Thesis for The Degree of Licentiate of Engineering

Efficient, Adaptable, and Scalable Synopses for Data-Intensive Systems

Vinh Quang Ngo

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2026

Efficient, Adaptable, and Scalable Synopses for Data-Intensive Systems

Vinh Quang Ngo

ISSN 1652-876X

Department of Computer Science and Engineering
Division of Networks and Systems
Distributed Computing and Systems
Chalmers University of Technology | University of Gothenburg
SE-412 96 Göteborg,
Sweden
Phone: +46(0)31 772 1000

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2026.

"There is beauty in imperfection."

Efficient, Adaptable, and Scalable Synopses for Data-Intensive Systems

Vinh Quang Ngo

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg

Abstract

Data-intensive systems generate data at rates and volumes that demand timely single-pass analysis with bounded memory. However, computing exact statistics in a single pass often requires memory that grows with the data under consideration; this is manageable for small windows, but becomes infeasible as the scope or number of computations grows. *Data summarization*, or *synopses*, addresses this by capturing statistics in sublinear space with bounded error, trading accuracy for memory and throughput. Although they are well-established algorithmically, practical use depends on hardware and deployment details. Most synopsis algorithms are sequential, yet production data rates often exceed a single core's capacity; throughput and latency also depend on cache behavior and access patterns, not just asymptotic complexity. Furthermore, they are initialized at fixed settings and approximation bounds, and cannot adapt to changing budgets and workloads. Multi-core and distributed processors can absorb higher rates, but they bring contention, cache coherence costs, state migration when scaling, and require merging differently-sized summaries.

Targeting these challenges, this thesis studies two core summarization primitives, *heavy-hitter detection* and *frequency estimation*, and contributes as follows. Chapter A analyzes the trade-offs among throughput, memory usage, and accuracy in heavy-hitter detection algorithms; the insights led to the design of the *Cuckoo Heavy Keeper* (CHK) algorithm, which introduces a process for distinguishing frequent from infrequent items that unlocks synergies inaccessible to conventional approaches, such as reduced per-item instruction cost and improved cache behavior. Chapter A also introduces a categorization of parallelization approaches and the *multi-CHK* (mCHK) framework, which can parallelize *any* sequential heavy-hitter algorithm, with support for concurrent updates and queries. Chapter B identifies three properties that target the above challenges: *resizability* (adjusting memory at runtime), *enhanced mergeability* (combining differently-sized summaries), and *partitionability* (splitting state for elastic scaling and load rebalancing). Building on these properties, Chapter B proposes ReSketch, a frequency estimation sketch design that achieves all three while maintaining a beneficial memory-to-accuracy ratio, together with the *instance provenance DAG*, which tracks how approximation bounds evolve through arbitrary sequences of these operations. Together, these results provide complementary building blocks for efficient, adaptable, and scalable summarization in modern data-intensive systems.

Keywords: Data Summarization, Synopsis, Efficiency, Adaptability, Scalability, Concurrency & Parallelism, Data-Intensive Systems

List of Articles

Appended Articles

Parts of the work presented in this thesis are also reported and documented in the following articles:

[**Article A**] **V. Q. Ngo** and M. Papatriantafilou, "Cuckoo Heavy Keeper and the Balancing Act of Maintaining Heavy Hitters in Stream Processing," *Proceedings of the VLDB Endowment*, Vol. 18, No. 9, pp. 3149–3161, May 2025.
DOI: `https://doi.org/10.14778/3746405.3746434`

[**Article B**] **V. Q. Ngo**, M. Hilgendorf, and M. Papatriantafilou, "ReSketch: A Mergeable, Partitionable, and Resizable Sketch," *Under submission*, 2026.

Other Articles

The following articles were published during my PhD studies, or are currently in submission/under revision. However, they are not appended to this thesis, due to contents overlapping that of main chapters or contents not related to the thesis.

[**Article a**] **V. Q. Ngo**, "Continuous and Concurrent Data Summarization," *Proceedings of the 19th ACM International Conference on Distributed and Event-based Systems (DEBS '25) – Doctoral Symposium*, Gothenburg, Sweden, June 2025.

[**Article b**] L. Magnusson, R. Thorsson, **V. Q. Ngo**, M. Papatriantafilou, J. van Rooij, and M. Chigrichenko, "CLUE — Clustering-Based Load Understanding and Exploration: Summarizing High-Dimensional Electricity Grid Data for Scenario Analysis," *Workshop on Relaxed Semantics in Data Analytics Pipelines (RELAX 2025), at DEBS '25*, Gothenburg, Sweden, June 2025.
DOI: `https://doi.org/10.5281/zenodo.18740102`

[**Article c**] M. K. Ngo-Huu, **V. Q. Ngo**, D. T. Luu, B. N. Pham, and V. T. Nguyen, "STERR-GAN: Spatiotemporal Rerendering for Facial Video Restoration," *IEEE MultiMedia*, Vol. 32, No. 4, pp. 45–57, Oct.–Dec. 2025.
DOI: `https://doi.org/10.1109/MMUL.2025.3611072`

Research Contribution

For both articles, I am the lead author and took the primary role in shaping the research directions, formulating the problems, and leading the writing; all co-authors actively participated in discussions, brainstorming, and contributing to the manuscript. For Article A, I was the main designer and implementor. For Article B, I designed and implemented the data structures; the experimental evaluation was carried out in collaboration with Martin Hilgendorf. Both projects were conducted under the advice and supervision of Marina Papatriantafilou.

Acknowledgment

First and foremost, I would like to express my deepest gratitude to my main supervisor, Marina. Thank you for always being there, for every discussion, every encouragement, and for your constant support. More than anything, thank you for pushing me to explore unfamiliar topics and aspects beyond what I was comfortable with. Those explorations always led to some of the most fascinating insights and results, and for that I am truly grateful. I would also like to thank my co-supervisors, Philippas, Vincenzo, and Dimitrios, for your insightful suggestions and support. Thank you also to Joris, for guiding me through a fruitful internship at Göteborg Energi.

I am truly honoured to have had Professor Papapetrou Odysseas as my discussion leader, and I am grateful to Ulf for his support as my examiner.

Looking further back, I owe so much to the people who believed in me before this journey even started. Thank you, Tiep Nguyen and Son Thai Mai, for encouraging me and supporting me in pursuing a PhD. And a very special thank you to Khanh Ngo, my former classmate/teammate/roommate, who truly inspired me to follow this research path.

Thank you, Martin, Linus, and Rasmus, for the fruitful collaborations we have had together. To my colleagues, Jacob, Jingyu, Yixing, Kåre, Atmane, and Yenan: thank you for making this journey so much more enjoyable. Research can be lonely at times, and having you around made all the difference. I also want to thank all the wonderful people I have crossed paths with in the corridors, classrooms, and over many lunches, for making the workplace feel warm and fun. This includes, but is certainly not limited to: Muoi, Romaric, Magnus, Hashim, Masoom, Umer, Rhouma, Ilias, Shubham, Weijia, Shahrooz, Elad, Torbjörn, Azadeh, Huaifeng, Danish, Francisco and Thomas.

A special thank you to the RELAX-DN network, which has supported me in so many ways: funding, training, expert guidance, and a wonderful community of researchers, which truly enriched my doctoral experience.

Last but not least, to my family. To my fantastic parents: thank you for always supporting me, even from so far away. Your love and encouragement have been constant throughout this journey, and I could not be more grateful. To my sister: honestly, I think you should be thanking me instead, since you are always the one bothering me. Thank you, Ngan, for always being by my side, for cheering me up when things got tough, and for giving me your perspective on the mathematical side of research whenever I was struggling. And to all of my amazing Vietnamese friends in Sweden: you are simply too many to name here. Thank you for listening to my stories, for the laughter, and for all the good times. You have made Sweden feel like a second home.

Funding Sources. This work has been supported Marie Skłodowska-Curie Doctoral Network RELAX-DN, funded by EU under Horizon Europe 2021-2027 FP Grant Agreement nr. 101072456 (www.relax-dn.eu/).

Contents

Abstract iii

List of Articles v

Acknowledgment vii

I Thesis Overview 1

1 Introduction 3

2 Stream Processing and Data Summarization 5
2.1 Stream Processing Models 6
2.2 Data Summarization 7
2.2.1 Historical Context 7
2.2.2 Summarization Techniques and Tasks 8
2.3 Frequency Estimation 10
2.3.1 The Count-Min Sketch 11
2.3.2 The Count Sketch 11
2.4 Heavy-Hitter Detection 12

3 Hashing 14
3.1 k-Wise Independent Hashing 14
3.2 Cuckoo Hashing 15
3.3 Consistent Hashing 16

4 Scaling Up and Scaling Out 17
4.1 Scale-Up: Single-Node Performance 18
4.2 Scale-Out: Distributed Deployments 19

5 Research Questions and State-of-the-Art 21
5.1 Efficient Data Summarization 21
5.2 Adaptable Data Summarization 22
5.3 Scalable Data Summarization 22

6 Thesis Contributions 23
6.1 Chapter A: Cuckoo Heavy Keeper and mCHK 23
6.2 Chapter B: ReSketch 24

7 Conclusions and Future Work 27

Bibliography 29

II Main Chapters **37**

Chapter A - Cuckoo Heavy Keeper and the Balancing Act of Maintaining Heavy Hitters in Stream Processing **39**

1 Introduction . . . 42
2 Problem Description . . . 44
3 Related Work and Problem Analysis . . . 45
3.1 Traditional Approaches . . . 45
3.2 Recent Advances . . . 46
4 Sequential Cuckoo Heavy Keeper . . . 47
4.1 Cuckoo Heavy Keeper Data Structure Layout . . . 47
4.2 Cuckoo Heavy Keeper Key Ideas . . . 48
4.3 Cuckoo Heavy Keeper Main Operations . . . 49
4.4 Weighted Update . . . 53
4.5 Optimizations . . . 53
5 Analysis of Cuckoo Heavy Keeper . . . 54
6 Concurrent Operations . . . 56
6.1 Parallel Designs and Trade-offs . . . 56
6.2 Parallel Cuckoo Heavy Keeper . . . 57
6.3 Accuracy of hh-Queries . . . 60
7 Evaluation . . . 61
7.1 Study of The Sequential Algorithms . . . 62
7.2 Study of The Parallel Algorithms . . . 65
8 Conclusions and Future Work . . . 69
Bibliography . . . 71

Chapter B - ReSketch: A Mergeable, Partitionable, and Resizable Sketch **75**

1 Introduction . . . 78
2 Preliminaries . . . 82
3 Problem Description . . . 84
4 ReSketch . . . 85
4.1 ReSketch Data Structure Layout . . . 85
4.2 Redistributing Bucket Contents . . . 86
4.3 Data Manipulation Operations . . . 89
4.4 Structure-Defining Operations . . . 90
5 ReSketch Analysis . . . 93
5.1 Static Single ReSketch Bound . . . 93
5.2 Dynamic Distributed ReSketch Bound . . . 94
6 Evaluation . . . 98
6.1 Sensitivity Analysis . . . 100
6.2 Benchmarks . . . 101
6.3 Application Example . . . 104
6.4 Further Discussion of Results . . . 106
7 Other Related Work . . . 107
8 Conclusions . . . 107
Bibliography . . . 109

Part I

Thesis Overview

1. Introduction

Data lies at the heart of modern decision-making. Organizations across every domain, from technology and finance to healthcare and scientific research, collect vast quantities of data with the goal of extracting actionable insights [9], [31]. The benefits of analysis are well-established. In databases, analyzing query workloads and data distributions allows optimizers to select execution plans that reduce response times by orders of magnitude [2]. In networking, monitoring traffic flows enables the detection of anomalous patterns, such as sudden surges characteristic of distributed denial-of-service (DDoS) attacks before service disruption occurs [44]. In both cases, value is realized not through the mere accumulation of data, but through its timely analysis.

The data deluge and the need for stream processing. The very success of data-driven approaches has produced unprecedented growth in the volume and velocity of data in modern systems. A single Internet exchange point can observe hundreds of gigabits of network traffic per second [13]; IoT deployments generate continuous telemetry from millions of sensors; database systems serving global applications process millions of transactions per minute. Such systems are commonly characterized as *data-intensive* [42]: the primary challenge lies in the scale and rate of data to ingest, process, and analyze. In these settings, the traditional *store-then-process* paradigm, in which data is first materialized to storage and only then analyzed as a bounded batch, is increasingly untenable [9], [61]. Storage costs alone can be prohibitive: retaining every packet traversing a 100 Gbps link for a single hour would require roughly 45 terabytes. Moreover, writing to persistent storage before analysis introduces latency incompatible with applications such as financial fraud detection, where delays of even a few microseconds can render a result useless [16]. These constraints motivate the *stream processing* paradigm, in which data is treated as a continuous, potentially unbounded sequence of items that are processed incrementally and in a single pass: each item is examined once upon arrival, and the algorithm's internal state is updated without requiring the raw data to be retained.

From stream processing to data summarization (or synopses). Even within the stream processing paradigm, the memory available to maintain state is finite and far smaller than the volume of data being processed. Computing exact aggregate statistics often requires memory that grows linearly with the data under consideration [6]: this is manageable when the scope of computation is small, but becomes infeasible as the scope or number of computations grows. *Data summarization* (or *synopses*) techniques address this by constructing compact, sublinear-space representations that capture the key statistical properties of the stream incrementally while using only a fraction of the memory that exact computation would require [17]. A Count-Min Sketch [21], for instance, can estimate the frequency of any item in a stream using space that depends only on the desired accuracy, independent of the number of distinct items or the stream length.

Efficiency, adaptability, and scalability. While the algorithmic foundations of data summarization are well established, deploying these in modern data-intensive systems requires further considerations. Most summarization algorithms are designed and analyzed as sequential procedures, yet production systems often have data rates that exceed the capacity of a single core; practical throughput is also affected by how a data structure interacts with the hardware (e.g., cache behavior, memory access patterns, instruction costs), not solely by its asymptotic complexity. Furthermore, conventional sketches fix their dimensions and error parameters at initialization and offer no mechanism to adapt afterward, yet available memory fluctuates and workload characteristics shift over time. Multi-core processors and distributed clusters could absorb higher data rates, but exploiting this parallelism for summarization structures raises its own difficulties, from contention and cache coherence on shared memory to nodes joining and leaving clusters, state migration due to elastic scaling, and merging of differently-sized summaries across distributed nodes. These observations point to three properties that practical synopses must address: *efficiency* in throughput, latency, memory usage, and accuracy on real hardware; *adaptability* to changing resource constraints and workload conditions; and *scalability* across multi-core and distributed environments.

Scope and Contributions. Among the many tasks that data summarization supports, this thesis studies two core summarization primitives, *frequency estimation* (approximating the count of individual items) and *heavy-hitter detection* (identifying the most frequent items in a stream) [21], [52], [66], and contributes as follows. These are among the most fundamental summarization primitives: frequency estimation provides the basic point-query building block, and heavy-hitter detection is one of its most important applications, used in network monitoring [20], [44], database optimization [10], [27], and analytics platforms [7], [22], [57]. Chapter A [55] analyzes the trade-offs among throughput, memory usage, and accuracy in heavy-hitter detection algorithms; the insights led to the design of *Cuckoo Heavy Keeper* (CHK), which introduces a process for distinguishing frequent from infrequent items that unlocks synergies inaccessible to conventional approaches, such as reduced per-item instruction cost and improved cache behavior. Chapter A also introduces a categorization of parallelization approaches and the *multi-CHK* (*m*CHK) framework, which can parallelize *any* sequential heavy-hitter algorithm, with support for concurrent updates and queries, even non-mergeable ones. Chapter B [54] identifies three properties that target the above challenges: *resizability* (adjusting memory at runtime), *enhanced mergeability* (combining differently-sized summaries), and *partitionability* (splitting state for elastic scaling and load rebalancing). Building on these properties, Chapter B proposes ReSketch, a frequency estimation sketch design that achieves all three while maintaining a beneficial memory-to-accuracy ratio, together with the *instance provenance DAG*, which tracks how approximation bounds evolve through arbitrary sequences of these operations. Together, these results provide complementary building blocks for efficient, adaptable, and scalable summarization, validated on both synthetic and real-world workloads.

Thesis organization. The remainder of this overview proceeds as follows. §2 introduces stream processing models, data summarization techniques, and the core tasks of frequency estimation and heavy-hitter detection. Since these techniques rely heavily on hash functions, §3 presents the hashing foundations used throughout the thesis. §4 discusses scale-up (single-node performance) and scale-out (distributed deployment) system foundations for summarization. §5 reviews related work, identifies the research gaps, and formulates three research questions. §6 presents the thesis contributions, mapping each paper to its research question. §7 concludes and discusses future directions. Detailed algorithm designs, formal analyses, and experimental evaluations appear in the main chapters (Chapter A and Chapter B in Part II).

2. Stream Processing and Data Summarization

Data processing has traditionally followed a *store-then-process* paradigm: data is first written to persistent storage and later read back for batch analysis [9]. This approach is conceptually simple, but it requires storage proportional to the volume of data and introduces latency between data arrival and result availability. As data rates grow, both costs become increasingly prohibitive. The *stream processing* paradigm avoids these costs by treating data as a continuous flow of items that are processed in a single pass: each item is examined once upon arrival, the system's internal state is updated, without requiring the raw data to be retained. Fig. 2.1 contrasts the two paradigms.

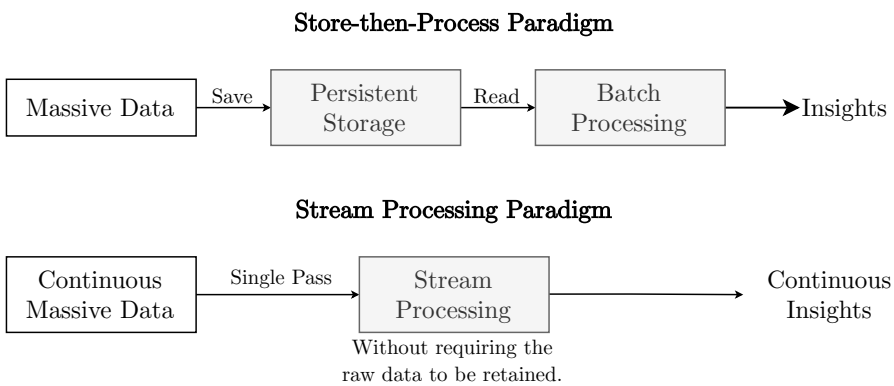


Figure 2.1: Two data processing paradigms. In the store-then-process paradigm (top), data is first persisted and later read for batch analysis. In the stream processing paradigm (bottom), data is processed in a single pass as it arrives, producing continuous insights without requiring the raw data to be retained.

The remainder of this chapter formalizes stream processing models (§2.1), then introduces data summarization, the family of algorithms that make stream processing more efficient (§2.2).

2.1 Stream Processing Models

This section introduces the streaming model used throughout this thesis and then briefly discusses other complementary aspects such as windowing and systems concerns.

Problem formulation. Following the standard treatment in the data summarization literature [9], [31], [52], a *data stream* is formally a sequence

$$S = (a_1, a_2, \ldots, a_t, \ldots),$$

where each element $a_t = (e_t, w_t)$ consists of an *item* e_t drawn from a universe $\mathcal{U}$ and a *weight* $w_t \in \mathbb{R}$. The item identifies the entity being observed (e.g., an IP address, a financial instrument, or a network flow identifier); the weight quantifies the magnitude of the observation (e.g., packet byte count, trade volume, or simply 1 for an unweighted occurrence).

As the stream progresses, it induces a *frequency vector* $\mathbf{f} \in \mathbb{R}^{|\mathcal{U}|}$, whose e-th entry accumulates the total weight attributed to item e:

$$\mathbf{f}[e] \;=\; \sum_{t\,:\,e_t = e} w_t, \qquad e \in \mathcal{U}. \tag{2.1}$$

A broad class of analytical tasks can be reduced to computing a function $\mathcal{F}(\mathbf{f})$ of this vector [5], [31], [52]: frequency estimation (computing $\mathbf{f}[e]$ for a given item), heavy-hitter detection (identifying items with large $\mathbf{f}[e]$), set-membership testing (determining whether $\mathbf{f}[e] > 0$), distinct counting (computing $|\{e : \mathbf{f}[e] > 0\}|$), and quantile estimation, among others. However, computing any such function *exactly* in a single pass generally requires memory proportional to the size of the universe $\mathcal{U}$ [6], which is prohibitive for large domains. This motivates *data summarization*: compact, sublinear-space representations that maintain approximate answers with bounded error, which will be discussed in §2.2.

The constraints imposed on the weights w_t give rise to different models, each with different algorithmic implications, as summarized in Fig. 2.2.

Standard streaming models [31], [52]

Let $\mathbf{f}^{(t)} \in \mathbb{R}^{|\mathcal{U}|}$ denote the frequency vector after processing the first t elements. The three standard models are defined by constraints on the weights w_t:

- *Cash-register model* ($w_t > 0$): all weights are positive, so the frequency vector is monotonically non-decreasing: $\forall e \in \mathcal{U} : \mathbf{f}^{(t)}[e] \geq \mathbf{f}^{(t-1)}[e]$.
- *Turnstile model* ($w_t \in \mathbb{R}$): weights are arbitrary reals, which models both insertions and deletions, so entries of $\mathbf{f}$ may become negative.
- *Strict turnstile model* ($w_t \in \mathbb{R}$, $\mathbf{f}^{(t)}[e] \geq 0$): weights are arbitrary reals, but the frequency vector satisfies $\mathbf{f}^{(t)}[e] \geq 0$ for all $e \in \mathcal{U}$ at every t in the stream. Deletions are permitted, yet the frequency for any item never goes negative.

Figure 2.2: Standard streaming models.

This thesis focuses primarily on the *cash-register* model, which is the natural setting for frequency estimation and heavy-hitter detection. Extending the techniques developed here to the strict turnstile and general turnstile models is identified as a direction for future work.

Other aspects of stream processing. In addition to the computational models above, the temporal scope of a computation is also an important consideration. The function of interest can be evaluated over the W most recent elements through *sliding windows*, where elements that fall outside the window expire as new ones arrive [8], [14]. Windowing captures the practical requirement that recent data is often more relevant than older records, and when the window fits in memory it admits exact answers. When the scope extends to the full observed stream, the window is too large for exact computation, or many windows must be maintained simultaneously, approximate summarization techniques become necessary and can be used in conjunction with windowing.

Additionally, in this very active research field, continuous query languages [1], [8] specify how summarization primitives are scoped over time through windows and triggers, while distributed stream processing engines [4], [14], [33], [63] handle partitioning, fault recovery, out-of-order events, and merging of partial states across nodes [3], [9]. These aspects compose naturally: advances in the summarization primitives benefit the systems that use them, and the requirements of those systems inform the design of the primitives. This thesis focuses on improving the summarization primitives themselves; the design decisions are informed by practical system requirements such as efficient mergeability and adaptability to changing resource conditions in distributed deployments.

2.2 Data Summarization

As noted in §2.1, computing exact statistics from a stream in a single pass generally requires memory that grows with the data under consideration. *Data summarization* techniques (also called *synopses*) address this by constructing compact, sublinear-space representations that are maintained incrementally and support approximate queries with bounded approximation error [17]. The remainder of this section covers the historical development of core summarization techniques and the main summarization tasks relevant to this thesis.

2.2.1 Historical Context

The conceptual origins of data summarization can be traced to random sampling, a technique with a history extending well before the advent of computers [18]. Within computer science, this idea materialized as *reservoir sampling*, which maintains a uniform random sample from a data stream when the total number of elements is not known in advance [64].

In the 1970s, Bloom [12] introduced a compact probabilistic data structure for set-membership testing, trading a controlled false-positive rate for dramatic space savings. Morris [50] showed that an approximate count of n events

can be maintained using only $O(\log \log n)$ bits by incrementing a counter probabilistically. Munro and Paterson [51] studied the problem of selection and sorting when the number of passes over the data is limited, establishing early connections between pass complexity and space. Flajolet and Martin [29] developed a probabilistic algorithm for counting the number of distinct elements in a large dataset using logarithmic space, a precursor to modern distinct-counting sketches such as HyperLogLog [28].

In the 1990s, Alon, Matias, and Szegedy [6] established the formal streaming model. Their paper posed a simple question: given a sequence of elements arriving one at a time, how much memory does an algorithm need to compute or estimate the *frequency moments* $F_k = \sum_{e \in \mathcal{U}} \mathbf{f}[e]^k$, where $\mathbf{f}[e]$ is the number of occurrences of item e, if *each element may be seen only once*? They established that computing any frequency moment *exactly* in a single pass requires memory linear in the stream length, $\Omega(n)$, even for F_0. Having established this lower bound, they then showed that *randomized approximation* can circumvent it significantly. For F_0 (distinct element count) and F_2 (sum of squared frequencies, or self-join size), randomized algorithms require only $O(\text{poly}(\log n, 1/\varepsilon))$ space, sublinear in the stream length, to achieve a $(1 \pm \varepsilon)$ multiplicative approximation. For higher moments (F_k, $k \geq 6$), however, even approximation requires space that grows polynomially with the stream length.

This seminal work laid a significant cornerstone for the field of data summarization. Recognized with the Gödel Prize in 2005, the paper inspired a rich body of follow-up work establishing summarization algorithms and tight space bounds for problems spanning frequency estimation, distinct counting, quantile estimation, and graph analysis [52]. The next section highlights several of these developments most relevant to this thesis.

2.2.2 Summarization Techniques and Tasks

Summarization Techniques. The three most common techniques are random sampling, histograms, and sketches.

- *Random sampling* retains a uniformly random subset of stream elements, providing a general-purpose mechanism that is straightforward to implement. Reservoir sampling [52] extends this to the streaming setting, maintaining a fixed-size representative sample of the stream as new elements arrive. Its accuracy degrades for rare items or tail-sensitive queries, since infrequent items are unlikely to appear in a small random sample.
- *Histograms* partition the data domain into a bounded number of buckets and maintain per-bucket aggregate statistics. Their accuracy depends on how well bucket boundaries align with the underlying data distribution.
- A *sketch* is a random linear projection of the frequency vector $\mathbf{f}$ defined in Eq. 2.1. More concretely, the sketch state is $\mathbf{Af}$, where $\mathbf{A}$ is a matrix determined by hash functions [17], [31]. Because matrix-vector multiplication is linear, i.e., $\mathbf{A}(\mathbf{f}_1 + \mathbf{f}_2) = \mathbf{Af}_1 + \mathbf{Af}_2$, sketches are *linear*: the entry-wise sum of two sketches built over disjoint sub-streams is a valid sketch for their union [3]. This linearity has benefits for distributed computation via mergeable summaries, as discussed later in this section.

All three techniques serve the same purpose: maintaining a compact internal state that approximates the frequency vector $\mathbf{f}$ defined in Eq. 2.1. Each stream element (e_t, w_t) updates this state $\mathbf{f}$ upon arrival (an UPDATE), and analytical tasks are answered by computing functions of the state (a QUERY).

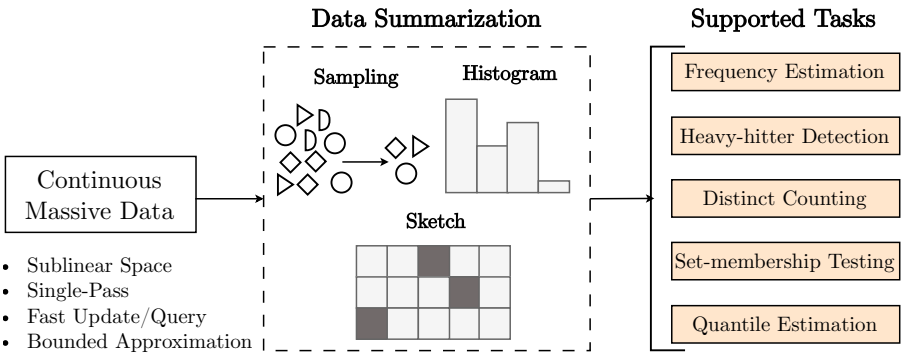


Figure 2.3: Overview of data summarization. Continuous massive data (left) is processed through summarization techniques (center), including sampling, histograms, and sketches, to support downstream analytical tasks (right) such as frequency estimation, heavy-hitter detection, distinct counting, set-membership testing, quantile estimation and more. Key properties of the summarization process are listed below the input: sublinear space, single-pass processing, fast updates and queries, and bounded approximation.

Summarization Tasks. A wide range of analytical tasks reduce to computing specific functions of $\mathbf{f}$. These include, but are not exhaustive:

- *Frequency estimation*: estimate a point query $\mathbf{f}[e]$ for a given item e. Sketch-based solutions include the Count-Min Sketch [21] and the Count Sketch [15], which we discuss in detail in §2.3.
- *Heavy-hitter detection*: identify all items e whose frequency $\mathbf{f}[e]$ exceeds a given fraction of the total stream weight $N = \|\mathbf{f}\|_1$. This task is discussed in more detail in §2.4.
- *Distinct counting*: estimate $F_0 = |\{e \in \mathcal{U} : \mathbf{f}[e] > 0\}|$, the number of distinct items observed. HyperLogLog [28] is the most widely used algorithm, which achieves remarkably small memory usage (e.g., kilobytes for counting billions of distinct items).
- *Set-membership testing*: determine whether $\mathbf{f}[e] > 0$ for a given item e. The Bloom filter [12] is the canonical example, which allows a controlled rate of false positives but no false negatives.
- *Quantile estimation*: given a target quantile $q \in (0, 1)$, return an item whose rank in the stream is approximately $q \cdot N$, where items are ordered by value. Representative algorithms include the GK algorithm [32], t-digest [25], and the KLL sketch [38], [41].

Mergeability. Beyond updates and queries, a third operation is highly desirable, especially in parallel and distributed settings: MERGE. A summary is *mergeable* if, given two summaries A_1 and A_2 computed on datasets D_1 and D_2 respectively, a MERGE operation produces a summary A_{merged} that is valid for the union $D_1 \cup D_2$ with the same approximation guarantees, without re-processing any original data [3]. This property underpins distributed computation: each node computes a local summary from its portion of the data, and a coordinator combines the partial results.

Sketches achieve mergeability with ease because of their linearity. The Count-Min Sketch is merged by entry-wise addition of counter arrays [21]; HyperLogLog by entry-wise maximum of registers [28]; the Bloom filter by bitwise OR of bit arrays [12]; and the KLL sketch by combining compactors level by level [41]. Other techniques can also support merging, but with more procedures (e.g., histograms require compatible bucket boundaries).

However, not all summarization algorithms are mergeable. Some heavy-hitter algorithms such as HeavyKeeper [69], HeavyGuardian [67], and ElasticSketch [68] achieve state-of-the-art performance but, to our knowledge, do not support merging. Among quantile algorithms, the GK summary [32] admits only a restricted form of merging that may increase the summary size, and its full mergeability was left as an open question by Agarwal et al. [3]. This lack of mergeability creates an additional challenge: when summaries that are distributed across parallel threads or cluster nodes cannot simply be combined, the system must instead provide efficient mechanisms for coordinating updates and answering queries without merging.

Even for algorithms that do support mergeability, the standard requirement is that all summaries have *identical initialized parameters*: the same width, depth, and hash functions (for sketches), or the same number of registers and hash family (for HyperLogLog), or the same bit-array length and hash functions (for Bloom filters).

These two limitations, non-mergeability of certain algorithm families and the identical-parameter constraint for those that are mergeable, and their implications are discussed further in §5.

2.3 Frequency Estimation

Among the summarization tasks introduced in §2.2, frequency estimation is arguably the most fundamental: given a queried item $e \in \mathcal{U}$, approximate the frequency $\mathbf{f}[e]$ of e in the stream (i.e., $\hat{f}(e)$), where $\mathbf{f}$ is the frequency vector defined in Eq. 2.1. The central question is how much accuracy can be preserved given a fixed memory budget, and how that trade-off is parameterized.

The standard formulation [17], [21], [31] for an (ε, δ)-*approximate frequency estimator* is as follows: for any queried item e, the estimator produces an estimate $\hat{f}(e)$ such that:

$$\Pr\Big[\ \Big|\hat{f}(e) - \mathbf{f}[e]\Big| \leq \varepsilon N\ \Big] \ \geq\ 1 - \delta,$$

where $N = \sum_{e \in \mathcal{U}} \mathbf{f}[e]$ is the stream size. The two parameters control the

quality of the approximation: $\varepsilon \in (0, 1)$ specifies the maximum additive error as a fraction of the total stream size, and $\delta \in (0, 1)$ bounds the probability that the error exceeds this guarantee.

2.3.1 The Count-Min Sketch

The Count-Min Sketch (CMS) (Fig. 2.4), introduced by Cormode and Muthukrishnan [21], is the most widely used frequency estimation sketch, owing to its simplicity, strong theoretical guarantees, and practical efficiency. It consists of a two-dimensional array C of $d \times w$ counters, each initialized to zero, together with d pairwise independent hash functions $h_1, h_2, \ldots, h_d$, each mapping items from $\mathcal{U}$ to the set $\{0, 1, \ldots, w-1\}$; the hash functions define the projection matrix $\mathbf{A}$ described in §2.2, and C stores the resulting sketch $\mathbf{Af}$. The UPDATE operation is straightforward: upon arrival of an element (e, w_t), for each row $i \in \{1, \ldots, d\}$, the algorithm increments counter $C[i][h_i(e)]$ by w_t, which costs $O(d)$ time per update. The QUERY operation estimates $\mathbf{f}[e]$ by returning $\hat{f}(e) = \min_{i=1}^{d} C[i][h_i(e)]$, i.e., the minimum counter value across d rows.

The guarantees of the Count-Min Sketch follow directly from its construction. With width $w = \lceil \mathrm{e}/\varepsilon \rceil$ (e denotes Euler's number) and depth $d = \lceil \ln(1/\delta) \rceil$, the sketch satisfies two properties: first, $\hat{f}(e) \geq \mathbf{f}[e]$ always holds, because every counter that e hashes to includes e's true counts plus counts from other items that collide into the same counter; second, $\Pr[\hat{f}(e) \leq \mathbf{f}[e] + \varepsilon N] \geq 1 - \delta$, meaning that the overestimation is bounded by εN with high probability. The total space requirement is $O\big(\frac{1}{\varepsilon} \log \frac{1}{\delta}\big)$ counters, independent of the universe size and stream length, and matches the lower bound for additive error relative to the total stream size.

CountMin(w, d)

```
C[1][1] ... C[d][w] = 0
for j ← 1 to d do
    Initialize h_j
foreach e, w_t do
    for j ← 1 to d do
        C[j][h_j(e)] += w_t
```

CountSketch(w, d)

```
C[1][1] ... C[d][w] = 0
for j ← 1 to d do
    Initialize h_j, g_j
foreach e, w_t do
    for j ← 1 to d do
        C[j][h_j(e)] += g_j(e) · w_t
```

Figure 2.4: Algorithms for Frequency Estimation

2.3.2 The Count Sketch

The Count Sketch (Fig. 2.4), introduced by Charikar, Chen, and Farach-Colton [15], offers an alternative formulation that provides an *unbiased* estimator. In addition to the d position hash functions $h_i : \mathcal{U} \to \{0, 1, \ldots, w-1\}$, the Count Sketch uses d sign functions $g_i : \mathcal{U} \to \{-1, +1\}$. During an update, each counter $C[i][h_i(e)]$ is incremented by $g_i(e) \cdot w_t$ rather than by w_t directly.

During a query, the estimate is computed as $\hat{f}(e) = \text{median}_{i=1}^{d}\, g_i(e) \cdot C[i][h_i(e)]$, using the median across rows to amplify the probability of an accurate answer.

With $w = O(1/\varepsilon^2)$ columns and $d = O(\log(1/\delta))$ rows, it guarantees:

$$\Pr\Big[\ \Big|\hat{f}(e) - \mathbf{f}[e]\Big| \ \le\ \varepsilon\sqrt{F_2}\ \Big] \ \ge\ 1-\delta,$$

where $F_2 = \sum_{e\in\mathcal{U}} \mathbf{f}[e]^2$ is the second frequency moment of the stream. This bound can be substantially tighter than the Count-Min's εN bound when the frequency distribution is skewed. However, the Count Sketch requires $O(1/\varepsilon^2)$ counters per row (width) instead of $O(1/\varepsilon)$, and its depth d has larger constant factors in the $O(\log(1/\delta))$ term. This combination makes it less space-efficient than Count-Min for many data distributions. The choice between the two sketches therefore depends on the application's error metric and the expected skewness of the data distribution.

2.4 Heavy-Hitter Detection

Heavy-hitter detection identifies the items in a data stream whose frequency exceeds a given fraction ϕ of the total stream size. The problem was first formally described by Misra and Gries [48]. Following the standard formulation [19], [55], given a data stream S, a heavy-hitter threshold $\phi \in (0,1)$, and an approximation parameter $\varepsilon \in (0,\phi)$, the *ε-ϕ-heavy hitters problem* requires an algorithm to return a set $\hat{R}$ of estimated heavy hitters satisfying:

C1: If $\mathbf{f}[e] \ge \phi N$, then $e \in \hat{R}$ (no false negatives).
C2: If $\mathbf{f}[e] \le (\phi-\varepsilon)N$, then $e \notin \hat{R}$ (limited false positives).
C3: For each $e \in \hat{R}$, $|\hat{f}(e) - \mathbf{f}[e]| \le \varepsilon N$ (bounded deviation).

The (ε,δ)-*ϕ-heavy hitters* variant relaxes these conditions probabilistically:

C4: For each condition (C1–C3), if the premise is met, then with probability at least $1-\delta$, the stated outcome holds.

This relaxation allows for even more compact summaries.

There are three main algorithmic families for solving the heavy-hitter detection problem, each with different trade-offs in terms of accuracy, space, and update/query efficiency.

Key-Value (KV)-based algorithms. Algorithms such as Frequent [48], Lossy Counting [46], and Space-Saving [47] maintain a fixed set of approximately $1/\phi$ KV counters tracking candidate heavy hitters (Fig. 2.5). When an arriving item matches an existing entry, its counter is incremented; otherwise, the algorithm either allocates a new slot or evicts an existing entry according to an algorithm-specific policy. These algorithms are deterministic, satisfying C1–C3 exactly, and exhibit strong recall. Their limitations are throughput (lookups may require scanning or maintaining a hash table) and frequency estimation precision, since the εN error floor applies regardless of true frequency.

FREQUENT(k)

```
1  n ← 0; T ← ∅
2  foreach e do
3      n ← n + 1
4      if e ∈ T then
5          c_e ← c_e + 1
6      else if |T| < k − 1 then
7          T ← T ∪ {e}
8          c_e ← 1
9      else
10         forall j ∈ T do
11             c_j ← c_j − 1
12             if c_j = 0 then
                   T ← T \ {j}
```

SPACESAVING(k)

```
1  n ← 0; T ← ∅
2  foreach i do
3      n ← n + 1
4      if i ∈ T then
5          c_i ← c_i + 1
6      else if |T| < k then
7          T ← T ∪ {i}
8          c_i ← 1
9      else
10         j ← arg min_{j∈T} c_j
11         c_i ← c_j + 1
12         T ← T ∪ {i} \ {j}
```

LOSSYCOUNTING(k)

```
1  n ← 0; Δ ← 0; T ← ∅
2  foreach e do
3      n ← n + 1
4      if e ∈ T then
5          c_e ← c_e + 1
6      else
7          T ← T ∪ {e}
8          c_e ← 1 + Δ
9      if ⌊n/k⌋ ≠ Δ then
10         Δ ← ⌊n/k⌋
11         forall j ∈ T do
12             if c_j < Δ then
                   T ← T \ {j}
```

Figure 2.5: Algorithms for Heavy-Hitter Detection

Sketch-based algorithms. Sketch-based approaches augment the sketches (e.g., Count-Min Sketch) (§2.3) with a min-heap of top-k candidates. The sketch provides frequency estimates for all items; items whose estimated count exceeds the threshold are added to the heap. Because sketches do not inherently distinguish heavy from non-heavy items, the εN overestimation can elevate infrequent items above the detection threshold, producing false positives.

Hybrid algorithms. Hybrid approaches pair a *heavy part* (a KV structure for candidates) with a *light part* (a sketch for the tail). Items are checked against the heavy part first, fall through to the light part, and are promoted from the light part when their estimated count crosses a threshold. Representatives include HeavyGuardian [67], ElasticSketch [68], AugmentedSketch [59], CuckooCounter [60], and HeavyKeeper [69]. Several share the technique of *count-with-exponential-decay*: on a collision, an occupant's counter is decre-

mented by 1 with probability b^{-C}, biasing retention toward frequent items. Note that the path-dependent eviction policies in these hybrid designs make them non-mergeable (§2.2.2), which has implications for their use in parallel and distributed settings and will be discussed further in §5. A detailed analysis of the trade-offs among these families is developed in Chapter A [55].

3. Hashing

The summarization techniques surveyed in §2 share a common algorithmic substrate: hash functions. In the Count-Min Sketch, hash functions determine which counters are incremented for each arriving item; in HyperLogLog and Bloom filters, hash functions map items into compact bit/register representations [12], [21], [28]. Hash functions supply the *randomness* that makes formal approximation bounds possible.

This chapter begins with the probabilistic foundation and then presents two algorithms that build on it. §3.1 introduces *k-wise independent hash families*, which formalize the limited randomness that hash functions must provide for the error bounds of sketches, hash tables, and distributed key-assignment protocols to hold. §3.2 covers *cuckoo hashing*, a hash-table design that achieves worst-case constant-time lookup and serves as the underlying data structure of the CHK algorithm developed in this thesis. §3.3 covers *consistent hashing*, a key-assignment scheme that decouples item-to-bucket mapping from the number of buckets, which enables minimal redistribution when the structure is resized or partitioned, which is central to the ReSketch algorithm developed in this thesis.

3.1 k-Wise Independent Hashing

Theoretically, a truly random hash function $h : \mathcal{U} \to [m]$ is analytically ideal for algorithmic analysis, as it provides perfect randomness. However, storing such a function requires $|\mathcal{U}| \log m$ bits, which is infeasible for large universes [49]. Therefore, k-wise independent hash families are used as a practical substitute, providing sufficient randomness for algorithmic guarantees.

Definition 3.1 (k-wise independence [49])**.** *A family $\mathcal{H}$ of functions $h : \mathcal{U} \to [m]$ is* k-wise independent *if, for every k distinct elements $x_1, \ldots, x_k \in \mathcal{U}$ and every target tuple $(v_1, \ldots, v_k) \in [m]^k$,*

$$\Pr_{h \sim \mathcal{H}}\big[h(x_1) = v_1 \ \wedge \ \cdots \ \wedge \ h(x_k) = v_k\big] \ = \ \frac{1}{m^k}.$$

A common way to construct k-wise independent hash families is to use random polynomials over a finite field $\mathbb{F}_p$ with $p \geq |\mathcal{U}|$: a degree-$(k-1)$ polynomial $h(x) = \sum_{j=0}^{k-1} a_j x^j \bmod p$ with coefficients drawn uniformly from $\mathbb{F}_p$ yields a k-wise independent family [65]. For example, with $k = 2$, the family $\mathcal{H}_2 = \{h_{a,b}(x) = ax + b \bmod p : a, b \in \mathbb{F}_p\}$ is pairwise independent and suffices for the Count-Min Sketch analysis.

Concentration (tail) Inequalities and Their Independence Requirements k-wise independence is valuable since many concentration inequalities (also called *tail inequalities*) require only *limited* independence. The following box collects the common inequalities used throughout this thesis.

Concentration (tail) inequalities and their independence requirements

Let $X_1, \ldots, X_n$ be random variables satisfying the stated independence condition, and let $X = \sum_{i=1}^{n} X_i$ with $\mu = E[X]$ and $\sigma^2 = \mathrm{Var}[X]$.

- **Markov's inequality** *(no independence required; $X \geq 0$):* $\Pr[X \geq t] \leq \mu/t$. Yields $O(1/t)$ tail decay. Used in the Count-Min Sketch analysis.
- **Chebyshev's inequality** *(2-wise independence suffices):* $\Pr[|X - \mu| \geq t] \leq \sigma^2/t^2$. Pairwise independence ensures $\mathrm{Var}[X] = \sum_i \mathrm{Var}[X_i]$, giving $O(1/t^2)$ tails. Used in the Count Sketch analysis.
- **Chernoff bound** *(n or $O(\log n)$-wise independence required):* For $X_i \in [0, 1]$: $\Pr[X \geq (1+\delta)\mu] \leq \exp(-\delta^2\mu/3)$. Yields exponential tail decay. Used in cuckoo hashing and consistent hashing analyses.

Example proof sketch: Count-Min Sketch error bound via Markov's inequality [21]. Fix a row i and a queried item e. The collision noise $Z_i = \sum_{e' \neq e} \mathbf{f}[e'] \cdot \mathbf{1}[h_i(e') = h_i(e)]$ satisfies $E[Z_i] \leq N/w$ by pairwise independence (due to construction). Since $Z_i \geq 0$, Markov's inequality gives $\Pr[Z_i > \varepsilon N] \leq 1/(w\varepsilon)$. Setting $w = \lceil \mathrm{e}/\varepsilon \rceil$ and taking the minimum over $d = \lceil \ln(1/\delta) \rceil$ independent rows yields the (ε, δ) guarantee using only 2-wise independent hash functions. Note that the proof relies only on the collision probability $\Pr[h_i(e') = h_i(e)] = 1/w$, which is strictly weaker than the full pairwise independence condition (Definition 3.1). This observation, that the (ε, δ) guarantee depends on collision probability rather than on full pairwise independence, is revisited in the context of ReSketch (§6), where it motivates replacing modulo-based bucket assignment with consistent hashing.

3.2 Cuckoo Hashing

Conventional open-addressing hash tables achieve $O(1)$ expected-case lookup but can degrade to $O(1/(1-\alpha))$ probes at load factor α [43], which is problematic for latency-sensitive applications. *Cuckoo hashing*, introduced by Pagh and Rodler [56], achieves *worst-case* $O(1)$ lookup time while supporting $O(1)$ amortized expected-case insertion.

The scheme maintains a single table of size m with two independent hash functions $h_0, h_1 : \mathcal{U} \to [m]$. Every stored key e resides in either position $h_0(e)$ or position $h_1(e)$, so a Lookup inspects exactly these two positions, guaranteeing worst-case $O(1)$ time. Delete locates the key via the same two-probe lookup and clears the slot in worst-case $O(1)$ time. Insert is where the scheme derives its name. Fig. 3.1 illustrates three cases; in each panel, the left table shows the state before the operation and the right table shows the state after. In panel (a), item i is inserted into an empty candidate slot. In panel (b), item h finds one

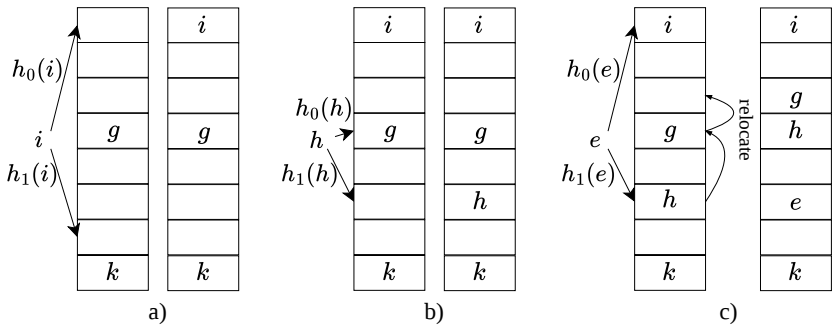


Figure 3.1: Cuckoo hashing insertion (before → after). (a) Direct placement into an empty slot. (b) Placement without displacement. (c) Displacement chain: e evicts h and g to their alternative positions.

candidate occupied but the other free, so it is placed without displacement. In panel (c), both candidates for item e are occupied, triggering a displacement chain: e evicts h, which displaces g to its alternative position. If the chain exceeds a predefined length, the table is rehashed with new hash functions.

Regarding hash function independence, Pagh and Rodler [56] showed that $O(\log n)$-wise independence, where n is the number of stored keys, suffices for the full theoretical guarantees. In the same work, they also showed that even 2-wise or 3-wise independent families work well [56]. In practice, *bucketized* cuckoo hashing [26] groups multiple slots into a single cache-line-sized bucket, so that all slots are fetched in one memory access, exploiting spatial locality (cf. §4.1). Bucketization was also shown analytically to achieve better load factors compared to standard cuckoo hashing [23], [30].

3.3 Consistent Hashing

When a collection of n nodes jointly manages a key space, a key question is: how should keys be assigned to nodes so that the assignment remains stable as nodes join or leave? The simplest approach, modular hashing $h(e) \bmod n$, distributes keys uniformly but is brittle under changes. Adding or removing a single node changes the modulus, which remaps a fraction of all keys to different nodes and forces costly redistribution of state.

The core insight of *consistent hashing*, introduced by Karger et al. [40], is to decouple key assignment from the exact number of nodes. Rather than computing a modulus that depends on n, both keys and nodes are mapped onto a shared space, so that each key's assignment is determined by geometric proximity to a node. This way, when a single node is added or removed, only the keys in its immediate neighborhood are affected, while the vast majority of assignments remain unchanged.

Concretely, the construction maps both items and nodes onto a logical ring $\mathcal{R} = [0, 2^{32})$ via a hash function h. One can think of the ring as a clock face: each node occupies a position on the ring, and each key e is assigned to the first

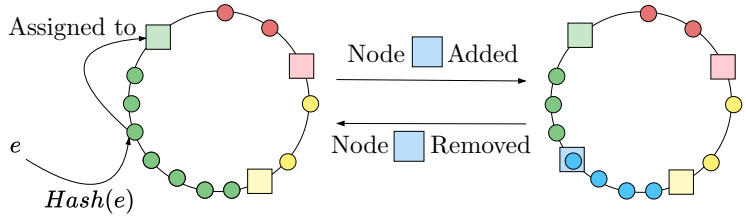


Figure 3.2: Consistent hashing. Left: item e is hashed onto the ring and assigned to the next clockwise node (green). Right: adding or removing a node (blue) reassigns only a fraction of keys.

node encountered when walking clockwise from position $h(e)$, i.e., its *successor node*. The ring is thus partitioned into contiguous arcs, one per node, where each arc contains all keys between a node and its clockwise predecessor. When a new node joins, it lands at some position on the ring and takes ownership of the arc between itself and its predecessor; only the keys in that arc, a $1/(n+1)$ fraction in expectation, are reassigned. Symmetrically, when a node departs, its arc is absorbed by its clockwise successor, again affecting only $O(1/n)$ keys. Fig. 3.2 illustrates the ring layout and the effect of a topology change.

Lookup requires a binary search over the sorted node positions in $O(\log n)$ time. Regarding independence requirements, pairwise independence of the hash function suffices for the expected-load analysis, while the sharper maximum-load concentration bound follows from a Chernoff-type argument under $O(\log n)$-wise independence [49] and additional techniques (i.e., virtual nodes [40]).

4. Scaling Up and Scaling Out

The summarization algorithms described in the preceding sections are sequential and single-threaded. In practice, however, the rate at which data arrives frequently exceeds the processing capacity of a single core, and even on a single core, practical throughput depends on how well an algorithm interacts with the hardware, not just its asymptotic complexity. Modern processors provide tens or hundreds of cores on a single chip, which offers an opportunity to scale up performance through parallelism. Beyond a single machine, real-world deployments are often distributed across clusters of networked nodes, sometimes because the data itself are geographically dispersed (e.g., edge routers, regional data centers) and sometimes simply because the production infrastructure was designed that way. Exploiting these resources requires understanding two complementary strategies: *scaling up*, which improves the performance on a single machine by leveraging its hardware and multiple cores, and *scaling out*, which distributes computation across multiple machines connected by a network. This section covers the hardware and system foundations underlying both strategies. Research questions and related work building on these foundations are detailed in §5 and the main chapters.

4.1 Scale-Up: Single-Node Performance

Modern multi-core processors organize memory in a hierarchy of decreasing speed and increasing capacity [24], [34]. Fig. 4.1 illustrates this hierarchy. Each core has private caches: an L1 cache (typically 32–64 KB, with 1–4 cycle latency) and an L2 cache (typically 256 KB–1 MB, with 5–12 cycle latency). A shared last-level cache (L3, typically 8–64 MB, with 20–40 cycle latency) serves all cores on a socket. Main memory (DRAM, 100–300 cycle latency) is shared across all cores. On multi-socket systems (NUMA topology), each socket has its own memory controller; accessing memory attached to a remote socket incurs additional latency. So computation within a core is fast, but moving data between caches, between cores, or between sockets is expensive.

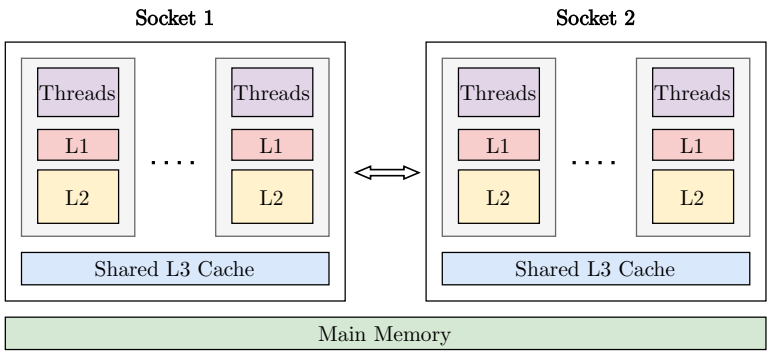


Figure 4.1: Multi-socket cache hierarchy.

Cache Locality. This hierarchy is designed around two empirical observations about program behavior: recently accessed memory tends to be accessed again (*temporal locality*), and memory close in address to a recently accessed location tends to be accessed soon thereafter (*spatial locality*). So every access to main memory brings a contiguous block of data (a *cache line*, typically 64 bytes) into the L1 cache, and subsequent accesses to that line are much faster; less-used data is evicted to farther (slower) levels to make room for hot data. These observations imply that the design of data structures and their access patterns is important for performance regardless of parallelism [24], [34].

Cache coherence. When multiple cores access the same memory location, hardware coherence protocols (e.g., MESI, MOESI [53]) ensure that all cores observe a consistent view of shared memory. A write by one core invalidates all copies of the affected cache line in other cores' private caches, forcing those cores to re-fetch the line before they can access it. This invalidation traffic is an inherent cost of shared-memory parallelism. *False sharing* is a particular manifestation: when two cores access logically independent variables that happen to reside on the same cache line (typically 64 bytes), every write by one core invalidates the other's cached copy, serializing operations that should have

been independent. For sketch data structures, false sharing can arise when adjacent counters in a shared array are updated by different threads [36].

Synchronization primitives. Concurrent access to shared data structures requires synchronization to prevent data races. *Mutual exclusion locks* (mutexes) are the simplest approach: a thread acquires a lock before accessing the shared structure and releases it afterward. Locks introduce performance hazards: *contention* (many threads waiting for the same lock), *priority inversion* (a high-priority thread blocked by a low-priority lock holder), and *convoying* (all threads stalled when the lock holder is descheduled). *Reader-writer locks* allow concurrent reads but require exclusive access for writes, which helps when reads are frequent. *Atomic operations* (compare-and-swap, CAS; fetch-and-add, FAA) provide finer-grained, non-blocking synchronization: they are hardware-supported instructions that modify a single memory location without acquiring a lock. *Lock-free* algorithms guarantee that at least one thread makes progress in any finite interval; *wait-free* algorithms guarantee that every thread completes its operation in a bounded number of steps [35], [36]. The spectrum from coarse-grained locking through fine-grained atomics to lock-free and wait-free designs trades implementation complexity for scalability under high contention.

Design principles for scalable parallel data structures. The key insight is that *any synchronization on the path of an update reduces throughput*; therefore, several design principles have emerged for high-performance parallel data structures to reduce synchronization overhead: partition data across threads to minimize sharing (each thread owns a portion of the data and accesses it without synchronization); use thread-local buffers to batch work and reduce the frequency of cross-thread communication; align data structures to cache-line boundaries to prevent false sharing; prefer designs where concurrent readers do not need to write (avoiding invalidation traffic). Beyond parallelism, per-item instruction cost matters as well: expensive operations on the update path (e.g., floating-point arithmetic) can become a throughput bottleneck even on a single core. These principles apply to the parallelization of many data structures, including synopsis structures (§2.3, §2.4), and will be discussed further in relation to related work and research questions in §5 [58], [62].

4.2 Scale-Out: Distributed Deployments

While shared-memory parallelism exploits the multiple cores within a single machine, workloads can also exceed the capacity of a single machine, motivating the use of multiple machines to keep up with the data rate. A related setting arises when the data itself is inherently distributed: it originates at geographically dispersed sources (e.g., edge routers, IoT gateways, regional data centers), and no single machine can observe the full stream. In such settings, distributed computing is not merely a performance optimization but a fundamental constraint of the system architecture.

The Dataflow Model. Recall from §2.1 that the streaming model processes data as an unbounded sequence of items arriving one at a time. In distributed settings, a widely adopted computational model is the *dataflow model*, in which a computation is expressed as a directed acyclic graph (DAG) of operators connected by data channels [4], [14]. Each operator consumes one or more input streams, applies a transformation (e.g., filtering, mapping, or aggregating), and emits results to downstream operators. The dataflow model has two forms of parallelism. *Task parallelism* occurs when operators that are independent in the DAG execute concurrently on different nodes, each performing a distinct computation. In practice, task parallelism is often handled implicitly by the stream processing engine via task scheduling. *Data parallelism* occurs when a single logical operator is replicated across multiple physical instances, each processing a disjoint partition of the input stream. Data parallelism is the primary mechanism for scaling individual operators to match the rate of incoming data, and it is the form of parallelism most relevant to the distributed summarization contributions of this thesis.

Distributed Aggregation via Partial Results. Data parallelism for aggregation operators follows a natural two-phase pattern. In the first phase, the input stream is partitioned across p parallel instances of the same logical operator, and each instance independently computes a local result from its partition. In the second phase, a *combine* stage merges the p partial results into a single global result that represents the entire input. This pattern, commonly referred to as *partial aggregation*, is the standard mechanism for scaling aggregate computations in distributed stream processing systems [14], [63]. The pattern is general: it applies to exact aggregations (e.g., distributed sums or counts) as well as approximate ones, provided that the local results can be combined correctly. For approximate data summarization, mergeable data structures (§2.2.2) supply exactly this property: two sketches or synopses computed independently on disjoint sub-streams can be combined into a single structure representing their union, with the same approximation guarantees and without re-processing any original data.

Practical Considerations in Distributed Deployments In practice, distributed deployments exhibit characteristics that go beyond the basic partial-aggregation pattern. Nodes may be added or removed at runtime as workloads change (*elastic scaling* [33]). Different nodes may have different hardware capabilities and memory budgets (*resource heterogeneity*). Data distribution across nodes may be uneven and may shift over time (*load imbalance*), sometimes requiring the system to rebalance by migrating accumulated state between nodes. These characteristics of modern distributed environments motivate the investigation of more flexible summarization operations, as discussed in §5.

5. Research Questions and State-of-the-Art

This section identifies the research gaps that motivate this thesis and formulates the research questions addressed by the main chapters. Each subsection corresponds to one of the three properties in the thesis title: efficiency (§5.1), adaptability (§5.2), and scalability (§5.3).

5.1 Efficient Data Summarization

Traditionally, data summarization algorithms are analyzed and compared primarily through asymptotic complexity: space bounds. While such bounds are essential for understanding theoretical limits, they are insufficient for predicting real-world performance, as discussed in §4. Practical throughput depends heavily on cache behavior, memory access patterns, instruction costs, factors that asymptotic analysis does not capture.

In the context of membership testing, a system-aware perspective has been extensively studied and has produced a rich family of practical designs. Bloom filters [12] have been refined into blocked variants that align each probe to a single cache line, register-blocked and sectorized layouts that further optimize SIMD and memory-level parallelism, and cache-sectorized designs that balance false-positive rate against cache efficiency [45]. Cuckoo filters [26] similarly exploit cache-friendly bucket layouts with compact fingerprints. However, heavy-hitter detection has not benefited from the same level of system-aware optimization. A complementary aspect of efficiency concerns the relationship between memory usage and accuracy. As discussed in §4, smaller data structures are more likely to fit in cache, leading to substantially higher throughput. Many algorithms achieve space-optimal bounds for a specific error metric (e.g., the Count-Min Sketch for εN additive error), but such bounds are quite loose and not always optimal for the accuracy metrics used in practice, such as average relative error (ARE) and average absolute error (AAE). Algorithms based on different norms, such as the Count Sketch with its $\varepsilon\sqrt{F_2}$ bound, can yield better practical accuracy for common skewed distributions while using different space trade-offs. Even hash function requirements, the core substrate of most summarization algorithms, can be relaxed to be more efficient: just as cuckoo hashing requires $O(\log n)$-wise independence in theory but works well with simpler families (i.e., 3-wise) in practice [56] (§3.2), hash function designs for sketches can be relaxed to allow faster computation with controlled trade-offs. These observations suggest that efficiency improvements remain available across multiple dimensions of the design space.

Research Question 1 (Efficiency)

How can data summarization algorithms be designed for higher practical efficiency in terms of throughput, latency, memory usage, and accuracy?

5.2 Adaptable Data Summarization

As introduced in §2.3, conventional frequency estimation sketches fix their dimensions and approximation bounds at initialization. Consider a sketch of size $d \times w$: it employs d hash functions $h_1, \ldots, h_d$ drawn from a pairwise independent family, and these functions determine the bucket index $h_i(e)$ for an item e via a modulo operation ($\pmod w$). When resizing a sketch by changing its width from w to w', many bucket contents must be moved, and because each bucket stores only an aggregate count, there is no information to guide remapping during resizing, merging, or partitioning. For example, if a counter in one bucket needs to be partitioned across two new buckets, known methods cannot determine how to partition its value because we do not know *which items are present* or *their individual counts* to map to the new locations. This rigidity is problematic in the dynamic, heterogeneous environments described in §4.2, where available memory fluctuates over time and across nodes.

Existing resizable designs address this only partially. DCMS [70] supports expansion by maintaining a chain of progressively larger sub-sketches but cannot reclaim memory. Geometric Sketch [11] supports fine-grained expansion and partial memory release but cannot shrink below its initial allocation, and its query throughput degrades as the internal chain grows. To our knowledge, neither produces structures that are compatible with straightforward merging with differently sized instances after resizing. This limitation is consequential for distributed deployments: standard mergeability requires identical parameters, so heterogeneous nodes must either adopt the smallest common sketch size, wasting memory on well-provisioned nodes, or forgo merging entirely (i.e., keep separate instances and scan them all when querying). This is already an important practical problem. We argue that adaptability in sketch structure can also facilitate scalability, since the ability to resize and merge differently-sized instances is precisely what is needed for elastic parallelism.

Research Question 2 (Adaptability)

How can data summarization algorithms be made more adaptable to changing resource and deployment conditions?

5.3 Scalable Data Summarization

As discussed in §4, concurrent and distributed computing are established fields, yet research on how data summarization structures perform under such settings remains limited. Conventional approaches to parallelizing summarization algorithms follow two main strategies: *single-shared* designs, where all threads operate on a single shared structure with synchronization; and *thread-local* designs, where each thread maintains its own private structure and results are merged at query time. They have drawbacks: single-shared designs suffer from contention on hot spots, while thread-local designs require mergeability (which is not always supported by the underlying algorithms as discussed in §2.2.2 and

§2.4) and cannot support concurrent queries during processing. The recently introduced peer-collaborative design based on domain splitting [37], [39], [62] addresses several of these limitations: it enables concurrent insertions and frequency queries, achieves high throughput through contention-free private structures, and does not require mergeability. However, it also raises new questions. Global queries, such as retrieving heavy hitters or computing quantiles across all domains, require scanning all P local structures, introducing latency proportional to the degree of parallelism. Locking or freezing the entire structure to answer such queries is possible but not scalable. Furthermore, the degree of parallelism itself is fixed at initialization: the domain partition is determined by P, and no existing design supports dynamically changing the number of partitions, which is essential when the workload or available resources change. Partitioning the accumulated state of a summarization structure, splitting its historical information into independent pieces that can be migrated or rebalanced, remains an open problem that connects adaptability and scalability in parallel settings.

Research Question 3 (Scalability)

How can data summarization algorithms be made scalable for parallel and distributed processing while supporting efficient concurrent updates and queries?

6. Thesis Contributions

This section and the Table 6.1 outline the contributions of the thesis and their relationship to the research questions defined in §5.

Table 6.1: Research questions and the chapters that address them.

	Chapter A CHK & mCHK	**Chapter B** ReSketch
RQ1: Efficiency	•	•
RQ2: Adaptability	○	•
RQ3: Scalability	•	•

6.1 Chapter A: CHK and mCHK

Chapter A contributes to both efficiency and scalability of heavy-hitter detection in two orthogonal directions.

Cuckoo Heavy Keeper (RQ1). As discussed in §5.1, system-aware optimization has yielded substantial improvements for membership-testing structures, yet heavy-hitter detection has not benefited from the same level of attention. Hybrid algorithms (§2.4) often combine a KV heavy part with a sketch-based light part; the conventional approach inserts items into the heavy part first and falls back to the light part when the heavy part is full. We analyze how

this data flow interacts with input distributions and hardware, and identify that it forces every arriving item, including the infrequent, through the expensive heavy-part path before being redirected. *Cuckoo Heavy Keeper* (CHK) inverts this flow: items first enter the light part and are promoted to the heavy part only after accumulating evidence of significance. This inversion unlocks algorithmic synergies that are inaccessible with prior designs: expensive collision-resolution mechanisms (specifically, cuckoo hashing, §3.2) are confined to a small number of proven frequent items rather than applied to every arriving item, and the separated data flow enables optimizations such as hash precalculation, system-aware instruction optimization, and improved cache behavior, in line with the design principles discussed in §4.1. CHK provides (ε, δ)-ϕ-heavy hitter guarantees with improved accuracy bounds for the same memory budget, achieving 1.7–5.7× higher throughput and up to four orders of magnitude better accuracy compared to state-of-the-art algorithms including Space-Saving [47], Count-Min Sketch [21], AugmentedSketch [59], and HeavyKeeper [69], across real-world network traces and synthetic data. The throughput and accuracy advantage generalizes across workloads, even under constrained conditions (e.g., low ϕ or low skew).

***m*CHK: a parallel framework for heavy-hitter detection (RQ3).** As identified in §5.3, domain splitting enables concurrent insertions and point-frequency queries, but *global* queries, such as reporting all heavy hitters across the entire stream, remain problematic because they require scanning all P local structures. Chapter A introduces a parallel framework based on domain splitting that addresses this challenge. The key insight is that heavy-hitter detection is inherently approximate (§2.4), so the parallelism itself can tolerate controlled additional approximation while remaining theoretically error-bounded. The framework is *algorithm-agnostic*: it wraps any sequential heavy-hitter algorithm, including the non-mergeable designs identified in §2.2.2 and §2.4, without requiring modifications to the underlying data structure. It provides two complementary designs (mCHK-I and mCHK-Q) targeting different workloads. An important observation is that CHK's sequential accuracy improvement also enables the parallel one: higher accuracy means fewer false positives, which reduces synchronization overhead and data movement during global queries. Evaluated with varying underlying algorithms (CHK, AugmentedSketch [59], Count-Min Sketch [21], and Space-Saving [47]), both designs scale nearly linearly with the number of threads, and mCHK-Q maintains heavy-hitter query latency below 150 μsec even under high thread counts and query rates. The framework also scales across diverse hardware, including NUMA and UMA architectures, with and without hyper-threading.

6.2 Chapter B: ReSketch

As identified in §5.2, conventional frequency estimation sketches cannot resize, cannot merge with differently-sized instances, and cannot partition their accumulated state, because the modulo-based hash functions tie item assignment

to a specific bucket and because each bucket stores only an aggregate count with no information about which items contributed to it. Chapter B studies all of these limitations and proposes ReSketch, a sketch design that addresses them through three complementary ideas, together with a formal framework for reasoning about the resulting approximation guarantees.

Ring-based bucket assignment (RQ2). The first idea targets the assignment problem: when the number of buckets changes, the modulo operation forces a global remapping of items to buckets. The key property needed is that adding or removing buckets should reassign only a small, local set of items. This is precisely what consistent hashing provides (§3.3). ReSketch adapts this idea to sketches: each row's buckets are defined as contiguous segments on a logical ring, and an item is assigned to the bucket whose segment covers its hash value. Because the item-to-ring mapping remains fixed, resizing amounts to adjusting segment boundaries, reassigning only items near the boundary rather than all items (RQ2).

Per-bucket distribution summaries (RQ2). The second idea targets the redistribution problem: even with the ring-based mapping, adjusting a boundary requires knowing how to divide the affected bucket's aggregate count between the old and new segments. ReSketch addresses this by maintaining a per-bucket mergeable distribution summary. When a boundary moves, this summary estimates how many counts belong to the portion being transferred. Because the summary is itself mergeable, it also enables combining differently-sized sketch instances (*enhanced mergeability*, RQ2).

Partition-aware hashing (RQ3). Together, the ring-based assignment and per-bucket summaries also address partitioning. The challenge is that: in a standard sketch, each item maps to *different* columns across rows, so splitting the matrix by columns would scatter a single item's counters across multiple partitions, requiring each update to be sent to several partitions and each query to gather from all of them. ReSketch solves this with *partition-aware hashing*: each item carries a fingerprint, recovered from stored hash values via modular multiplicative inverses at no additional storage cost, that maps it to exactly one partition. This enables dynamic redistribution of accumulated state for load balancing in distributed settings (§5.3). ReSketch also complements recent work on domain-splitting parallelism [37], [39], [55], [62], providing the dynamic partitioning mechanism that these frameworks currently lack.

Efficiency and generality (RQ1). The ability to shrink a sketch when accuracy requirements permit means that the working set can fit in faster cache levels (§4.1), and the ability to expand when memory becomes available improves accuracy without reinitializing the structure. ReSketch maintains competitive throughput and the (ε, δ) guarantee, and because the design is compatible with matrix-based sketches in general, other sketches can adopt the same mechanism.

Connection to hashing foundations. From the perspective of the hashing foundations developed in §3, ReSketch's design can also be understood as a systematic relaxation of standard hash function properties. As shown in the proof sketch of §3.1, the Count-Min Sketch analysis only requires that, for any two distinct items $x \neq y$, the collision probability satisfies $\Pr[h(x) = h(y)] = 1/w$; pairwise independence (§3.1) demands the strictly stronger condition $\Pr_{h\sim\mathcal{H}}\big[h(x) = a \wedge h(y) = b\big] = 1/w^2$ for all distinct x, y and all $a, b \in [w]$. This gap between what the proofs use and what the hash families provide is precisely what motivates ReSketch's design: by replacing discrete modulo mapping with consistent hashing on a logical ring (§3.3), the design decouples item assignment from the physical number of buckets while preserving the collision-probability property that the (ε, δ) guarantee actually depends on.

Instance provenance DAG. When a sketch instance has been resized, merged, and partitioned through an arbitrary sequence of operations, reasoning about its approximation guarantees becomes non-trivial: each operation may introduce additional error, and the cumulative effect depends on the full history. Chapter B introduces the *instance provenance DAG*, a formal framework that records the lineage of each sketch instance and tracks how approximation bounds propagate through these operations, showing that expand, merge, and partition introduce zero expected additional error.

Evaluation highlights. An extensive evaluation on real-world datasets compares ReSketch against Count-Min Sketch [21] (for static settings only), DCMS [70], and Geometric Sketch [11]. ReSketch achieves orders of magnitude better estimation accuracy across expansion and shrinking scenarios, while maintaining stable update throughput. The evaluation also shows its suitability for long-lived analytics processes with dynamic memory availability: the sketch can grow to improve accuracy, shrink to release memory for other work, and continue processing throughout. In contrast, Geometric Sketch's performance degrades progressively with each expansion, to the point where ReSketch surpasses it in update throughput, and DCMS suffers significant query throughput loss as its internal chain of sub-sketches grows. None of the baselines supports enhanced merging or partitioning; empirically, ReSketch's EnhancedMerge and Partition operations do not degrade estimation accuracy, consistent with the theoretical analysis, with merged sketches matching the accuracy of a same-sized sketch that processed the full input directly, and partitioned sketches matching independently constructed ones.

7. Conclusions and Future Work

This thesis investigated three properties that practical data summarization must satisfy in modern data-intensive systems: efficiency, adaptability, and scalability.

For heavy-hitter detection, the thesis contributes CHK and a parallel framework mCHK. CHK inverts the conventional hybrid flow: items enter the light part first and are promoted to the heavy part later. This confines expensive operations (i.e., collision resolution) to a small set of frequent items and enables the optimizations studied in this thesis, including hash precalculation, instruction-level optimization, and improved cache behavior. The accompanying parallel framework mCHK can extend any sequential heavy-hitter algorithms to work in parallel, including non-mergeable ones, and supports concurrent updates and global queries with controlled additional approximation.

For frequency estimation, the thesis contributes ReSketch. ReSketch addresses the rigidity caused by modulo-based bucket assignment, which prevents resizing, differently-sized merging, and state partitioning in conventional sketches. It replaces modulo mapping with consistent hashing on a logical ring and maintains per-bucket mergeable distribution summaries to estimate count redistribution when boundaries move. With partition-aware hashing and the instance provenance DAG, ReSketch also supports partitioning while tracking approximation guarantees across operation sequences.

Generalizing domain splitting. Reflecting on these results, it is possible to note that mCHK's domain-splitting design is algorithm-agnostic and not specific to heavy-hitter detection. Non-mergeable or restricted-merge algorithms appear in other summarization tasks as well; for example, the GK summary and t-digest for quantile estimation [25], [32] face similar parallelization barriers. Whether domain splitting with bounded additional approximation can serve as a general parallelization strategy for such algorithms is worth investigating, particularly in conjunction with tighter or distribution-specific error bounds.

Toward adaptable summaries. On the sketch side, ReSketch's ring-based assignment and per-bucket distribution summaries are not tied to the current design specifically. Any matrix-based sketch (Count-Min Sketch, Count Sketch, and variants) could, in principle, adopt the same mechanism to gain resizability, enhanced mergeability, and partitionability. For algorithms that already maintain per-item identity, such as Space-Saving or the KLL sketch, adaptation may be more direct, since the information needed for redistribution is already present. Taken together, these observations point toward a broader class of *adaptable summaries*: structures whose internal organization can change at runtime in response to resource availability, workload shifts, or the degree of parallelism. A key open question is what theoretical guarantees can be preserved and whether the instance provenance DAG generalizes to track error propagation for these structures.

Adaptability and efficiency trade-offs. Adaptability also introduces its own trade-off. The per-bucket distribution summaries that enable adaptability add overhead to every update, even when the structure is not being resized. More generally, this kind of slowdown is not unique to ReSketch. Any sketch that supports resizing must maintain additional structure or perform additional bookkeeping beyond a bare frequency sketch, so some throughput cost is expected. Note that for ReSketch, we use the KLL quantile sketch, which is a full quantile summary, and might be richer than necessary for the redistribution task. Whether a simpler, lighter-weight distribution summary or estimator can achieve a similar (ε, δ) guarantee while reducing the per-update cost, and potentially also the cost of structural operations that redistribute these summaries, is an open question worth investigating.

Coordination and integration into deployed systems. Finally, integrating these ideas into deployed systems raises engineering challenges and opens new research questions. In stream processing engines, ReSketch's partition-aware state splitting could support live repartitioning of operator state during elastic scaling, without discarding accumulated statistics. Analytics platforms that maintain sketches as persistent aggregates (e.g., Apache Druid [7], Redis [57]) currently require all instances to share identical parameters; enhanced mergeability would let each node size its sketch to available memory while still producing correct combined results. In database query optimizers, where Count-Min Sketches inform optimization decisions, runtime resizability would allow sketches to grow or shrink as table statistics evolve. Across such settings, a common challenge is deciding *when* and *by how much* to resize, merge, or partition a summary. This calls for coordination policies that balance redistribution overhead against the expected gains in accuracy and system performance, in both sequential and parallel deployments. Developing such policies, and evaluating them under conditions such as frequent churn, repeated oscillation, varying node capacities, and long sequences of structural operations, is a natural next step. In parallel environments, these coordination mechanisms may be especially valuable, since ReSketch's structural operations could complement domain-splitting frameworks [37], [39], [55], [62] by enabling dynamic rebalancing as resources and workloads change.

Overall, the thesis opens up a design space for efficient, adaptable, and scalable summarization, and exploring this space both theoretically and practically constitutes an impactful research direction worth pursuing in the future.

Bibliography

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul and S. Zdonik, 'Aurora: A new model and architecture for data stream management,' *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003. DOI: 10.1007/s00778-003-0095-z.

[2] S. Acharya, P. B. Gibbons, V. Poosala and S. Ramaswamy, 'Join synopses for approximate query answering,' in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '99, New York, NY, USA: Association for Computing Machinery, Jun. 1999, pp. 275–286. DOI: 10.1145/304182.304207.

[3] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei and K. Yi, 'Mergeable summaries,' in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, ser. PODS '12, New York, NY, USA: Association for Computing Machinery, May 2012, pp. 23–34. DOI: 10.1145/2213556.2213562.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom and S. Whittle, 'MillWheel: Fault-tolerant stream processing at internet scale,' in *Proceedings of the VLDB Endowment*, vol. 6, 2013, pp. 1033–1044. DOI: 10.14778/2536222.2536229.

[5] N. Alon, Y. Matias and M. Szegedy, 'The space complexity of approximating the frequency moments,' in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96, Association for Computing Machinery, Jul. 1996, pp. 20–29. DOI: 10.1145/237814.237823.

[6] N. Alon, Y. Matias and M. Szegedy, 'The space complexity of approximating the frequency moments,' *Journal of Computer and System Sciences*, vol. 58, no. 1, pp. 137–147, 1999. DOI: 10.1006/jcss.1997.1545.

[7] Apache Druid, *Topn queries - apache druid documentation*, https://druid.apache.org/docs/latest/querying/topnquery/, Accessed: 2025, Apache Software Foundation, 2024.

[8] A. Arasu, S. Babu and J. Widom, 'The CQL continuous query language: Semantic foundations and query execution,' *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006. DOI: 10.1007/s00778-004-0147-z.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani and J. Widom, 'Models and issues in data stream systems,' in *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02, New York, NY, USA: Association for Computing Machinery, 2002, 1–16. DOI: 10.1145/543613.543615.

[10] K. Beyer and R. Ramakrishnan, 'Bottom-up computation of sparse and iceberg CUBE,' *Sigmod Record*, vol. 28, no. 2, pp. 359–370, 1999. DOI: 10.1145/304181.304214.

[11] D. Biton, R. Friedman and R. Shahout, 'Geometric Sketch: The Inflatable-Shrinkable Sketch,' en, in *Advanced Information Networking and Applications*, L. Barolli, Ed., Cham: Springer Nature Switzerland, 2025, pp. 270–281. DOI: 10.1007/978-3-031-87766-7_24.

[12] B. H. Bloom, 'Space/time trade-offs in hash coding with allowable errors,' *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. DOI: 10.1145/362686.362692.

[13] CAIDA, *The CAIDA UCSD Anonymized Internet Traces - 2018-03-15*, 2018.

[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi and K. Tzoumas, 'Apache Flink: Stream and batch processing in a single engine,' in *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 38, 2015, pp. 28–38.

[15] M. Charikar, K. Chen and M. Farach-Colton, 'Finding frequent items in data streams,' *Theoretical Computer Science*, Automata, Languages and Programming, vol. 312, no. 1, pp. 3–15, Jan. 2004. DOI: 10.1016/S0304-3975(03)00400-6.

[16] C. Cook. 'When a microsecond is an eternity: High performance trading systems in C++.' Presented at CppCon 2017, CppCon. (8th Oct. 2017).

[17] G. Cormode, 'Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches,' *Foundations and Trends in Databases*, vol. 4, no. 1-3, pp. 1–294, 2011. DOI: 10.1561/1900000004.

[18] G. Cormode, 'Applications of Sketching and Pathways to Impact,' in *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '23, Association for Computing Machinery, Jun. 2023, pp. 5–10. DOI: 10.1145/3584372.3589937.

[19] G. Cormode and M. Hadjieleftheriou, 'Finding frequent items in data streams,' *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1530–1541, Aug. 2008. DOI: 10.14778/1454159.1454225.

[20] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck and D. Srivastava, 'Holistic UDAFs at streaming speeds,' in *Proceedings of the 2004 ACM SIGMOD international conference on management of data*, ser. Sigmod '04, Association for Computing Machinery, 2004, pp. 35–46. DOI: 10.1145/1007568.1007575.

[21] G. Cormode and S. Muthukrishnan, 'An improved data stream summary: The count-min sketch and its applications,' *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005. DOI: 10.1016/j.jalgor.2003.12.001.

[22] Databricks, *Approx_top_k function - Databricks SQL reference*, 2024.

[23] M. Dietzfelbinger and C. Weidling, 'Balanced allocation and dictionaries with tightly packed constant size bins,' in *Proceedings of the 32nd international conference on automata, languages and programming*, ser. ICALP'05, Springer-Verlag, 2005, pp. 166–178. DOI: 10.1007/11523468_14.

[24] U. Drepper, 'What every programmer should know about memory,' *Red Hat, Inc.*, 2007, Available at https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[25] T. Dunning, 'The t-digest: Efficient estimates of distributions,' *Software Impacts*, vol. 7, p. 100 049, Feb. 2021. DOI: 10.1016/j.simpa.2020.100049.

[26] B. Fan, D. G. Andersen, M. Kaminsky and M. D. Mitzenmacher, 'Cuckoo filter: Practically better than bloom,' in *Proceedings of the 10th ACM international on conference on emerging networking experiments and technologies*, ser. CoNEXT '14, Association for Computing Machinery, 2014, pp. 75–88. DOI: 10.1145/2674005.2674994.

[27] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani and J. D. Ullman, 'Computing iceberg queries efficiently,' in *Proceedings of the 24rd international conference on very large data bases*, ser. Vldb '98, Morgan Kaufmann Publishers Inc., 1998, pp. 299–310.

[28] P. Flajolet, É. Fusy, O. Gandouet and F. Meunier, 'HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm,' *Discrete Mathematics & Theoretical Computer Science*, vol. DMTCS Proceedings vol. AH,... No. Proceedings, p. 3545, Jan. 2007. DOI: 10.46298/dmtcs.3545.

[29] P. Flajolet and G. N. Martin, 'Probabilistic counting algorithms for data base applications,' *J. Comput. Syst. Sci.*, vol. 31, no. 2, 182–209, Sep. 1985. DOI: 10.1016/0022-0000(85)90041-8.

[30] N. Fountoulakis, M. Khosla and K. Panagiotou, 'The multiple-orientability thresholds for random hypergraphs,' in *Proceedings of the twenty-second annual ACM-SIAM symposium on discrete algorithms*, ser. Soda '11, Society for Industrial and Applied Mathematics, 2011, pp. 1222–1236.

[31] M. Garofalakis, J. Gehrke and R. Rastogi, Eds., *Data Stream Management: Processing High-Speed Data Streams* (Data-Centric Systems and Applications). Berlin, Heidelberg: Springer, 2016. DOI: 10.1007/978-3-540-28608-0.

[32] M. Greenwald and S. Khanna, 'Space-efficient online computation of quantile summaries,' in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, ACM, 2001, pp. 58–66. DOI: 10.1145/375663.375670.

[33] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente and P. Valduriez, 'Streamcloud: An elastic and scalable data streaming system,' *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012. DOI: 10.1109/TPDS.2012.24.

[34] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th. Morgan Kaufmann, 2019.

[35] M. Herlihy, 'Wait-free synchronization,' *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991. DOI: 10.1145/114005.102808.

[36] M. Herlihy, *The Art of Multiprocessor Programming*, Rev. 1st ed. Morgan Kaufmann, 2012, 1 p.

[37] M. Hilgendorf and M. Papatriantafilou, 'LMQ-Sketch: Lagom Multi-Query Sketch for High-Rate Online Analytics,' in *39th International Symposium on Distributed Computing (DISC 2025)*, D. R. Kowalski, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 356, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 36:1–36:24. DOI: 10.4230/LIPIcs.DISC.2025.36.

[38] N. Ivkin, E. Liberty, K. Lang, Z. Karnin and V. Braverman, *Streaming Quantiles Algorithms with Small Space and Update Time*, arXiv:1907.00236 [cs], Jun. 2019. DOI: 10.48550/arXiv.1907.00236.

[39] V. Jarlow, C. Stylianopoulos and M. Papatriantafilou, 'QPOPSS: Query and Parallelism Optimized Space-Saving for finding frequent stream elements,' *Journal of Parallel and Distributed Computing*, vol. 204, p. 105 134, Oct. 2025. DOI: 10.1016/j.jpdc.2025.105134.

[40] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine and D. Lewin, 'Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web,' en, in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC '97*, El Paso, Texas, United States: ACM Press, 1997, pp. 654–663. DOI: 10.1145/258533.258660.

[41] Z. Karnin, K. Lang and E. Liberty, 'Optimal Quantile Approximation in Streams,' in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, New Brunswick, NJ, USA: IEEE, Oct. 2016, pp. 71–78. DOI: 10.1109/FOCS.2016.17.

[42] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[43] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Reading, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[44] A. Lakhina, M. Crovella and C. Diot, 'Characterization of network-wide anomalies in traffic flows,' in *Proceedings of the 4th ACM SIGCOMM conference on internet measurement*, ser. Imc '04, Association for Computing Machinery, 2004, pp. 201–206. DOI: 10.1145/1028788.1028813.

[45] H. Lang, T. Neumann, A. Kemper and P. Boncz, 'Performance-optimal filtering: Bloom overtakes Cuckoo at high throughput,' *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 502–515, Jan. 2019. DOI: 10.14778/3303753.3303757.

[46] G. S. Manku and R. Motwani, 'Approximate frequency counts over data streams,' *Proc. VLDB Endow.*, vol. 5, no. 12, p. 1699, 2012. DOI: 10.14778/2367502.2367508.

[47] A. Metwally, D. Agrawal and A. E. Abbadi, 'An integrated efficient solution for computing frequent and top-k elements in data streams,' *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1095–1133, Sep. 2006. DOI: 10.1145/1166074.1166084.

[48] J. Misra and D. Gries, 'Finding repeated elements,' *Science of Computer Programming*, vol. 2, no. 2, pp. 143–152, Nov. 1982. DOI: 10.1016/0167-6423(82)90012-0.

[49] M. Mitzenmacher and E. Upfal, *Probability and computing: randomized algorithms and probabilistic analysis*, eng. Cambridge: Cambridge University Press, 2012.

[50] R. Morris, 'Counting large numbers of events in small registers,' *Commun. ACM*, vol. 21, no. 10, 840–842, Oct. 1978. DOI: 10.1145/359619.359627.

[51] J. Munro and M. Paterson, 'Selection and sorting with limited storage,' *Theoretical Computer Science*, vol. 12, no. 3, pp. 315–323, 1980. DOI: 10.1016/0304-3975(80)90061-4.

[52] S. Muthukrishnan, 'Data streams: Algorithms and applications,' *Foundations and Trends in Theoretical Computer Science*, vol. 1, no. 2, pp. 117–236, 2005. DOI: 10.1561/0400000002.

[53] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, R. Balasubramonian, A. Jain, C. Lin, L. A. Barroso, U. Hölzle, P. Ranganathan, J. Szefer, T. M. Aamodt, W. W. L. Fung and T. G. Rogers, 'A Primer on Memory Consistency and Cache Coherence, Second Edition,'

[54] V. Q. Ngo, M. Hilgendorf and M. Papatriantafilou, 'ReSketch: A mergeable, partitionable, and resizable sketch,' *Under Submission*, 2026.

[55] V. Q. Ngo and M. Papatriantafilou, 'Cuckoo Heavy Keeper and the Balancing Act of Maintaining Heavy Hitters in Stream Processing,' en, *Proceedings of the VLDB Endowment*, vol. 18, no. 9, pp. 3149–3161, May 2025. DOI: 10.14778/3746405.3746434.

[56] R. Pagh and F. F. Rodler, 'Cuckoo hashing,' *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004. DOI: https://doi.org/10.1016/j.jalgor.2003.12.002.

[57] Redis, *Top-k*, Accessed: 2025, 2024.

[58] A. Rinberg, A. Spiegelman, E. Bortnikov, E. Hillel, I. Keidar, L. Rhodes and H. Serviansky, 'Fast concurrent data sketches,' *ACM Transactions on Parallel Computing*, vol. 9, no. 2, 2022. DOI: 10.1145/3512758.

[59] P. Roy, A. Khan and G. Alonso, 'Augmented sketch: Faster and more accurate stream processing,' in *Proceedings of the 2016 international conference on management of data*, ser. Sigmod '16, Association for Computing Machinery, 2016, pp. 1449–1463. DOI: 10.1145/2882903.2882948.

[60] Q. Shi, Y. Xu, J. Qi, W. Li, T. Yang, Y. Xu and Y. Wang, 'Cuckoo Counter: Adaptive Structure of Counters for Accurate Frequency and Top-k Estimation,' *IEEE/ACM Transactions on Networking*, vol. 31, no. 4, pp. 1854–1869, Aug. 2023. DOI: 10.1109/TNET.2022.3232098.

[61] M. Stonebraker, U. Çetintemel and S. Zdonik, 'The 8 requirements of real-time stream processing,' *SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005. DOI: 10.1145/1107499.1107504.

[62] C. Stylianopoulos, I. Walulya, M. Almgren, O. Landsiedel and M. Papatriantafilou, 'Delegation sketch: A parallel design with support for fast and accurate concurrent operations,' in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–16. DOI: 10.1145/3342195.3387542.

[63] A. Toshniwal, S. Tanber, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal and D. Ryaboy, 'Storm@Twitter,' in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM, 2014, pp. 147–156. DOI: 10.1145/2588555.2595641.

[64] J. S. Vitter, 'Random sampling with a reservoir,' *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985. DOI: 10.1145/3147.3165.

[65] M. N. Wegman and J. L. Carter, 'New hash functions and their use in authentication and set equality,' *Journal of Computer and System Sciences*, vol. 22, no. 3, pp. 265–279, Jun. 1981. DOI: 10.1016/0022-0000(81)90033-7.

[66] D. P. Woodruff, 'Sketching as a tool for numerical linear algebra,' *Foundations and Trends in Theoretical Computer Science*, vol. 10, no. 1–2, pp. 1–157, 2014. DOI: 10.1561/0400000060.

[67] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi and X. Li, 'HeavyGuardian: Separate and Guard Hot Items in Data Streams,' in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18, New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 2584–2593. DOI: 10.1145/3219819.3219978.

[68] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li and S. Uhlig, 'Elastic sketch: Adaptive and fast network-wide measurements,' in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 561–575. DOI: 10.1145/3230543.3230544.

[69] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen and X. Li, 'HeavyKeeper: An Accurate Algorithm for Finding Top- k Elephant Flows,' *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, Oct. 2019. DOI: 10.1109/TNET.2019.2933868.

[70] X. Zhu, G. Wu, H. Zhang, S. Wang and B. Ma, 'Dynamic Count-Min Sketch for Analytical Queries Over Continuous Data Streams,' in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, Bengaluru, India: IEEE Computer Society, Dec. 2018, pp. 225–234. DOI: 10.1109/HiPC.2018.00033.

Part II

Main Chapters

Chapter A

Cuckoo Heavy Keeper and the Balancing Act of Maintaining Heavy Hitters in Stream Processing

V. Q. Ngo, M. Papatriantafilou

Proceedings of the VLDB Endowment, Vol. 18, No. 9, pp. 3149–3161, 2025.

DOI: https://doi.org/10.14778/3746405.3746434

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 trade-offs 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.

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 [27]. Besides, it is often needed by downstream applications also to return the estimated heavy hitters' frequencies. Corporations such as AT&T [9], Google [32], and Cloudflare [7] extensively use heavy-hitter detection solutions for network traffic analysis [37], [40], [41], anomaly detection [22], and iceberg query processing [4], [16]. These algorithms are also implemented in analytics platforms including Druid, Redis, and Databricks [3], [12], [33]. 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 [18], 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* [27], *LossyCounting* [25], and *Space-Saving* [26] 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* [10] and *CountSketch* [6] 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* [39], *ElasticSketch* [40], *AugmentedSketch* [36], *CuckooCounter* [37], *Topkapi* [24], and *HeavyKeeper* [41] 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* [18]. While there is some recent work realizing such needs [20], [34], [35], [38], parallelizing ϵ-ϕ-*heavy hitters* algorithms has focused mainly on parallelizing insertions (cf. [24], [42]), with only one exception [21], 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 [36], [39], [40] — 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* [38] (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 [30], 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.

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 [27] as follows:

Table 1: Notation Summary for Heavy Hitters Problem

Notation	Description
S	Input stream of item tuples $(a_1, \ldots, a_t, \ldots)$
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

ϵ-ϕ-heavy hitters: Given a stream $S = (a_1, \ldots, a_t, \ldots)$ 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 $e \in \hat{R}$, $\mid f(e) - \hat{f}(e) \mid \leq \epsilon N$ (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* [27], *LossyCounting* [25], and *Space-Saving* [26] 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 [8].

Sketch-based algorithms, such as the *CountSketch* and *Count-MinSketch* [6], [10], 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 [8].

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* [8], *HeavyKeeper* [41], and *Topkapi* [24], or by adopting faster hashing schemes like *cuckoo hashing* [31] in *CuckooCounter* [37].

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* [39], *ElasticSketch* [40], and *AugmentedSketch* [36] 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* [39] and *HeavyKeeper* [41]. This technique modifies how counter values are updated when hash collisions occur. Unlike the key-value counter in the Frequent algorithm [27] 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* [39] and *ElasticSketch* [40]. However, these techniques can show opposite effects under different conditions.

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

In lower-skew datasets, *HeavyGuardian* and *ElasticSketch* may suffer accuracy loss due to frequent collisions, and *AugmentedSketch* [36] 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. §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

[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 [31]), which guarantees distinct candidate locations.

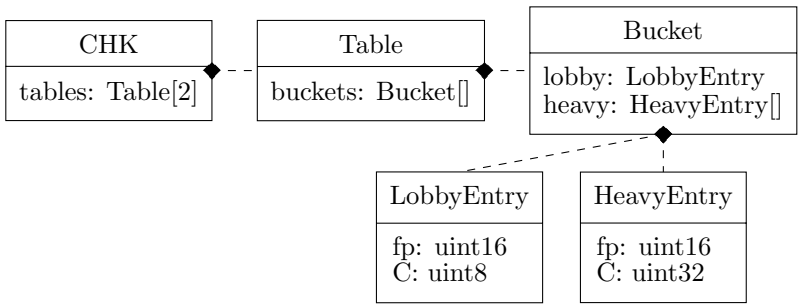


Figure 1: *Cuckoo Heavy Keeper* consists of two tables of buckets. Each bucket has one lobby entry to filter infrequent items and multiple heavy entries to maintain heavy-hitter candidates.

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 [36], [39], [40] 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 [36], [39], [40], 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* [31] 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* [37] 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

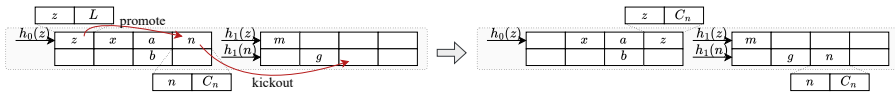


Figure 2: lobby item z replaces the minimum-counter item n in the heavy part, which then triggers relocation via cuckoo hashing. The process may continue recursively until finding an empty slot, or reaching `MAX_KICKS`, or encountering a below-threshold item.

efficiency, data movement patterns, and computational costs of floating-point arithmetic. These factors, however, significantly influence algorithms' running time (cf. [14] 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 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 [13], [17]. 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* [15], 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. [15]). 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

Algorithm 1: Cuckoo Heavy Keeper - Main Operations

```
1  Procedure Update(e, w)
2      N ← N + w
3      fp, idx_0, idx_1 ← 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 f̂(e)
7      // Found and updated in lobby
8      if CheckAndUpdateLobby(fp, idx_0, idx_1, w) then
9          return f̂(e)
       // If empty lobby exists
10     if exists i ∈ {0, 1} such that T[i][idx_i].lobby is empty then
11         Insert ⟨fp, w⟩ into empty T[i][idx_i].lobby
12         if w ≥ L then
13             TryPromote(T[i][idx_i])
14         return f̂(e)
15     // Count with exponential decay
16     i ← fp mod 2
17     C_new ← DecayCounter(T[i][idx_i].lobby.C, w)
18     if C_new = 0 then
19         Update T[i][idx_i].lobby with ⟨fp, w − de[C]⟩
20     else
21         T[i][idx_i].lobby.C ← C_new
22     if T[i][idx_i].lobby.C ≥ L then
23         TryPromote(T[i][idx_i])
24     return f̂(e)
25 Procedure f-Query(e)
26     fp, idx_0, idx_1 ← GenerateFpAndIndexes(e)
27     return f̂(e) if found, 0 otherwise
```

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 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* ($fp \bmod 2$) 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 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.

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.

[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 ← fingerprint(e), idx_0 ← hash(e) mod B
3      idx_1 ← (hash(fp) ⊕ idx_0) mod 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 ≥ 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 ← 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 ← de[C] − de[C − 1]
25         prob ← w/min_decay
26         return (Random(0,1) < prob) ? C − 1 : C
27     if w ≥ de[C] then
28         return 0
29     return largest i where de[i] + w ≥ 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 ← smallest entry in bucket.heavy
34     if Random(0,1) < (bucket.lobby.C − L)/(min.C − L) then
35         evicted_item ← min
36         min.fp ← bucket.lobby.fp
37         Clear bucket.lobby and Kickout(evicted_item)
38     else
39         bucket.lobby.C ← L
40 Procedure Kickout(entry)
41     for kicks ← 1 to MAX_KICKS do
42         if entry.C < φN 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
```

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 [33]).

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 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

[3] Our microbenchmarks [30] 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 1 formula calculation.

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 2 (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:*

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

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.

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: $f(e) \geq Z \cdot \big(Y_{h(e)} - f(e)\big) + L$ 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 3 (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 2). 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:

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

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):

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

For the overestimation bound:

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

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* [11]. 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,

[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.

this approach requires querying all thread-local data structures to collect heavy hitters, which can be inefficient for high-performance scenarios. For example, *Topkapi* [24] 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.

6.1.3 Peer-collaborative.

The peer-collaborative design [38] 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* [21], 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

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.

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

We now describe first how the two algorithms perform insertion-delegations and f-queries, extending [38]. 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.

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 4). 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 [28]) 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 3: Parallel Cuckoo Heavy Keeper Wrapper

```
1  Procedure Update(e, w)
2      ctid, otid ← current thread ID, owner(e) // Domain splitting
3      Add ⟨e, w⟩ to B_ctid[otid] // Buffer for delegated processing
4      if |B_ctid[otid]| ≥ MAX_BUF or B_ctid[otid][e] ≥ 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 ← 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 ← 0 to P − 1 do
19         if PQ_ctid[tid] has unprocessed f-query e then
20             count ← 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
```

6.2.2 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.

Algorithm 4: mCHK-I operations

```
1  Procedure ProcessPendingUpdates
2      ctid ← current thread ID
3      if cannot acquire lock on ctid's data then
4          return
5      while Q_ctid has unprocessed references do
6          B_ref ← get next unprocessed reference from Q_ctid
7          foreach ⟨e, w⟩ ∈ B_ref do
8              CHK_ctid.Update(e, w) // Delegated updates
9              Atomic add w to N_processed // Track stream size
10         Mark B_ref as processed in Q_ctid
11     release lock on thread ctid's data
12 Procedure hh-Query
13     R̂, remaining, made_progress ← ∅, P, false
14     while remaining > 0 do
15         made_progress ← false
16         for tid ← 0 to P − 1 do
17             if tid's data is not scanned and can acquire lock then
18                 cand ← CHK_tid.hh-Query()
19                 foreach ⟨e, count⟩ ∈ cand do
20                     if count ≥ φN_processed then
21                         Add ⟨e, count⟩ to R̂
22                 Mark thread tid's data as scanned and release lock
23                 remaining, made_progress ← remaining − 1, true
24         if remaining > 0 and not made_progress then
25             ProcessPendingUpdates
26     return R̂
```

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 [23]) 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

Algorithm 5: mCHK-Q operations

```
1  Procedure ProcessPendingUpdates
2      ctid ← current thread ID
3      while Q_ctid has unprocessed references do
4          B_ref ← get next unprocessed reference from Q_ctid
5          foreach ⟨e, w⟩ ∈ B_ref do
6              count ← CHK_ctid.Update(e, w) // Delegated updates
7              Atomic add w to N_processed // Track stream size
8              if count ≥ φN_processed then
9                  Update ⟨e, count⟩ in HH // Maintain global HH
10         Mark B_ref as processed
11 Procedure hh-Query
12     R̂ ← ∅
       // HH: Global concurrent hash table of heavy hitters
13     foreach entry position i in HH do
14         repeat
15             ⟨e1, count1⟩ ← HH[i] // First read
16             ⟨e2, count2⟩ ← HH[i] // Second read for consistency
17         until e1 = e2 and count1 = count2 // double collecting
18         if count1 ≥ φN_processed then
19             Add ⟨e1, count1⟩ to R̂
20     return R̂
```

($\mathbb{B}(e) \leq P \times MAX\_W$, since each thread can buffer at most MAX_W weight for any item).

Theorem 4 (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 3 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 Xeon E5-2695 (NUMA dual-socket)	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

Data sets: We used three datasets: (1) *CAIDA_L*, a real-world dataset of source IPs (skewness $\sim$ 1.2); (2) *CAIDA_H*, a real-world dataset of source ports (skewness $\sim$ 1.5). Both are derived from the CAIDA Anonymized Internet Traces 2018 [5], a broadly used benchmark in heavy-hitter detection literature [37], [40], [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 [29], [38].

Baselines: We compared *CHK* against representative state-of-the-art heavy-hitter detection algorithms: *Space-Saving* (SS) [26], *Count-MinSketch* (CMS) [10], *AugmentedSketch* (AS) [36] and *HeavyKeeper* (HK) [41], all implemented in C++, compiled with -O2 and available in our repository [30]. 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 [13], [15], [17] 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 [15] and decay factor $b = 1.08$ following [41].

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

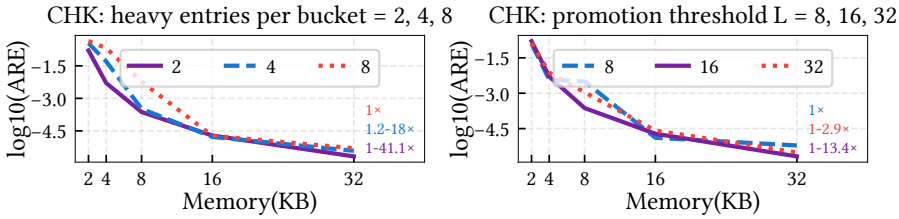


Figure 3: Sensitivity to decay base and heavy entries per bucket

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 [5], 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. [19]), 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.

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.

7.1.2 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

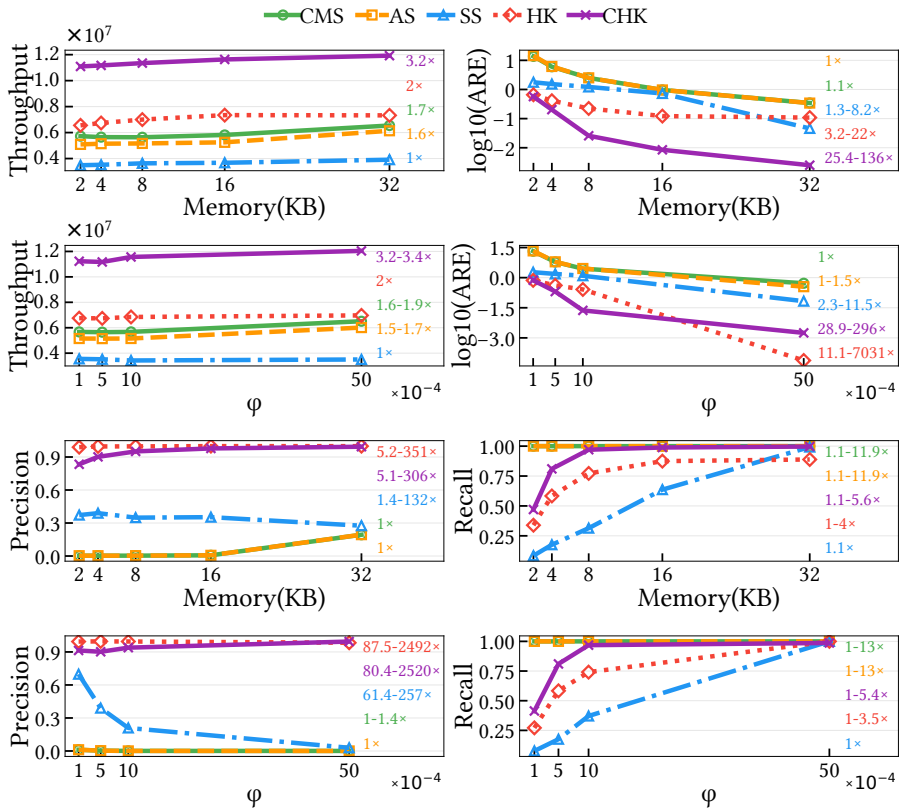


Figure 4: Plots show performance of sequential algorithms on *CAIDA_L* dataset for throughput, log_{10}(ARE), precision, and recall across varying memory, and ϕ. See §7.1.1 for calculation details.

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 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.

7.1.3 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,

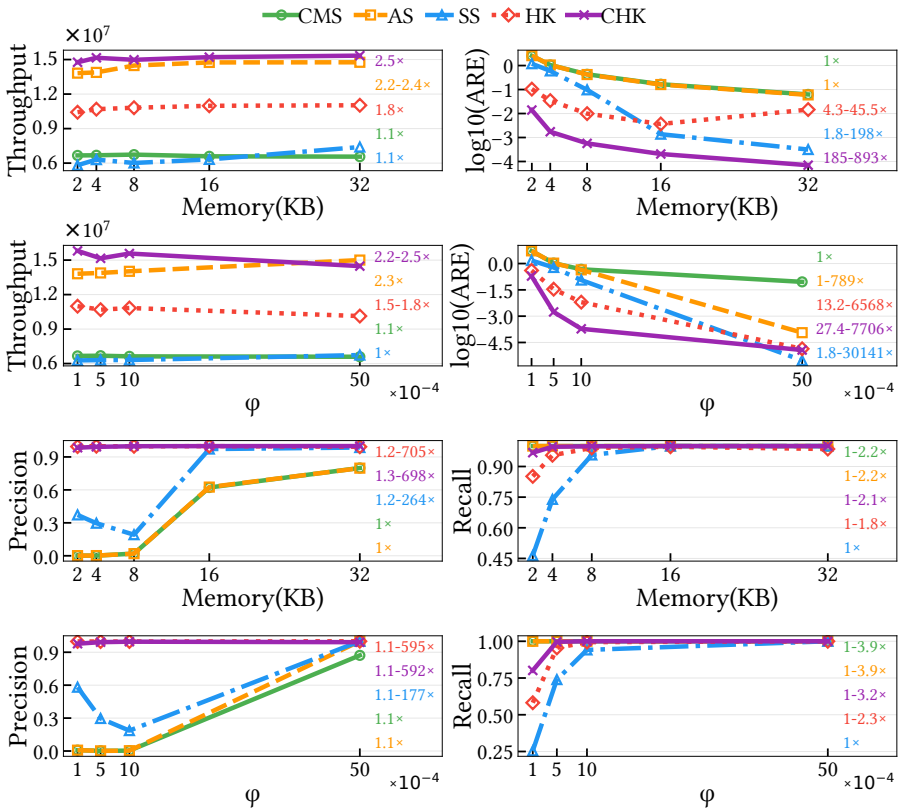


Figure 5: Plots show performance of sequential algorithms on *CAIDA_H* dataset for throughput, log_{10}(ARE), precision, and recall across varying memory, and ϕ. See §7.1.1 for calculation details.

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.

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-

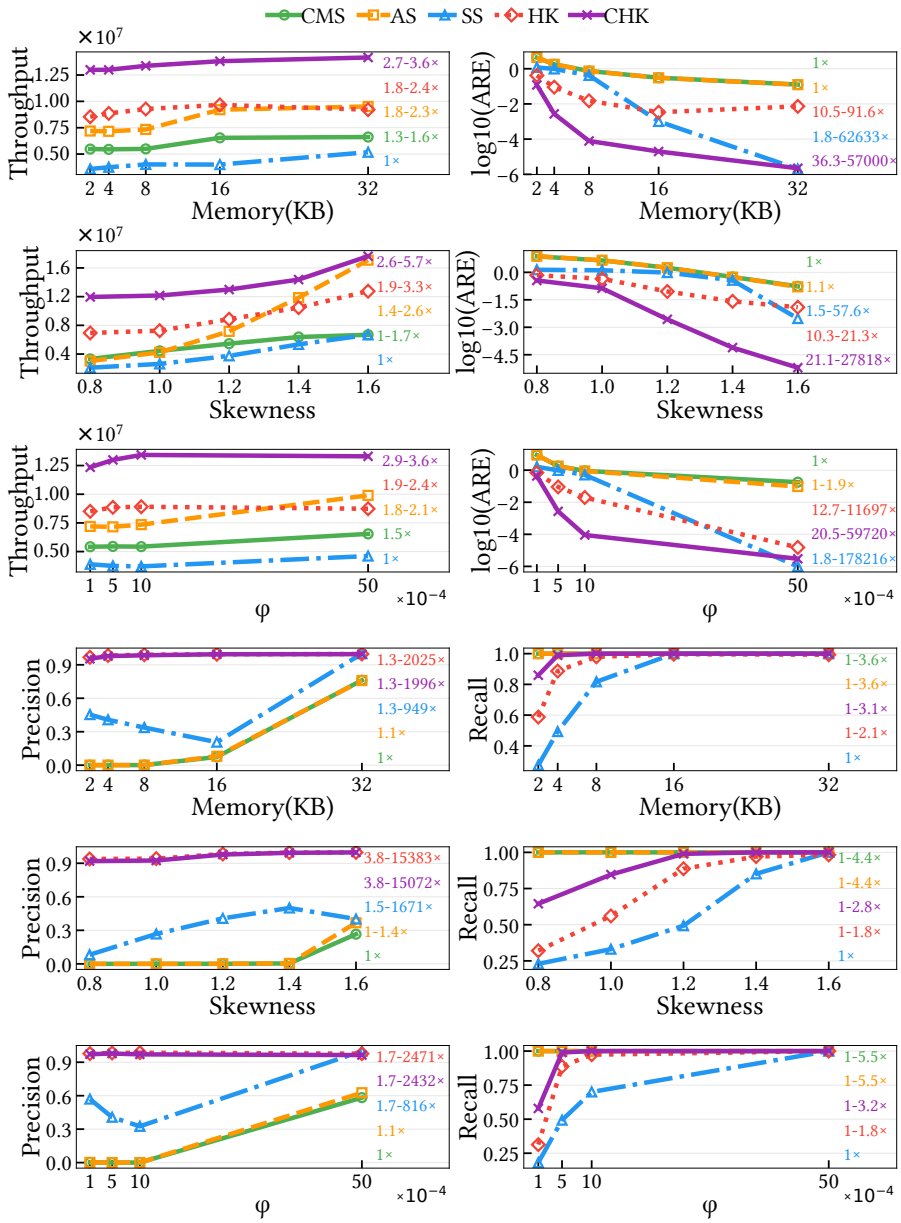


Figure 6: Plots show performance of sequential algorithms on *Synthetic data* for throughput, log_{10}(ARE), precision, and recall across varying skewness, memory, and ϕ. See §7.1.1 for calculation details.

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 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.

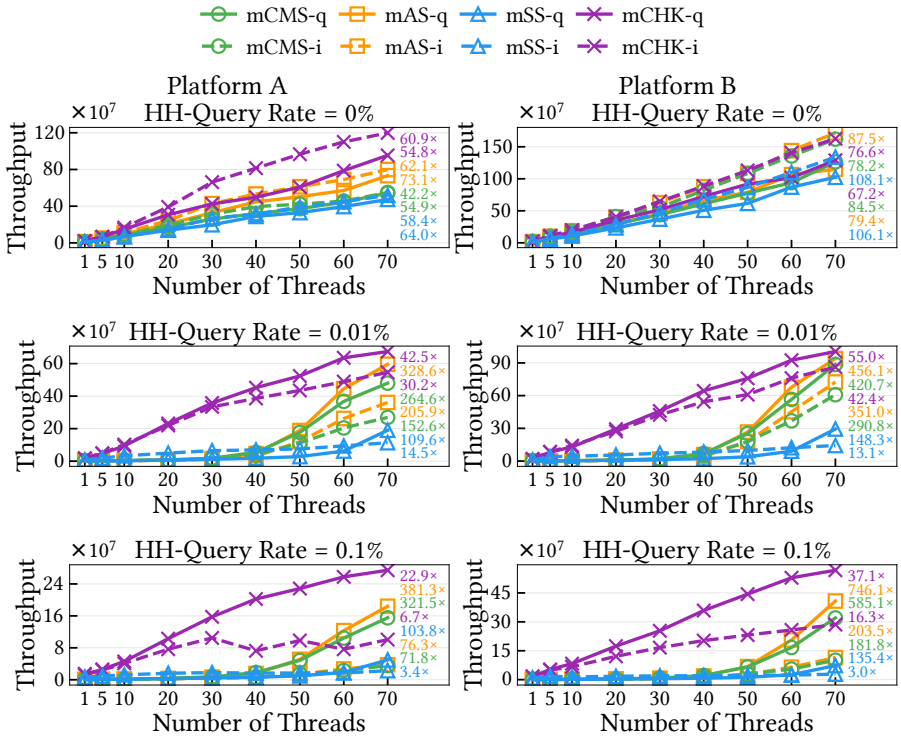


Figure 7: Plots show throughput and relative improvement compared to single-thread implementation of parallel variants on *CAIDA_H* across varying hh-query rates, thread counts, and platforms.

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.

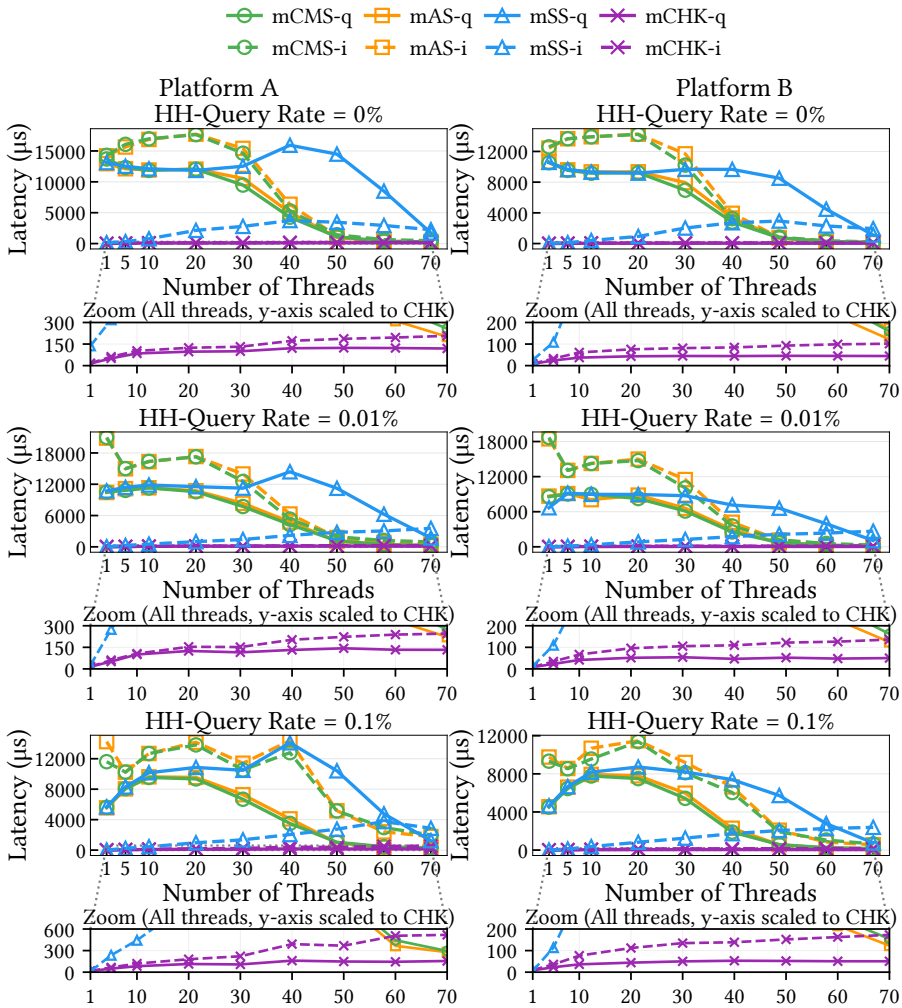


Figure 8: Plots show hh-query latency of parallel variants on *CAIDA_H* across varying hh-query rates, thread counts, platforms.

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 [20], [21], [38]), 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.

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 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.

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; 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".

Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit, 'Atomic snapshots of shared memory,' *Journal of The Acm*, vol. 40, no. 4, pp. 873–890, 1993. DOI: 10.1145/153724.153741.

[2] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei and K. Yi, 'Mergeable summaries,' in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, ser. PODS '12, New York, NY, USA: Association for Computing Machinery, May 2012, pp. 23–34. DOI: 10.1145/2213556.2213562.

[3] Apache Druid, *Topn queries - apache druid documentation*, https://druid.apache.org/docs/latest/querying/topnquery/, Accessed: 2025, Apache Software Foundation, 2024.

[4] K. Beyer and R. Ramakrishnan, 'Bottom-up computation of sparse and iceberg CUBE,' *Sigmod Record*, vol. 28, no. 2, pp. 359–370, 1999. DOI: 10.1145/304181.304214.

[5] CAIDA, *The CAIDA UCSD Anonymized Internet Traces - 2018-03-15*, 2018.

[6] M. Charikar, K. Chen and M. Farach-Colton, 'Finding frequent items in data streams,' *Theoretical Computer Science*, Automata, Languages and Programming, vol. 312, no. 1, pp. 3–15, Jan. 2004. DOI: 10.1016/S0304-3975(03)00400-6.

[7] Cloudflare, *RakeLimit: A rate limiter for distributed systems*, 2020.

[8] G. Cormode and M. Hadjieleftheriou, 'Finding frequent items in data streams,' *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1530–1541, Aug. 2008. DOI: 10.14778/1454159.1454225.

[9] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck and D. Srivastava, 'Holistic UDAFs at streaming speeds,' in *Proceedings of the 2004 ACM SIGMOD international conference on management of data*, ser. Sigmod '04, Association for Computing Machinery, 2004, pp. 35–46. DOI: 10.1145/1007568.1007575.

[10] G. Cormode and S. Muthukrishnan, 'An improved data stream summary: The count-min sketch and its applications,' *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005. DOI: 10.1016/j.jalgor.2003.12.001.

[11] S. Das, S. Antony, D. Agrawal and A. E. Abbadi, 'CoTS: A scalable framework for parallelizing frequency counting over data streams,' in *2009 IEEE 25th international conference on data engineering (ICDE'09)*, 2009, pp. 1323–1326. DOI: 10.1109/ICDE.2009.231.

[12] Databricks, *Approx_top_k function - Databricks SQL reference*, 2024.

[13] M. Dietzfelbinger and C. Weidling, 'Balanced allocation and dictionaries with tightly packed constant size bins,' in *Proceedings of the 32nd international conference on automata, languages and programming*, ser. ICALP'05, Springer-Verlag, 2005, pp. 166–178. DOI: 10.1007/11523468_14.

[14] U. Drepper, 'What every programmer should know about memory,' *Red Hat, Inc*, vol. 11, no. 2007, p. 2007, 2007.

[15] B. Fan, D. G. Andersen, M. Kaminsky and M. D. Mitzenmacher, 'Cuckoo filter: Practically better than bloom,' in *Proceedings of the 10th ACM international on conference on emerging networking experiments and technologies*, ser. CoNEXT '14, Association for Computing Machinery, 2014, pp. 75–88. DOI: 10.1145/2674005.2674994.

[16] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani and J. D. Ullman, 'Computing iceberg queries efficiently,' in *Proceedings of the 24rd international conference on very large data bases*, ser. Vldb '98, Morgan Kaufmann Publishers Inc., 1998, pp. 299–310.

[17] N. Fountoulakis, M. Khosla and K. Panagiotou, 'The multiple-orientability thresholds for random hypergraphs,' in *Proceedings of the twenty-second annual ACM-SIAM symposium on discrete algorithms*, ser. Soda '11, Society for Industrial and Applied Mathematics, 2011, pp. 1222–1236.

[18] M. Garofalakis, J. Gehrke and R. Rastogi, Eds., *Data Stream Management: Processing High-Speed Data Streams* (Data-Centric Systems and Applications). Berlin, Heidelberg: Springer, 2016. DOI: 10.1007/978-3-540-28608-0.

[19] D. Harris, A. Rinberg and O. Rottenstreich, 'Compressing Distributed Network Sketches With Traffic-Aware Summaries,' *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 1962–1975, Jun. 2023. DOI: 10.1109/TNSM.2022.3172299.

[20] M. Hilgendorf and M. Papatriantafilou, 'LMQ-Sketch: Lagom Multi-Query Sketch for High-Rate Online Analytics,' in *39th International Symposium on Distributed Computing (DISC 2025)*, D. R. Kowalski, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 356, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 36:1–36:24. DOI: 10.4230/LIPIcs.DISC.2025.36.

[21] V. Jarlow, C. Stylianopoulos and M. Papatriantafilou, 'QPOPSS: Query and Parallelism Optimized Space-Saving for finding frequent stream elements,' *Journal of Parallel and Distributed Computing*, vol. 204, p. 105 134, Oct. 2025. DOI: 10.1016/j.jpdc.2025.105134.

[22] A. Lakhina, M. Crovella and C. Diot, 'Characterization of network-wide anomalies in traffic flows,' in *Proceedings of the 4th ACM SIGCOMM conference on internet measurement*, ser. Imc '04, Association for Computing Machinery, 2004, pp. 201–206. DOI: 10.1145/1028788.1028813.

[23] X. Li, D. G. Andersen, M. Kaminsky and M. J. Freedman, 'Algorithmic improvements for fast concurrent Cuckoo hashing,' in *Proceedings of the ninth european conference on computer systems*, ser. EuroSys '14, Association for Computing Machinery, 2014. DOI: 10.1145/2592798.2592820.

[24] A. Mandal, H. Jiang, A. Shrivastava and V. Sarkar, 'Topkapi: Parallel and fast sketches for finding top-K frequent elements,' in *Proceedings of the 32nd international conference on neural information processing systems*, ser. NIPS'18, Curran Associates Inc., 2018, pp. 10 921–10 931.

[25] G. S. Manku and R. Motwani, 'Approximate frequency counts over data streams,' *Proc. VLDB Endow.*, vol. 5, no. 12, p. 1699, 2012. DOI: 10.14778/2367502.2367508.

[26] A. Metwally, D. Agrawal and A. E. Abbadi, 'An integrated efficient solution for computing frequent and top-k elements in data streams,' *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1095–1133, Sep. 2006. DOI: 10.1145/1166074.1166084.

[27] J. Misra and D. Gries, 'Finding repeated elements,' *Science of Computer Programming*, vol. 2, no. 2, pp. 143–152, Nov. 1982. DOI: 10.1016/0167-6423(82)90012-0.

[28] A. Morrison and Y. Afek, 'Fast concurrent queues for x86 processors,' *SIGPLAN Not.*, vol. 48, no. 8, pp. 103–112, 2013. DOI: 10.1145/2517327.2442527.

[29] M. Newman, 'Power laws, Pareto distributions and Zipf's law,' *Contemporary Physics*, vol. 46, no. 5, pp. 323–351, 2005. DOI: 10.1080/00107510500052444.

[30] V. Ngo and M. Papatriantafilou, *Artifact of the paper: Cuckoo Heavy Keeper and the balancing act of maintaining heavy hitters in stream processing*, Sep. 2025. DOI: 10.5281/zenodo.15593109.

[31] R. Pagh and F. F. Rodler, 'Cuckoo hashing,' *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004. DOI: https://doi.org/10.1016/j.jalgor.2003.12.002.

[32] R. Pike, S. Dorward, R. Griesemer and S. Quinlan, 'Interpreting the data: Parallel analysis with sawzall,' vol. 13, no. 4, pp. 277–298, 2005. DOI: 10.1155/2005/962135.

[33] Redis, *Top-k*, Accessed: 2025, 2024.

[34] A. Rinberg and I. Keidar, 'Intermediate value linearizability: A quantitative correctness criterion,' *J. ACM*, vol. 70, no. 2, 2023. DOI: 10.1145/3584699.

[35] A. Rinberg, A. Spiegelman, E. Bortnikov, E. Hillel, I. Keidar, L. Rhodes and H. Serviansky, 'Fast concurrent data sketches,' *ACM Transactions on Parallel Computing*, vol. 9, no. 2, 2022. DOI: 10.1145/3512758.

[36] P. Roy, A. Khan and G. Alonso, 'Augmented sketch: Faster and more accurate stream processing,' in *Proceedings of the 2016 international conference on management of data*, ser. Sigmod '16, Association for Computing Machinery, 2016, pp. 1449–1463. DOI: 10.1145/2882903.2882948.

[37] Q. Shi, Y. Xu, J. Qi, W. Li, T. Yang, Y. Xu and Y. Wang, 'Cuckoo Counter: Adaptive Structure of Counters for Accurate Frequency and Top-k Estimation,' *IEEE/ACM Transactions on Networking*, vol. 31, no. 4, pp. 1854–1869, Aug. 2023. DOI: 10.1109/TNET.2022.3232098.

[38] C. Stylianopoulos, I. Walulya, M. Almgren, O. Landsiedel and M. Papatriantafilou, 'Delegation sketch: A parallel design with support for fast and accurate concurrent operations,' in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–16. DOI: 10.1145/3342195.3387542.

[39] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi and X. Li, 'HeavyGuardian: Separate and Guard Hot Items in Data Streams,' in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18, New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 2584–2593. DOI: 10.1145/3219819.3219978.

[40] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li and S. Uhlig, 'Elastic sketch: Adaptive and fast network-wide measurements,' in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 561–575. DOI: 10.1145/3230543.3230544.

[41] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen and X. Li, 'HeavyKeeper: An Accurate Algorithm for Finding Top- k Elephant Flows,' *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, Oct. 2019. DOI: 10.1109/TNET.2019.2933868.

[42] Y. Zhang, Y. Sun, J. Zhang, J. Xu and Y. Wu, 'An efficient framework for parallel and continuous frequent item monitoring,' *Concurrency and Computation: Practice and Experience*, vol. 26, no. 18, pp. 2856–2879, 2014. DOI: https://doi.org/10.1002/cpe.3182.

[43] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett, Z. Wang and B. Chen, 'H2O: Heavy-hitter oracle for efficient generative inference of large language models,' in *Proceedings of the 37th international conference on neural information processing systems*, ser. Nips '23, Curran Associates Inc., 2023.

Chapter B

ReSketch: A Mergeable, Partitionable, and Resizable Sketch

V. Q. Ngo, M. Hilgendorf, M. Papatriantafilou

Under submission.

Abstract

Tracking items' frequency in data streams is a fundamental problem with applications ranging from network monitoring to database query optimization, machine learning, and more. Sketches offer practical, sublinear-memory solutions that provide high-throughput updates and queries with provable accuracy approximation bounds. Furthermore, sketches are mergeable, which allows multiple ones of identical parameters to be combined into a single, representative sketch, which enables their use in parallel and distributed systems.

Still, there are limitations in known sketch designs that restrict their applicability in systems characterized by resource heterogeneity across nodes, workload fluctuation over time, and the need for efficient distributed data aggregation. We identify and formalize three critical properties that can address these limitations: *resizability*, *enhanced mergeability*, and *partitionability*. We propose ReSketch, a matrix-based sketch algorithmic design, which, through a combination of consistent hashing and quantile sketching, fused with a partition-aware hashing technique, leads to the ability to satisfy all three properties, with a beneficial memory-to-accuracy ratio. We propose an analysis methodology for dynamic sketches and apply it to investigate the costs and benefits of ReSketch, in conjunction with a detailed empirical study that also includes its time-associated behavior. As ReSketch is orthogonal to other matrix-based sketches, we expect it can enable them to support the aforementioned properties and, in turn, lead to new significant use cases for frequency estimation sketches in modern systems.

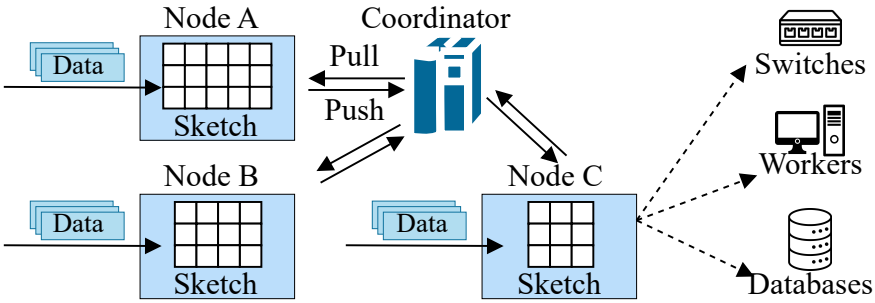


Figure 1: Distributed monitoring with heterogeneous nodes and dynamic workload maintaining sketches of varying sizes.

1 Introduction

Tracking item frequencies and derivative problems such as heavy hitter detection [7], [10], [29], [37], estimation of frequency moments [19], inner products [3], and range sums [16] are fundamental primitives for data-intensive systems. They are used for query optimization, approximate query processing, and join size estimation in databases [1], [30], [31]; identifying popular flows, DDoS mitigation, and change detection in networking [9], [18], [38]; load balancing and caching in distributed systems [32]; and more. Given the input rate of such systems, it is important to find algorithms that have favorable memory requirements and capabilities to process streams promptly in a single pass. Knowing that the exact solutions require memory at least linear to the number of distinct items in the data [15], and that many applications accept some approximation, a substantial volume of work focuses on succinct (sublinear) representations from which items' frequencies can be queried approximately.

A common approach achieving this is through frequency estimation sketches [7], [11], which provide (ε, δ)-approximation; i.e. item frequencies are estimated with a bounded approximation factor ε, with probability at least 1 - δ. They are commonly implemented as $w \times d$-matrices (width $\times$ depth), in which w influences ε, while d influences the probability bound δ. A valuable property of sketches is mergeability [2], meaning that multiple sketches of size $w \times d$ can be combined into one with the same size and (ε, δ) guarantee. This property makes sketches even more powerful, providing better scalability and flexibility, which enables broader use-cases such as distributed data processing and in-network aggregation [2]. However, as systems turn more *distributed*, *heterogeneous*, and *dynamic*, there is an increasing need for solutions that go beyond what conventional sketch designs and their mergeability can address.

The need for more flexible sketches. Let's consider an example distributed network monitoring system as illustrated in Figure 1, where network devices monitor traffic flows using sketches. Note that (i) Due to *heterogeneity* in device capabilities and *dynamicity* in resources and traffic, both the available memory and the needed space for sketching vary both across devices and over

Table 1: Analysis and state-of-the-art of sketch properties in focus: motivation, related work, and gaps.

Property	Motivation	Related Work	Gap
Resizability	Error bounds (ε, δ) are fixed at initialization, based on dimensions $w \times d$ and cannot be changed without initializing new sketches. *Expandability* can improve accuracy when resources become available; *shrinkability* can free memory for other tasks or reduce transmission overhead.	Some methods support expandability [41], others shrinkability [38], mostly at coarse granularity (doubling or halving the size) rather than fine-grained adjustments. Only Geometric Sketch (GS) [4] supports fine-grained expansion and memory release.	GS cannot shrink below its initial allocation, and its throughput reduces as it expands. If initialized too small (which allows it to shrink to a smaller size when needed), frequent expansion reduces throughput and degrades accuracy.
Enhanced Mergeability	• Conventional mergeability requires identical dimensions (cannot merge differently-sized sketches). • Cannot control the output size when merging to control precision. E.g., when p devices maintain sketches of size $w \times d$, they consume $p \times w \times d$ space but can only achieve single-sketch accuracy after merging.	Resizing the sketches before merging (*resize-then-merge*) seems like a straightforward solution. However, existing resizable sketches GS and DCMS [4], [41] still logically stack multiple fixed-size sketches internally, so resizing does not make them compatible for merging.	• No support for merging different-sized sketches, which implies scanning all sketches during queries, plus bookkeeping overhead. • Even when merging the same-sized sketches, cannot control the output size to preserve higher precision.
Partitionability	As workloads scale, partitioning traffic across multiple sketch instances (e.g., offloading a subset of keys and their counts to a new node) is essential for load balancing and parallel processing. Dynamic environments further require moving partitions (and their state) between nodes at runtime.	Partitioning streams to independent sketches improves throughput and accuracy [17], [19], [21], [29], [34]. Known approaches rely on *static partitioning* defined at initialization.	Cannot dynamically rebalance partitions or migrate state (historical data) between sketches.

time, which creates the need for adaptive memory allocation. (ii) For query processing, devices periodically send their sketches to a coordinator, which merges them (if possible) before querying or otherwise scans all of them; when sketch sizes are different, there is no known method to merge them, and all sketches must be scanned. (iii) As devices join or leave the network due to load balancing or reconfiguration, the system must redistribute monitoring responsibilities while preserving historical information.

These observations motivate three properties: ❶ **Resizability**—the ability to dynamically expand or shrink the sketch size at fine granularity while maintaining accuracy and throughput comparable to a sketch initialized at the target size; ❷ **Enhanced Mergeability**—the ability to combine sketches of different sizes and control the output size to preserve as much precision as the input sketches collectively offer; ❸ **Partitionability**—the ability to partition the sketch state itself (not just the input) to enable dynamic redistribution of monitoring responsibilities while preserving historical information. Table 1 summarizes the motivation, related work, and existing gaps for each property.

Insights. The observed limitations stem from a single, fundamental design choice inherent in customary sketches: the hash functions. For a sketch of size $w \times d$, existing designs employ d functions $h_1, \ldots, h_d$ drawn from a pairwise independent family. In the de facto standard approach, these functions determine the bucket index $h_i(e)$ for an item e via a modulo operation ($\pmod w$). This mapping is static and does not change over time, which is not flexible enough to support the aforementioned requirements. Consider resizing a sketch by changing its width from w to w'; there will be excessive need for moving contents of buckets to other ones, and, moreover, as the counters within a bucket are aggregations of counts, when a bucket's contents must be remapped due to resizing, merging, or partitioning, there is no information to guide the process. E.g., if a counter at a single bucket needs to be partitioned across two new buckets, known methods cannot determine how to partition its value because we do not know *which items are present* or *their individual counts* to map to the new locations.

Contributions. We identify and formalize three critical properties—*resizability*, *enhanced mergeability*, and *partitionability*—that extend the applicability of frequency estimation sketches, and propose ReSketch, a *sketch algorithmic design* that achieves all three. In particular, ReSketch builds on a novel coupling of *three key ideas*:

First, instead of using the conventional modulo-based bucket mapping, we transform each bucket's responsible domain from discrete to continuous using *consistent hashing* [22]. The hash space over which bucket responsibility is defined is conceptualized as a continuous logical ring (e.g., the interval $[0, 1)$). The sketch's buckets are then defined as contiguous, non-overlapping *segments* that partition the ring. An item is assigned to the single bucket whose segment covers the item's hash value. The hash function *mapping items to the ring remains fixed* for the sketch's lifetime. Structure-defining operations, such as resizing or partitioning, are performed not by changing the function, but by *adjusting the boundaries* that define the segments. These boundaries are initialized as points chosen uniformly at random on the ring

by pairwise independent hash functions, which ensures a balanced distribution across buckets.

Second, based on the above, redistributing items during resizing or partitioning reduces to identifying the portion of a segment (i.e., which *few* items) should be moved. Hence, we define a "sketch of sketches" design that maintains a per-bucket mergeable distribution summary (e.g., a quantile sketch) that approximates the distribution of items' hash values within that bucket. When a segment is resized or moved, we query that summary to estimate the counts corresponding to the portion of the segment being moved. Also, since the quantile summary is mergeable, it makes the aforementioned *resize-then-merge* strategy possible, thereby enabling both *resizability* and *enhanced mergeability*. The extra information in the quantile summary also enables a new, improved estimator for items' frequencies, which compensates for the extra memory used.

Third, to enable sketch partitioning, we introduce **Partition-Aware Hashing**. In sketches, an item is scattered to random buckets across rows. Consequently, simply partitioning the sketch by column index (e.g., assigning the first half of columns to one sketch and the rest to another) is ineffective: a single item's hash locations would span both partitions, which makes future updates be sent to multiple sketches. Our approach solves this by assigning every item a *fingerprint* that maps it to exactly one partition at any given time. These partitions correspond to segments in a global partition ring (i.e., different from the per-row rings discussed), with each segment corresponding to a sketch. During a sketch partition operation, an item may be reassigned to a new partition (sketch). Its history and newer updates can find the new partition easily through the *fingerprint* and its segment in the partition ring. Importantly, we recover this fingerprint directly from the stored hash values in the aforementioned quantile sketch using *modular multiplicative inverses*, which requires no additional storage overhead.

Besides these ideas and their synthesis forming ReSketch, we provide a methodology to analyse properties of dynamic sketches, proposing the concept of an *instance-provenance graph*, show analytical bounds for the algorithm, present an open-source repository with an implementation [28], as well as a comprehensive experimental evaluation on both real-world and synthetic datasets. The results demonstrate that ReSketch achieves high accuracy and competitive throughput even with small space needs, while providing the three desired properties, outperforming state-of-the-art baselines that only partly address some of these properties.

Importantly, because the proposed mechanisms are compatible with matrix-based sketches, other sketches can adopt this design to inherit resizability, enhanced mergeability, and partitionability. Moreover, ReSketch complements and extends recent works including [19], [21], [29], [34], which have shown that splitting the input domain and sketching each such partition separately yields orders-of-magnitude improved accuracy/memory ratio for frequency moments and frequent items sketching, while also enabling concurrency. ReSketch provides significant steps towards addressing the questions of how partitions of keys to such sketches can happen in general and with what dynamic properties. We expect ReSketch's approach to unlock significant new possibilities for

frequency estimation sketches in modern, dynamic systems.

Roadmap. The rest of the paper is organized as follows. section 2 provides preliminaries on frequency estimation sketches and their mergeability. section 3 formalizes the three properties. section 4 presents the design of ReSketch. section 5 provides theoretical analysis. section 6 presents experimental results. section 7 discusses other related work. Finally, section 8 concludes the paper.

2 Preliminaries

Count-Min Sketch. Count-Min Sketch (CMS) [11] is a probabilistic data structure that provides approximate frequency counts of items in a data stream. Let $S = \langle e_1, e_2, \ldots \rangle$ be a data stream where each element e_t is drawn from a universe $\mathcal{U}$. The true frequency of an item $e \in \mathcal{U}$, denoted $f(e)$, is the number of times it appears in the stream: $f(e) = |\{t \mid e_t = e\}|$. The total stream length is $N = \sum_{e \in \mathcal{U}} f(e)$. CMS uses a series of scalar counters arranged in a $w \times d$ array, where each row i corresponds to a hash function h_i drawn from a pairwise independent family. The de facto standard approach defines these functions as $h_i(x) = ((a_i x + b_i) \bmod p) \bmod w$, where p is a prime number and a_i, b_i are random coefficients. When an item e arrives, CMS increments the counter at position $h_i(e)$ in each row i. The estimated frequency of an item is calculated by taking the minimum count across its hashed positions: $\widehat{f}(e) = \min_{i=1}^{d} \text{count}[i][h_i(e)]$. Due to hash collisions, $\widehat{f}()$ can overestimate the actual count. However, by setting $d = \ln(1/\delta)$ and $w = \mathrm{e}/\varepsilon$ (where e is Euler's constant), CMS guarantees that with probability at least $1 - \delta$, the estimated frequency satisfies $\widehat{f}(e) \leq f(e) + \varepsilon N$ for any item e.

Algorithm 1: KLL Quantile Sketch (q_sketch)

```
// k:  Parameter controlling sketch size and accuracy
// Compactors:  List of arrays storing sampled values per level
Procedure UpdateKLL(value)
    Append value to the lowest level compactor (level 0)
    if compactor at level 0 is full then CompactKLL(0)
Procedure QueryCountInRange(start, end)
    estimated_count ← 0
    for each level l do
        for each value in compactor at level l do
            if start < value ≤ end then estimated_count += 2^l
    return estimated_count
Procedure QueryRank(value)
    return QueryCountInRange(−∞, value)
Procedure MergeKLL(other_sketch)
    for each level l do
        Add all items from other_sketch.Compactors[l] to this Compactors[l]
        if compactor at level l is full then CompactKLL(l)
Procedure CompactKLL(l)
    Sort items in Compactors[l]
    Randomly discard either the odd-indexed or even-indexed positions
    Move the remaining items to the compactor at level l + 1
    if compactor at level l + 1 is full then CompactKLL(l + 1)
```

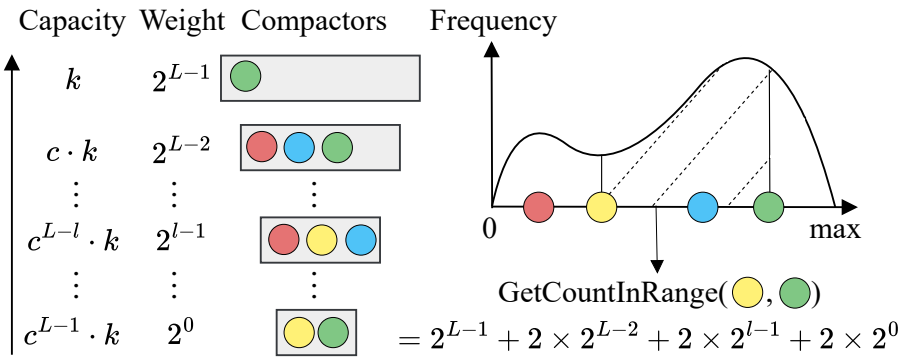


Figure 2: KLL sketch maintains multi-level compactors where items at level l have weight 2^{l-1}. The GetCountInRange operation estimates counts within any range, which generalizes both rank queries and single-item frequency estimation.

Quantile Summary. A quantile summary approximates the distribution of items from a totally-ordered universe $\mathcal{U}$. Given a stream of N items, the *rank* query of a value $e \in \mathcal{U}$, denoted `Rank(`e`)`, is the number of items in the stream that are less than or equal to e. A quantile query, specified by a fraction $\phi \in [0, 1]$, asks for the item e such that `Rank(`e`)` $\approx \phi N$. More generally, a quantile summary can support a `QueryCountInRange(`*start*, *end*`)` that estimates the number of items falling within a specified range (*start*, *end*], which generalizes rank queries.

The KLL sketch [23] is state-of-the-art, mergeable quantile summary that provides strong approximation error guarantees. It operates by maintaining a collection of L varying-capacity arrays of sampled items, called *compactors*, organized in multiple levels; although the logical height of the hierarchy grows logarithmically with the stream size, the sketch maintains a bounded number of arrays L by replacing the bottom levels with a weighted sampler when necessary. The core idea is that an item at level l has a weight of 2^{l-1}, representing exponentially more original data points than an item at a lower level, as illustrated in Figure 2. The operations of a KLL sketch are shown in Algorithm 1. When a new item arrives, `UpdateKLL` adds it to the compactor at the lowest level (level 0). If this addition causes the compactor to exceed its capacity, a `CompactKLL` operation is triggered, which sorts the compactor, randomly samples half of the items, and "promotes" them to the next level up. This process continues until the compactor is within its capacity limits. The `Rank(`e`)` of a value is estimated by summing the weights of all sampled items less than or equal to it. The accuracy and memory usage of the KLL sketch are controlled by a parameter k. For the mergeable version, setting $k = \mathcal{O}((1/\varepsilon)\sqrt{\log(1/\delta)})$ guarantees that the estimated rank is within an additive error of εN with probability at least $1 - \delta$, using $\mathcal{O}(k)$ space.

Mergeability. Formally, a summarization method is considered mergeable if, given two summaries $\mathcal{A}_1$ and $\mathcal{A}_2$ computed on datasets D_1 and D_2 respectively, there exists a `Merge` operation that produces a new summary $\mathcal{A}_{merged}$ that is

Table 2: Summary of Notations.

Symbol	Description
General Stream Notations	
S	Data stream, a sequence of items.
e_t,$\mathcal{U}$	An item from the item universe $\mathcal{U}$.
$f(e)$, $\widehat{f}(e)$	True and estimated frequency of an item e.
N	Total number of items in the stream.
KLL Sketch (Inner Sketch)	
k	Accuracy parameter controlling size and precision.
ε_{KLL}, δ_{KLL}	Rank error guarantee and failure probability parameters.
Compactors	Internal data structure for storing samples.
ReSketch Notations	
$\mathcal{A}$, w, d	A ReSketch instance, its width (buckets per row), and depth (rows).
ε, δ	Sketch approximation and probability parameters.
seeds	An array of d unique seeds for the hash functions.
rings	An array of d consistent hash rings.
buckets	The $d \times w$ array holding the sketch's buckets.
buckets$[i][j]$	Bucket at row i, column j; holds a counter and a q-sketch.
.count	The counter for items mapped to a bucket.
.q_sketch	The inner KLL sketch within a bucket.
Model for Mergeable, Redistributable, and Resizable Sketches	
$\mathcal{A}_{id}$	A sketch instance with a unique identifier *id*.
S_{id}	The logical stream associated with instance $\mathcal{A}_{id}$.
$f_{id}(e)$, $\widehat{f}_{id}(e)$	True and estimated frequency of item e in stream S_{id}.
N_{id}	The total number of items of the logical stream S_{id}.

valid for the union dataset $D_1 \cup D_2$ without needing to re-process the original data [2]. The merged summary $\mathcal{A}_{merged}$ should satisfy the same guarantees as the individual summaries (e.g., same bounds ε, δ). Many summaries exhibit some form of mergeability. For frequency estimation, these include the Count-Min Sketch [11] and Count-Sketch [7], which require sketches to have the same width w and depth d to merge. For heavy hitter detection, counter-based algorithms like Misra-Gries [26] and SpaceSaving [25] are mergeable [2]. For quantile estimation, KLL [22] and t-digest [14] are mergeable regardless of their configured size parameters (e.g., parameter k for KLL). However, to the best of our knowledge, the resulting error bounds for merging sketches with different k parameters are not formally established; existing analysis is limited to merging sketches of identical size parameters. For membership testing, the Bloom filter [5] is also mergeable by performing a bitwise OR operation on the filter arrays.

3 Problem Description

Let $\mathcal{A}(w, d)$ denote a $w \times d$ frequency estimation sketch (we drop (w, d) when the context does not require it). When discussing multiple sketches, we use subscripts to distinguish them (e.g., $\mathcal{A}_1$, $\mathcal{A}_2$). Operations on sketches are categorized into two types: (1) *Data Manipulation:* `Update`$(\mathcal{A}, e)$ and

`Query`$(\mathcal{A}, e)$ operate on a single sketch to process items and estimate frequencies. (2) *Structure-Defining:* `Resize`, `EnhancedMerge`, and `Partition` create new sketches from existing ones, potentially changing their dimensions or combining their states. Table 2 summarizes the notations used in this paper. We now define the properties of structure-defining operations.

Definition 1 (Resizability)**.** *A sketch $\mathcal{A}(w, d)$ is resizable if it supports the operation* `Resize`*$(\mathcal{A}, w')$*[1]*: given $\mathcal{A}(w, d)$, this operation produces $\mathcal{A}'(w', d)$. If $w' > w$, the operation expands the sketch, resulting in an improved error bound $\varepsilon' < \varepsilon$. If $w' < w$, it shrinks the sketch. A sketch is truly resizable if w' can be any positive integer.*

Definition 2 (Enhanced Mergeability)**.** *A sketch supports enhanced mergeability if it can merge sketches of different sizes into a new sketch of a user-defined target; i.e., the operation* `EnhancedMerge`*$(\mathcal{A}_1, \mathcal{A}_2)$ given two sketches $\mathcal{A}_1(w_1, d)$ and $\mathcal{A}_2(w_2, d)$, produces $\mathcal{A}_{merged}(w_1 + w_2, d)$, followed by a resize if $w' \neq w_1 + w_2$. WLOG, operations are defined for two sketches, as more sketches can be merged by successive pair-wise operations. The operation should preserve accuracy, i.e., when $w' = w_1 + w_2$, querying an item e from the resulting sketch yields accuracy comparable to the aggregated result of querying the ancestor sketches.*

Definition 3 (Partitionability)**.** *A sketch is partitionable if it supports the operation* `Partition`*$(\mathcal{A}, w_1, w_2)$: given $\mathcal{A}(w, d)$, and target widths w_1 and w_2 s.t. $w_1 + w_2 = w$, the operation produces $\mathcal{A}_1(w_1, d)$ and $\mathcal{A}_2(w_2, d)$, to operate on disjoint data subsets $\mathcal{U}_1$ and $\mathcal{U}_2$ of $\mathcal{U}$, while the historical state (i.e., info for frequency estimation) corresponding to any item $e \in \mathcal{U}_i$ is migrated to $\mathcal{A}_i$. The migration should preserve accuracy, i.e. querying an item e from its responsible sketch $\mathcal{A}_i$ yields accuracy comparable to querying the original $\mathcal{A}$.*

4 ReSketch

Building upon the rationale outlined in the introduction, this section presents ReSketch, a frequency estimation sketch algorithmic design that achieves *resizability*, *enhanced mergeability*, and *partitionability*. We first introduce the data structure's layout. Next, we describe a core mechanism in ReSketch, that enables all structure-defining operations, namely, the method for redistributing bucket contents using *consistent hashing* and *quantile estimation*. After detailing the data manipulation operations, we present the structure-defining ones (`Resize`, `EnhancedMerge`, and `Partition`), along with the *partition-aware hashing* scheme to support them.

4.1 ReSketch Data Structure Layout

Similar to a Count-Min Sketch, ReSketch is built on a $d \times w$ matrix, with a "sketch of sketches" architecture, where buckets are composite structures rather

[1]We use the terms `Expand` and `Shrink` when distinction between expanding ($w' > w$) and shrinking ($w' < w$) behaviors is required.

Algorithm 2: ReSketch – Update, Query Operations

```
// seeds[d]:  Array of d unique seeds for hashing
// rings[d]:  d consistent hash rings, sorted by hash point
// buckets[d][w]:  2D array of {count, q_sketch} structs
Procedure FindBucket(e_hash, ring)
    Search for first (p, id) in ring where p ≥ e_hash
    if no such pair found then return first id in ring
    else return id
Procedure Update(e)
    for i ← 0 to d − 1 do
        y ← Hash(e, seeds[i])
        id ← FindBucket(y, rings[i])
        buckets[i][id].count += 1
        buckets[i][id].q_sketch.UpdateKLL(y)
Procedure Query(e)
    min_count ← ∞; estimates[d] ← []
    for i ← 0 to d − 1 do
        y ← Hash(e, seeds[i])
        id ← FindBucket(h, rings[i])
        row_kll_est ← buckets[i][id].q_sketch.QueryFrequency(y)
        estimates[i] ← row_kll_est
    return median(estimates)              // Estimator:  median of KLL estimates
```

than simple counters. As detailed in Table 2, each bucket contains: (1) A primary counter, $buckets[i][j].count$, tracking the total number of items mapped to the bucket. (2) A compact secondary quantile sketch, $buckets[i][j].q\_sketch$, *summarizing the distribution of hash values of the items*[2] mapped to that bucket. For the latter, we use a KLL sketch [23] (with the same parameter k each) due to its analytical guarantees, and efficient algorithmic implementations in existing works [20].

4.2 Redistributing Bucket Contents

As discussed in the introduction, conventional sketches cannot redistribute counts during structural changes because they lack information about which items need to be moved and their individual counts. ReSketch addresses this through two key mechanisms: *consistent hashing* transforms bucket domains from discrete to continuous, which allows to know *which portion of items* needs to move when boundaries change (e.g., during a `Resize`); and *per-bucket quantile sketches* estimate *how many items* fall into that portion. Together, these enable bucket content redistribution for all structure-defining operations (`Resize`, `EnhancedMerge`, and `Partition`). We now describe how this redistribution works in detail.

Consistent Hashing for Bucket Selection. In the de facto standard implementations of Count-Min sketch, the pairwise independent hash functions are realized for each row i as $h_i(x) = ((a_i x + b_i) \bmod p) \bmod w$. The inner term $(a_i x + b_i) \bmod p$ generates pairwise-independent hash values in a hash space $[0, p-1]$, while the outer term $\bmod w$ serves as a *bucket-mapping function* that

[2] Note it keeps hash values, not items, allowing to track the distribution over the continuous hash domain, essential for redistribution, as detailed in subsection 4.4.

Algorithm 3: ReSketch – Redistribute Row

```
1  Procedure RedistRow(in_ring, in_buckets, out_ring)
2      out_buckets ← new array of empty buckets for out_ring
3      all_points ← sorted list of unique hash points from in_ring and out_ring
       // Iterate through the disjoint ranges
4      for i ← 0 to length of all_points − 2 do
5          start, end ← all_points[i], all_points[i + 1]
           // For each, find its source and destination
6          in_id ← FindBucket(start, in_ring)
7          out_id ← FindBucket(start, out_ring)
8          in_bucket ← in_buckets[in_id]
9          count ← in_bucket.q_sketch.QueryCountInRange(start, end)
10         if count > 0 then
               // Move the data for this range to its destination
11             out_buckets[out_id].count += count
12             sub_sketch ← in_bucket.q_sketch.FilterKLL(start, end)
13             out_buckets[out_id].q_sketch.MergeKLL(sub_sketch)
14     return out_buckets
```

reduces this range to the smaller range of sketch bucket indices $[0, w-1]$. ReSketch retains the pairwise-independent hash generation but replaces the rigid outer bucket mapping function with consistent hashing [22]. This conceptually maps the generated hash values to a circular ring in each row, partitioned into w contiguous segments, each segment corresponding to a bucket in the row (illustrated in Figure 3, Figure 4, and Figure 5). As shown in the `Update` operation (Algorithm 2), the hash generation call `Hash`(e, *seeds*[i]) (the red highlighted line in Algorithm 7) computes $y = (a_i \cdot e + b_i \bmod p)$, which conceptually places the item on this ring. The w buckets in each row correspond to the segments defined by boundary points stored in the sorted array $rings[i]$; initially, these are generated by sampling w random points from the hash space. To process an item, we determine the bucket responsible for the segment containing y using `FindBucket`, which searches the sorted boundary points in $rings[i]$ to locate the first one greater than or equal to y, i.e., the upper bound of the item's assigned segment and thus the correct bucket. The key advantage is that the pairwise-independent hash generation function `Hash` for each row remains unchanged throughout the sketch's lifetime. All structural changes, such as resizing, are handled by adjusting the segment boundaries on the ring, not by altering the hash functions. The statistical properties of this hashing scheme are analyzed in section 5.

Quantile Estimation for Count Redistribution. The role of the secondary quantile sketch is to redistribute aggregated counts when bucket boundaries are modified. As items are added via `Update`, each bucket's inner quantile sketch (*q_sketch*) is also updated with the item's hash value y using `UpdateKLL` to maintain a compact summary of the distribution of consistent hash values of the bucket's items. A quantile sketch enables to estimate the number of items whose hash values fall within any range (*start*, *end*) by using `QueryCountInRange`. When a structural change occurs (e.g., during `Resize` or `EnhancedMerge`), the segment boundaries on the consistent hash ring are modified, creating new intervals. To handle this, the function `RedistRow` (Algorithm 3) first creates a unified, sorted list of all boundary points, i.e., the union of the original and the

Algorithm 4: Extended KLL Quantile Sketch (*q_sketch*)

```
1  Procedure QueryFrequency(e_hash)
2      estimated_count ← 0
3      for each level l do
4          for each value in compactor at level l do
5              if value = e_hash then estimated_count += 2^l
6      return estimated_count
7  Procedure FilterKLL(start, end)
8      new_sketch ← new KLLSketch()
       // Filter values within specified range to build new KLL
9      for each level l of this sketch's Compactors do
10         for each value in compactor at level l do
11             if start < value ≤ end then
12                 Add value to compactor at level l in new_sketch
13                 new_sketch.total_count += 2^l
14     return new_sketch
```

new ring. The function then iterates through these intervals. For an interval (*start*, *end*), it identifies the source bucket in the old layout and the destination bucket in the new layout. It then queries the source bucket's quantile sketch using `QueryCountInRange` to estimate the count of items whose hash values fall within this range. This estimate is then added to the destination bucket's counter.

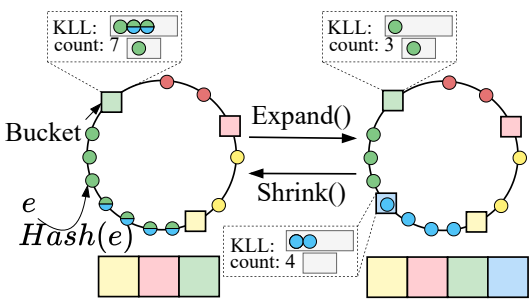


Figure 3: Resize operation: Adding or removing boundary points on the consistent hash ring redistributes counts using per-bucket KLL sketches.

Importantly, simply transferring the count is not enough; the underlying quantile information must also be redistributed to support future operations. This is achieved in *two steps* (detailed in Algorithm 4). First, our proposed `FilterKLL` operation (line 7), extending the functionality of KLL creates a new, temporary KLL sub-sketch that preserves the structure of the original quantile (i.e., the same parameter k and compactor configuration), then filters the original quantile's contents level-wise, retaining only the sampled hash values that fall within the (*start*, *end*) range. Second, this new sub-sketch is combined with the destination bucket's *q_sketch* using `MergeKLL`, which performs a level-wise union of the two sketches' internal compactors and triggers compaction as needed. The entire process is visualized in Figure 3 and Figure 4.

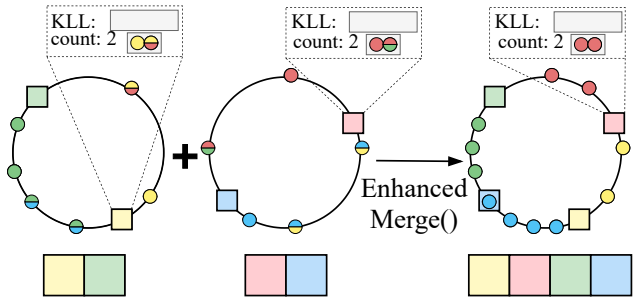


Figure 4: EnhancedMerge operation: Creates a unified ring with a combined width and redistributes both input sketches to the new ring structure.

Properties. The redistribution operation takes $\mathcal{O}((w+w')(\log(w+w')+k))$ steps, which can be broken down as follows: First, creating the sorted list of boundary points takes $\mathcal{O}(w + w')$ steps, where w' is the target width. Second, the algorithm iterates through the resulting $\mathcal{O}(w + w')$ intervals; for each, identifying the source and destination buckets using binary search takes $\mathcal{O}(\log(w + w'))$ steps, and the quantile operations (`FilterKLL` and `MergeKLL`) perform linear scans of compactors in $\mathcal{O}(k)$ time. The formal guarantees on the associated approximation error are provided in section 5.

4.3 Data Manipulation Operations

Update. The `Update` operation (Algorithm 2) processes an item e by updating one bucket in each row i: the item is first hashed using the pairwise-independent hash function of the row to obtain a hash value $y = (a_i \cdot e + b_i) \bmod p$ on the continuous ring (as described before, in subsection 4.2, regarding hashing). (Note that when `Partition` support is required, this is replaced by the partition-aware hashing detailed in subsection 4.4.) The bucket index is determined by `FindBucket`$(y, rings[i])$, through a binary search on the boundary points in $rings[i]$ to find the segment containing y. The bucket's counter is incremented, and the hash value y is inserted into the bucket's inner quantile sketch to update its distribution summary.

Query. The `Query` operation (Algorithm 2) leverages the distributional information stored in the inner quantile sketches to estimate item frequency. As detailed in line 16, the algorithm queries the quantile sketch of each mapped bucket using KLL's `QueryFrequency` (Algorithm 4) to obtain d independent estimates. The final estimate is the *median* of these values, since KLL estimates have two-sided errors, implying the minimum estimator (as in Count-min) is unsuitable (as it amplifies under-estimation) and, unlike the mean, the median is robust against outliers caused by heavy collisions in specific rows.

Properties. Updates take $\mathcal{O}(d(\log w + \log k))$ steps, which accounts for d consistent hashings (i.e. $\log w$ each) and KLL updates ($\mathcal{O}(\log k)$ each). The term $\log k$ is achieved by using the lazy compaction strategies from [20] (compared to $\mathcal{O}(k)$ in the standard version [23]). Queries take $\mathcal{O}(d(\log w + k))$

steps, which can be optimized to $\mathcal{O}(d(\log w + \log k))$ for batch queries if the internal compactors are sorted beforehand. The approximation bounds for this estimator are derived in section 5.

4.4 Structure-Defining Operations

This section applies the count redistribution mechanism to implement structure-defining operations. For the `EnhancedMerge` and `Partition`, sketches must share the same hash seeds to ensure consistent mapping on the logical ring across instances, which allows their structures to be combined or partitioned meaningfully.

Resize. The `Resize` operation (Algorithm 5) adjusts the row width from w to w'. It first calculates the difference $\Delta = w' - w$ to update the consistent hashing ring structure for each row. If $\Delta > 0$ (expansion), Δ new boundary points are randomly added to the ring to increase granularity; if $\Delta < 0$ (shrinking), $|\Delta|$ boundary points are randomly removed, which merges adjacent segments (illustrated in Figure 3). Once the ring is updated, the function invokes `RedistRow` on each row to transfer frequencies and quantile summaries from the old bucket array to the new one.

Algorithm 5: ReSketch – Resize Operation

```
1 Procedure Resize(A, w')
2     buckets ← new bucket array of size A.d × w'; Δ ← w' − A.w
3     for i ← 0 to A.d − 1 do
4         out_ring ← A.rings[i]
5         if Δ > 0 then Add Δ random points to out_ring
6         else Remove |Δ| random points from out_ring
7         buckets[i] ← RedistRow(A.rings[i], A.buckets[i], out_ring)
8         A.rings[i] ← out_ring;
9     A.buckets ← buckets; A.w ← w'
```

Enhanced Merge. The `EnhancedMerge` operation (Algorithm 6) combines two sketches $\mathcal{A}_1(w_1, d)$ and $\mathcal{A}_2(w_2, d)$ into $\mathcal{A}_{merged}(w_1 + w_2, d)$. First, a new `merged_sketch` is initialized. For each row, its ring is formed by the union of the points from the rings of $\mathcal{A}_1$ and $\mathcal{A}_2$ (Figure 4 illustrates this process). Then, for each of the original sketches, `RedistRow` redistributes its contents into temporary bucket arrays that conform to the new merged ring layout. Finally, the function iterates through the new buckets, and aggregates the results by summing the primary counters and calling `MergeKLL` on the quantile sketches element-wise. To merge more sketches, `EnhancedMerge` can be generalized by creating a new ring in each row with the union of the boundaries from all sketches; alternatively, it can be called iteratively to merge them pairwise, until all are combined into one.

Partition. The `Partition` operation (Algorithm 7) splits $\mathcal{A}(w, d)$ into two smaller, separate instances $\mathcal{A}_1(w_1, d)$ and $\mathcal{A}_2(w_2, d)$ (where $w_1 + w_2 = w$). A naive approach might simply split the bucket array of each row at index w_1. However, as established in Algorithm 2, the hash value generation is pairwise-independent; therefore, an item e might map to one of the first w_1 buckets in one row and to one of the subsequent w_2 buckets in another row. Consequently,

Algorithm 6: ReSketch – Merge Operation

```
Procedure EnhancedMerge(A_1, A_2)
    new_width ← A_1.w + A_2.w
    A_merged ← new ReSketch(A_1.d, new_width, A_1.seeds)
    for i ← 0 to A_1.d − 1 do
        T1 ← RedistRow(A_1.rings[i], A_1.buckets[i], A_merged.rings[i])
        T2 ← RedistRow(A_2.rings[i], A_2.buckets[i], A_merged.rings[i])
        for j ← 0 to new_width − 1 do
            A_merged ← A_merged.buckets[i][j]
            A_merged.count ← T1[j].count + T2[j].count
            A_merged.q_sketch ← T1[j].q_sketch
            A_merged.q_sketch.MergeKLL(T2[j].q_sketch)
    return A_merged
```

future updates or queries for e would need to be sent to both instances, which would undermine the purpose of partitioning in distributed processing. To address this, we introduce **Partition-Aware Hashing** (highlighted in green in Algorithm 7, visualized in Figure 5), which replaces the standard pairwise-independent hashing (highlighted in red).

Algorithm 7: ReSketch – Partition Operation

```
Procedure Hash(e, seed_i)
    return seed_i.a · e + seed_i.b                  // Original 2-wise independent hash
    // Partition-aware hashing; replaces the original hash logic when
       Partition is supported
    fp ← h_fp(e)                                    // Step 1: Partition (uniform) Hash
    return seed_i.a · fp + seed_i.b                 // Step 2: Placement (2-indep.) Hash
Procedure ModInverse(a)
    return a^{-1}                          // (a · a^{-1}) mod 2^W = 1, where W is word width
Procedure RecoverFingerprint(y, seed_i)
    // y = seed_i.a · fp + seed_i.b ⟹ fp = (y − seed_i.b) · a^{-1}_{seed_i.a}
    return (y − seed_i.b)·ModInverse(seed_i.a)
Procedure Partition(A, w_1, w_2)
    A_1 ← new ReSketch(A.d, w_1, A.seeds)
    A_2 ← new ReSketch(A.d, w_2, A.seeds)
    // Define split point on the primary partition space [0,1)
    split_point ← w_1/(w_1 + w_2)
    foreach (i, y, weight) in A do
        fp ← RecoverFingerprint(y, A.seeds[i])
        if fp < split_point then Reinsert(A_1, i, y, weight)
        else Reinsert(A_2, i, y, weight)
    return A_1, A_2
Procedure Reinsert(A, i, y, weight)
    Add weighted sample (y, weight) to the correct bucket in row i of A
```

This technique treats hash value generation as a composition of two steps: ❶ **Partition Hashing**: The item e is mapped to a fingerprint fp via a uniform partition hash function $h_{fp}(e)$. This fingerprint acts as an immutable identifier for the item's partition assignment for its entire lifetime. ❷ **Placement Hashing**: The pairwise-independent transformation for row i then operates on this fingerprint: $y = a_i \cdot fp + b_i$ rather than on the raw item (see Figure 5). The reasoning of this 2-step process is analogous to how `Resize` relies on slicing the

continuous bucket ring to redistribute items between buckets; i.e., partitioning relies on slicing the continuous fingerprint domain to redistribute items between sketches, which allows partitioned sketches to be partitioned further or merged similarly.

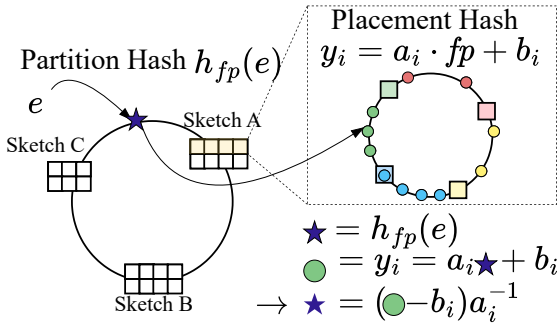


Figure 5: Partition-aware hashing: An item is mapped to a fingerprint via a uniform hash function $h_{fp}(e)$, then to a point on a ring via pair-wise independent placement hashing $y = a_i \cdot fp + b_i$, which determines bucket mapping. The fingerprint fp can be recovered from y through modular multiplicative inverse, introducing no additional storage usage.

To enable state migration during a `Partition`, the fingerprint fp must be recoverable from the hash values stored within the sketch's buckets; otherwise, we would need to store the fingerprint alongside every hash value, which would double the space needs. However, since the hash value is the result of a modulo operation ($\bmod p$), multiple fingerprints can map to the same hash value, hence cannot be uniquely recovered. To do this on W-bit registers, we can map the fingerprint to a general finite field $\mathbb{GF}(2^W)$ and perform arithmetic operations (i.e., $a_i \cdot fp + b_i$) in this field [27]. In this domain, the linear transformation $a_i x + b_i$ is a bijection for all $a_i \neq 0$, which can allow to always recover the fingerprint from the hash value. `Partition` employs this to split the sketch based on a split point in the partition space. As shown in Algorithm 7, it iterates through all summarized samples, recovers fp to determine which of the two new sketches the sample belongs to, and re-inserts the samples and their weights accordingly.

Time Complexity. The cost of `Resize` is determined by the cost of `RedistRow` across d rows, which is $\mathcal{O}(d(w + w')(\log(w + w') + k))$. For `EnhancedMerge`, the complexity consists of the redistribution overhead for the input sketches plus the cost of merging the resulting buckets. Since merging two KLL sketches takes $\mathcal{O}(k)$ [23] steps, the additional merging cost is $\mathcal{O}(d(w_1 + w_2)k)$. Adding this to the redistribution cost yields a total complexity of $\mathcal{O}(d((w_1 + w_2)\log(w_1 + w_2) + k))$. Finally, `Partition` involves iterating over every sample stored in the sketch to recover its fingerprint and re-insert it. With $d \times w$ buckets each storing $\mathcal{O}(k)$ items, and considering that re-inserting each item requires finding the target bucket in $\mathcal{O}(\log w)$ and adding it as a weighted sample in $\mathcal{O}(1)$, this operation has a total time complexity of $\mathcal{O}(dwk \log w)$.

Practical Implementation of Partition-Aware Hashing

Arithmetic in $\mathbb{GF}(2^W)$ is computationally expensive in commodity hardware, as it requires polynomial multiplication and modular reductions. Consequently, we adopt a more efficient approach that leverages the native 2's complement representation integer arithmetic of modern processors (e.g., x86) . We treat the fingerprint *fp* and hash values as standard W-bit machine words (e.g., $W = 64$), and apply the linear transformation $y = a_i \cdot fp + b_i$, constrained that a_i is odd, without an explicit modulo p operation. Figure 5 illustrates this hashing mechanism. On W-bit registers, multiplication automatically discards overflow bits, which is equivalent to performing operations modulo 2^W. This guarantees both statistical uniformity and reversibility. Regarding uniformity, we rely on Fact 3.4 from Thorup [35], which states that for any modulus m and multiplier a coprime to m, the linear map x to $ax + b \pmod m$ preserves the uniform distribution of the input. Due to the fact that *fp* is uniformly distributed, and our multiplier a_i is odd (coprime to 2^W), the resulting hash values are uniform over the ring. Regarding reversibility, also because a_i is coprime to 2^W, a multiplicative inverse a_i^{-1} is guaranteed to exist. This allows us to recover the original fingerprint *fp* from a stored hash value y via the operation $fp = (y - b_i) \cdot a_i^{-1} \pmod{2^W}$.

5 ReSketch Analysis

In this section, we analyze ReSketch's frequency estimation approximation bounds. We begin in subsection 5.1 by establishing the bound for a static single sketch and then extend to dynamic and distributed cases in subsection 5.2, where sketches undergo structure-defining operations. Towards the latter, we introduce an *instance provenance* graph to track error accumulation across operations, analyze errors from both data processing and structure-defining operations, and derive our main theorem (Theorem 15), which provides a bound for any sketch instance produced by an arbitrary sequence of operations.

5.1 Static Single ReSketch Bound

We first analyze the expected bucket load in ReSketch's consistent hashing scheme. Let $C^{i,j}$ denote the count in bucket j of row i.

Lemma 4 (Expected Bucket Load)**.** *For a static $\mathcal{A}(w, d)$ processing a stream of length N, given any arbitrary item e, the expected count in row i bucket j, where j is the bucket that e is hashed to for row i, is $\mathbb{E}[C^{i,j}] \approx \frac{2N}{w}$.*

Proof. The placement of w random bucket boundaries and one specific item's hash value on the ring, partitions it into $w + 1$ intervals, each with expected length $\frac{1}{w+1}$ by uniformity. The bucket containing e consists of the interval between the boundary preceding e's hash value and the one succeeding it. Thus, its expected length is $\frac{2}{w+1} \approx \frac{2}{w}$, and the expected count is $\frac{2N}{w}$. □

Lemma 5 (KLL Frequency Estimation Error)**.** *Consider a KLL sketch with parameter k, that processed N items. Define the estimated frequency of an item e as $\widehat{f}(e) = rank(e) - rank(e^-)$, where e^- denotes the value immediately*

preceding e in the totally-ordered domain. Then, $|\widehat{f}(e) - f(e)| \leq 2\varepsilon_{KLL} \cdot N$ with probability at least $1 - \delta_{KLL}$ for any item e, where $f(e)$ is the true frequency.

Proof. A KLL sketch provides rank queries with error at most $\varepsilon_{KLL} \cdot N$ with probability at least $1 - \delta_{KLL}$. Since both rank queries contribute at most $\pm\varepsilon_{KLL} \cdot N$ error, the total error satisfies $|\widehat{f}(e) - f(e)| \leq 2\varepsilon_{KLL} \cdot N$ with probability at least $1 - \delta_{KLL}$. □

Lemma 6 (Median Amplification)**.** *Consider estimating the frequency $f(e)$ of a specific item e using d independent estimators. Suppose each estimator fails to satisfy an approximation bound $|\widehat{f}(i)e - f(e)| \leq \varepsilon$ with probability at most $p < 1/2$. Then, the median of these estimators fails to satisfy the same bound with probability at most δ, provided $d = \mathcal{O}(\log(1/\delta))$.*

Proof. Let X_i be the indicator random variable that estimator i fails, and let $X = \sum_{i=1}^{d} X_i$. The median fails if and only if at least $d/2$ estimators fail, i.e., $X \geq d/2$. We have $\mathbb{E}[X] = p \cdot d < d/2$. By the Chernoff bound, $\Pr[X \geq d/2] = \exp(-\Theta(d))$. Setting $d = \mathcal{O}(\log(1/\delta))$ ensures the failure probability is at most δ. □

Theorem 7 (Static Single ReSketch Error Bound)**.** *For a static $\mathcal{A}(w, d)$ processing a stream of length N, the estimate satisfies $|\widehat{f}(e) - f(e)| \leq \varepsilon N$ with probability $\geq 1 - \delta$ using space $\mathcal{O}\left(\frac{1}{\varepsilon} \log \frac{1}{\delta}\right)$.*

Proof. Let $C^{i,j}$ denote the count in e's bucket in row i. By Lemma 4, $\mathbb{E}[C^{i,j}] \approx 2N/w$. The row-$i$ estimate fails if either (1) the KLL fails internally (prob. $\leq \delta_{KLL}$), or (2) the bucket load satisfies $2\varepsilon_{KLL} \cdot C^{i,j} > \varepsilon N$ (the factor 2 follows from Lemma 5). By Markov, $\Pr[(2)] \leq \frac{2\varepsilon_{KLL} \cdot 2N/w}{\varepsilon N} = \frac{4\varepsilon_{KLL}}{\varepsilon w}$, so $P_{\text{row\_fail}} \leq \delta_{KLL} + \frac{4\varepsilon_{KLL}}{\varepsilon w}$.

The final estimator takes the median of d independent row estimates. To apply Lemma 6, we require $P_{\text{row\_fail}} < 1/2$. Let's choose $P_{\text{row\_fail}} \leq 1/3$, and allocate the budget evenly: set $\delta_{KLL} = 1/6$ for event (1), and $w = 24\varepsilon_{KLL}/\varepsilon$ for event (2). By Lemma 6 with $d = \mathcal{O}(\log(1/\delta))$, the median fails with probability $\leq \delta$.

The space is $d \times w \times k$, where each bucket stores one KLL of the same size $\mathcal{O}(k)$. Since $k = \mathcal{O}(1/\varepsilon_{KLL})$ for fixed $\delta_{KLL} = 1/6$, we have: Space $= d \cdot w \cdot k = \mathcal{O}(\log \frac{1}{\delta}) \cdot \frac{24\varepsilon_{KLL}}{\varepsilon} \cdot \frac{1}{\varepsilon_{KLL}} = \mathcal{O}(\frac{1}{\varepsilon} \log \frac{1}{\delta})$. □

5.2 Dynamic Distributed ReSketch Bound

Theorem 7 bounds approximation errors for a single sketch instance at fixed width. However, dynamic (distributed or parallel) systems have multiple sketch origins processing different streams and undergoing structure-defining operations (Resize, EnhancedMerge, Partition). Therefore, the error bound must account for: (a) data processing across multiple instances, each potentially at different widths, (b) errors introduced by structure-defining operations (c) errors persisted from prior operations.

To reason about error accumulation in such systems, we formalize sketch instances and their relationships. Each *sketch instance*, denoted $\mathcal{A}_{id}$ with

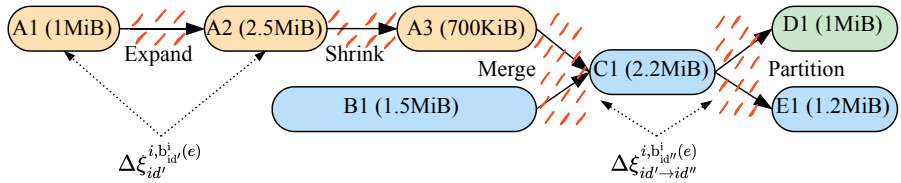


Figure 6: Instance provenance DAG illustrating error tracking across operations. Each node represents a sketch instance with its size, also represents data processing period with error reference $\xi^{i,\mathrm{b}^i_{id}(e)}_{id'}$; solid edges represent structure-defining operations with error reference $\xi^{i,\mathrm{b}^i_{id'}(e)}_{id'\to id''}$. Error accumulation follows provenance paths to $\mathcal{A}_{id}$ (e.g., $\mathcal{A}_{C1}$ inherits errors from $\mathcal{A}_{A1} \to \mathcal{A}_{A2} \to \mathcal{A}_{A3}$ and $\mathcal{A}_{B1}$).

a unique identifier id, conceptually summarizes a *logical stream*, S_{id}, the conceptual multiset of all items it processed, with a total number of items denoted N_{id}. This logical stream includes both items directly processed by the instance and items inherited from ancestor instances through the provenance paths, which will be defined shortly. For example, when two instances $\mathcal{A}_1$ and $\mathcal{A}_2$ merge to create $\mathcal{A}_{merged}$, the logical stream summarized by the merged sketch, S_{merged}, contains all items from both S_1 and S_2, and thus $N_{merged} = N_1 + N_2$. The relationships between sketch instances form a directed acyclic graph (DAG) defined as follows:

Definition 8 (Instance Provenance DAG)**.** *The* instance provenance DAG *(example illustrated in Figure 6) is a directed acyclic graph where: (i)* Nodes and processing periods*: Each node represents a sketch instance $\mathcal{A}_{id}$ with an associated* logical stream*, i.e., the substream of data items it processes. A processing period is an interval during which a sketch instance processes updates (also represented as part of the respective node of the graph). (ii)* Edges *represent structure-defining operations (Resize, EnhancedMerge, or Partition) that create descendant instances from ancestors. For a given sketch instance $\mathcal{A}_{id}$, consider the* provenance paths *originating from source nodes (i.e., sketch instances with no predecessors) and terminating at $\mathcal{A}_{id}$. Based on them, define: (iii) $\mathcal{P}_{id}$: The set of all sketch instance IDs along those provenance paths. (iv) $\mathcal{Q}_{id}$: The set of all structure-defining transitions $id' \to id''$ occurring along these paths.*

Unlike the static single sketch case, where an item is always mapped to the same bucket j in each row and bucket count $C^{i,j}$ increases monotonically, structure-defining operations can move items between buckets, which may alter the bucket index for item e and cause bucket counts to both increase and decrease. Let $\mathrm{b}^i_{id}(e)$ be the index of the bucket in row i to which item e maps in the analyzed instance $\mathcal{A}_{id}$. We denote by $C^{i,\mathrm{b}^i_{id}(e)}_{id}$ the total item count within this bucket in row i.

Bucket counts change through two types of operations: (1) *Data processing at instance id*: The change is denoted $\Delta C^{i,\mathrm{b}^i_{id}(e)}_{id}$, which represents items added to instance id during data processing period. The subscript id alone indicates

processing happening at this instance. (2) *Structure-defining operation* $id \to id'$: The change is denoted $\Delta C^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}$, which represents bucket changes during the transformation from instance id to id'. The subscript $id \to id'$ indicates a transition between instances. $\Delta C^{i,\mathrm{b}^i_{id}(e)}_{id}$ and $\Delta C^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}$ do not directly bound the error increment because KLL compaction errors are irreversible. KLL error arises from compaction: as more items are processed, more compaction occurs, which creates error. For example, Expand may reduce the bucket count ($\Delta C^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'} < 0$), but prior compaction errors from items already compacted cannot be recovered. We introduce an *error reference variable* $\xi^{i,\mathrm{b}^i_{id}(e)}$ which accounts for the fact that only operations increasing bucket count cause new compaction.

Definition 9 (Error Reference Variable)**.** *For item e in row i, we define the* incremental error reference variable $\xi^{i,\mathrm{b}^i_{id}(e)}$ *for each operation or processing period. The* cumulative error reference *for item e in row i at instance $\mathcal{A}_{id}$, denoted $\Xi^{i,\mathrm{b}^i_{id}(e)}_{id}$ is the sum of all incremental error contributions along the provenance paths from source nodes to id:*

$$\Xi^{i,\mathrm{b}^i_{id}(e)}_{id} = \sum_{id'\in\mathcal{P}_{id}} \xi^{i,\mathrm{b}^i_{id'}(e)}_{id'} + \sum_{id'\to id''\in\mathcal{Q}_{id}} \xi^{i,\mathrm{b}^i_{id''}(e)}_{id'\to id''}$$

Lemma 10 (Data Processing Error Reference)**.** *Consider data processing at $\mathcal{A}_{id}(w_{id}, d)$ that processes ΔN_{id} items. For any item e in any row i, the incremental error reference satisfies: $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id}(e)}_{id}] = 2\Delta N_{id}/w_{id}$.*

Proof. By Lemma 4, if instance $\mathcal{A}_{id}$ has width w_{id} and processes ΔN_{id} items during this period, the expected count increment in e's bucket is $2\Delta N_{id}/w_{id}$, thus: $\mathbb{E}[\Delta C^{i,\mathrm{b}^i_{id}(e)}_{id}] = 2\Delta N_{id}/w_{id}$. New items cause new KLL compaction, so the incremental error reference equals the bucket count increment: $\xi^{i,\mathrm{b}^i_{id}(e)}_{id} = \Delta C^{i,\mathrm{b}^i_{id}(e)}_{id}$. Therefore: $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id}(e)}_{id}] = \mathbb{E}[\Delta C^{i,\mathrm{b}^i_{id}(e)}_{id}] = 2\Delta N_{id}/w_{id}$. □

Lemma 11 (Expand Error Reference)**.** *Consider the Resize operation from $\mathcal{A}_{id}(w, d)$ to $\mathcal{A}_{id'}(w' > w, d)$, i.e., expanding the sketch. For any item e in any row i, the operation introduces no additional error reference: $\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'} = 0$.*

Proof. The Resize operation with $w' > w$ adds $w' - w$ new boundary points to each row's consistent hash ring, which splits some existing buckets into smaller segments. There is no re-hashing, random sampling, or compaction that occurs during this process. Since the original hash values are preserved and no new compaction is performed, the operation introduces no new estimation errors beyond those already present in the old KLL sketches. Although the bucket count $C^{i,\mathrm{b}^i_{id}(e)}_{id'}$ may decrease (as items redistribute to finer buckets), the existing KLL compaction errors from prior compactions remain unchanged. Thus: $\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'} = 0$. In fact, it could reduce errors for future data processing since less contention in buckets lead to less compaction. □

Lemma 12 (Shrink Error Reference)**.** *Consider the Resize operation from $\mathcal{A}_{id}(w, d)$ to $\mathcal{A}_{id'}(w' < w, d)$, i.e., shrinking the sketch, where N_{id} is the total number of items processed by $\mathcal{A}_{id}$ before the resize. For any item e in any row i, the incremental error reference satisfies:* $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}] = \mathbb{E}[\Delta C^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}] = 2N_{id}(1/w' - 1/w)$.

Proof. The Shrink operation removes $w - w'$ boundary points, which merges some buckets together. Let N_{id} denote the total number of items processed by instance $\mathcal{A}_{id}$ before the shrink. By Lemma 4, before shrink, the expected e's bucket count is $\mathbb{E}[C^{i,\mathrm{b}^i_{id}(e)}_{id}] = 2N_{id}/w$. After shrink, buckets are merged, which increases the expected bucket count to $\mathbb{E}[C^{i,\mathrm{b}^i_{id}(e)}_{id'}] = 2N_{id}/w'$. The bucket count increment is: $\mathbb{E}[\Delta C^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}] = 2N_{id}/w' - 2N_{id}/w = 2N_{id}(1/w' - 1/w)$. This increase in bucket count causes additional KLL compaction when the KLL sketches are merged (via MergeKLL operation), which introduces additional error reference proportional to the count increment. Thus: $\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'} = \Delta C^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}$, giving $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}] = 2N_{id}(1/w' - 1/w)$. □

Lemma 13 (Enhanced Merge Error Reference)**.** *Consider the EnhancedMerge operation combining $\mathcal{A}_1(w_1, d)$ and $\mathcal{A}_2(w_2, d)$ into $\mathcal{A}_{id'}(w_1 + w_2, d)$. For any item e in any row i, the operation introduces no additional expected error reference:* $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{(id_1,id_2)\to id'}] = 0$.

Proof. The EnhancedMerge operation first expands both $\mathcal{A}_1$ and $\mathcal{A}_2$ to width $w_1 + w_2$, then merges buckets element-wise. Let N_1 and N_2 denote the total number of items processed by the two instances. By Lemma 11, expanding both sketches to width $w_1 + w_2$ introduces no additional error: $\xi^{i,\mathrm{b}^i_{id'}(e)}_{id_1\to id'_1} = 0$ and $\xi^{i,\mathrm{b}^i_{id'}(e)}_{id_2\to id'_2} = 0$. After expansion, by Lemma 4, the expected bucket counts are $\mathbb{E}[C^{i,\mathrm{b}^i_{id}(e)}_{id'_1}] = 2N_1/(w_1 + w_2)$ and $\mathbb{E}[C^{i,\mathrm{b}^i_{id}(e)}_{id'_2}] = 2N_2/(w_1 + w_2)$. The element-wise KLL MergeKLL merges the two buckets, which gives expected bucket count: $\mathbb{E}[C^{i,\mathrm{b}^i_{id}(e)}_{id'}] = \frac{2(N_1+N_2)}{w_1+w_2}$. This is identical to the expected bucket count from processing the union stream directly into a sketch of width $w_1 + w_2$. Thus, the merge operation introduces zero additional error reference: $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{(id_1,id_2)\to id'}] = 0$. In practice, merging two sketches may trigger additional compaction in some buckets. However, some others become emptier and thus have reduced error reference for future incoming items. This is why $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{(id_1,id_2)\to id'}] = 0$. □

Lemma 14 (Partition Error Reference)**.** *Consider the Partition operation of $\mathcal{A}_{id}(w, d)$ into $\mathcal{A}_1(w_1, d)$ and $\mathcal{A}_2(w_2, d)$, where $w_1 + w_2 = w$. For any item e in any row i, the operation introduces no additional expected error reference:* $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to(id_1,id_2)}] = 0$.

Proof. The Partition operation uses reversible hashing to recover partition keys $v = h_{fp}(e)(e)$ for each item, then partitions items based on whether $v < v_{\text{split}}$ for some split point $v_{\text{split}} \in (0, 1)$. Let N denote the total number of items processed by $\mathcal{A}_{id}$. By the uniform distribution property of $h_{fp}(e)$, the expected

number of items in partition 1 is: $\mathbb{E}[N_1] = v_{\text{split}} \cdot N$. The partition widths are allocated proportionally: $w_1 = v_{\text{split}} \cdot w$ and $w_2 = (1 - v_{\text{split}}) \cdot w$. Before split, by Lemma 4, the expected e's bucket count is $\mathbb{E}[C_{id}^{i,\mathrm{b}_{id}^i(e)}] = 2N/w$. For item e assigned to $\mathcal{A}_1$, the expected bucket count is: $\mathbb{E}[C_{id_1}^{i,\mathrm{b}_{id}^i(e)}] = \frac{2\mathbb{E}[N_1]}{w_1} = \frac{v_{\text{split}} \cdot N}{v_{\text{split}} \cdot w} = \frac{2N}{w}$. This ensures the e's bucket count is preserved in expectation. Therefore, the operation introduces no additional expected error reference: $\mathbb{E}[\xi_{id \to (id_1, id_2)}^{i,\mathrm{b}_{id'}^i(e)}] = 0$. □

Theorem 15 (Dynamic Distributed ReSketch Error Bound)**.** *For $\mathcal{A}_{id}(w, d)$ processing a total stream length $N_{id} = \sum_{id' \in \mathcal{P}_{id}} \Delta N_{id'}$, the frequency estimate satisfies $|\widehat{f}(e) - f(e)| \le \varepsilon N_{id}$ with probability $\ge 1 - \delta$, provided:*

$$\sum_{id' \in \mathcal{P}_{id}} \frac{\Delta N_{id'}}{w_{id'}} + \sum_{\substack{(id' \to id'') \in \mathcal{Q}_{id} \\ \textit{is Shrink}}} N_{id'} \left(\frac{1}{w_{id''}} - \frac{1}{w_{id'}} \right) \le \frac{\varepsilon N_{id}}{24 \varepsilon_{KLL}}$$

Proof. For row i, by Lemma 5, the KLL sketch satisfies $|\widehat{f}_i(e) - f(e)| \le 2\varepsilon_{KLL} \cdot \Xi_{id}^{i,\mathrm{b}_{id}^i(e)}$ with probability $1 - \delta_{KLL}$, where $\Xi_{id}^{i,\mathrm{b}_{id}^i(e)}$ serves as stream length for the KLL in that bucket. By Definition 9, and Lemmas 10–14, we have: $\mathbb{E}[\Xi_{id}^{i,\mathrm{b}_{id}^i(e)}] = \sum_{id' \in \mathcal{P}_{id}} \frac{2\Delta N_{id'}}{w_{id'}} + \sum_{\substack{(id' \to id'') \in \mathcal{Q}_{id} \\ \text{is Shrink}}} \left(\frac{2N_{id'}}{w_{id''}} - \frac{2N_{id'}}{w_{id'}} \right)$.

The row-i estimate fails if either: (1) The KLL fails internally with probability $\le \delta_{KLL}$, or (2) The cumulative error $2\varepsilon_{KLL} \cdot \Xi_{id}^{i,\mathrm{b}_{id}^i(e)}$ exceeds εN_{id}. By Markov's inequality, $\Pr[2\varepsilon_{KLL} \cdot \Xi_{id}^{i,\mathrm{b}_{id}^i(e)} \ge \varepsilon N_{id}] \le \frac{2\varepsilon_{KLL} \cdot \mathbb{E}[\Xi_{id}^{i,\mathrm{b}_{id}^i(e)}]}{\varepsilon N_{id}}$. Let X be the left-hand side of the condition inequality. Then $\mathbb{E}[\Xi_{id}^{i,\mathrm{b}_{id}^i(e)}] = 2X$. Substituting $X \le \frac{\varepsilon N_{id}}{24\varepsilon_{KLL}}$, we get: $\Pr[(2)] \le \frac{2\varepsilon_{KLL} \cdot 2(\frac{\varepsilon N_{id}}{24\varepsilon_{KLL}})}{\varepsilon N_{id}} = \frac{4\varepsilon_{KLL} \cdot \varepsilon N_{id}}{24\varepsilon_{KLL} \cdot \varepsilon N_{id}} = \frac{1}{6}$. Thus, $P_{\text{row\_fail}} \le \delta_{KLL} + 1/6$. Setting $\delta_{KLL} = 1/6$ gives $P_{\text{row\_fail}} \le 1/3$. Taking the median of $d = \mathcal{O}(\log(1/\delta))$ independent row estimates, Lemma 6 ensures the estimate succeeds with probability $\ge 1 - \delta$. □

This theorem provides two practical implications: First, it serves as a verification mechanism: given an initial target error bound ε, one can verify if the guarantee holds after an arbitrary sequence of operations by checking if the cumulative errors (the left-hand side of the inequality) remain below the derived threshold. Second, it enables dynamic bound derivation: e.g., in cases where the sequence of shrinking operations forces the sketch error go beyond its original parameters, the inequality can be inverted to find the new effective error. This allows the system to quantify the accuracy trade-offs incurred by dynamic resizing, merging, and partitioning.

6 Evaluation

Here, we provide an evaluation of the performance of ReSketch in three parts. First, a *sensitivity analysis* is performed, both to associate the performance of ReSketch with the bounds in the analysis, and to select suitable parameters

for subsequent benchmarks. Second, we perform *benchmarks of the operations* supported by ReSketch: Update, Query, Resize, EnhancedMerge, and Partition. Third, based on the example application from section 1, we demonstrate the benefits enabled by ReSketch in a *realistic network monitoring* scenario. To begin, we describe the evaluation environment, datasets, baselines, and parameters. After presenting detailed benchmark results, we summarize the main takeaways.

Environment. All experiments are conducted on an Intel Xeon E5-2695 v4 processor running at 2.1 GHz with three-level cache hierarchy (32 KiB L1, 256 KiB L2, and 45 MiB L3). ReSketch and baselines are implemented in C++[3], compiled with GCC 15.2.1 using `-O3` on openSUSE Tumbleweed 20251127. All data are averages over 30 runs, with shaded regions illustrating run-to-run variance.

Dataset. We use the CAIDA Anonymized Internet Traces 2018 [6], a broadly used benchmark in network monitoring literature [33], [38], [39]. We extract source IP addresses from the first 10 M packets, representing network traffic monitoring scenarios with realistic skewness that is commonly observed in real-world environments [12].

Baselines. The performance of ReSketch is evaluated relative to: 1. Count-Min Sketch (CMS), as described in section 2, an efficient, established sketch, however lacking support for resizing and enhanced mergeability. 2. Dynamic Count-Min Sketch (DCMS) [41], a design for coarse-grained expansion using an ever-growing linked list of complete CMS instances in order to maintain certain estimation error bounds for skewed data, but does not support shrinking to reclaim memory. 3. Geometric Sketch (GS) [4], which supports fine-grained dynamic resizing in both directions; however, it cannot shrink below its initial allocation. As noted, no prior work supports enhanced merging and the ability to adjust partitioning.

Hashing. We use 64-bit xxHash [8] for partition hashing $h_{fp}(e)$ (section 12), due to its high throughput on small inputs and good quality regarding collisions and random dispersion of hash values [36].

Metrics of interest. We evaluate several metrics, defined as follows: 1. *Update Throughput* measures the number of processed items per second. 2. *Query throughput* measures the number of queries processed per second. 3. *Memory usage* measures the space needed for data structures. 4. *Average Relative Error (ARE)* ($\frac{1}{|S|}\sum_{e\in S}\frac{|f(e)-\widehat{f}(e)|}{f(e)}$) measures the deviation between true and estimated frequencies across all distinct items in S. 5. *Average Absolute Error (AAE)* ($\frac{1}{|S|}\sum_{e\in S}|f(e)-\widehat{f}(e)|$) similarly measures the absolute deviation between true and estimated frequencies. 6. *ARE variance* ($\frac{1}{|S|}\sum_{e\in S}(\frac{|f(e)-\widehat{f}(e)|}{f(e)}-\mathrm{ARE})^2$) measures the dispersion of relative errors across items within a single run. 7. *AAE variance* ($\frac{1}{|S|}\sum_{e\in S}(|f(e)-\widehat{f}(e)|-\mathrm{AAE})^2$) measures the dispersion of absolute errors across items within a single run. The variance metrics are designed to measure the dispersion of relative and absolute errors across all items in the sketch, testing the stability of the returned estimates; i.e. large errors for a few keys will weigh more than small errors for many keys, a behavior

[3] Open-source, available at `https://github.com/vinhqngo5/ReSketch` [28]

that ARE/AAE cannot reveal due to taking the mean.

Parameters. To evaluate ReSketch, the parameters k, d, and w (as in Table 2) need to be configured. The total memory M of the structure is given by $d \cdot w \cdot 3k$, where $3k$ is the upper bound on the size of a KLL with parameter k. To this end, we perform a sensitivity analysis to select suitable parameters for subsequent benchmarks.

6.1 Sensitivity Analysis

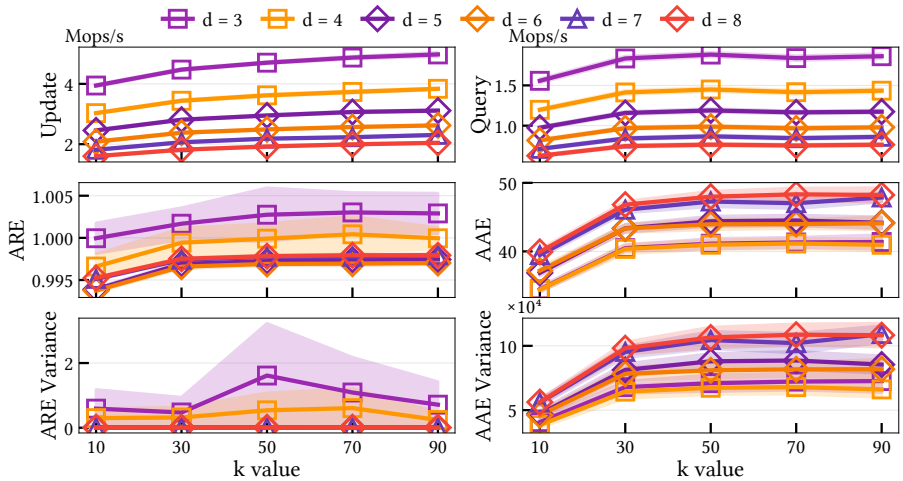


Figure 7: Varying k and d (rows) for ReSketch with fixed $M = 64\,\text{KiB}$.

Method. Using different memory budgets $M = 32$, 64, 256 and 1024 KiB, we process 10 M items from the CAIDA dataset with various combinations of k and d, setting w accordingly to satisfy the chosen M. For brevity, we show results for $M = 64\,\text{KiB}$ here, while the remaining figures can be found alongside the implementation, in the open repository [28]. Values of M are chosen based on the memory needed to count all unique items in the input dataset accurately; there are 154 k unique items, requiring $154000 \cdot 2 \cdot 4 \approx 1203\,\text{KiB}$ — this is the upper bound on M. We explore the performance of these parameter choices on the aforementioned metrics: throughput for update and query operations, accuracy measured as ARE and AAE, and the within-run variance of each.

Results. Results for all 6 metrics are shown in Figure 7. Throughput decreases with increasing d as each additional row requires an additional row update/read operation. Increasing k for fixed d shows improved throughput for updates, as larger capacity compactors perform compaction operations less frequently. Impact of k on query throughput is less significant, as queries take a similar amount of time regardless of w. ARE with few rows ($d = 3$ or 4) is higher than for deeper sketches, as the median estimator has fewer input estimators to sample. Run-to-run variance is somewhat larger for these configurations, while using $d \geq 5$ yields stable ARE performance regardless of k. Results are generally close to 1.0, as a large number of light keys will

have their occurrences evicted from their KLLs/buckets, and will hence be underestimated by 100 % of their true count, i.e., a relative error of 1.0. With increased M, more keys can be stored in the KLLs and fewer keys have their frequency estimated as 0, decreasing ARE. Absolute error increases with more rows for any fixed k, as each per-row estimator has fewer buckets, inducing more collisions and updates per KLL, in turn leading to more compactions and more items evicted from KLLs. The pairwise arrangement is due to the median estimator taking the mean of the two central estimates for even d, which offsets the reduction in accuracy compared to the wider estimators when using depth $d - 1$. Within-run variance is overall low, indicating stable estimator accuracy, and further supports the conclusion that fewer but wider per-row estimators achieve better accuracy.

Based on these results, we select $k = 10$ and $d = 4$ for subsequent experiments, to balance throughput and accuracy. From the analysis (Theorem 7), the choice of k does not impact the worst-case error bound, hence we select k based on the empirical results observed here. We observe that smaller k has significantly better accuracy with only a minor impact on throughput. Although a smaller k entails more frequent KLL compaction, the larger w permitted in the same M yields overall better accuracy (many small KLLs with fewer collisions is more accurate than fewer large ones) in both ARE and AAE. Similarly, selecting a low depth d permits a larger w in the same M, reducing collisions per row hence yielding better per-row estimators to take the median over.

6.2 Benchmarks

6.2.1 Resize

We evaluate the performance of ReSketch when resizing, in terms of throughput and accuracy. We compare with the resizable baselines as well as a static, non-resized CMS and ReSketch instance to see potential performance or accuracy changes due to resizing, while highlighting the benefit of resizability compared to using a static sketch configuration.

Method. The benchmark proceeds in two phases. First, all sketches are initialized at $M_0 = 64\,\text{KiB}$ (as in the sensitivity analysis), and begin processing updates while recording update throughput. After every 100 k updates, we measure ARE, AAE, within-run variance, and query throughput. Then, the memory budget for each sketch is expanded by $M_\Delta = 8\,\text{KiB}$, and the process continues until 10 M items have been processed and the sketches have grown to $M_1 = 864\,\text{KiB}$. Note that DCMS only expands once the budget has doubled, due to the coarse-grain expansion scheme used. The second phase, using copies of the previously populated sketches, proceeds with a series of shrink operations to a final size of $M_2 = 16\,\text{KiB}$. Accuracy is again recorded after each shrink. One copy of each sketch is kept pristine (denoted $\oslash$), while the other receives additional updates in between shrink operations to determine the impact of shrinking both in isolation and in conjunction with data manipulation.

Results. Figure 8 shows results in all metrics for the expand and shrink

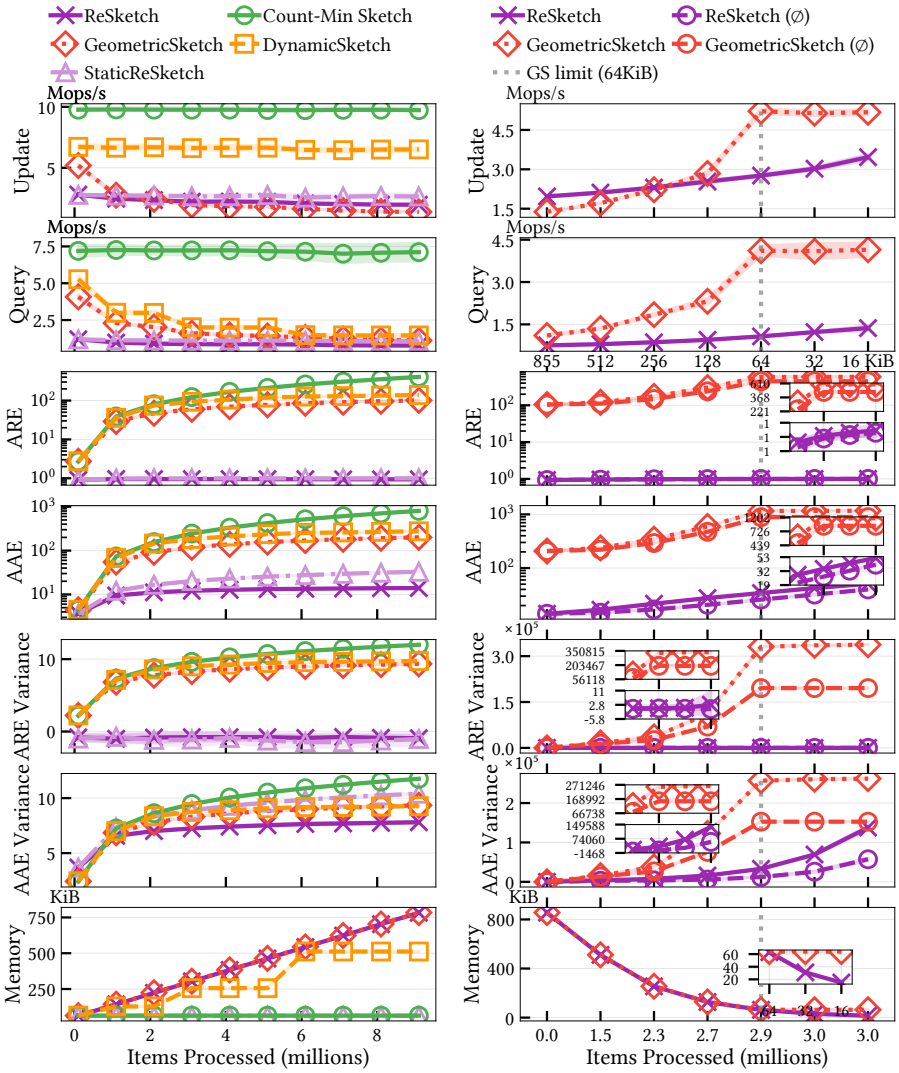


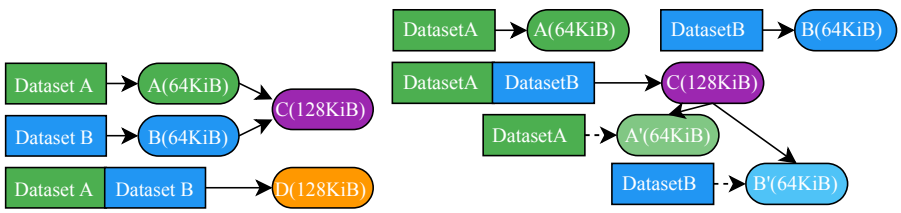
Figure 8: Comparison of performance subject to resizing operations compared to baselines (partially) supporting them; expand on the left and shrink on the right.

phases on the left and right, respectively. Update throughput of ReSketch decreases as the sketch size outgrows L2 cache, but stays close to the static ReSketch, while GS decays more as the sketch expands further. CMS and DCMS utilize less complex hashing schemes and are therefore able to sustain higher update throughput, while query throughput for DCMS degrades significantly as the chain of internal CMS instances grows. In comparison to baseline approaches, ReSketch achieves orders of magnitude improved accuracy. AAE and its variance reveal the benefit of expansion on accuracy when processing the same input, permitting the sketch to store more heavy items in the buckets, shown by the increasing gap between ReSketch and the static variant.

In the shrink phase, throughput of ReSketch increases as the sketch shrinks. GS cannot shrink below its initial size M_0, and hence stagnates there while ReSketch continues to shrink down to M_2 (bottommost figure). ReSketch again exhibits lower estimation errors, also in presence of continued update operations, while error increases significantly for GS if updates are performed.

6.2.2 EnhancedMerge & Partition

We proceed to evaluate the performance of ReSketch EnhancedMerge and Partition operations. The aim is to compare the impact of these operations on estimation error. No baseline supports partitioning, hence these operations are benchmarked in isolation, while parameters are selected to align with the results of the sensitivity analysis.



(a) EnhancedMerge (b) Partition

Figure 9: DAG of sketch instances for Merge & Partition experiments.

Method. To this end, Figure 9 shows two DAGs of input datasets, sketch instances, and the structure-defining operations that link them. For both experiments, again a base memory budget of $M = 64\,\text{KiB}$ is assigned. For the EnhancedMerge experiment, the input dataset is partitioned into Dataset A (DA) and Dataset B (DB), and processed to yield sketches A and B, respectively. A and B are then merged to produce sketch C with a size of 128 KiB. The accuracy of queries C is then compared that of queries on a 128 KiB baseline sketch D, which receives the full input dataset.

For the Partition experiment, the input dataset is summarized in a sketch C (64 KiB) which is then partitioned into sketches A' and B', of half the size each, acting as if they had processed only their respective partition of the dataset.

The accuracy of these sketches is compared to 'ground truth' sketches A and B, which each have processed the corresponding partition of the input data.

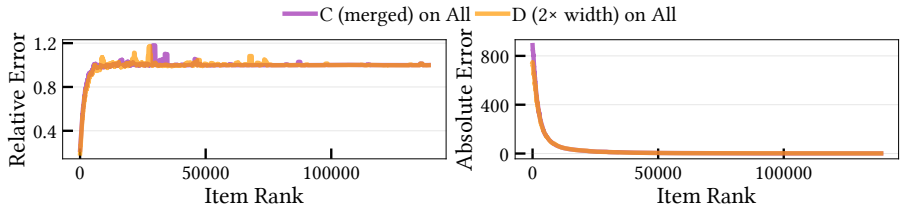


Figure 10: Accuracy after EnhancedMerge.

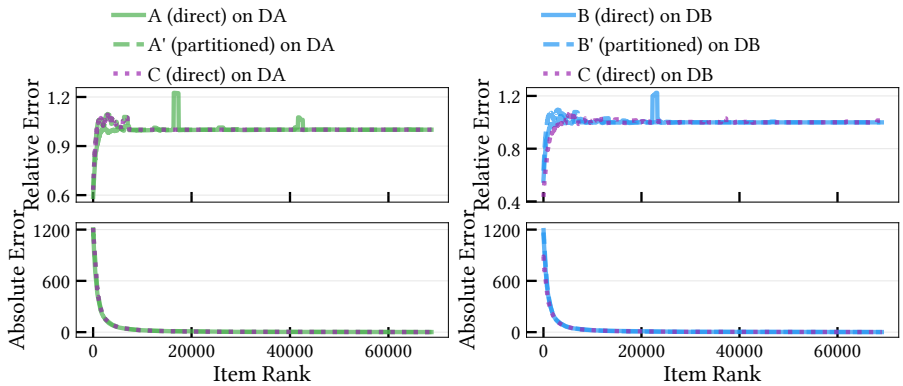


Figure 11: Accuracy after Partition.

Results. The relative and absolute error of every unique item in the input dataset (ordered by rank) are shown in Figure 10. Accuracy of merged sketch C, produced from merging two partition sketches A and B, is very similar to that of sketch D which has processed the complete input itself, as well as of A and B (shown in Table 3 as ARE and AAE along with run-to-run variances), as expected from Lemma 13. Similarly, for Partition, Figure 11 shows the accuracy for the two dataset partitions DA and DB achieved by partitioned sketches A' and B' compared to the respective baseline sketches A and B, again exhibiting very similar accuracy. Table 4 shows the estimation accuracy of each sketch instance. We also compare with the larger ancestor sketch C, and find that the partition operation has succeeded in maintaining the accuracy that could be achieved by querying C for the respective partition's items (cf. Lemma 14).

6.3 Application Example

Finally, we evaluate the performance of ReSketch for supporting a realistic end-to-end application (Figure 1) requiring all types of structural operations. We consider a representative execution of this system, shown in Figure 6, which prior work could not support.

Method. Each processing period (node) processes 2 M items from the

Table 3: EnhancedMerge accuracy.

Sketch	ARE	AAE
A	0.9919 ± 0.0017	27.95 ± 0.39
B	0.9914 ± 0.0034	25.96 ± 0.35
C	0.9927 ± 0.0021	27.09 ± 0.26
D	0.9898 ± 0.0019	25.69 ± 0.27

Table 4: Partition accuracy.

Sketch	ARE	AAE
A'	0.9994 ± 0.0017	37.05 ± 0.87
B'	0.9977 ± 0.0024	30.89 ± 0.70
A	0.9970 ± 0.0026	34.73 ± 1.03
B	0.9967 ± 0.0017	34.33 ± 0.91
C	0.9957 ± 0.0014	33.00 ± 0.44

CAIDA dataset (except D1 and E1 which process no more updates). Update throughput and query accuracy in terms of ARE and AAE is recorded for each processing period, and the latency of each structure-defining operation is measured.

Execution begins with one node (yellow in the DAG) summarizing an input stream to A1, and as more data arrives and ample memory is available, the sketch is expanded to A2 to improve accuracy as the process continues. After some time, a second node (blue) joins the system and begins processing a separate data stream in sketch B1. Meanwhile, the first node has to reallocate memory to a higher priority task, and the sketch relinquishes a large fraction of its memory. Eventually, node 1 departs from the system, so all processing is moved to the second node, and the two active sketch instances are merged to form C1. Finally, a new node (green) arrives, and the large C1 sketch is partitioned into D1 and E1 to load balance over both available nodes according to their capacities.

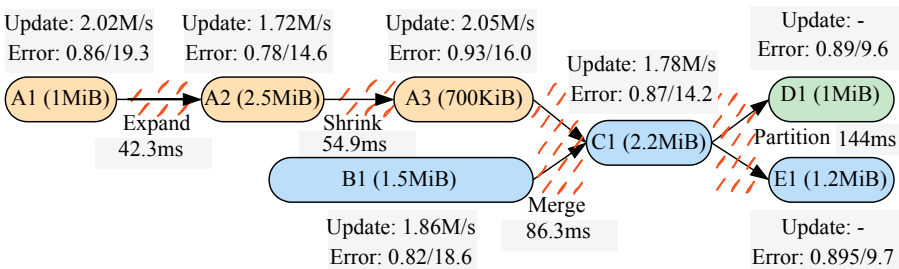


Figure 12: Application execution modelled as DAG (Figure 6. Gray boxes contain performance results; nodes (sketch instances) are annotated with achieved update throughput and accuracy (ARE/AAE), edges (structure-defining operations) with latency.

Results. The results are shown in Figure 12. The asymptotic complexities are reflected in the small latencies for structure-defining operations, which are expected to be invoked much less frequently than data manipulation; expand and shrink and merge are all similar, scaling depending on their input size, while the complexity of Partition and the size of its inputs determine its 144 ms latency.

Key Takeaways From Evaluation

Summary: 1. Sensitivity analysis reveals the balancing trade-off between memory, accuracy, and performance, showing how to allocate that memory between M, w, and k. 2. Importantly, the analysis shows the benefit of using multiple smaller KLL instances per row, aligning with and extending earlier results [19], [21], [29], [34] about partitioning leading to better memory-accuracy ratios. 3. Resize, Enhanced-Merge, and Partition are a powerful set of operations for supporting elasticity; they maintain their theoretical accuracy and performance as per the lemmas in section 5 also in realistic benchmarks. 4. A realistic use-case for ReSketch is evaluated to demonstrate its suitability for real-world applications.
These results show the suitability of ReSketch for long-lived data analytics processes with dynamic memory availability, due to its ability to be resized, partitioned, and merged dynamically during execution. The sketch maintains its throughput performance and accuracy throughout these operations, while growing into available memory to improve estimation accuracy, or shrinking to free up memory for other work, continuing to process items while maintaining orders of magnitude better accuracy than prior methods. ReSketch natively captures the dynamic nature of distributed monitoring systems through corresponding structure-defining operations, addressing the gap left by limitations in existing approaches which lack the necessary generalized flexibility.

6.4 Further Discussion of Results

Throughput trade-off. As shown throughout the benchmarks (Figure 8), ReSketch achieves substantially better estimation accuracy than all baselines, but at a lower update throughput compared to the plain Count-Min Sketch. This gap stems from the per-bucket distribution summaries (in our case, KLL quantile sketches) that ReSketch maintains on every insertion to support redistribution during structural operations. More generally, this kind of slowdown is not unique to ReSketch. Any sketch that supports resizing must maintain additional structure or perform additional bookkeeping beyond bare CMS, so some throughput cost relative to static CMS is expected. Depending on the sketch size and execution stage, ReSketch, DCMS, and Geometric Sketch each lead in different parts of the throughput curves, rather than one resizable design dominating the others. Note that for ReSketch, the KLL is a full quantile summary, which might be richer than necessary for the redistribution task. Whether a simpler, lighter-weight distribution summary or estimator can achieve a similar (ε, δ) guarantee while reducing the per-update cost, and potentially also the cost of structural operations that redistribute these summaries, is an open question worth investigating.

Coordination policies. While ReSketch provides the algorithmic primitives for resizing, merging, and partitioning (Figure 12), deciding *when* and *by how much* to invoke these operations remains an open practical question. Automating this process requires coordination policies that carefully balance redistribution overhead against expected improvements in accuracy across both sequential and parallel settings. Developing automated algorithms for such policies, alongside evaluating their effectiveness under stress scenarios (e.g., frequent churn, repeated oscillation), additional heterogeneity settings,

and sensitivity to operation sequence length, constitutes an important direction for future work. In parallel environments, coordination becomes further challenging, yet also more rewarding: ReSketch's primitives could enable elastic scaling for domain-splitting frameworks [19], [21], [29], [34], where prior work has demonstrated superlinear performance improvements from workload distribution across workers.

7 Other Related Work

Dynamic resizing has been studied extensively in approximate membership testing because practical deployments often involve changing set cardinalities, memory budgets and throughput requirements over time, so the structure must adapt. Representative examples include the consistent-hashing-based cuckoo-filter line of work, including the Consistent Cuckoo Filter and the Bucket-Level Elastic Cuckoo Filter, which supports bucket-level elasticity [24], [40], and InfiniFilter, which studies a different approach to supporting continued filter growth while maintaining favorable cost trade-offs [13]. As discussed throughout this work, dynamic resizing has also been considered for frequency estimation, including Dynamic Count-Min Sketch [41], ElasticSketch [38], and Geometric Sketch [4]. This line of work is undeniably important because frequency estimation is not only for point-query, but also a building block for a broad range of stream summarization tasks, including heavy hitter detection, frequency moments, inner products, quantile estimation, and range aggregates, so improvements in flexibility can benefit many downstream summaries. At the same time, the problem is less straightforward than in membership testing, since approximate membership structures typically retain per-item fingerprints or directly relocatable item representations, whereas frequency sketches store aggregated state, which makes structural adaptation substantially less direct. In this context, ReSketch targets a broader combination of properties, namely fine-grained resizability together with enhanced mergeability and partitionability, which, to the best of our knowledge, has not been studied in the literature and carries broader implications as motivated throughout this article. Accordingly, ideas from elastic membership filters, especially consistent-hashing-based remapping and elastic hashed layouts, are related in spirit and partly inspirational, while sketching for frequency estimation requires further novel algorithmic components that work in tandem to achieve the desired properties, as contributed in this work.

8 Conclusions

ReSketch is a novel sketch algorithmic design that delivers three critical properties — *resizability*, *enhanced mergeability*, and *partitionability*. Beyond that, this work also lays the foundation for analyzing approximation error in dynamic distributed sketches through the *instance provenance DAG*, a formal framework that enables to reason about and bound the approximation error of sketch instances produced by arbitrary sequences of operations, which is

really important to ensure that the resulting sketches maintain rigorous guarantees. Empirically, ReSketch demonstrates high accuracy and competitive throughput on evaluation, validating its practicality. As noted in subsection 6.4, the structural flexibility that ReSketch provides introduces trade-offs, such as lower update throughput compared to plain Count-Min Sketch; potential mitigations for these overheads are also discussed therein and worth further investigation in future work. As further noted in subsection 6.4, developing coordination policies to automate structural operations in both sequential and parallel settings remains an open practical question that warrants a dedicated study. Overall, by offering a blueprint for transforming rigid matrix-based sketches into (enhanced) mergeable, partitionable, and resizable structures, we believe ReSketch and future work along these lines will pave the way for a new generation of adaptive approximate query processing systems capable of seamless elastic scaling and rebalancing.

Acknowledgments

Work supported by Marie Skłodowska-Curie Doctoral Network RELAX-DN, funded by EU under Horizon Europe 2021-2027 FP Grant Agreement nr. 101072456; 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".

Bibliography

[1] S. Acharya, P. B. Gibbons, V. Poosala and S. Ramaswamy, 'Join synopses for approximate query answering,' in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '99, New York, NY, USA: Association for Computing Machinery, Jun. 1999, pp. 275–286. DOI: 10.1145/304182.304207.

[2] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei and K. Yi, 'Mergeable summaries,' in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, ser. PODS '12, New York, NY, USA: Association for Computing Machinery, May 2012, pp. 23–34. DOI: 10.1145/2213556.2213562.

[3] N. Alon, P. B. Gibbons, Y. Matias and M. Szegedy, 'Tracking join and self-join sizes in limited storage,' in *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. PODS '99, New York, NY, USA: Association for Computing Machinery, May 1999, pp. 10–20. DOI: 10.1145/303976.303978.

[4] D. Biton, R. Friedman and R. Shahout, 'Geometric Sketch: The Inflatable-Shrinkable Sketch,' en, in *Advanced Information Networking and Applications*, L. Barolli, Ed., Cham: Springer Nature Switzerland, 2025, pp. 270–281. DOI: 10.1007/978-3-031-87766-7_24.

[5] B. H. Bloom, 'Space/time trade-offs in hash coding with allowable errors,' *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. DOI: 10.1145/362686.362692.

[6] CAIDA, *The CAIDA UCSD Anonymized Internet Traces - 2018-03-15*, 2018.

[7] M. Charikar, K. Chen and M. Farach-Colton, 'Finding frequent items in data streams,' *Theoretical Computer Science*, Automata, Languages and Programming, vol. 312, no. 1, pp. 3–15, Jan. 2004. DOI: 10.1016/S0304-3975(03)00400-6.

[8] Y. Collet, *xxHash*, Jan. 2026.

[9] G. Cormode and M. Garofalakis, 'Sketching streams through the net: Distributed approximate query tracking,' in *Proceedings of the 31st international conference on Very large data bases*, ser. VLDB '05, Trondheim, Norway: VLDB Endowment, Aug. 2005, pp. 13–24.

[10] G. Cormode and M. Hadjieleftheriou, 'Finding frequent items in data streams,' *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1530–1541, Aug. 2008. DOI: 10.14778/1454159.1454225.

[11] G. Cormode and S. Muthukrishnan, 'An improved data stream summary: The count-min sketch and its applications,' *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005. DOI: 10.1016/j.jalgor.2003.12.001.

[12] G. Cormode and S. Muthukrishnan, 'Summarizing and Mining Skewed Data Streams,' en, in *Proceedings of the 2005 SIAM International Conference on Data Mining*, Newport Beach, CA, USA: Society for Industrial and Applied Mathematics, Apr. 2005, pp. 44–55. DOI: 10.1137/1.9781611972757.5.

[13] N. Dayan, I. Bercea, P. Reviriego and R. Pagh, 'Infinifilter: Expanding filters to infinity and beyond,' *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–27, 2023. DOI: 10.1145/3589285.

[14] T. Dunning, 'The t-digest: Efficient estimates of distributions,' *Software Impacts*, vol. 7, p. 100 049, Feb. 2021. DOI: 10.1016/j.simpa.2020.100049.

[15] M. Garofalakis, J. Gehrke and R. Rastogi, Eds., *Data Stream Management: Processing High-Speed Data Streams* (Data-Centric Systems and Applications), en. Berlin, Heidelberg: Springer, 2016. DOI: 10.1007/978-3-540-28608-0.

[16] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan and M. Strauss, 'Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries,' in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sep. 2001, pp. 79–88.

[17] L. Gu, Y. Tian, W. Chen, Z. Wei, C. Wang and X. Zhang, 'Per-Flow Network Measurement With Distributed Sketch,' *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, pp. 411–426, Feb. 2024. DOI: 10.1109/TNET.2023.3286879.

[18] D. Harris, A. Rinberg and O. Rottenstreich, 'Compressing Distributed Network Sketches With Traffic-Aware Summaries,' *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 1962–1975, Jun. 2023. DOI: 10.1109/TNSM.2022.3172299.

[19] M. Hilgendorf and M. Papatriantafilou, 'LMQ-Sketch: Lagom Multi-Query Sketch for High-Rate Online Analytics,' in *39th International Symposium on Distributed Computing (DISC 2025)*, D. R. Kowalski, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 356, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 36:1–36:24. DOI: 10.4230/LIPIcs.DISC.2025.36.

[20] N. Ivkin, E. Liberty, K. Lang, Z. Karnin and V. Braverman, *Streaming Quantiles Algorithms with Small Space and Update Time*, arXiv:1907.00236 [cs], Jun. 2019. DOI: 10.48550/arXiv.1907.00236.

[21] V. Jarlow, C. Stylianopoulos and M. Papatriantafilou, 'QPOPSS: Query and Parallelism Optimized Space-Saving for finding frequent stream elements,' *Journal of Parallel and Distributed Computing*, vol. 204, p. 105 134, Oct. 2025. DOI: 10.1016/j.jpdc.2025.105134.

[22] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine and D. Lewin, 'Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web,' en, in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC '97*, El Paso, Texas, United States: ACM Press, 1997, pp. 654–663. DOI: 10.1145/258533.258660.

[23] Z. Karnin, K. Lang and E. Liberty, 'Optimal Quantile Approximation in Streams,' in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, New Brunswick, NJ, USA: IEEE, Oct. 2016, pp. 71–78. DOI: 10.1109/FOCS.2016.17.

[24] L. Luo, D. Guo, O. Rottenstreich, R. T. Ma, X. Luo and B. Ren, 'The Consistent Cuckoo Filter,' in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, Paris, France: IEEE Press, Apr. 2019, pp. 712–720. DOI: 10.1109/INFOCOM.2019.8737454.

[25] A. Metwally, D. Agrawal and A. E. Abbadi, 'An integrated efficient solution for computing frequent and top-k elements in data streams,' *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1095–1133, Sep. 2006. DOI: 10.1145/1166074.1166084.

[26] J. Misra and D. Gries, 'Finding repeated elements,' *Science of Computer Programming*, vol. 2, no. 2, pp. 143–152, Nov. 1982. DOI: 10.1016/0167-6423(82)90012-0.

[27] M. Mitzenmacher and E. Upfal, *Probability and computing: randomized algorithms and probabilistic analysis*, eng. Cambridge: Cambridge University Press, 2012.

[28] V. Q. Ngo and M. Hilgendorf, *ReSketch: A Mergeable, Redistributable, and Resizable Sketch*, 2025.

[29] V. Q. Ngo and M. Papatriantafilou, 'Cuckoo Heavy Keeper and the Balancing Act of Maintaining Heavy Hitters in Stream Processing,' en, *Proceedings of the VLDB Endowment*, vol. 18, no. 9, pp. 3149–3161, May 2025. DOI: 10.14778/3746405.3746434.

[30] O. Papapetrou, M. Garofalakis and A. Deligiannakis, 'Sketch-based querying of distributed sliding-window data streams,' *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 992–1003, Jun. 2012. DOI: 10.14778/2336664.2336672.

[31] W. R. Punter, O. Papapetrou and M. Garofalakis, 'OmniSketch: Efficient Multi-Dimensional High-Velocity Stream Analytics with Arbitrary Predicates,' *Proc. VLDB Endow.*, vol. 17, no. 3, pp. 319–331, Nov. 2023. DOI: 10.14778/3632093.3632098.

[32] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel and B. Sericola, 'Efficient key grouping for near-optimal load balancing in stream processing systems,' en, in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, Oslo Norway: ACM, Jun. 2015, pp. 80–91. DOI: 10.1145/2675743.2771827.

[33] Q. Shi, Y. Xu, J. Qi, W. Li, T. Yang, Y. Xu and Y. Wang, 'Cuckoo Counter: Adaptive Structure of Counters for Accurate Frequency and Top-k Estimation,' *IEEE/ACM Transactions on Networking*, vol. 31, no. 4, pp. 1854–1869, Aug. 2023. DOI: 10.1109/TNET.2022.3232098.

[34] C. Stylianopoulos, I. Walulya, M. Almgren, O. Landsiedel and M. Papatriantafilou, 'Delegation sketch: A parallel design with support for fast and accurate concurrent operations,' in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–16. DOI: 10.1145/3342195.3387542.

[35] M. Thorup, *High Speed Hashing for Integers and Strings*, arXiv:1504.06804 [cs], May 2020. DOI: 10.48550/arXiv.1504.06804.

[36] xxHash Contributors, *Performance comparison - Benchmarks concentrating on small data*, en, May 2024.

[37] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi and X. Li, 'HeavyGuardian: Separate and Guard Hot Items in Data Streams,' in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18, New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 2584–2593. DOI: 10.1145/3219819.3219978.

[38] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li and S. Uhlig, 'Elastic sketch: Adaptive and fast network-wide measurements,' in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 561–575. DOI: 10.1145/3230543.3230544.

[39] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen and X. Li, 'HeavyKeeper: An Accurate Algorithm for Finding Top- k Elephant Flows,' *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, Oct. 2019. DOI: 10.1109/TNET.2019.2933868.

[40] Y. Zhang, Q. Xiao, C. Guo, G. Pan, J. Huang, K. He, Y. Miao and W. Li, 'Bucket-Level Elastic Cuckoo Filter for Dynamic Set Membership Query and Encoded Set Operations,' *IEEE Transactions on Networking*, vol. 33, no. 5, pp. 2256–2275, Oct. 2025. DOI: 10.1109/TON.2025.3565545.

[41] X. Zhu, G. Wu, H. Zhang, S. Wang and B. Ma, 'Dynamic Count-Min Sketch for Analytical Queries Over Continuous Data Streams,' in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, Bengaluru, India: IEEE Computer Society, Dec. 2018, pp. 225–234. DOI: 10.1109/HiPC.2018.00033.