/accuracy bounds.

Baselines: We compare our open-source QPOPSS [26]-along with the code for generating synthetic data and links to the real data used in this evaluation-to the following baselines. These baselines are also discussed in Section 3, with additional details provided here for clarity.

1. A single-threaded QOSS
2. QPOPSS using Space-Saving as the inner algorithm
3. PRIF [44]
4. Topkapi [29]

To examine the speedup between a single-threaded QOSS and the multithreaded QPOPSS, (1) was selected as a baseline, while (2) facilitates studying the impact of QOSS on query response time. (3) and (4) were chosen since they are representative multithreaded approaches to the frequent elements problem. The *owner* function described in Section 4.2 uses the modulo operator: $owner(e) = e \mod T$.

PRIF [44] entails a reserved merging thread that periodically merges updates from thread-local algorithm instances. As an algorithmic component, the authors present OWFrequent, an optimized version of Frequent [34] that supports weighted updates. Due to the merging thread, extra latencies are introduced. The authors propose an update coefficient β that controls the rate at which the merging thread receives updates from the thread-local OWFrequent algorithm instances. The authors give a rigorous analysis and evaluation of the approach. The evaluation shows that PRIF is somewhat precise in reporting the frequent elements and that there is a good speedup compared to a single-threaded version. Due to the absence of open-source implementations of PRIF, one was created for evaluation purposes [26]. As in the authors' implementation, a shared-bounded buffer was implemented with semaphores [39] to handle the communication between the sub-threads and the merging thread. OWFrequent was implemented using the frequent elements sketch algorithm package from [13] as a base. The implementation of QOSS was also based on said algorithm package.

The *Topkapi Sketch* [29] combines the concepts of the Frequent Algorithm with the Count-Min Sketch by keeping a Frequent counter in each cell of a Count-Min Sketch matrix. A Topkapi Sketch with $\log(2\frac{N}{\delta})$ rows and $\frac{1}{\epsilon}$ counters (see analysis section in [15]), solves the ϕ-approximate frequent elements problem outlined in Definition 1. The authors present a multithreaded approach wherein multiple streams are processed concurrently. At the end of processing, the summaries are merged into a final result, aiming to deliver the frequent elements of the combined stream, adhering to the outlined theoretical limits. The code is open source and was adopted for the relative study in this paper.

In summary, a motivating factor in the selection of these algorithms among others is the support of concurrent queries and updates and compatibility in computational model.

Baseline adaptations: To ensure a fair comparison, certain adaptations were made. Since PRIF supports concurrent queries and updates, we implemented it *as is*. Regarding Topkapi, as it lacks design elements that enable concurrent queries and updates, when it comes to throughput, the experiments allowed it to perform *thread-unsafe* queries without synchronization, since its original design did not target concurrent updates [29]. This adaptation clearly favors the throughput performance of Topkapi, which would otherwise require a synchronization mechanism. However, this allows us to establish a best-case estimate for Topkapi's parallel performance and compare it to our approach. This also does not affect the accuracy evaluation of Topkapi since that is measured via a query at the end of the stream, irrespective of the synchronization mechanisms used.

Experiment Parameters: Across experiments, the following parameters are varied: *skewness* of the input distribution, *frequent element threshold parameter (ϕ, controlling query size)*, *number of dispatched threads*, *query rate*, and *stream length*. This simulates the point during execution when a query occurs. To reduce the size of the parameter space, ϵ (present in all baselines) is set to $\frac{1}{m} = \epsilon = 0.1\phi$. This is in alignment with previous studies [43,2,14] to ensure consistency and comparability with existing research. The PRIF-specific β parameter controlling the delay at which elements are sent to the merging thread is set to $\beta = 0.9\epsilon$, as in the authors' evaluation [44]. The Topkapi-specific *rows* (also present in the Count-Min Sketch [15]) parameter controlling the probability of failure to estimate an element count within a certain error is affixed to 4, which was also the case in the authors' evaluation [29]. The analysis in Section 5 implies that QPOPSS finds all frequent elements with $\frac{1}{\epsilon}$ counters for both the real-world data sets and the Zipf distribution data sets with $a \leq 1$. However, for Zipf data sets with $a > 1$, $\frac{1}{\epsilon T}^{\frac{1}{a}}$ counters suffice.

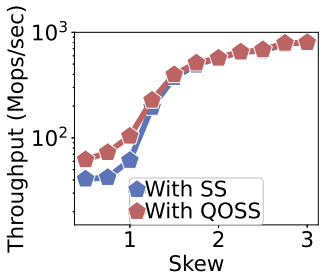
6.2. Query Optimized Space-Saving

We begin the evaluation by examining the impact of the inner algorithm employed by QPOPSS. We study the differences in latency and throughput between QOSS and Space-Saving since both algorithms have identical accuracy and memory consumption.

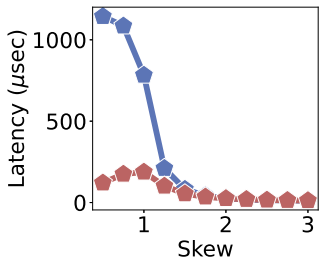
Fig. 4 contains the experimental results, where a query is carried out every 10000 updates, and ϕ is set to $10^{-4} = 10\epsilon$ to limit the parameter space.

Figs. 4a and 4b show the latency and throughput over varied skew levels. QOSS yields a higher throughput than the baseline for all skew values, and the latency is significantly lower with QOSS (up to 5x lower) for low skew values of 0.5-1.25. The improvement is less noticeable at higher skew levels in both figures. QOSS performs significantly better than Space-Saving when the skew level $a < 1$. This is due to QOSS's more efficient, low-latency query procedure and the fact that both algorithms use $m = \frac{1}{\epsilon}$ counters instead of $\left(\frac{1}{\epsilon}\right)^{\frac{1}{a}}$ for higher skew levels. Additionally, lower skew levels contain more frequent elements, as shown in Table 2.

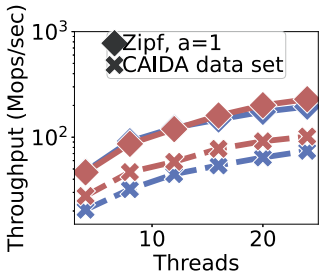
In Figs. 4c and 4d, the scalability of the approaches is evaluated as the number of dispatched threads varies. The experiments use two data sets: the synthetic Zipf data with skew parameter a = 1 and the CAIDA IP-packet trace. QOSS achieves up to 1.3x higher throughput for the CAIDA data set and up to 4x lower latency for both data sets. In particular,



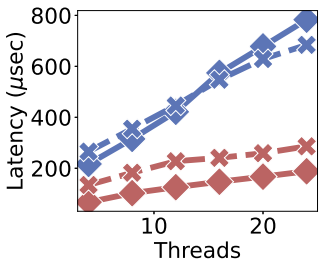
(a) Throughput as skew varies along the x-axis, T=24 threads. Note the logarithmic y-axis



(b) Latency as skew varies along the x-axis, T=24 threads

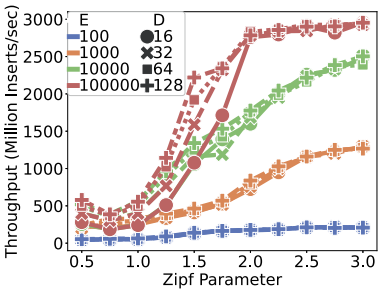


(c) Throughput as threads vary along the x-axis while processing Zipf and CAIDA data sets. Note the logarithmic y-axis.

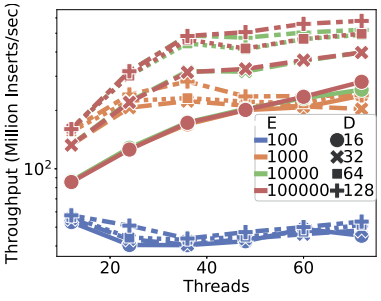


(d) Latency as threads vary along the x-axis while processing Zipf and CAIDA data sets.

Fig. 4. Throughput and query latency when QPOPSS employs QOSS or Space-Saving as the inner algorithm. Platform A, queries make up 0.01% of the operations and $\phi = 10^{-4}$.



(a) Platform B, Zipf skew level varies along the x-axis, $T = 72$ threads.



(b) CAIDA data set on Platform B, threads vary along the x-axis. Note the logarithmic y-axis.

Fig. 5. Throughput of QPOPSS in million operations per second. Values of E and D are varied. The query rate is 0.02%.

Fig. 4d shows that the latency of QOSS scales significantly better with the increasing number of threads, compared to Space-Saving.

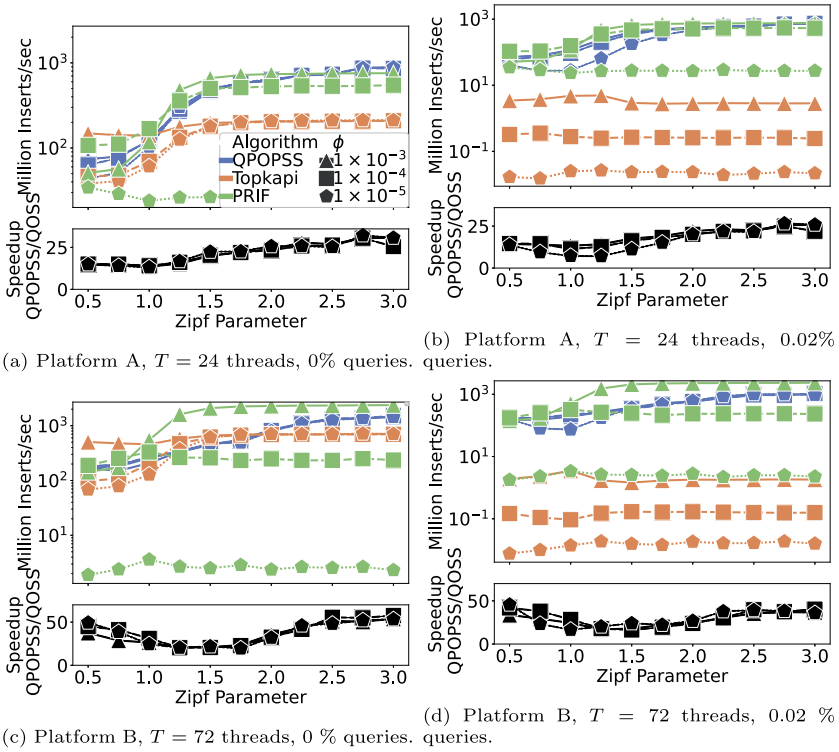
As the number of threads increases, the performance gap grows in favor of QOSS. This improvement is because QOSS only checks a subset of counters to identify frequent elements, while Space-Saving needs to inspect all of its counters.

6.3. Throughput and scalability of QPOPSS

We evaluate the throughput and scalability of QPOPSS by studying the impact of parameters E and D across varying skewness and thread counts. Recall that E is the maximum number of elements a thread processes before pushing filters and D defines the maximum number of unique elements per filter.

Fig. 5 presents a sensitivity analysis on Platform B, which provides a higher degree of multithreading. Specifically, Fig. 5a shows QPOPSS throughput for 16 combinations of E and D across varying skew levels. For Zipf parameters greater than 1.5, the results exhibit a clear separation between lines, indicating a correlation between throughput and E. This matches expectations: higher E allows more thread-local work and reduces synchronization overhead. When the Zipf parameter is below 1.5, E has less effect on throughput, as shown by the overlapping lines. This is expected: at low skew, filters usually fill with D elements before E items are processed. Consequently, the push frequency is determined by D rather than E, reducing the impact of E on throughput. Regarding Fig. 5b, a low value of E consistently results in reduced throughput across all numbers of dispatched threads. In contrast, higher values of E and D generally lead to increased throughput, with the best performance observed when both E and D are maximized. However, higher values of E can also cause increased buffering and delays in reporting element counts. This highlights the importance of carefully selecting E and D, as optimizing for throughput may negatively impact accuracy, as stated in Theorem 2. Interestingly, when $E > 1000$, throughput scales linearly and only flattens after 36 cores, likely due to two-way hyperthreading.

To understand how QPOPSS compares to Topkapi [29] and PRIF [44], we now focus on throughput and scalability when varying ϕ, the number of threads and the number of concurrent queries. We also compare



(a) Platform A, $T = 24$ threads, 0% queries.

(b) Platform A, $T = 24$ threads, 0.02% queries.

(c) Platform B, $T = 72$ threads, 0 % queries.

(d) Platform B, $T = 72$ threads, 0.02 % queries.

Fig. 6. Throughput in million operations per second and multicore speedup of QPOPSS for different skew parameters of the synthetic Zipf data sets. The skew level varies along the x-axis. Note the logarithmic y-axes.

the internal throughput scalability of QPOPSS (i.e., speedup) to a single-threaded QOSS algorithm, across Platforms A and B.

Fig. 6 shows the update and query throughput relative to the skew of the input data. The three different values of ϕ represent queries of varying sizes based on the number of frequent elements (see Table 2). In each execution, 24 threads are dispatched on Platform A, and 72 threads on Platform B. QPOPSS shows a higher throughput than Topkapi in all cases except for when no queries are carried out, and the skew value is between 0.5 and 1 (Fig. 6a). As the query rate increases to 0.02 in Fig. 6a, QPOPSS maintains a high throughput across all skew levels and values of ϕ, several times higher than Topkapi, which was shown to be highly scaleable in [29]. As expected, for query rates above 0, the computationally heavy merge operations carried out by Topkapi yield a diminished overall throughput. On the contrary, our algorithms continue to maintain a high throughput even with concurrent queries.

Due to its query-prioritized design, PRIF copes well with an increased query rate. However, the update throughput of PRIF seems to be strongly dependent on the threshold parameter ϕ, as setting $\phi = 10^{-5}$ yields very low throughput across all skew levels and for all query rates. Overall, it is observed that QPOPSS is the balanced choice, performing well in most circumstances and combinations of parameter variations. This trend can be observed for both platforms.

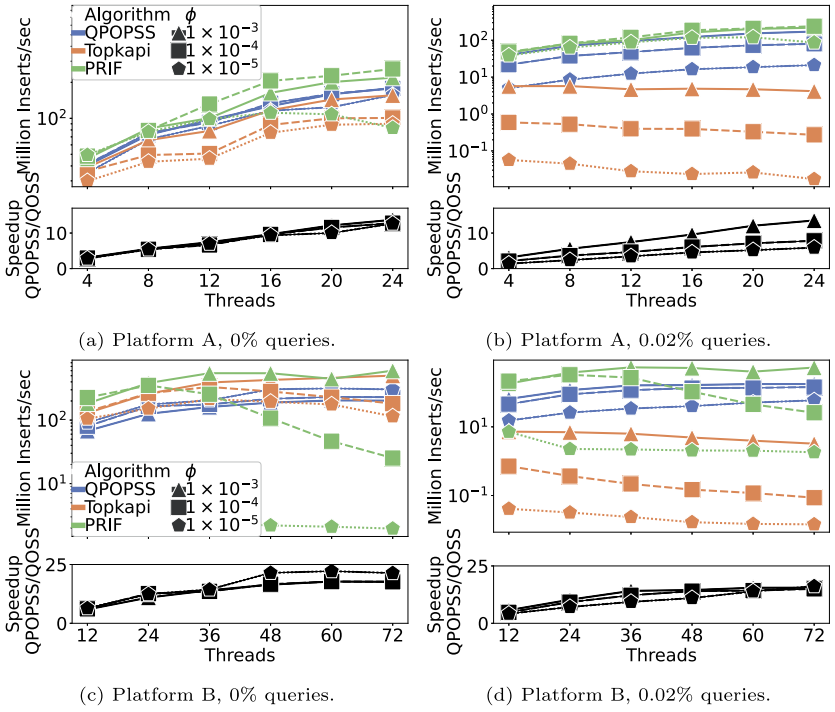
The plots containing black lines in Fig. 6 show the speedup relative to a single-threaded QOSS. The speedup of QPOPSS with 24 and 72 dispatched threads compared to a single-threaded QOSS is around 10-30x for Platform A and around 20-50 for Platform B across all combinations of ϕ, query rate, and skew. Interestingly, due to the efficiency of the delegation filters, QPOPSS achieves a speedup greater than the number of dispatched threads in higher skew levels. As worker-threads in QPOPSS swiftly insert elements in filters (an operation consisting of linear searching through a small fixed-size array and incrementing a counter), the single-threaded execution of QOSS must update a more complex tree data structure, potentially performing multiple time-consuming swap operations to ensure maintained heap properties.

The throughput and multicore scalability results for the CAIDA data set are shown in Fig. 7. When no queries are carried out (Figs. 7a and 7c), the throughput of Topkapi and QPOPSS is similar. However, as the query rate increases (Figs. 7b and 7d), Topkapi's performance rapidly decreases, highlighting the inefficiencies of the query process. Comparing Topkapi and QPOPSS, there is no major difference between platforms, aside from the generally lower throughput on Platform A than B, due to fewer threads. As for PRIF, the throughput does not vary with the query rate but instead depends on the threshold parameter ϕ. The different algorithms' throughput clearly correlates with the frequent elements per value of ϕ in Table 2, with numerous frequent elements corresponding to lower throughput and vice-versa. When $\phi = 10^{-3}$, PRIF outperforms QPOPSS. Still, in all other cases, QPOPSS can be observed to be the more balanced approach, maintaining high throughput when responding to large and small queries. QPOPSS has a positive trend, wherein throughput increases with the number of dispatched threads. This is not the case for PRIF, which, especially for low values of support parameter ϕ, seems to have declining throughput as more threads are added. This is likely due to PRIF's single merging thread becoming a bottleneck.

In the case of real-world data input as in Fig. 7, the throughput speedup of QPOPSS scales linearly with the number of threads compared to single-threaded Space-Saving, independently of ϕ and the query rate. Interestingly, the throughput of the different algorithms in Figs. 7a and 7c correlates very well with the frequent elements for each data set and value of ϕ in Table 2, with numerous frequent elements corresponding to lower throughput and vice-versa.

6.4. Memory consumption and query accuracy

We now compare each approach's memory requirements and the ability to report the frequent elements of a stream correctly. Correctness is measured by the metrics recall, precision, and ARE, previously mentioned in 6.1. Due to the space-accuracy trade-off associated with the ϵ-approximate ϕ-frequent problem, we set the memory consumption



(a) Platform A, 0% queries. (b) Platform A, 0.02% queries.

(c) Platform B, 0% queries. (d) Platform B, 0.02% queries.

Fig. 7. Throughput in million operations per second and multicore speedup of QPOPSS using the CAIDA backbone router data set. The number of threads varies along the x-axis. Note the logarithmic y-axes.

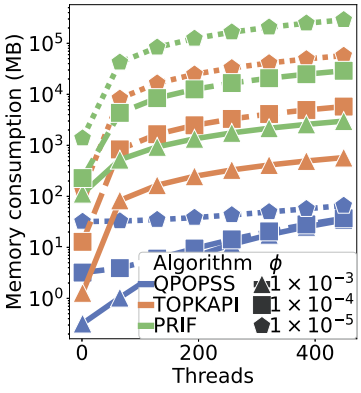


Fig. 8. Memory consumed by each approach in megabytes. The number of threads varies along the x-axis. Note the logarithmic y-axis.

of each approach to be equal to that of the respective analysis [29,44] (for QPOPSS, see Corollaries 1 and 2). Each counter equals 32 bytes.

Fig. 8 shows the megabytes consumed by each approach as the number of dispatched threads increases. Three values of ϕ are plotted for each baseline.

QPOPSS consumes the least space for each threshold parameter value of ϕ and scales up to 450 threads with at most 65.8 MB consumed in the case of $\phi = 10^{-5}$, compared to the 288.1 GB required by PRIF and 57 GB required by Topkapi. The memory consumption of QPOPSS also scales very well with different values of ϕ, as seen in Fig. 8. For example, at 450 threads, $\phi = 10^{-3}$ requires 34 MB, $\phi = 10^{-4}$ requires 37 MB, and $\phi = 10^{-5}$ requires 65 MB, which is very low and shows high scalability.

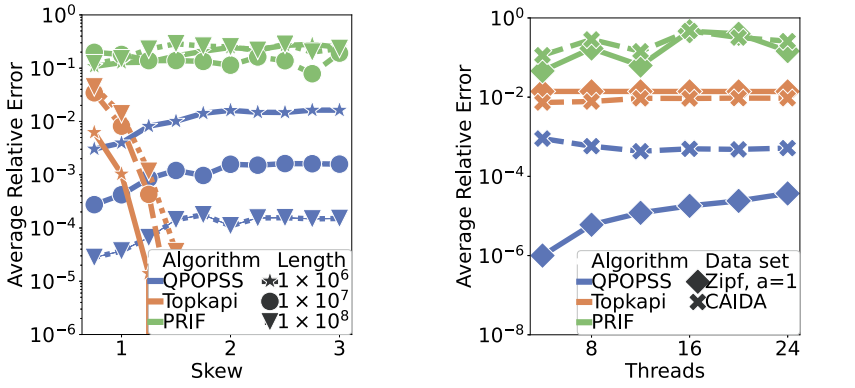
The PRIF memory requirements are $2\frac{T+1}{\epsilon-\beta}$, where $\beta < \epsilon$, and T is the number of dispatched threads (compared to QPOPSS, the latter uses $\frac{1}{\epsilon}$ counters, with an additional $T^2 D$ counters Delegation Filters, where D is the maximum number of unique elements in a filter).

Given the space-accuracy trade-off typical of synopsis algorithms, QPOPSS is also highly accurate. Using the above-described amount of memory bytes, we now compare the accuracy of each approach. To compare fairly between baselines, the experiments entail processing a stream of elements followed by a single query, which is compared to the ground truth. The following results, therefore, show the accuracy of each approach without considering the effects of concurrency. Fig. 9 contains the ARE, which is the arithmetic mean of the error in each element count divided by the actual count. ARE was measured across different datasets, stream lengths, and thread counts. In Fig. 9a, the x-axis shows the level of input data skew. The stream lengths are simulated by querying after a certain number of elements have been observed. As the length of the stream increases, QPOPSS displays decreasing ARE over all skew levels. This behavior is predicted by Theorem 4, i.e., as the stream length tends to infinity, the ARE tends to 0. PRIF's ARE is relatively high compared to the other baselines and seems not to correlate with either skew level or stream length. For Topkapi, however, the ARE seems to depend less on stream length and more on the skew level, as the accuracy improves in the higher levels. For skew values above 2, Topkapi often achieved zero ARE, meaning all estimated counts were exact. Fig. 9b contains the ARE for two input data sets while varying the number of dispatched threads. The data sets were Zipf with skew level 1 and for the real CAIDA data set. QPOPSS is the approach with the least ARE, both in the case of real and synthetic data. However, for the synthetic data, there seems to be an upward trend as the number of dispatched threads increases, while for the real data, the ARE seems to stabilize from 16 to 24 threads. Topkapi and PRIF have a high ARE, which remains somewhat stable as the number of threads increases.

The results presented in Table 3 describe how the baselines compare on precision and recall when the support ϕ and skew level of the synthetic data sets are varied. The results show that QPOPSS maintains perfect precision and recall in all cases. All approaches show high precision and recall; however, QPOPSS is alone in achieving perfect precision and recall for all parameter combinations. Table 3 shows that Topkapi achieves sub-optimal precision in skew levels between 0.75 and 1.25. This can be attributed to Topkapi being based on the probabilistic Count-Min Sketch. In Table 3, both PRIF and Topkapi have varied outcomes, as PRIF's recall drops to 0.87 in one case.

Table 3
Accuracy in terms of precision and recall for different threshold values of ϕ and Zipf skew.

Method		QPOPSS			PRIF			Topkapi		
Skew	Metric \ ϕ	10^{-3}	10^{-4}	10^{-5}	10^{-3}	10^{-4}	10^{-5}	10^{-3}	10^{-4}	10^{-5}
0.5	Precision	1	1	1	1	1	1	1	1	1
	Recall	1	1	1	1	1	1	1	1	1
0.75	Precision	1	1	1	1	1	1	1	1	0.793
	Recall	1	1	1	0.751	0.972	0.971	1	1	0.681
1	Precision	1	1	1	1	1	1	0.982	0.991	0.814
	Recall	1	1	1	0.965	0.973	0.941	0.982	0.991	0.545
1.25	Precision	1	1	1	1	1	1	0.997	0.997	0.827
	Recall	1	1	1	0.973	0.977	0.984	1	0.997	0.827
1.5	Precision	1	1	1	1	1	1	1	1	1
	Recall	1	1	1	0.980	0.982	0.913	1	1	1
1.75	Precision	1	1	1	1	1	1	1	1	1
	Recall	1	1	1	0.971	0.981	0.983	1	1	0.705
2	Precision	1	1	1	1	1	1	1	1	1
	Recall	1	1	1	1	0.994	0.991	1	1	0.520
2.25	Precision	1	1	1	1	1	1	1	1	1
	Recall	1	1	1	0.944	1	1	1	1	0.432
2.5	Precision	1	1	1	1	1	1	1	1	1
	Recall	1	1	1	0.928	0.986	1	1	1	0.397
2.75	Precision	1	1	1	1	1	1	1	1	1
	Recall	1	1	1	1	0.981	1	1	1	0.366
3	Precision	1	1	1	1	1	1	1	1	1
	Recall	1	1	1	1	1	1	1	1	0.348



(a) Different stream lengths as Zipf skew varies along the x-axis, $T = 24$ Threads.

(b) Different data sets as the number of threads vary along the x-axis.

Fig. 9. Average relative error. The Zipf skew level varies along the x-axis. $\phi = 10^{-4}$. Note the logarithmic y-axes.

6.5. Query latency

In this section, we evaluate the latency of QPOPSS by analyzing thread sensitivity in relation to parameters E and D on Platform B, which supports higher multithreading. Additionally, we compare QPOPSS, PRIF, and Topkapi on Platform A to assess QPOPSS's performance relative to other state-of-the-art methods.

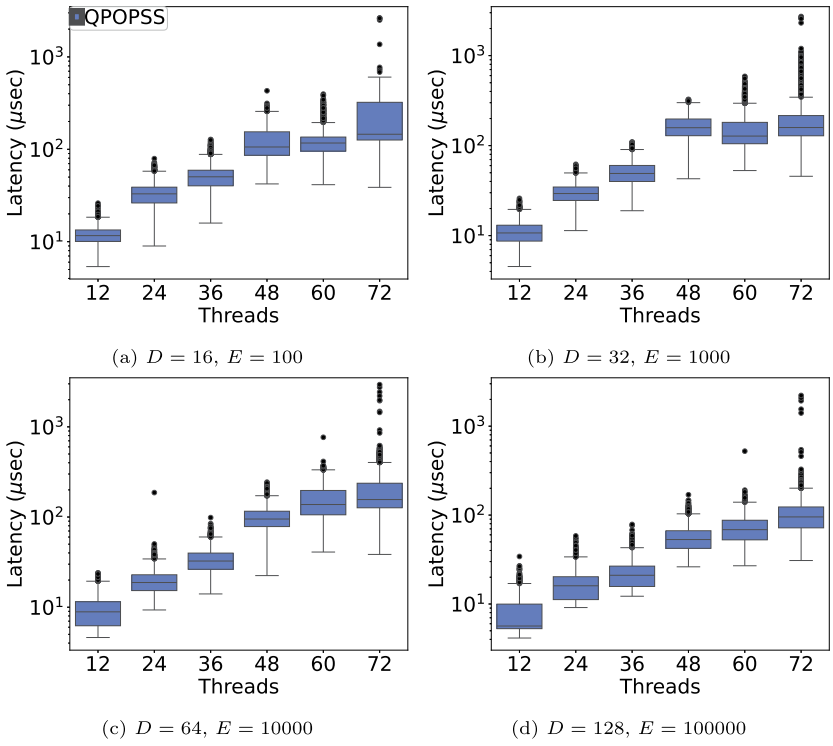
Starting with the thread sensitivity experiment, we conducted a series of experiments to evaluate how D, E, and T affect query latency, as shown in Fig. 10. Each box plot represents the response delays for 4,000 queries during runtime. Notably, latency increases sharply when T rises from 36 to 48 threads, likely due to 2-way hyperthreading.

Larger values of D and E lead to more local computation, likely explaining the tighter latency distributions at higher T in Figs. 10b, 10c, and 10d. For $T = 72$, the mean latency and distribution tails are in the range of hundreds of microseconds, with only some outliers reaching up to 3 ms.

We also conducted experiments to compare the different approaches in terms of query latency. Here, we present the mean latency out of 4000 queries carried out during runtime. In each run of a query latency experiment, the query and update workload is evenly distributed across all threads, with 0.01% of the operations carried out being queries and the other 99.9% being update operations. For the experiments in Fig. 11, the threshold parameter was set to $\phi = 10^{-4} = 10\epsilon$. A synthetic Zipf data set with shape parameter $a = 1$ was used as input to the baselines.

As seen in Fig. 11a, PRIF takes the least time to perform a query in all showcased parameter combinations. This is a result of its query-dedicated merging thread. A design that foregoes memory conservativeness in favor of minimal query latency. A query by QPOPSS takes on the order of 10 to 100 microseconds, suiting many real-world applications. The Topkapi approach takes on the order of 100 milliseconds, which introduces delays unsuitable for real-world high-throughput applications. The query latency of QPOPSS decreases with skew level, while the latency of the two other approaches is constant regardless of stream distribution. This is most likely due to the small number of required counters for QPOPSS in the higher skew levels.

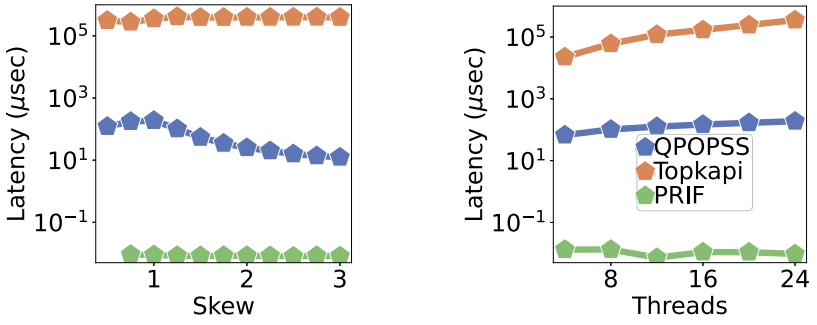
Fig. 11b contains the results of the experiments where the number of threads was varied. These results are consistent with previous ones. Due to the design of PRIF, where a single merging thread is queried, it is not affected by increasing the number of threads. QPOPSS has a slight upward trend, meaning that the number of threads dispatched affects the query latency since more QOSS instances need to be merged. Topkapi has a slightly steeper upward trend due to its cumbersome merging process, initiated each time a query is carried out. Nonetheless, QPOPSS maintains at least an order of magnitude lower latency compared to Topkapi across all numbers of threads.



(a) $D = 16,\ E = 100$ (b) $D = 32,\ E = 1000$

(c) $D = 64,\ E = 10000$ (d) $D = 128,\ E = 100000$

Fig. 10. Box plots of the query latency as the number of dispatched threads of QPOPSS vary along the x-axis. Using Platform B, the CAIDA data set, and 0.01% queries. The box limits from top to bottom represent the 75th and 25th percentiles, respectively. Whiskers are located the 25th percentile - 1.5*IQR and the 75th percentile + 1.5*IQR, where IQR is the distance between the quartiles.



(a) Latency as the Zipf skew level varies along the x-axis. T=24 threads.

(b) Latency as the number of threads vary along the x-axis

Fig. 11. Query latency with 0.01% queries, platform A and $\phi = 10^{-4}$. Note the logarithmic y-axes.

6.6. Summary

When it comes to the comparison between Space-Saving and QOSS as the inner algorithm employed by QPOPSS, throughput and latency greatly improve, especially in the lower skew levels. The results of the comparative evaluation between the state-of-the-art methods are summarized in Table 4, using both platforms A and B.

The throughput of QPOPSS excels when processing streams with high data skew and while answering large queries. Compared to PRIF and Topkapi, QPOPSS handles this scenario exceptionally well. When processing the CAIDA data set, especially when queries are present in the workload, Topkapi lacks the competitive throughput of QPOPSS and PRIF.[4] PRIF handles both the low and high query rates equally well due to its query-favored design. The precision and recall of QPOPSS are perfect when processing Zipf data sets, and the ARE is low, diminishing quickly as the stream length grows. PRIF and Topkapi have higher ARE when processing real-world data, and Topkapi has excellent ARE when processing high-skew synthetic data. When scalable memory consumption is important, QPOPSS has a clear advantage over both PRIF and Topkapi due to their counters increasing with a factor of T, which is not the case for QPOPSS. PRIF excels in latency due to its design of constant merging by a dedicated thread, while QPOPSS and Topkapi appear less favorable due to employing a merge-on-demand style of querying. Our sensitivity analysis reveals that throughput is strongly influenced by E, particularly for Zipf parameters above 1.5, where increased E enhances thread-local processing and reduces synchronization overhead. At lower skew levels, throughput becomes more dependent on D rather than E. Higher values of E and D improve performance but may introduce buffering delays, affecting accuracy as outlined in Theorem 2.

Our latency analysis highlights the significant impact of T. Higher E and D values result in more local computation, leading to reduced query

[4] Recall that this is despite the fact that Topkapi was given an advantage regarding throughput in the presence of concurrent queries, by not enforcing thread-safe synchronization, as they were not part of its design.

Table 4
Summary of the evaluation results, given a high number of dispatched threads, moderate to high data skewness, and considerable stream length. The number of ↑ and ↓ symbols indicate positive and negative comparative performance on a specific metric, respectively (i.e., ↑ on latency and ARE means a low and therefore desirable metric value).

Baseline	Query aspects		Metric					
			Throughput	Precision	Recall	ARE	Memory	Latency
QPOPSS	Few	Small	↑↑	↑↑	↑↑	↑↑	↑↑	↑
		Large	↑	↑↑	↑↑	↑↑	↑↑	↑
	Many	Small	↑	↑↑	↑↑	↑↑	↑↑	↑
		Large	↑↑	↑↑	↑↑	↑↑	↑↑	↑
PRIF	Few	Small	↑↑	↑↑	↑	↓	↓↓	↑↑
		Large	↓↓	↑↑	↑↑	↓	↓↓	↑↑
	Many	Small	↑↑	↑↑	↑	↓	↓↓	↑↑
		Large	↓↓	↑↑	↑↑	↓	↓↓	↑↑
Topkapi	Few	Small	↑	↑↑	↓↓	↑	↓	↓
		Large	↑	↑	↓	↑	↓	↓
	Many	Small	↓↓	↑↑	↓↓	↑	↓	↓
		Large	↓↓	↑	↓	↑	↓	↓

delays at larger T. For $T = 72$, mean latencies and tail distributions remain in the range of hundreds of microseconds, with occasional outliers reaching up to 3 milliseconds. These results demonstrate the trade-offs between throughput, latency, and memory efficiency, emphasizing the need for careful parameter selection based on workload characteristics.

7. Other related work

As mentioned in Section 3, several algorithms target variants of the ϵ-approximate frequent elements using multithreading, with emphasis on thread-local approaches as discussed in that section and the evaluation baselines [29,40,44,9,16]. What follows are descriptions of representative approaches and the rationale for including them in our evaluation.

The Augmented Sketch (ASketch) [40] is a highly accurate stream processing algorithm for element frequency estimation. The design comprises two interconnected data structures: a sketch and a filter. The fixed-size filter tracks elements and their occurrence. When the filter becomes full and a non-tracked element appears in the stream, the element is inserted both in the filter and in the underlying sketch. ASketch improves accuracy and increases throughput when processing streams with high data skew. The authors also provide designs for parallel processing with either the filter and the sketch running on different cores or where a complete ASketch runs on a reserved core. However, although frequent elements estimation is possible, the approach focuses on the point estimation of the frequency count of specific elements.

The HeavyKeeper algorithm [21] targets combining counter-based and sketch-based approaches for element-count tracking. By periodically decaying element counts, freshness is ensured as input data distributions shift over time. HeavyKeeper can be combined with a min-heap to track the most frequent elements of a stream. It is a sequential algorithm, though, and its parallelization, allowing concurrent updates with queries and the appropriate supporting data structures, is not discussed in the work.

M. Cafaro et al. [9] present a parallel design utilizing Space-Saving as the core algorithm in a purely thread-local design. Queries merge each Space-Saving algorithm instance in a tree-like fashion until only a final algorithm instance that contains the frequent elements is left. The design gives perfect speedup compared to a sequential Space-Saving algorithm instance. Moreover, the accuracy of the presented design was precisely equal to the single-threaded version. However, no scheme for concurrent queries is given and is therefore not usable in real-life applications where continuous queries are required, which are targeted here. This fact, coupled with our inability to find a readily available open-source implementation, made us choose not to include the approach in our evaluation.

The Cooperative Thread Scheduling Framework (CoTS) [16] and its multi-stream extension [17] are approaches for parallelizing the frequent elements problem. Similar to QPOPSS, this approach builds on the Space-Saving algorithm. The design features a single Space-Saving algorithm instance on which each thread operates. Update operations are carried out directly on the space-saving algorithm instance or handed over to whichever thread currently has exclusive access. The synchronization primitives used are lock-free, promoting high throughput.

The approach was evaluated using synthetic Zipfian data sets, showing relative throughput gains between CoTS and a lock-based design. However, the work lacks discussion on the effects of overlapping updates and queries on query accuracy and rely solely on the analysis of the sequential Space-Saving algorithm. Additionally, due to the combination of the complexity of the approach and the absence of a readily available open-source implementation, implying risks of misinterpretation of the work, it was not possible to include this approach in our evaluation.

8. Conclusions

High-throughput data analytics is important in many fields, such as network optimization, cybersecurity, and online data analysis. It requires novel algorithmic solutions that exploit concurrency to achieve a higher degree of parallelism. To this end, we analyzed the problem of finding the frequent elements of high throughput data streams with concurrent updates and queries, identifying a set of challenges.

To address these challenges, we designed and extensively evaluated Query and Parallelism Optimized Space-Saving, both analytically and empirically, exploring the extended trade-off space in the presence of concurrent queries and updates for the frequent elements problem. To address concurrency-associated accuracy challenges, we provided a bound on the space required by QPOPSS to accurately report the frequent elements of a stream. Furthermore, we bounded the frequent elements reported and their estimated occurrence when queries overlap concurrently with updates. To our knowledge, this has not been done before in the context of frequent elements estimation.

In addressing timeliness challenges, we proposed the Query Optimized Space-Saving algorithmic implementation, which drastically reduces the query latency compared to the original approach. We evaluated QPOPSS through comparison with representative methods in the literature, using synthetic and real-life data sets and using two different NUMA hardware platforms. The results clearly show that QPOPSS accurately and swiftly reports the frequent elements of a stream compared to other approaches, using extremely few bytes of memory, addressing challenges related to memory footprint and accuracy. This can be attributed to the unique combination of domain partitioning, inter-thread

filters, and query-optimized synopsis data structures, which together ensure high throughput, low latency, low memory, and high accuracy for an overall balanced approach. The code and the data used (alt. code to generate the synthetic data-sets) are openly available [26].

Future work could extend this method beyond frequent elements to quantiles, histograms, wavelets, and related problems. Moreover, recent advances where parallelization improvements are of interest include methods for finding frequent elements in streams of both updates and removals [45]. Such works can lay the groundwork for efficient mechanisms for tracking the most recent frequent elements [6,7], as the stream data distribution may change over time. It is of particular interest to study efficient mechanisms for maintaining frequent elements in a sliding-window framework, given that real-world data streams often exhibit non-stationary behavior.

CRediT authorship contribution statement

Victor Jarlow: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Charalampos Stylianopoulos:** Writing – review & editing, Writing – original draft, Software, Methodology, Formal analysis. **Marina Papatriantafilou:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the following grants: the Marie Skłodowska-Curie Doctoral Network project RELAX-DN, funded by the European Union under Horizon Europe 2021-2027 Framework Programme Grant Agreement nr. 101072456; the Chalmers AoA frameworks Energy and Production, WPs INDEED, and “Scalability, Big Data and AI”, respectively; the Swedish Research Council grant “EPITOME” (VR 2021-05424); the Wallenberg AI, Autonomous Systems and Software Program and Wallenberg Initiative Materials for Sustainability prj. WASP-WISE STRATIFIER; and the TANDEM project (project no. IEM2022-08) within the framework of the Swedish Electricity Storage and Balancing Centre (SESBC), co-funded by the Swedish Energy Agency (project no. 2021-035871).

Data availability

The data and code is shared through references in the manuscript.

References

[1] Anonymized Internet traces 2019, https://catalog.caida.org/dataset/passive_2019_pcap. (Accessed 11 March 2025).
[2] Yehuda Afek, Anat Bremler-Barr, Edith Cohen, Shir Landau Feibish, Michal Shagam, Efficient distinct heavy Hitters for DNS DDoS attack detection, arXiv:1612.02636 [cs]. Available from http://arxiv.org/abs/1612.02636, 2016.
[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, Hedera: dynamic flow scheduling for data center networks, in: USENIX Conference on Networked Systems Design and Implementation, vol. 7(1), 2010, pp. 281–296.
[4] Daniel Anderson, Pryce Bevan, Kevin Lang, Edo Liberty, Lee Rhodes, Justin Thaler, A high-performance algorithm for identifying frequent items in data streams, in: Proceedings of the 2017 Internet Measurement Conference, 2017, pp. 268–282.
[5] Mike D. Atkinson, Jörg-R. Sack, Nicola Santoro, Thomas Strothotte, Min-max heaps and generalized priority queues, Commun. ACM 29 (10) (1986) 996–1000.
[6] Ran Ben-Basat, Gil Einziger, Roy Friedman, Yaron Kassner, Heavy Hitters in streams and sliding windows, IEEE Int. Conf. Comput. Commun. 35 (2016) 1–9.
[7] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, Erez Waisbard, Memento: making sliding windows efficient for heavy Hitters, IEEE/ACM Trans. Netw. 30 (4) (2022) 1440–1453.
[8] Richard A. Brualdi, Introductory Combinatorics, 5th edition, Pearson/Prentice Hall, Upper Saddle River, N.J., 2010.
[9] Massimo Cafaro, Marco Pulimeno, Piergiulio Tempesta, A parallel space saving algorithm for frequent items and the Hurwitz zeta distribution, Inf. Sci. 329 (2016) 1–19.
[10] Moses Charikar, Kevin Chen, Martin Farach-Colton, Finding frequent items in data streams, Theor. Comput. Sci. 312 (1) (2004) 3–15.
[11] Graham Cormode, Data summarization and distributed computation, in: Proceedings of the ACM Symposium on Principles of Distributed Computing, 2018, pp. 167–168.
[12] Graham Cormode, Current trends in data summaries, SIGMOD Rec. 50 (4) (2022) 6–15.
[13] Graham Cormode, Marios Hadjieleftheriou, Finding frequent items website, http://hadjieleftheriou.com/frequent-items/, 2005. (Accessed 11 March 2025).
[14] Graham Cormode, Marios Hadjieleftheriou, Finding frequent items in data streams, VLDB Endow. 1 (2) (2008) 1530–1541.
[15] Graham Cormode, Shan Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, J. Algorithms 55 (1) (2005) 58–75.
[16] Sudipto Das, Shyam Antony, Divyakant Agrawal, Amr El Abbadi, CoTS: a scalable framework for parallelizing frequency counting over data streams, IEEE Int. Conf. Data Eng. 25 (1) (2009) 1323–1326.
[17] Sudipto Das, Shyam Antony, Divyakant Agrawal, Amr El Abbadi, Thread cooperation in multicore architectures for frequency counting over multiple data streams, Proc. VLDB Endow. 2 (1) (2009) 217–228.
[18] Erik D. Demaine, Alejandro López-Ortiz, Ian Munro, Frequency Estimation of Internet Packet Streams with Limited Space, Algorithms — ESA 2002, vol. 2461, Springer, 2002, pp. 348–360.
[19] Cristian Estan, George Varghese, New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice, ACM Trans. Comput. Syst. 21 (3) (2003) 270–313.
[20] Minos Garofalakis, Johannes Gehrke, Rajeev Rastogi, Data Stream Management: A Brave New World, Springer, Berlin Heidelberg, 2016, pp. 1–9.
[21] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, Xiaoming Li, HeavyKeeper: an accurate algorithm for finding top-k elephant flows, in: USENIX Annual Technical Conference, 2018, pp. 909–921.
[22] Rob Harrison, Qizhe Cai, Arpit Gupta, Jennifer Rexford, Network-wide heavy hitter detection with commodity switches, in: Proceedings of the Symposium on SDN Research, 2018, pp. 1–7.
[23] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, Ana Sokolova, Quantitative relaxation of concurrent data structures 48 (1) (jan 2013) 317–328.
[24] Aitor Hernandez, Bin Xiao, Valentin Tudor, Eraia-enabling intelligence data pipelines for iot-based application systems, in: IEEE International Conference on Pervasive Computing and Communications, 2020, pp. 1–9.
[25] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, Ali Vakilian, Learning-based frequency estimation algorithms, in: International Conference on Learning Representations, 2019.
[26] Victor Jarlow. QPOPSS Github, https://github.com/jarlow/QPOPSS, 2025. (Accessed 11 March 2025).
[27] Richard M. Karp, Scott Shenker, Christos H. Papadimitriou, A simple algorithm for finding frequent elements in streams and bags, ACM Trans. Database Syst. 28 (1) (2003) 51–55.
[28] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, Jia Wang, Data streaming algorithms for efficient and accurate estimation of flow size distribution, in: Proceedings of the Joint ACM International Conference on Measurement and Modeling of Computer Systems, 2004, pp. 177–188.
[29] Ankush Mandal, He Jiang, Anshumali Shrivastava, Vivek Sarkar, Topkapi: parallel and fast sketches for finding top-k frequent elements, Proc. Int. Conf. Neural Inf. Process. Syst. 32 (2018) 10921–10931.
[30] Nishad Manerikar, Themis Palpanas, Frequent items in streaming data: an experimental evaluation of the state-of-the-art, Data Knowl. Eng. 68 (4) (2009) 415–430.
[31] Gurmeet Singh Manku, Rajeev Motwani, Approximate frequency counts over data streams, Proc. VLDB Endow. 5 (2012) 346–357.
[32] Chandler May, Kevin Duh, Benjamin Van Durme, Ashwin Lall, Streaming word embeddings with the space-saving algorithm, ArXiv, 2017.
[33] Ahmed Metwally, Divyakant Agrawal, Amr El Abbadi, An integrated efficient solution for computing frequent and top- k elements in data streams, ACM Trans. Database Syst. 31 (3) (2006) 1095–1133.
[34] Jaydev Misra, David Gries, Finding repeated elements, Sci. Comput. Program. 2 (2) (1982) 143–152.
[35] Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, Philippas Tsigas, A consistency framework for iteration operations in concurrent data structures, in: IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 239–248.
[36] Kostas Pagiamtzis, Ali Sheikholeslami, Content-Addressable Memory (CAM) circuits and architectures: a tutorial and survey, IEEE J. Solid-State Circuits 41 (3) (2006) 712–727.

[37] Arik Rinberg, Idit Keidar, Intermediate value linearizability: a quantitative correctness criterion, International Symposium on Distributed Computing 179 (2020) 2:1–2:17.
[38] Arik Rinberg, Idit Keidar, Intermediate value linearizability: a quantitative correctness criterion, J. ACM 70 (2) (2023).
[39] Kay A. Robbins, Steven Robbins (Eds.), Unix Systems Programming: Communication, Concurrency, and Threads, Prentice Hall PTR, Upper Saddle River, NJ, 2003.
[40] Pratanu Roy, Arijit Khan, Gustavo Alonso, Augmented sketch: faster and more accurate stream processing, in: ACM International Conference on Management of Data, 2016, pp. 1449–1463.
[41] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, Jennifer Rexford, Heavy-hitter detection entirely in the data plane, in: Proceedings of the ACM Symposium on SDN Research, 2017, pp. 164–176.
[42] Sebastian U. Stich, Jean-Baptiste Cordonnier, Martin Jaggi, Sparsified sgd with memory, Adv. Neural Inf. Process. Syst. 31 (2018) 4452–4463.
[43] Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, *Delegation sketch*: a parallel design with support for fast and accurate concurrent operations, Eur. Conf. Comput. Syst. 15 (2020) 1–16.
[44] Yu Zhang, Yue Sun, Jianzhong Zhang, Jingdong Xu, Ying Wu, An efficient framework for parallel and continuous frequent item monitoring, Concurr. Comput., Pract. Exp. 26 (2014) 2856–2879.
[45] Fuheng Zhao, Divyakant Agrawal, Amr El Abbadi, Ahmed Metwally, Spacesaving ±: an optimal algorithm for frequency estimation and frequent items in the bounded-deletion model, Proc. VLDB Endow. 15 (6) (2022) 1215–1227.
[46] George Kingsley Zipf, Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology, Addison-Wesley Press, 1949.
[47] Vinh Quang Ngo, Marina Papatriantafilou, Cuckoo Heavy Keeper and the balancing act of maintaining heavy hitters in stream processing, Proc. VLDB Endow. 18 (9) (2025).

Victor Jarlow is a researcher at AstaZero AB. He received his master's degree in Computer Science with a specialization in distributed systems and computer networks from the University of Gothenburg. His research interests are in the areas of parallel and distributed systems, parallel architectures, network systems, and methods for testing connected autonomous vehicles with a focus on big data, AI, and distributed algorithms.

Charalampos Stylianopoulos is a senior software engineer at emnify. He received the bachelor's and master's degrees in Electrical and Computer Engineering from the National Technical University of Athens. He holds a PhD in Computer Science from Chalmers University of Technology. His research interests are in the areas of parallel and distributed systems, parallel architectures, and network systems.

Marina Papatriantafilou is an Associate professor and co-head of the Distributed Computing Systems group at Chalmers University of Technology. She holds a PhD in Computer Engineering and Informatics from Patras University. She has also been with the National Research Institute for Mathematics and Computer Science, Amsterdam (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken. Her research is focused on robust and efficient distributed algorithms and their applications in multiprocessor/multicore systems and network-based distributed systems, consistency, and fine-grain synchronization, including data-stream/big-data processing, efficient processing of varying volumes of data; fault-tolerance in multicore/distributed systems, cyber-physical systems, and digitalization.