Cuckoo Heavy Keeper and the balancing act of maintaining heavy hitters in stream processing

Vinh Quang Ngo
Chalmers Un. of Technology and Gothenburg Un.
Gothenburg, Sweden
vinhq@chalmers.se

Marina Papatriantafilou
Chalmers Un. of Technology and Gothenburg Un.
Gothenburg, Sweden
ptrianta@chalmers.se

ABSTRACT

Finding heavy hitters in databases and data streams is a fundamental problem with applications ranging from network monitoring to database query optimization, machine learning, and more. Approximation algorithms offer practical solutions, but they present tradeoffs involving throughput, memory usage, and accuracy. Moreover, modern applications further complicate these trade-offs by demanding capabilities beyond sequential processing that require both parallel scaling and support for concurrent queries and updates.

Analysis of these trade-offs led us to the key idea behind our proposed streaming algorithm, *Cuckoo Heavy Keeper* (CHK). The approach introduces an inverted process for distinguishing frequent from infrequent items, which unlocks new algorithmic synergies that were previously inaccessible with conventional approaches. By further analyzing the competing metrics with a focus on parallelism, we propose an algorithmic framework that balances scalability aspects and provides options to optimize query and insertion efficiency based on their relative frequencies. The framework is capable of parallelizing any heavy-hitter detection algorithm.

Besides the algorithms' analysis, we present an extensive evaluation on both real-world and synthetic data across diverse distributions and query selectivity, representing the broad spectrum of application needs. Compared to state-of-the-art methods, CHK improves throughput by 1.7-5.7× and accuracy by up to four orders of magnitude even under low-skew data and tight memory constraints. These properties allow its parallel instances to achieve near-linear scale-up and low latency for heavy-hitter queries, even under a high query rate. We expect the versatility of CHK and its parallel instances to impact a broad spectrum of tools and applications in large-scale data analytics and stream processing systems.

PVLDB Reference Format:
Vinh Quang Ngo and Marina Papatriantafilou. Cuckoo Heavy Keeper and the balancing act of maintaining heavy hitters in stream processing. PVLDB, 18(9): 3149 - 3161, 2025.
doi:10.14778/3746405.3746434

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.5281/zenodo.15593109.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746434

1 INTRODUCTION

The problem of finding heavy hitters in a set or stream of items requires identifying the items that appear more times than a specified fraction ϕ of the set or the processed stream size [29]. Besides, it is often needed by downstream applications also to return the estimated heavy hitters' frequencies. Corporations such as AT&T [10], Google [33], and Cloudflare [8] extensively use heavy-hitter detection solutions for network traffic analysis [20, 38, 41], anomaly detection [24], and iceberg query processing [5, 17]. These algorithms are also implemented in analytics platforms including Druid, Redis, and Databricks [4, 13, 34]. Most recently, heavy-hitter detection has also seen prominent applications in large language models, where identifying frequently accessed tokens can improve inference throughput significantly [43].

Considering the size and input rate of the data, it is important to find algorithms that have favorable memory requirements and system alignment (e.g., cache-friendliness), as well as capabilities to process stream items promptly. Knowing that the exact solution requires memory linear to the number of distinct items in the stream [19], and that many applications accept some approximation, a substantial volume of literature focuses on succinct (sublinear) representations from which heavy hitters and their frequencies can be queried approximately. A common formulation is the *ϵ-ϕ-heavy hitters* problem, where the requirement is to return the estimated heavy hitters and their estimated frequencies, approximated within a bounded difference ϵ from their true frequencies. If, furthermore, the requirement allows that the estimated heavy hitter and its frequency are within the bounded difference with a probability of at least $1 - \delta$, the problem is known as the *(ϵ, δ)-ϕ-heavy hitters*. Such a relaxation can align with an even lower memory footprint.

Challenges. Given the same amount of *memory*, *ϵ-ϕ-heavy hitters* algorithms can be compared based on their *throughput and accuracy* across different workloads. They can be grouped into key-value *(KV)-based*, *sketch-based*, and *hybrid algorithms*. *KV-based* algorithms such as *Frequent* [29], *LossyCounting* [27], and *Space-Saving* [28] maintain a fixed number of key-value counters to find the heavy hitters deterministically. While these perform well in finding heavy hitters due to their deterministic nature, they suffer from significant frequency estimation errors and throughput challenges. *Sketch-based* algorithms such as *Count-MinSketch* [11] and *CountSketch* [7] typically use a series of scalar counters arranged in a two-dimensional array with multiple hash functions to map input items to counters; however, collisions can cause infrequent items to be mistaken for frequent ones. *Hybrid* algorithms such as *HeavyGuardian* [40], *ElasticSketch* [41], *AugmentedSketch* [37], *CuckooCounter* [38], *Topkapi* [26], and *HeavyKeeper* [20] combine the strengths of both KV-based and sketch-based approaches to provide better results regarding throughput, memory usage, and accuracy. Nevertheless, depending on the ways techniques are combined and used, there are synergies and trade-offs, which call for better balancing of competing requirements and further improvements.

Moreover, as data volumes and rates continue to grow while applications evolve, there is an increasing need for solutions that go beyond sequential processing capabilities, posing *challenges on both scalable parallel performance and support for concurrent queries and updates* [19]. While there is some recent work realizing such needs [22, 35, 36, 39], parallelizing *ϵ-ϕ-heavy hitters* algorithms has focused mainly on parallelizing insertions (cf. [26, 42]), with only one exception [23], to the best of our knowledge, on concurrent queries and insertions.

Contributions. We provide ways to balance and improve the multi-faceted trade-offs of the problem, in two orthogonal directions:

First, we propose *Cuckoo Heavy Keeper* (CHK), a fast, accurate, and space-efficient *ϵ-ϕ-heavy hitters* algorithm. CHK inverts and repurposes components in the conventional data flow seen in hybrid approaches [37, 40, 41] — instead of placing items directly into the *heavy part* and redirecting them to a fallback storage (where estimated frequencies have higher relative error) when needed, CHK requires items to first pass through the *lobby part* and prove their significance before being promoted to the *heavy part*; i.e. the *lobby* acts as a lightweight filter to identify potential heavy hitters, while the *heavy part* maintains accurate counts of heavy candidates through hash collision resolution. This inversion unlocks new algorithmic synergies (e.g., applying hash collision resolution selectively to heavy-hitter candidates), which were inaccessible with common existing approaches, thus resulting in better throughput, memory efficiency, and accuracy, even with low-skew data. This key idea also enables an algorithmic implementation of CHK with a system-aware layout and calculation optimizations.

Second, we introduce a parallel algorithmic framework for heavy-hitter detection that supports concurrent insertions and queries via *domain-splitting* [39] (partitioning the input and assigning each partition to a thread). This framework offers two complementary designs: *mCHK-I*, targeting workloads where insertions and frequency queries are predominant compared to heavy-hitter queries, and *mCHK-Q*, for scenarios where heavy-hitter queries are more frequent. Notably, our parallel designs are compatible with any heavy-hitter detection algorithm without requiring the underlying data structures to support mergeability [2]. This flexibility allows a wider range of algorithms to be parallelized using our approach.

Moreover, we provide analytical bounds for the algorithms, an open-source repository with their implementations [31], as well as a comprehensive experimental evaluation on both real-world and synthetic datasets with diverse distributions, across different hardware platforms and varying query selectivity, in comparison with an array of representative established methods in the literature. The results show that CHK and its parallel counterparts generalize across varied workloads and offer predictable performance in practical deployments where the stream properties and memory requirements cannot be known in advance. They can process streams of diverse skewness with orders of magnitude improved accuracy, even under memory limitations. This improvement directly translates to parallel performance benefits — with fewer infrequent items incorrectly identified as heavy hitters, threads need less synchronization to maintain consistency, resulting in less data movement overhead. This implies clear benefits in both timeliness and scalability. All these properties make the CHK algorithm family a powerful and potentially influential component in tool-chains for large-scale data analytics.

Table 1: Notation Summary for Heavy Hitters Problem

Notation	Description
S	Input stream of item tuples $(a_1, \dots, a_t, \dots)$
N	Total weighted size of the processed data stream
$a_t = (e, w)$	Tuple at timestamp/position t, where $e \in U$ is an item drawn from the input universe U with weight w
ϕ	Heavy-hitter frequency threshold
ϵ	Approximation bound for frequency estimation
δ	Confidence parameter for probabilistic guarantees
$f(e)$	True frequency of item e
$\hat{f}(e)$	Estimated frequency of item e
R	Set of true heavy hitters $\{\langle e, f(e)\rangle \mid f(e) \geq \phi N\}$
$\hat{R}$	Set of estimated heavy hitters returned

Roadmap. In §2 we describe the problem, followed by its analysis relative to related work in §3. In §4 and §5 we detail the CHK algorithm design and its analysis, while §6 is about the parallel algorithm designs, also in association with related work. In §7 we present our detailed empirical evaluation and we conclude in §8.

2 PROBLEM DESCRIPTION

Given a data stream, heavy hitters are items whose frequency exceeds a threshold ϕ of the processed stream size N. The problem was first formally described by Misra and Gries [29] as follows:
ϵ-ϕ-heavy hitters: Given a stream $S = (a_1, \dots, a_t, \dots)$ where each $a_t = (e, w)$ represents a tuple with e being an item from the input universe U, t being a timestamp, w (a positive integer) being the weight of the tuple, and $f(e)$ denoting the true frequency of item e, where each update $a_t = (e, w)$ increments $f(e)$ by w. Let $N = \sum_{e \in U} f(e)$ denote the total weighted size of the processed stream, $\phi \in (0, 1)$ denote the heavy-hitter threshold and $\epsilon \in (0, 1)$ ($\epsilon \ll \phi$) denote the approximation bound. Let $R = \{\langle e, f(e)\rangle \mid f(e) \geq \phi N\}$ be the set of true heavy hitters in the processed stream and $\hat{R} \subset \{\langle e, \hat{f}(e)\rangle \mid e \in U\}$ be the set of estimated heavy hitters returned by the query. The *ϵ-ϕ-heavy hitters* must satisfy:
C1: If $f(e) \geq \phi N$, then $e \in \hat{R}$ (no false negatives)
C2: If $f(e) \leq (\phi - \epsilon)N$, then $e \notin \hat{R}$ (limited false positives)
C3: For each $e \in \hat{R}$, $|f(e) - \hat{f}(e)| \leq \epsilon N$ (bounded deviation)

Let $\delta \in (0, 1)$ denote the confidence parameter for probabilistic guarantees; an **(ϵ, δ)-ϕ-heavy hitters** algorithm guarantees:
C4: For each condition (C1–C3), if the premise is met, then with probability at least $1 - \delta$, the stated outcome holds.

We primarily focus on the (ϵ, δ)-ϕ-heavy hitters problem, and use deterministic variants as baselines for theoretical and empirical comparisons. The notation summary is provided in Table 1.

Heavy-hitter data structures support the following operations:
Update(e, w): Given an item $e \in U$ and weight w, processes the stream tuple (e, w) and maintains necessary data structure state.
f-Query(e): returns e's estimated frequency $\hat{f}(e)$. If e is a heavy hitter ($e \in R$), the estimate satisfies conditions (C1-C4).
hh-Query(): Returns the set $\hat{R}$ of estimated heavy hitters along with their estimated frequencies $\hat{f}(.)$. The returned results must satisfy conditions (C1-C4).

Metrics of interest. (1) *Precision* $\left(\frac{|R \cap \hat{R}|}{|\hat{R}|}\right)$ measures the fraction of reported heavy hitters that are true ones. (2) *Recall* $\left(\frac{|R \cap \hat{R}|}{|R|}\right)$ measures the fraction true heavy hitters identified. (3) *Average Relative Error (ARE)* $\left(\frac{1}{|R|}\sum_{e \in R}\frac{|f(e)-\hat{f}(e)|}{f(e)}\right)$ measures the deviation between true and estimated frequencies across true heavy hitters. (4) *Throughput* measures the number of operations processed per time unit. (5) *Query latency* measures the time to process a query. (6) *Memory usage* measures the space needed for data structures.

3 RELATED WORK AND PROBLEM ANALYSIS

3.1 Traditional approaches

Key-Value (KV)-based algorithms such as *Frequent* [29], *LossyCounting* [27], and *Space-Saving* [28] maintain a fixed set of approximately $1/\phi$ key-value counters to track the frequencies of heavy hitters. Because of this fixed size, only potential heavy-hitter candidates can be tracked. Upon item arrival, if the item is already tracked, its counter is incremented; otherwise, the algorithm must either allocate a new counter (if available) or reassign an existing one. The returned heavy hitters set satisfies **(C1-C3)** (cf. §2). However, their estimated frequencies may suffer from significant errors due to over-/under-estimation, and/or their update operations can be computationally expensive, which can affect downstream tasks [9].

Sketch-based algorithms, such as the *CountSketch* and *Count-MinSketch* [7, 11], use a series of scalar counters arranged in a $d \times w$ array. Each row corresponds to one of d pairwise-independent hash functions $h_1, h_2, \ldots, h_d$, each h_i mapping items from U to one of the w counters in row i. Each input tuple's item is hashed with each h_i to determine which counters to update. The estimated frequency of an item is calculated by aggregating the counts from its hashed positions (taking the average or minimum in the aforementioned methods), satisfying conditions **C1-C4** (cf. §2). However, since multiple items may be mapped to the same counter, low-frequency items may be mistakenly identified as high-frequency ones, leading to incorrect identification of heavy hitters [9].

3.2 Recent advances

Based on the strengths of each of the approaches, new techniques provided advances in throughput, memory usage, and accuracy.

3.2.1 *Advances regarding Throughput.*

Sketch-based approaches achieve better throughput compared to KV-based ones, due to lower time complexity for updates, through hashing. Later approaches target improved throughput in similar ways by combining hashing with counter-based methods, such as *Space-Saving Heap* [9], *HeavyKeeper* [20], and *Topkapi* [26], or by adopting faster hashing schemes like *cuckoo hashing* [32] in *CuckooCounter* [38].

3.2.2 *Advances regarding Memory Usage.*

In many real-world data streams, only a small fraction of items appear frequently enough to become heavy hitters. This suggests dividing the data structure into a *heavy* and a *light part*, each handling either frequent or infrequent items. Hence, more suitable data structures with proper sizes, potentially smaller for the less important parts, can be used for each substructure. Commonly, the *heavy part* is implemented as a simple $\langle Key, Frequency\rangle$ key-value data structure to store heavy hitters. When an item is inserted, the algorithm first checks the *heavy part*; if the item is there, its count is updated; else, the item is inserted into a sketch which acts as the *light part. HeavyGuardian* [40], *ElasticSketch* [41], and *AugmentedSketch* [37] adopt this approach.

3.2.3 *Advances regarding Accuracy.*

To improve the accuracy of the estimated frequency of heavy hitters, the *count-with-exponential-decay* method was introduced and used in *HeavyGuardian* [40] and *HeavyKeeper* [20]. This technique modifies how counter values are updated when hash collisions occur. Unlike the key-value counter in the Frequent algorithm [29] that uniformly decrements counters regardless of their values, the *count-with-exponential-decay* lowers the counter by 1 with probability b^{-C}, for every unit of the update weight, where b is a decay base and C is the current counter value. If the item is a heavy hitter, C is high, hence, the probability to decrement the counter will be lower, and the item retains a more accurate frequency value.

3.2.4 *Synergetic effects and Trade-offs.*

The aforementioned techniques are different high-level concepts with a variety of algorithmic design choices and combinations. Each choice or combination means different synergies or trade-offs. For example, frequent/infrequent separation not only reduces the memory used but potentially improves the throughput through a fast path created by the part storing frequent items, and also has better cache behavior, as seen in *HeavyGuardian* [40] and *ElasticSketch* [41]. However, these techniques can show opposite effects under different conditions. In lower-skew datasets, *HeavyGuardian* and *ElasticSketch* may suffer accuracy loss due to frequent collisions, and *AugmentedSketch* [37] may experience reduced throughput from frequent data movement between parts. Moreover, applying *count-with-exponential-decay* on all counters regardless of the item's status can hurt throughput due to floating point operations.

4 SEQUENTIAL CUCKOO HEAVY KEEPER

By studying the strengths and multi-faceted trade-offs of different designs, we aim to balance and improve the metrics of interest by combining key algorithmic elements in a novel, system-aware fashion, that harmonizes their benefits and enables suitability for varying input features. We propose a new algorithm, *Cuckoo Heavy Keeper*, which represents a fundamental rethinking of the frequent/infrequent separation concept for heavy-hitter detection. This reinterpretation — as will be clarified in the following — unlocks synergies that were previously inaccessible with conventional approaches. We start by outlining the data structure layout, followed by a discussion of the design choices. Next, we explain the algorithm's operations, the weighted update, and other optimizations.

4.1 Cuckoo Heavy Keeper data structure layout

Fig. 1 shows the layout of the *Cuckoo Heavy Keeper.* The notation related to the data structure is summarized in Table 2. *Cuckoo Heavy Keeper* maintains two tables $T[0]$ and $T[1]$[1], each consisting of an array of buckets. Each bucket has one lobby entry for filtering infrequent items and multiple heavy entries (e.g., 2-4 per bucket, cf.

[1]Cuckoo Heavy Keeper can be implemented using two separate tables or a single one with two hash functions; we use the two-table approach (as in the original cuckoo hashing work [32]), which guarantees distinct candidate locations.

Table 2: Notation Summary for Cuckoo Heavy Keeper

Notation	Description
$T[2][]$	Two tables of buckets
fp	Fingerprint of item e
idx_0, idx_1	Indices in the hash tables computed from e
$T[i][idx_i]$	Bucket at position idx_i in table i ($i \in \{0, 1\}$)
$T[i][idx_i].lobby$	Lobby entry in bucket $T[i][idx_i]$
$T[i][idx_i].heavy$	Array of heavy entries in bucket $T[i][idx_i]$
L	Promotion threshold for lobby entries
$\mathcal{B}$	Number of buckets in each table
b	Decay base used in counter decay
$de[k]$	Expected decays to reduce $C = k$ to 0
`MAX_KICKS`	Maximum number of kicks allowed

§7 for parameter selection rationale) for maintaining heavy-hitter candidates. Each bucket's lobby entry ($T[i][idx_i].lobby$) stores a tuple $\langle fp, C\rangle$ where fp is a fixed-size fingerprint and C is small counter implementing the *count-with-exponential-decay* method. The heavy entries ($T[i][idx_i].heavy$) use the same tuple format but with larger counters, for more precise tracking.

4.2 Cuckoo Heavy Keeper key ideas

Count-with-Exponential-Decay as a filter. We observe that *count-with-exponential-decay*, typically used for counting heavy items (in *heavy part*) in prior work, can be repurposed as a *lobby* that holds only *potential* heavy hitters, as it allows frequent items to accumulate count while infrequent items fade quickly. Our approach *inverts* the conventional workflow in frequent/infrequent separation: while algorithms [37, 40, 41] direct items first to the *heavy part* and use a *light part* as fallback storage when no space remains or none of the existing items in the *heavy part* can be replaced, our *Cuckoo Heavy Keeper* eliminates the *light part* entirely. Instead, items must first go through a *lobby* that filters potential heavy hitters eligible for promotion to the *heavy part*. Once an item is identified as a heavy-hitter candidate, it is moved to the *heavy part* where it is tracked accurately without decay operations.

Collision resolution in the *heavy part*. Most *heavy part* implementations use simple key-value data structures without hash collision handling [37, 40, 41], meaning that, for example, in low-memory settings, heavy items can be hashed to the same bucket, forcing one to be moved to the *light part* where it is tracked less accurately. This motivates our use of *cuckoo hashing* [32] inside the *heavy part* to give colliding heavy hitters a second chance, which significantly improves recall in diverse settings, particularly under memory constraints and low-skew distributions where collisions among true heavy hitters are more likely to occur. Importantly, we only resolve hash collisions for heavy-hitter candidates, which is only possible due to our repurposed *lobby*. In contrast, algorithms like *CuckooCounter* [38] resolve collisions for all items, which implies trade-offs in throughput depending on stream cardinality.

System-Aware Design. Most existing works focus primarily on asymptotic complexity, which cannot capture system-awareness aspects such as cache efficiency, data movement patterns, and computational costs of floating-point arithmetic. These factors, however, significantly influence algorithms' running time (cf. [15] and references therein). While maintaining good asymptotic complexity, we also recognize the importance of system-level considerations in our design. The bucket layout illustrated in Fig. 1 considers these system-level factors and offers two advantages. First, storing the

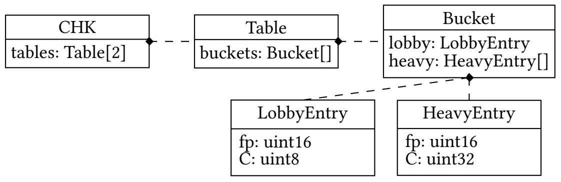

Figure 1: *Cuckoo Heavy Keeper* consists of two tables of buckets. Each bucket has one lobby entry to filter infrequent items and multiple heavy entries to maintain heavy-hitter candidates.

lobby entry and heavy entries together in the same bucket allows the bucket indices to be calculated only once, for both the *lobby* and the *heavy part*. Second, when accessing any entry in a bucket, the CPU cache line brings in all the entries in that bucket, which helps to check and update the frequencies of items in both parts efficiently, minimizing memory bandwidth contention. The approach combines the best design practices of *cuckoo hashing*, which was shown analytically to improve cache utilization and increase occupancy rates [14, 18]. Furthermore, our design restricts the need for floating point operations of *count-with-exponential-decay* within a small range, bounded by the threshold of the *lobby part* (as large counters "live" only in the *heavy part*, where they are no longer subject to floating point operations). This enables the possibility of pre-calculation and tabulation, to replace expensive operations with simple lookups, as we elaborate in subsequent sections.

4.3 Cuckoo Heavy Keeper main operations

For an item e, the algorithm stores a fixed-size fingerprint $fp = fingerprint(e)$; its two possible mapped bucket indices are calculated as $idx_0 = hash(e)$ and $idx_1 = idx_0 \oplus hash(fp)$. Given either idx_0 or idx_1 and the fingerprint, the other index can be derived using *XOR* operations. This technique, known as *partial-key cuckoo hashing* [16], reduces memory footprint by using smaller fingerprints while maintaining a low false positive rate for identity checks.

Update (Alg. 1): For each input tuple (e, w), the fingerprint fp and both bucket indices idx_0, idx_1 for $T[0], T[1]$ are generated using independent hash functions (Alg. 2, `GenerateFpAndIndexes` function, cf. [16]). The update process follows three cases:

Case 1 - Item is tracked in *heavy part* (Alg. 1, l. 4-6): If fp matches an entry in $T[i][idx_i].heavy$ of either possible bucket, the algorithm increments the matched entry's counter by w and returns. This is the *most common and fastest path* (green part in pseudo-code) — since tracked heavy-hitter candidates account for a large portion of the stream, this path will be taken most of the time. Additionally, since all heavy entries in a bucket are in the same cache line, they can be checked and updated without additional memory accesses.

Case 2 - Item is tracked in lobby part (Alg. 1, l. 7-9): If fp matches a lobby entry, the entry's counter C is incremented; if it exceeds L, a threshold parameter, the algorithm attempts promotion (Alg. 2, `TryPromote` function) for the item from *lobby* to *heavy part*:

- If an empty heavy entry exists in the bucket, the item is promoted directly to the *heavy part*, along with its counter.
- Otherwise, it tries promotion, which succeeds with probability $(C - L)/(C_{min}^{heavy} - L)$ where C_{min}^{heavy} is the smallest counter in the bucket's heavy entries (Fig. 2). If the promotion succeeds, the

Algorithm 1: Cuckoo Heavy Keeper - Main Operations

```
1  Procedure Update($e$, $w$)
2      $N \leftarrow N + w$
3      $fp, idx_0, idx_1 \leftarrow$ GenerateFpAndIndexes($e$)
4      // Found and updated in heavy ▷ most common and fastest path
5      if CheckAndUpdateHeavy($fp, idx_0, idx_1, w$) then
6          return $\hat{f}(e)$
7      // Found and updated in lobby
8      if CheckAndUpdateLobby($fp, idx_0, idx_1, w$) then
9          return $\hat{f}(e)$
       // If empty lobby exists
10     if exists $i \in \{0, 1\}$ such that $T[i][idx_i].lobby$ is empty then
11         Insert $\langle fp, w \rangle$ into empty $T[i][idx_i].lobby$
12         if w $\geq L$ then
13             TryPromote($T[i][idx_i]$)
14         return $\hat{f}(e)$
15     // Count with exponential decay
16     $i \leftarrow fp$ mod 2
17     $C_{new} \leftarrow$ DecayCounter($T[i][idx_i].lobby.C, w$)
18     if $C_{new} = 0$ then
19         Update $T[i][idx_i].lobby$ with $\langle fp, w - de[C] \rangle$
20     else
21         $T[i][idx_i].lobby.C \leftarrow C_{new}$
22     if $T[i][idx_i].lobby.C \geq L$ then
23         TryPromote($T[i][idx_i]$)
24     return $\hat{f}(e)$
25 Procedure f-Query($e$)
26     $fp, idx_0, idx_1 \leftarrow$ GenerateFpAndIndexes($e$)
27     return $\hat{f}(e)$ if found, 0 otherwise
```

promoted item's counter becomes C_{min}^{heavy} and cuckoo kickout is initiated, to relocate displaced entries (Alg. 2, Kickout function). The Kickout function relocates the displaced entry to its alternate bucket ("alt bucket" in Alg. 2, l.44) in the other table. If the alternate bucket has an empty heavy entry, the displaced entry moves there directly. Otherwise, it displaces the heavy entry with the minimum counter, which is then recursively kicked out. This recursive process continues until an empty heavy entry is found, or MAX_KICKS attempts are reached, or the counter of a displaced item falls below ϕN^2, indicating it is no longer a viable heavy-hitter candidate. If the promotion fails, the item remains in the *lobby part*, with its counter set to L. This probabilistic promotion ensures that higher-frequency items are more likely to be promoted to the *heavy part*.

Case 3 - Item is not tracked (Alg. 1, l. 10-22): If there is an empty lobby entry in either of the buckets that the item is hashed to, the algorithm inserts $\langle fp, w \rangle$ directly, promotes it if $w \geq L$, and returns. Otherwise, it applies *count-with-exponential-decay* to the target lobby entry, which is determined using *modulo divsion hashing* (fp mod 2) to ensure consistent bucket selection. For unweighted updates ($w = 1$), the procedure decays the counter with probability b^{-C}, where b is the decay base and C is the current counter value. After the decay operation, if the counter $C = 0$, the existing fingerprint in $T[i][idx_i].lobby$ is replaced with the fingerprint fp of the incoming item e. For weighted updates ($w > 1$), the DecayCounter function in Alg. 2 simulates a sequence of unweighted updates: it calculates how many decay operations would be needed for the existing counter to reach zero, and then compares this with the incoming weight. Based on this comparison, the algorithm either

[2]This threshold for viable heavy-hitter candidates can be adjusted. For example, setting it to 0 disables the early cuckoo kickout termination.

Algorithm 2: Cuckoo Heavy Keeper - Helper Functions

```
1  Procedure GenerateFpAndIndexes($e$)
2      $fp \leftarrow fingerprint(e)$, $idx_0 \leftarrow hash(e)$ mod $\mathcal{B}$
3      $idx_1 \leftarrow (hash(fp) \oplus idx_0)$ mod $\mathcal{B}$
4      return $fp, idx_0, idx_1$
5  Procedure CheckAndUpdateHeavy($fp, idx_0, idx_1, w$)
6      Search heavy entries in both tables for matching $fp$
7      if found matching entry or empty slot exists then
8          Update counter or insert entry return true
9      return false
10 Procedure CheckAndUpdateLobby($fp, idx_0, idx_1, w$)
11     Search lobby entries in both tables for matching $fp$
12     if found then
13         Update lobby counter
14         if counter $\geq L$ then
15             TryPromote($T[i][idx_i]$)
16         return true
17     return false
18 Procedure DecayCounter($C$, $w$)
19     // Decay with exponential probability
20     if $w = 1$ then
21         $prob \leftarrow b^{-C}$
22         return $(Random(0, 1) < prob)\ ?\ C - 1 : C$
       // Weighted update
23     if $w > 1$ and $w < min\_decay$ then
24         $min\_decay \leftarrow de[C] - de[C - 1]$
25         $prob \leftarrow w/min\_decay$
26         return $(Random(0, 1) < prob)\ ?\ C - 1 : C$
27     if $w \geq de[C]$ then
28         return 0
29     return largest $i$ where $de[i] + w \geq de[C]$
30 Procedure TryPromote($bucket$)
31     if empty slot exists in $bucket.heavy$ then
32         Move $bucket.lobby$ to empty slot return
33     $min \leftarrow$ smallest entry in $bucket.heavy$
34     if $Random(0, 1) < \frac{bucket.lobby.C - L}{min.C - L}$ then
35         $evicted\_item \leftarrow min$
36         $min.fp \leftarrow bucket.lobby.fp$
37         Clear $bucket.lobby$ and Kickout($evicted\_item$)
38     else
39         $bucket.lobby.C \leftarrow L$
40 Procedure Kickout($entry$)
41     for $kicks \leftarrow 1$ to MAX_KICKS do
42         if $entry.C < \phi N$ then
43             return
44         if empty heavy entry in alt bucket then
45             Move $entry$ to empty heavy entry return
46         else
47             Swap $entry$ with min entry in alt bucket
```

replaces the lobby item when the weight can fully decay the counter (with promotion if the remaining weight exceeds L), or partially decrements the existing counter based on statistical projection. This method ensures that those with large weights can quickly establish themselves as heavy-hitter candidates. The analysis of weighted update behavior is presented in §4.4. For timeliness and practical efficiency, DecayCounter, in this context can be realized through tabulation, as it applies to bounded counting, making this path be very fast as well. This optimization is detailed in §4.5.

f-Query(e): (Alg. 1, f-Query function) When querying the estimated frequency $\hat{f}(e)$ for a specific item e, the algorithm first computes its fingerprint fp and bucket indices idx_0, idx_1 as in update operations. Then, it checks all heavy entries in the corresponding buckets for a matching fingerprint. If found, the counter value is returned; otherwise, the item is not tracked, and the query returns 0.

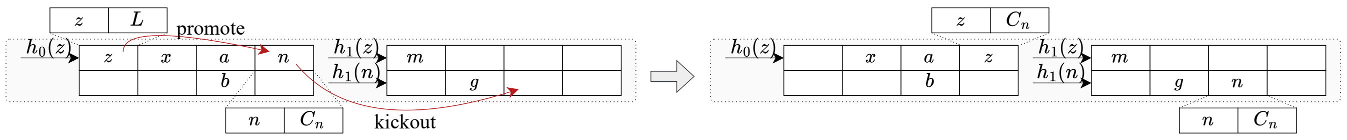

Figure 2: lobby item z replaces the minimum-counter item n in the heavy part, which then triggers relocation via cuckoo hashing. The process may continue recursively until finding an empty slot, or reaching `MAX_KICKS`, or encountering a below-threshold item.

hh-Query(): The implementation can vary by the specific use case (e.g., offline or streaming environment, if queries are continuous). For this reason, we omit the pseudo-code. However, for completeness, we describe a common approach using an auxiliary min-heap data structure that maintains the set of heavy hitters $\hat{R}$ continuously during stream processing. During `Update`, if $\hat{f}(e) \geq \phi N$, the item is pushed or updated in the heap. The algorithm then repeatedly removes the root item if its frequency falls below ϕN until all remaining items exceed the threshold. When querying heavy hitters, all items in the heap are returned with their frequencies.

4.4 Weighted update

Most (ϵ, ϕ)-heavy hitters algorithms focus on unweighted updates ($w = 1$). However, there can be a significant benefit in the ability to handle weighted updates ($w > 1$), e.g., process aggregated data from upstream tasks in distributed processing pipelines, which can substantially reduce communication costs between system components. When handling weighted updates, many algorithms perform multiple unweighted updates, which can degrade performance or produce incorrect results, especially in concurrent settings (cf. §6) where thread interference may occur during repeated updates. To address this limitation, we develop a weighted update extension for the *count-with-exponential-decay* that we incorporate into CHK. Note that this extension can be directly adopted by other systems using similar counting techniques (e.g., Redis Top-k [34]).

Let $dc_{C,w}$ denote the counter value after applying w consecutive decay operations to an initial counter value C. Each single decay operation ($w = 1$) sets $C = C - 1$ with probability b^{-C} or keeps it unchanged with probability $1 - b^{-C}$, where b is the decay base. For weighted updates, we need to determine $E[dc_{C,w}]$, the expected counter value after w decay operations.

Theorem 4.1. *For a counter value C with decay base b, where each decay operation sets the value $C = C - 1$ with probability b^{-C}, the expected counter value $E[dc_{C,w}]$ after w decay operations is:*

$$E[dc_{C,w}] = \log_b \left(b^C - \frac{w(b-1)}{b} \right) \tag{1}$$

Proof. Let $de[k]$ represent the expected number of decay operations needed to decrease a counter from k to 0. Since each decay has probability b^{-C}, one successful decay requires an expected b^C attempts. For $C > 0$, using geometric series sum: $de[C] = \sum_{k=1}^{C} b^k = \frac{b(b^C-1)}{b-1}$. Since each decay operation reduces the expected attempts by 1, after w operations: $de[E[dc_{C,w}]] = de[C] - w$. Substituting into both sides and simplifying, we get the result. □

4.5 Optimizations

Precomputed Decay Outcomes for Weighted Updates. Computing $E[dc_{C,w}]$ (cf. §4.4) for each update involves expensive floating-point operations. Recall that our design restricts these calculations to lobby counters with bounded values (L). This enables the possibility of pre-calculation and tabulation to replace expensive operations with simple lookups. We can precompute in the $de[]$ array the expected number of decay operations needed to reduce a counter from k to 0: $de[0] = 0$, and for $k = 1$ to L, $de[k] = de[k-1] + b^k$. When performing an update with weight w, if $w \geq de[C]$, that means the incoming weight is larger than the expected number of decay operations needed to reduce the existing counter C to 0. In this case, the incoming item's fingerprint fp replaces the old one, the remaining weight is calculated as $w' = w - de[C]$, and `TryPromote` will be triggered if $w' > L$. Otherwise, binary search can be used to find the expected value after w decay operations, thus updating C with C_{new}. The pseudo-code for weighted and unweighted updates is highlighted in the `DecayCounter` function (Alg. 2). This optimization trades $O(L)$ memory for $O(\log L)$ lookup time, eliminating all floating-point operations during updates while preserving the same guarantees[3].

Early heavy-entry placement. During the initial stages with empty heavy entries, forcing items through the lobby reduces accuracy due to exponential decay counting. Since heavy hitters often appear early in streams, we introduce early heavy entry placement. When an item arrives and finds an empty heavy entry, it is placed there directly. This allows precise counting of heavy hitters early on.

5 ANALYSIS OF CUCKOO HEAVY KEEPER

Lemma 5.1 (Heavy Hitter Promotion Guarantee). *Let $\mathcal{B}$ be the number of buckets in the hash table, N be the stream size, and e be any true heavy hitter $R = \{e \mid f(e) \geq \phi N\}$. Assuming no fingerprint collisions occur and the heavy part has sufficient space to accommodate promoted items without losing true heavy hitters during relocations, the probability that e will be promoted to the heavy part is bounded as:*

$$\Pr[e \textit{ is promoted to heavy part} \mid f(e) \geq \phi N] \geq 1 - \frac{1}{\phi \mathcal{B}}$$

Proof. For a heavy hitter e with frequency $f(e) \geq \phi N$, we analyze the worst-case scenario: when e arrives, its lobby position is already occupied by another item with a counter value at L. Due to the exponential decay with base b, approximately b^L collisions are required to cause a single decrement. In practice, e only collides with a subset of items hashed to the same bucket, but for our analysis, we conservatively assume it must collide with all other items hashed there. If $f(e)$ exceeds this worst-case collision threshold, then e will eventually be promoted.

[3]Our microbenchmarks [31] of the weighted upate, following **Recommended Parameter Configuration** in §7 ($L = 16$, $b = 1.08$) show that tabulation is 168× faster than repeated unweighted updates and 2.6× faster than theorem 4.1 formula calculation.

Let an indicator variable $I_{e',j} = 1$ if $h(e') = j$ and 0 otherwise, where h maps items to $\mathcal{B}$ buckets. Define $Y_j = \sum_{e' \in U} I_{e',j} \cdot f(e')$, which represents the total frequency of all items hashed to bucket j. Then: $E[Y_j] = \sum_{e' \in U} \frac{f(e')}{\mathcal{B}} = \frac{N}{\mathcal{B}}$. For e to eventually be promoted, the key inequality is: $f(e) \geq Z \cdot \left(Y_{h(e)} - f(e)\right) + L$ where Z represents the average number of collisions needed to cause a single decrement. Since $f(e) \gg L$, we can rearrange to get $(Z+1)f(e) \geq Z \cdot Y_{h(e)} + L$, which means promotion occurs when $Y_{h(e)} \lesssim \frac{Z+1}{Z} f(e)$.

Therefore, the probability that e is not promoted is bounded by Markov's inequality using $E[Y_{h(e)}] = \frac{N}{\mathcal{B}}$:

$$\Pr\left[Y_{h(e)} \geq \frac{1+Z}{Z} f(e)\right] \leq \frac{\frac{N}{\mathcal{B}}}{\frac{1+Z}{Z} \cdot \phi N} = \frac{1}{\mathcal{B}} \cdot \frac{Z}{(1+Z) \cdot \phi} \leq \frac{1}{\phi \mathcal{B}}$$

where the last inequality follows from $\frac{Z}{1+Z} < 1$ for any $Z > 0$. Thus, $\Pr[e \text{ is promoted to heavy part} \mid f(e) \geq \phi N] \geq 1 - \frac{1}{\phi \mathcal{B}}$. □

Theorem 5.2 (Approximation Bounds). *Let $\mathcal{B}$ be the number of buckets in the hash table, N be the stream size, and e be any item in the heavy part (including heavy hitters that appear there with high probability as shown in lemma 5.1). Assuming no fingerprint collisions occur and the heavy part has sufficient space to accommodate promoted items without losing true heavy hitters during relocations, for any positive $\epsilon < 1$, the probability of estimation error exceeding ϵN is bounded as:*

$$\Pr[|\hat{f}(e) - f(e)| \geq \epsilon N] \leq \frac{1}{\epsilon \mathcal{B}}$$

Proof. The true frequency $f(e)$ can be decomposed as $f(e) = f^d(e) + f^p(e) + f^h(e)$, where $f^d(e)$ counts occurrences during the *count-with-exponential-decay phase*, $f^p(e)$ counts occurrences during probabilistic promotion, and $f^h(e)$ counts occurrences after entering the heavy part. The algorithm estimates $\hat{f}(e) = f^d(e) - X^d(e) + m + f^h(e)$, where $X^d(e)$ represents decrements during the decay phase and m is the frequency added upon promotion.

For the underestimation bound:

$$\begin{aligned} & \Pr[\hat{f}(e) \leq f(e) - \epsilon N] = \Pr[-X^d(e) + m \leq f^p(e) - \epsilon N] \\ = \quad & \Pr[X^d(e) + f^p(e) \geq \epsilon N + m] \leq Pr[X^d(e) + f^p(e) \geq \epsilon N] \\ \leq \quad & \frac{E[X^d(e) + f^p(e)]}{\epsilon N} \quad \text{(by Markov's inequality)} \qquad (2) \end{aligned}$$

Using $Y_{h(e)}$ as defined above, since $X^d(e) + f^p(e) \leq f^d(e) + f^p(e) = f(e) - f^h(e) \leq f(e) \leq Y_{h(e)}$, we have: $E[X^d(e) + f^p(e)] \leq E[Y_{h(e)}] = \frac{N}{\mathcal{B}}$. Substituting into (2):

$$\Pr[\hat{f}(e) \leq f(e) - \epsilon N] \leq \frac{1}{\epsilon \mathcal{B}} \qquad (3)$$

For the overestimation bound:

$$\begin{aligned} & \Pr[\hat{f}(e) \geq f(e) + \epsilon N] = \Pr[-X^d(e) + m - f^p(e) \geq \epsilon N] \\ \leq \quad & \Pr[m - f^p(e) \geq \epsilon N] \leq \Pr[m \geq \epsilon N] \leq \Pr[Y_{h(e)} \geq \epsilon N] \\ \leq \quad & \frac{E[Y_{h(e)}]}{\epsilon N} = \frac{N/\mathcal{B}}{\epsilon N} = \frac{1}{\epsilon \mathcal{B}} \quad \text{(by Markov's inequality)} \qquad (4) \end{aligned}$$

By the union bound on (3) and (4), the theorem follows. □

From this theorem, we can establish that CHK satisfies the probabilistic heavy-hitter conditions (**C1-C4**) (cf. §2) with appropriate parameterization. A key assumption for our algorithm—that "the heavy part has sufficient space to accommodate promoted items"—represents, in fact, a significant constraint for other algorithms in practice (cf. §7.1.1 for more detail). It is also worth noting that our analysis's conservative bounds do not account for all beneficial aspects of the *Cuckoo Heavy Keeper* algorithm, such as cuckoo hashing, which creates alternative locations for items that might otherwise be lost in traditional approaches. Indeed, in our empirical evaluation (§7), we demonstrate that CHK's performance consistently exceeds these theoretical guarantees.

6 CONCURRENT OPERATIONS

As query operations need to execute while insertions are happening, the problem of concurrent queries and insertions poses challenging questions regarding the associated synchronization. As outlined in the introduction, there is growing interest in the respective issues. In this section, we outline the main results along with the algorithmic design space in synchronization of parallel operations, considering multi-threaded, shared-memory systems.

6.1 Parallel designs and trade-offs

Table 3: Comparison of Parallel Design Categories

Design	**Tentative scalability**	**Tentative f-query rate**	**Tentative hh-query rate**
Single-shared	Low	High	High
Thread-local	High	-	-
Peer-collaborative	High	Medium/High	Low/Medium
Global-collaborative	Medium/High	-	High
Hybrid Peer-Global	Medium/High	Medium/High	High

For parallel processing in shared memory systems, several main design options exist, with properties as summarized in the paragraphs below and sketched in Table 3.

6.1.1 Single-shared. Such a design features a single shared-memory data structure, accessible by all threads for insertions and queries. Insert operations require synchronization mechanisms like locks or atomic operations or helping mechanisms, as in *COTS* [12]. In highly parallel environments with high-rate input streams, this design poses challenges regarding scaling with the number of threads. However, queries only need to access a single data structure, potentially leading to faster responses [4].

6.1.2 Thread-local. The thread-local design assigns each thread its own local data structure. Threads insert items directly into their respective structures without synchronization, potentially facilitating scalability regarding insertions. However, this approach requires querying all thread-local data structures to collect heavy hitters, which can be inefficient for high-performance scenarios. For example, *Topkapi* [26] implements this approach but does not support concurrent insertions/queries, instead only allowing querying at the end and requiring mergeability [2] (ability to combine multiple sketches into one without losing accuracy), highlighting the challenge of efficiently aggregating results from multiple sources.

[4]Note that this depends on the synchronization method. For example, if a reader-writer lock with priority to the writers is employed, a query can starve.

6.1.3 *Peer-collaborative.*

The peer-collaborative design [39] enhances the thread-local approach, through *domain-splitting* and *delegated operations*. Each thread is responsible for a subset of items from the universe — a concept known as *domain-splitting*. If a thread receives operations associated with another thread's domain, it buffers them and delegates these operations accordingly. This method, employed by *QPOPSS* [23], maintains good scalability even with concurrent insertions and f-queries. However, hh-query consistency is relaxed, potentially introducing bounded staleness. Additionally, hh-queries still require scanning all thread-local data structures, which can introduce higher hh-query latency when the number of threads increases or when hh-queries are frequent.

6.1.4 *Global-collaborative.*

The global-collaborative design combines elements of the single-shared and thread-local approaches. Each thread periodically flushes its local heavy hitters into a single-shared data structure. This allows hh-queries to be answered quickly by accessing one location. Synchronization is less costly compared to the continuous synchronization required in the single-shared approach, since it is not needed at every update. An example of this design is *PRIF* [42], although it is noteworthy that *PRIF* permits only one dedicated thread for hh-queries and does not support f-queries, highlighting the challenges of efficiently handling both types of queries in a global-collaborative setup.

6.2 Parallel Cuckoo Heavy Keeper

Table 4: Additional Notation for Parallel Versions

Notation	Description
P	Number of parallel threads
tid	Thread identifier (0 to $P-1$)
$ctid, otid$	Current thread ID and owner thread ID for incoming item e
MAX_BUF	Maximum buffer size before flush
MAX_W	Maximum allowed weight for buffered items
$owner(e)$	Thread assignment function $hash(e)$ mod P
CHK_{tid}	Thread tid's CHK instance
$B_{tid}[j]$	Buffer of $\langle e, w\rangle$ pairs from thread tid to thread j
Q_{tid}	Queue of buffer references for thread tid
$PQ_{tid}[j]$	f-Query slot $\langle e, count, flag\rangle$ from thread j to tid
HH	Global concurrent hash table of heavy hitters
$N_{processed}$	Global atomic processed stream size counter

Studying the aforementioned parallel designs, we note multi-faceted trade-offs in terms of parallel scalability and the associated potential for f-query and hh-query efficiency. Hence, we seek to balance and improve upon these aspects. To this end, we propose a general algorithmic framework for parallelizing heavy-hitter detection, with two specific designs tailored to different contexts:

- *mCHK-I*, based on the peer-collaborative design and optimized for situations where insertions and f-queries are predominant.
- *mCHK-Q*, a hybrid peer-global collaborative approach, combining elements of both peer-collaborative and global-collaborative designs, suited for scenarios where hh-queries are more frequent.

Although we use *Cuckoo Heavy Keeper* as the underlying algorithm in the description, our parallel designs operate as a *wrapper*, compatible with any heavy-hitter algorithm. Those with native weighted update support are directly applicable, yet any other algorithm can be used by falling back to repeated unweighted updates.

We now describe first how the two algorithms perform insertion-delegations and f-queries, extending [39]. We then explore the differences between them when it comes to hh-queries. In Section 7, we evaluate them both regarding scalability while supporting concurrent insertions, f-queries, and hh-queries, discussing the balancing properties regarding the aforementioned trade-offs.

Algorithm 3: Parallel Cuckoo Heavy Keeper Wrapper

```
1  Procedure Update($e, w$)
2      $ctid, otid$ ← current thread ID, owner(e) // Domain splitting
3      Add $\langle e, w\rangle$ to $B_{ctid}[otid]$ // Buffer for delegated processing
4      if $|B_{ctid}[otid]| \geq MAX\_BUF$ or $B_{ctid}[otid][e] \geq MAX\_W$ then
5          Add reference of $B_{ctid}[otid]$ to $Q_{otid}$ // Delegate to otid
6          while $B_{ctid}[otid]$ not processed do
7              ProcessPendingUpdates // Process work while waiting
8  Procedure f-Query($e$)
9      $ctid$ ← current thread ID
10     $otid \leftarrow owner(e)$
11     Store f-query $e$ in slot $PQ_{otid}[ctid]$ // Delegate query to otid
12     while f-query in $PQ_{otid}[ctid]$ not processed do
13         ProcessPendingUpdates // Process work while waiting
14         ProcessPendingf-Queries()
15     return result from $PQ_{otid}[ctid]$
16 Procedure ProcessPendingf-Queries
17     $ctid$ ← current thread ID
18     for $tid \leftarrow 0$ to $P-1$ do
19         if $PQ_{ctid}[tid]$ has unprocessed f-query $e$ then
20             $count \leftarrow CHK_{ctid}$.f-Query($e$) // Process query
21             Store $count$ as result in $PQ_{ctid}[tid]$
22             Mark f-query in $PQ_{ctid}[tid]$ as processed
```

6.2.1 *Insertions and f-queries.*

Consider the notation in Table 4. Note that subscripts (e.g., CHK_{tid}, B_{tid}, PQ_{tid}) indicate thread-local data structures that are independently allocated, which prevents false sharing. MAX_BUF should limit buffer size to e.g., fit within one cache line for efficient transfers between threads, while MAX_W caps the maximum weight per buffered item to balance accuracy (cf. Theorem 6.1). Insertions and f-query are implemented as follows:

Insertion (Alg. 3, function `Update`): Insertions are delegated when thread $ctid$ receives items e owned by thread $otid$. Instead of immediate delegating, items are buffered in $B_{ctid}[otid]$. When the buffer size $|B_{ctid}[otid]| \geq MAX\_BUF$ or the buffer per item $B_{ctid}[otid][e] \geq MAX\_W$, thread $ctid$ adds a reference to the buffer into Q_{otid}. Here, Q_{tid} is a lock-free queue (e.g., LCRQ [30]) that stores references to buffers needing processing by thread tid. Delegation operations require the underlying heavy-hitter data structure to be able to handle *weighted updates*, which is supported by the *Cuckoo Heavy Keeper* algorithm (cf. §4.4). When the buffer $B_{ctid}[otid]$ is processed by `ProcessPendingUpdates`, the thread $otid$ updates its local CHK_{ctid} and increments the global atomic stream size counter $N_{processed}$ by w. Note that the `ProcessPendingUpdates` implementation differs between the two parallel designs: in *mCHK-Q* (Alg.5), it additionally identifies items that exceed the heavy-hitter threshold and adds them to a global hash table HH, whereas *mCHK-I* (Alg.4) only performs the local updates.

f-query (Alg. 3, function `f-Query`): Similar to updates, f-queries from thread $ctid$ to $otid$ are also delegated through pending f-query slots ($PQ_{otid}[ctid]$). Each slot stores a tuple $\langle e, count, flag\rangle$ where e is the queried item, $count$ stores the result, and $flag$ indicates processing status. When thread $ctid$ needs to f-query an item e owned by thread $otid$, it initializes a slot with $\langle e, 0, unprocessed\rangle$ in $PQ_{otid}[ctid]$. The querying thread $ctid$ monitors the $flag$ in its assigned slot until it changes to *processed*. At the same time, instead of waiting idly, the querying thread *ctid* continuously processes its own pending updates and queries. Meanwhile, delegated thread *otid* processes the f-query by querying the frequency from its local CHK_{otid}. Once processed, *otid* updates the slot with the final count and marks the *flag* as *processed*. This design allows continuous f-querying without blocking or freezing thread execution.

Algorithm 4: mCHK-I operations

```
Procedure ProcessPendingUpdates
    $ctid$ ← current thread ID
    if cannot acquire lock on $ctid$'s data then
        return
    while $Q_{ctid}$ has unprocessed references do
        $B_{ref}$ ← get next unprocessed reference from $Q_{ctid}$
        foreach $\langle e, w\rangle \in B_{ref}$ do
            $CHK_{ctid}$.Update($e, w$) // Delegated updates
            Atomic add $w$ to $N_{processed}$ // Track stream size
        Mark $B_{ref}$ as processed in $Q_{ctid}$
    release lock on thread $ctid$'s data
Procedure hh-Query
    $\hat{R}, remaining, made\_progress \leftarrow \emptyset, P, false$
    while $remaining > 0$ do
        $made\_progress \leftarrow false$
        for $tid \leftarrow 0$ to $P - 1$ do
            if $tid$'s data is not scanned and can acquire lock then
                $cand \leftarrow CHK_{tid}$.hh-Query()
                foreach $\langle e, count\rangle \in cand$ do
                    if $count \geq \phi N_{processed}$ then
                        Add $\langle e, count\rangle$ to $\hat{R}$
                Mark thread $tid$'s data as scanned and release lock
                $remaining, made\_progress \leftarrow remaining - 1, true$
        if $remaining > 0$ and not $made\_progress$ then
            ProcessPendingUpdates
    return $\hat{R}$
```

Algorithm 5: mCHK-Q operations

```
Procedure ProcessPendingUpdates
    $ctid$ ← current thread ID
    while $Q_{ctid}$ has unprocessed references do
        $B_{ref}$ ← get next unprocessed reference from $Q_{ctid}$
        foreach $\langle e, w\rangle \in B_{ref}$ do
            $count \leftarrow CHK_{ctid}$.Update($e, w$) // Delegated updates
            Atomic add $w$ to $N_{processed}$ // Track stream size
            if $count \geq \phi N_{processed}$ then
                Update $\langle e, count\rangle$ in $HH$ // Maintain global HH
        Mark $B_{ref}$ as processed
Procedure hh-Query
    $\hat{R} \leftarrow \emptyset$
    // HH: Global concurrent hash table of heavy hitters
    foreach entry position $i$ in $HH$ do
        repeat
            $\langle e_1, count_1\rangle \leftarrow HH[i]$ // First read
            $\langle e_2, count_2\rangle \leftarrow HH[i]$ // Second read for consistency
        until $e_1 = e_2$ and $count_1 = count_2$ // double collecting
        if $count_1 \geq \phi N_{processed}$ then
            Add $\langle e_1, count_1\rangle$ to $\hat{R}$
    return $\hat{R}$
```

6.2.2 *hh-queries.*

mCHK-I (Alg. 4): When a hh-query is executed, the algorithm performs a non-blocking scan across threads' local CHK_{tid} structures to collect items whose counts exceed the threshold ($count \geq \phi N_{processed}$) into the result set $\hat{R}$. For thread safety, mCHK-I uses opportunistic thread-level locking — if a thread's lock cannot be acquired immediately, the algorithm continues scanning other threads and processes pending updates, before retrying locked threads later. As this is a low-contention locking when the hh-query rate is not too high, it potentially does not cause a high number of retries.

mCHK-Q (Alg. 5): It improves hh-query latency and overall throughput under frequent hh-queries by maintaining a global concurrent hash table HH (e.g., libcuckoo [25]) for heavy hitters. When thread *ctid* processes updates (Alg. 5, `ProcessPendingUpdates`), items exceeding the threshold ($count \geq \phi N_{processed}$) are added to HH. Although this introduces synchronization overhead, the cost is amortized over multiple updates due to buffering. When the hh-query is executed, the algorithm performs a non-blocking scan of HH to collect items whose counts exceed the threshold ($count \geq \phi N_{processed}$), using double-collecting [1] (Alg. 5, l. 13-17, repeatedly collecting the data twice until values match) to avoid torn reads and adding them to the result set $\hat{R}$.

6.3 Accuracy of hh-queries

Consider the global state of the algorithm at a point in time, applicable to both *mCHK-I* and *mCHK-Q*. Let N_s be the total weighted size of the processed data stream at the start of the query, and $\mathbb{B}(e)$ be the total buffered weight of item e i.e., the weight of e that has not yet been processed by the algorithm ($\mathbb{B}(e) \leq P \times MAX\_W$, since each thread can buffer at most MAX_W weight for any item).

Theorem 6.1 (Parallel Approximation Bound). *Let e be any item in the heavy part, $f_{N_s}(e)$ be the true frequency of e up to N_s; the estimated frequency $\hat{f}(e)$ from hh-query satisfies:*

$$\Pr\left[|\hat{f}(e) - (f_{N_s}(e) - \mathbb{B}(e))| \geq \epsilon N_s \right] \leq \frac{1}{\epsilon \mathcal{B}}$$

Proof. At query time, the underlying sequential *Cuckoo Heavy Keeper* has processed $f_{N_s}(e) - \mathbb{B}(e)$ weight for item e. Applying theorem 5.2 with this adjusted frequency yields the result. This implies an important practical trade-off: smaller MAX_W and MAX_BUF values reduce buffering delay (improving accuracy) but increase synchronization frequency (reducing throughput). □

7 EVALUATION

This section presents a comprehensive evaluation of the contributed methods, organized in two parts: (1) evaluation of *CHK* on both real-world and synthetic data, compared to state-of-the-art algorithms, focusing on accuracy and throughput, with varying memory constraints and data distributions (skewness); (2) evaluation of our parallel designs on varying hardware features, studying scalability with thread counts and hh-query rates; the study includes both *CHK* and alternative underlying heavy-hitter detection algorithms. We begin by describing our experiment setup, including hardware specifications, datasets, and evaluation metrics.

Platforms: We conducted experiments on two hardware platforms. Table 5 provides the specifications:

Table 5: Hardware Platform Specifications

	Platform A	Platform B
Processor	Intel(R) Xeon(R) E5-2695 v4 (NUMA dual sockets)	AMD EPYC 9754 (UMA single socket)
Cores/Clock	36 cores/2.1 GHz	128 cores/2.25 GHz
Hyper Threading	Enabled	Disabled
Cache (L1/L2/L3)	32KB/256KB/45MB	32KB/1024KB/256MB
Used for	All experiments	Parallel experiments only

Data sets: We used three datasets: (1) *CAIDA_L*, a real-world dataset of source IPs (skewness ∼ 1.2); (2) *CAIDA_H*, a real-world dataset of source ports (skewness ∼ 1.5). Both are derived from the CAIDA Anonymized Internet Traces 2018 [6], a broadly used benchmark in heavy-hitter detection literature [20, 38, 41]. From these traces, we extracted source IPs and source ports from the first 10M packets, representing network traffic monitoring scenarios faced in real-world environments with different skewness. (3) *Synthetic data* containing 10M items generated using Zipf distributions with skewness α ranging from 0.8 to 1.6, commonly used to model real-world frequency distributions [3, 39].

Baselines: We compared *CHK* against representative state-of-the-art heavy-hitter detection algorithms: *Space-Saving* (SS) [28], *Count-MinSketch* (CMS) [11], *AugmentedSketch* (AS) [37] and *HeavyKeeper* (HK) [20], all implemented in C++, compiled with -O2 and available in our repository [31]. CHK uses parameters from **Recommended Parameter Configuration** (cf. §7.1.1 for experiment setup), while others follow their prescribed recommendations. All algorithms use auxiliary heaps for continuous heavy-hitter tracking (cf. §4.3), as needed in real-world monitoring scenarios.

Recommended Parameter Configuration

Bucket Configuration: Fig. 3 shows that 2 heavy entries per bucket achieve better accuracy, which aligns with prior research on cuckoo hashing [14, 16, 18] showing 2-4 entries maximizes space efficiency. We use a promotion threshold $L = 16$ based on sensitivity analysis in Fig. 3. For counter sizes, we use 8-bit counters in the *lobby part* (sufficient for L) and 32-bit counters in the *heavy part* (to track frequencies of heavy items). This also ensures each bucket fits within a CPU cache line for optimal memory access performance. **Other Parameters:** 16-bit fingerprints following [16] and decay factor $b = 1.08$ following [20].

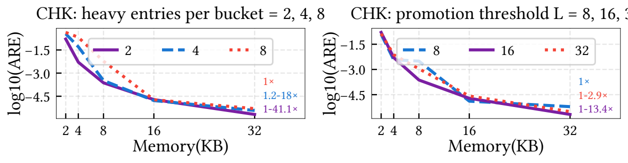

Figure 3: Sensitivity to decay base and heavy entries per bucket

7.1 Study of the sequential algorithms

7.1.1 Measurement Methodology. We use four metrics: precision, recall, ARE, and throughput as defined in §2. We vary different parameters depending on the dataset type. With synthetic data, we vary ϕ, memory usage, and skewness α, modifying one at a time. For the CAIDA datasets, where skewness is an inherent characteristic, we vary only ϕ and memory usage. When parameters are not varying, their values were set to: threshold ϕ of 0.0005, memory usage of 4KB, and skewness of 1.2, with the following reasoning: the skewness is a value in the range of real-world network traffic distribution [6], without favoring algorithms optimized for either high or low skew; ϕ corresponds to a rather low threshold, i.e., with low selectivity, matching cases of many heavy hitters, while memory is set to be just adequate to keep them. This enables evaluating the algorithms under limited memory rather than simply testing performance under abundant resources. Besides, memory size matters when the information is to be communicated (cf. e.g. [21]), so the smaller the sketch, the better. Each experiment was run 30 times, with average performance calculated and plotted. To enable easier comparisons, we also reported *relative improvements* rather than just plain absolute values. For each configuration, we calculated each algorithm's improvement normalized over the least-performing one and presented these numbers in sorted order, based on the average of point-wise improvement ratio.

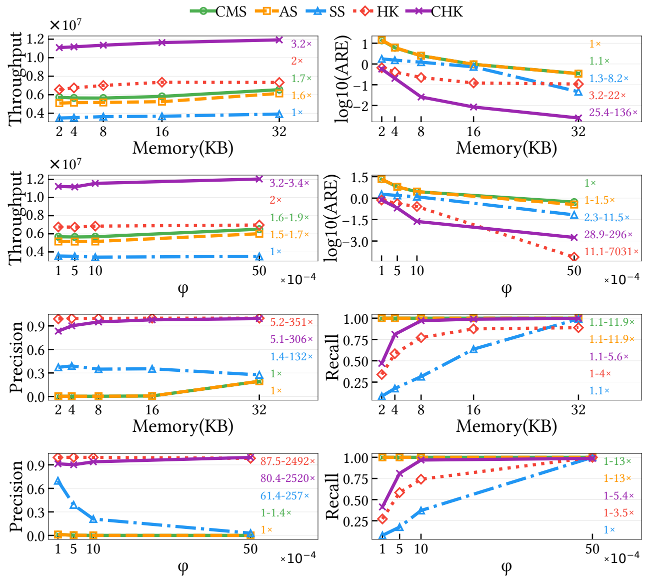

Figure 4: Plots show performance of sequential algorithms on *CAIDA_L* dataset for throughput, log_{10}(ARE), precision, and recall across varying memory, and ϕ. See §7.1.1 for calculation details.

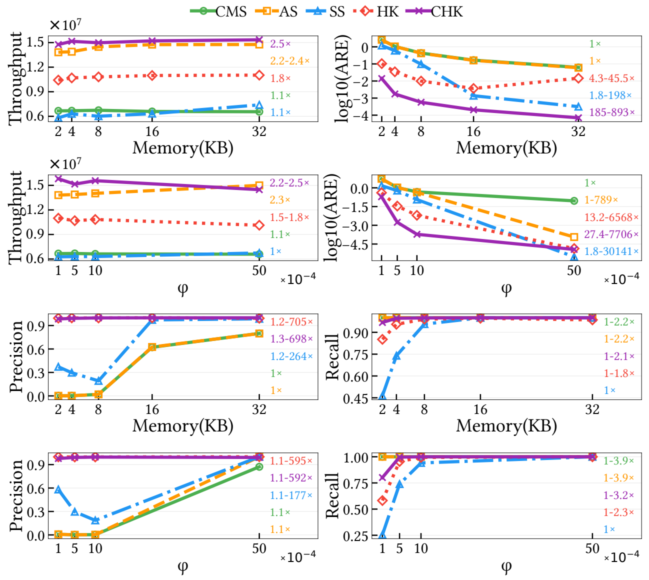

Figure 5: Plots show performance of sequential algorithms on *CAIDA_H* dataset for throughput, log_{10}(ARE), precision, and recall across varying memory, and ϕ. See §7.1.1 for calculation details.

7.1.2 Experiments on throughput. In streaming heavy-hitter algorithms, throughput is influenced by configuration parameters and dataset characteristics. Limited memory increases collisions, requiring more hash resolution operations; low skewness leads to more items competing for slots; and low thresholds classify more items as heavy hitters, increasing heap maintenance overhead. These factors reduce throughput by triggering more expensive algorithm paths. Our results across synthetic and real-world datasets (Fig. 4, Fig. 5, and Fig. 6) confirm these patterns and show the benefits of the CHK design, which is summarized in Key Takeaway 1.

Key Takeaway 1 – on sequential CHK throughput

CHK consistently delivers superior throughput across all tested configurations, by 2-3 times in most settings. While some methods, e.g., AS, demonstrate occasional performance spikes under high skew conditions (when few items dominate the traffic), they still underperform compared to CHK slightly, even in these favorable scenarios, and degrade substantially under low skew distributions. The improvements are possible through CHK's inverted filter (lobby) process and its enabled system-aware layout, where data movement is limited (and often within the same cache line), and hash collision resolution is applied selectively only to heavy-hitter candidates. As a result, CHK generalizes across varied workloads and offers more predictable performance in diverse scenarios, where stream characteristics cannot be known in advance and memory requirements are difficult to determine ahead of time.

7.1.3 Experiments on accuracy. Experiments on accuracy (precision, recall, and ARE of Fig. 4, Fig. 5, and Fig. 6) show varying behavior across different algorithms and configurations. In general, each algorithm shows similar improvement trends when varying parameters (for reasons similar to those discussed in §7.1.2), but the magnitude of improvement differs. At low memory, low skewness, or low threshold settings, algorithms exhibit distinct trade-offs: CMS/AS/SS achieve high recall but poor precision because they overestimate frequencies indiscriminately, which allows them to find most heavy hitters but results in many false positives. HK achieves high precision but lower recall as multiple heavy items can be mapped to the same bucket, causing collisions and omissions of true heavy hitters. CHK can balance, maintaining higher precision than CMS/AS/SS while delivering better recall than HK in most settings, as explained in Key Takeaway 2.

Key Takeaway 2 – on sequential CHK accuracy

Under constrained conditions (low memory usage or low skewness), CHK improves accuracy by a large margin. This is because CHK uses cuckoo collision resolution, which allows it to capture more heavy hitters, resulting in better recall, while maintaining a low false positive rate even under stringent memory constraints.

7.2 Study of the parallel algorithms

7.2.1 Measurement Methodology. We evaluated our parallel framework across multiple dimensions: hardware platforms (*Platform A* and *B*), underlying heavy-hitter detection algorithms, thread counts, and hh-query rates (ratio of hh-queries vs total operations performed by each thread), for insertion throughput and hh-query latency (§2). All experiments used the *CAIDA_H* dataset. Since both variants handle f-queries similarly via the *delegation mechanism*, we focus the comparison on their divergent approaches to hh-queries. To evaluate the framework's generality, we used various underlying algorithms that natively support weighted updates (CHK, AS, CMS, SS; HK was not included since it lacks this feature). The resulting

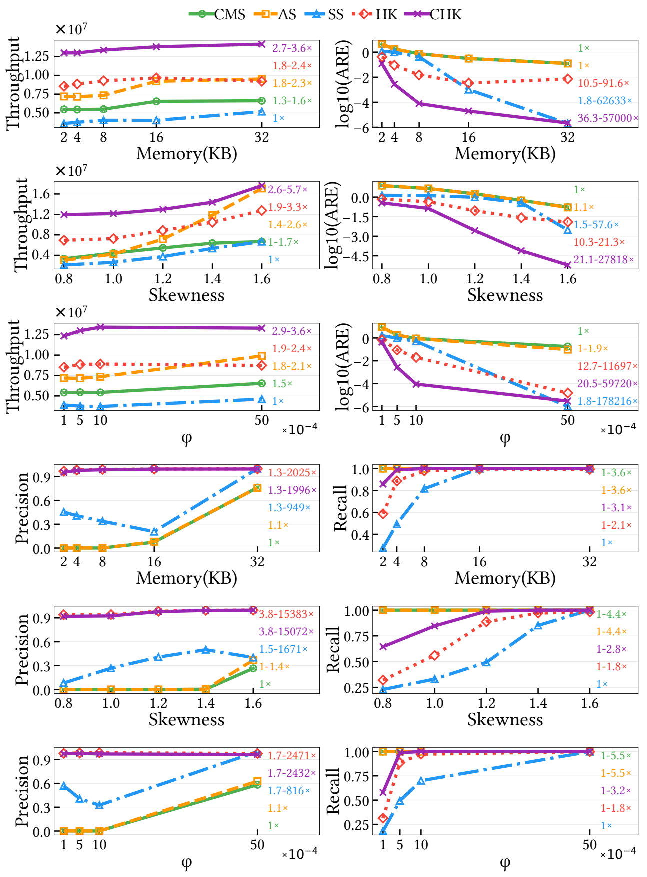

Figure 6: Plots show performance of sequential algorithms on *Synthetic data* for throughput, log_{10}(ARE), precision, and recall across varying skewness, memory, and ϕ. See §7.1.1 for calculation details.

parallel variants are denoted as *mCHK-I*, *mCHK-Q*, mAS-I, mAS-Q, mCMS-I, mCMS-Q, mSS-I, and mSS-Q, with -I and -Q indicating insertion- and query-optimization. With similar reasoning as in subsection 7.1.1, $\phi = 0.00005$ and memory = 1KB per thread, with algorithm parameters $MAX\_W = 1000$ and $MAX\_BUF = 16$, to simulate a resource-constrained environment. Each experiment ran 30 times, with results showing both average performance and relative speedup compared to single-thread implementations.

7.2.2 Experiments on throughput. Fig. 7 shows the average throughput of parallel variants with varying thread counts and hh-query rates across different underlying heavy-hitter detection algorithms. Both designs scale nearly linearly as thread count increases. When comparing the query-optimized (-Q) and insertion-optimized (-I) variants, we observe patterns consistent with the design trade-offs discussed in §6. The -I variants outperform their -Q counterparts in insertion-only workloads (0% query rate) due to their reduced synchronization overhead. However, as query rates increase, the -Q variants demonstrate significantly better performance. Notable results are summarized in Key Takeaway 3 and Key Takeaway 4.

7.2.3 Experiments on hh-query latency. Fig. 8 shows the hh-query latency of our parallel implementations with varying thread counts and hh-query rates across different underlying algorithms. The

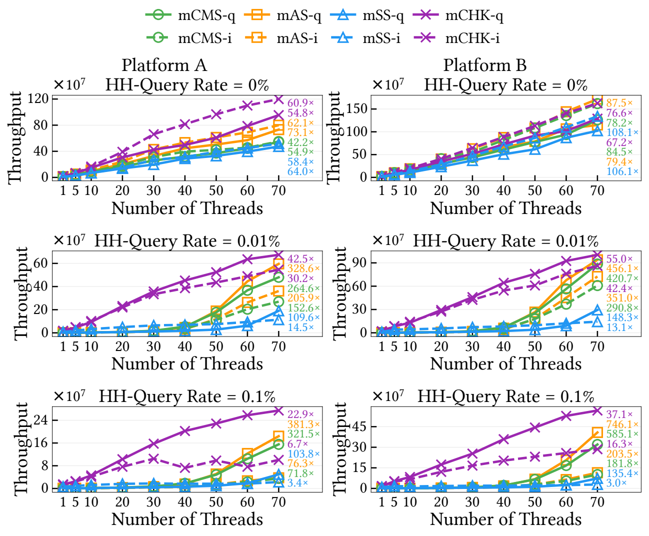

Figure 7: Plots show throughput and relative improvement compared to single-thread implementation of parallel variants on *CAIDA_H* across varying hh-query rates, thread counts, and platforms.

Key Takeaway 3 – on parallel throughput

Both *mCHK-I* and *mCHK-Q* consistently outperform other parallel variants in throughput by a substantial margin, especially at low to moderate thread counts. This difference arises from CHK's lean operations and accuracy improvements in heavy-hitter estimation, which together reduce the number of false positives that must be transferred during hh-queries. These results strengthen the claim that CHK and its parallel designs are adaptive to various workload conditions. Additionally, the performance gains at even lower thread counts suggests potential benefits in energy saving and elasticity possibilities, i.e. better resource utilization.

Key Takeaway 4 – on parallel framework portability

Our parallel framework can scale across diverse hardware, with differences in memory architecture (NUMA, UMA) and processor features (hyperthreading, non-hyperthreading). This portability extends even to cross-socket execution because of the delegated operations approach (also in [22, 23, 39]), enhanced by the weighted update capability, which allows threads to aggregate work and streamline it with synchronization, reducing communication overhead. Furthermore, in hyper-threading environments, where sibling threads must share execution resources, CHK outperforms other algorithms by enabling threads to complete more workload in the same amount of time.

results differ significantly between algorithms due to their query operation costs and overhead of transferring detected heavy hitters. When comparing -Q and -I variants, we observe patterns consistent with the design projections. The -I variants exhibit higher latency that increases with thread count due to the need to interact with each thread to access thread-local structures. In contrast, -Q variants typically maintain more stable latency, especially at higher thread counts, due to their centralized query architecture. However, an interesting exception occurs with CMS and AS variants under constrained memory conditions. For these algorithms, the -Q variants actually show higher latency than expected, as they must frequently synchronize large numbers of misclassified heavy hitters to the global structure, creating substantial overhead. The zoomed-in view of the results (with the y-axis scaled to match CHK's latency range) reveals that *mCHK-Q* shows very low and stable query latency compared to other algorithms. After reaching a certain thread count, *mCHK-Q* consistently maintains latency below 150 μsec even under high thread counts and query rates.

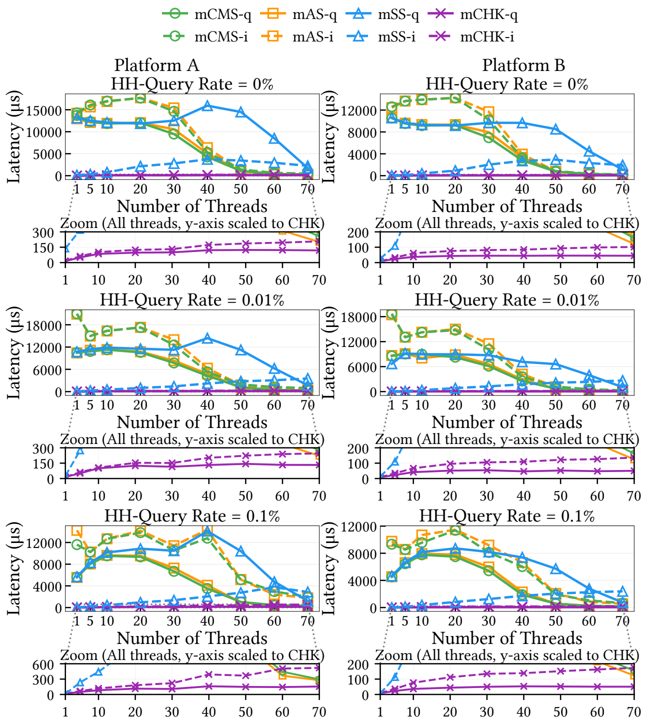

Figure 8: Plots show hh-query latency of parallel variants on *CAIDA_H* across varying hh-query rates, thread counts, platforms.

8 CONCLUSIONS AND FUTURE WORK

We introduced *Cuckoo Heavy Keeper* (CHK), a fast, accurate, and space-efficient algorithm that delivers orders of magnitude better throughput and accuracy compared to state-of-the-art methods, even with tight memory and low-skew data. For parallel scalability, we proposed *mCHK-I* and *mCHK-Q*, achieving near-linear scaling with low hh-query latencies. These parallel algorithms operate as a *wrapper* around any sequential heavy-hitter, without requiring mergeability, which enables modular integration into existing systems with minimal changes. This makes CHK and its parallel variants useful both as standalone algorithmic designs and as integrable building blocks within databases, stream processing engines, and data analytics frameworks. Future research directions include extending *Cuckoo Heavy Keeper* to support sliding windows with potential integration into established data processing systems.

ACKNOWLEDGMENTS

Work supported by Marie Skłodowska-Curie Doctoral Network RELAX-DN, funded by EU under Horizon Europe 2021-2027 FP Grant Agreement nr. 101072456 (www.relax-dn.eu/); Swedish Research Council prj."EPITOME" 2021-05424; prj TANDEM (Swedish Energy Agency SESBC, ref. nr. 2021-035871/IEM2022-08); Chalmers AoA Energy-INDEED & Production-"Scalability, Big Data and AI".

REFERENCES

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *J. ACM* 40, 4 (1993), 873–890. https://doi.org/10.1145/153724.153741

[2] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. 2012. Mergeable summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '12)*. Association for Computing Machinery, 23–34. https://doi.org/10.1145/2213556.2213562

[3] MEJ Newman and. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics* 46, 5 (2005), 323–351. https://doi.org/10.1080/00107510500052444

[4] Apache Druid. 2024. TopN Queries - Apache Druid Documentation. https://druid.apache.org/docs/latest/querying/topnquery/. Accessed: 2025.

[5] Kevin Beyer and Raghu Ramakrishnan. 1999. Bottom-up computation of sparse and Iceberg CUBE. *SIGMOD Rec.* 28, 2 (1999), 359–370. https://doi.org/10.1145/304181.304214

[6] CAIDA. 2018. The CAIDA Anonymized Internet Traces 2018 Dataset. Passive traffic traces from CAIDA's passive monitors in 2018, including the 'equinix-nyc' high-speed monitor. Data accessed March 2025.

[7] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theoretical Computer Science* 312, 1 (2004), 3–15. https://doi.org/10.1016/S0304-3975(03)00400-6

[8] Cloudflare. 2020. RakeLimit: A Rate Limiter for Distributed Systems. https://github.com/cloudflare/rakelimit Accessed: 2025.

[9] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proc. VLDB Endow.* 1, 2 (2008), 1530–1541. https://doi.org/10.14778/1454159.1454225

[10] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. 2004. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. Association for Computing Machinery, 35–46. https://doi.org/10.1145/1007568.1007575

[11] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[12] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. 2009. CoTS: A Scalable Framework for Parallelizing Frequency Counting over Data Streams. In *2009 IEEE 25th International Conference on Data Engineering (ICDE'09)*. 1323–1326. https://doi.org/10.1109/ICDE.2009.231

[13] Databricks. 2024. approx_top_k Function - Databricks SQL Reference. https://docs.databricks.com/en/sql/language-manual/functions/approx_top_k.html. Accessed: 2025.

[14] Martin Dietzfelbinger and Christoph Weidling. 2005. Balanced allocation and dictionaries with tightly packed constant size bins. In *Proceedings of the 32nd International Conference on Automata, Languages and Programming (ICALP'05)*. Springer-Verlag, 166–178. https://doi.org/10.1007/11523468_14

[15] Ulrich Drepper. 2007. What every programmer should know about memory. *Red Hat, Inc* 11, 2007 (2007), 2007.

[16] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. Association for Computing Machinery, 75–88. https://doi.org/10.1145/2674005.2674994

[17] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. 1998. Computing Iceberg Queries Efficiently. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., 299–310.

[18] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. 2011. The multiple-orientability thresholds for random hypergraphs. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '11)*. Society for Industrial and Applied Mathematics, 1222–1236.

[19] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. 2016. *Data stream management: processing high-speed data streams*. Springer.

[20] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. 2018. Heavykeeper: an accurate algorithm for finding top-k elephant flows. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*. USENIX Association, 909–921. https://doi.org/10.1109/TNET.2019.2933868

[21] Dor Harris, Arik Rinberg, and Ori Rottenstreich. 2023. Compressing Distributed Network Sketches With Traffic-Aware Summaries. *IEEE Transactions on Network and Service Management* 20, 2 (2023), 1962–1975. https://doi.org/10.1109/TNSM.2022.3172299

[22] Martin Hilgendorf and Marina Papatriantafilou. 2025. LMQ-Sketch: Lagom Multi-Query Sketch for High-Rate Online Analytics. arXiv:2506.16928 [cs.DS] https://arxiv.org/abs/2506.16928

[23] Victor Jarlow, Charalampos Stylianopoulos, and Marina Papatriantafilou. 2025. QPOPSS: Query and Parallelism Optimized Space-Saving for Finding Frequent Stream Elements. *J. Parallel and Distrib. Comput.* (2025), 105134. https://doi.org/10.1016/j.jpdc.2025.105134

[24] Anukool Lakhina, Mark Crovella, and Christiphe Diot. 2004. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement (IMC '04)*. Association for Computing Machinery, 201–206. https://doi.org/10.1145/1028788.1028813

[25] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent Cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. Association for Computing Machinery, Article 27. https://doi.org/10.1145/2592798.2592820

[26] Ankush Mandal, He Jiang, Anshumali Shrivastava, and Vivek Sarkar. 2018. Topkapi: parallel and fast sketches for finding top-K frequent elements. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., 10921–10931.

[27] Gurmeet Singh Manku and Rajeev Motwani. 2012. Approximate frequency counts over data streams. *Proc. VLDB Endow.* 5, 12 (2012), 1699. https://doi.org/10.14778/2367502.2367508

[28] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2006. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.* 31, 3 (2006), 1095–1133. https://doi.org/10.1145/1166074.1166084

[29] J. Misra and David Gries. 1982. Finding repeated elements. *Science of Computer Programming* 2, 2 (1982), 143–152. https://doi.org/10.1016/0167-6423(82)90012-0

[30] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. *SIGPLAN Not.* 48, 8 (2013), 103–112. https://doi.org/10.1145/2517327.2442527

[31] Vinh Ngo and Marina Papatriantafilou. 2025. *Artifact of the paper: Cuckoo Heavy Keeper and the balancing act of maintaining heavy hitters in stream processing*. https://doi.org/10.5281/zenodo.15593109

[32] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[33] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.* 13, 4 (2005), 277–298. https://doi.org/10.1155/2005/962135

[34] Redis. 2024. Top-K. https://redis.io/docs/latest/develop/data-types/probabilistic/top-k/ Accessed: 2025.

[35] Arik Rinberg and Idit Keidar. 2023. Intermediate Value Linearizability: A Quantitative Correctness Criterion. *J. ACM* 70, 2, Article 17 (2023). https://doi.org/10.1145/3584699

[36] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. 2022. Fast Concurrent Data Sketches. *ACM Trans. Parallel Comput.* 9, 2, Article 6 (2022). https://doi.org/10.1145/3512758

[37] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented Sketch: Faster and More Accurate Stream Processing. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, 1449–1463. https://doi.org/10.1145/2882903.2882948

[38] Qilong Shi, Yuchen Xu, Jiuhua Qi, Wenjun Li, Tong Yang, Yang Xu, and Yi Wang. 2023. Cuckoo Counter: Adaptive Structure of Counters for Accurate Frequency and Top-k Estimation. *IEEE/ACM Trans. Netw.* 31, 4 (2023), 1854–1869. https://doi.org/10.1109/TNET.2022.3232098

[39] Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. 2020. Delegation sketch: a parallel design with support for fast and accurate concurrent operations. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, Article 4. https://doi.org/10.1145/3342195.3387542

[40] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. HeavyGuardian: Separate and Guard Hot Items in Data Streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18)*. Association for Computing Machinery, 2584–2593. https://doi.org/10.1145/3219819.3219978

[41] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, 561–575. https://doi.org/10.1145/3230543.3230544

[42] Yu Zhang, Yue Sun, Jianzhong Zhang, Jingdong Xu, and Ying Wu. 2014. An efficient framework for parallel and continuous frequent item monitoring. *Concurrency and Computation: Practice and Experience* 26, 18 (2014), 2856–2879. https://doi.org/10.1002/cpe.3182

[43] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS '23)*. Curran Associates Inc., Article 1506.