ReSketch: A Mergeable, Partitionable, and Resizable Sketch

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

Martin Hilgendorf
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden
martin.hilgendorf@chalmers.se

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

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.

PVLDB Reference Format:
Vinh Quang Ngo, Martin Hilgendorf, and Marina Papatriantafilou. ReSketch: A Mergeable, Partitionable, and Resizable Sketch. PVLDB, 18(9): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/vinhqngo5/ReSketch.

1 INTRODUCTION

Tracking item frequencies and derivative problems such as heavy hitter detection [7, 10, 27, 35], estimation of frequency moments [18], inner products [3], and range sums [15] are fundamental primitives for data-intensive systems. They are used for query optimization, approximate query processing, and join size estimation in databases [1, 28, 29]; identifying popular flows, DDoS mitigation, and change detection in networking [9, 17, 36]; load balancing and caching in distributed systems [30]; 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 [14], 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.

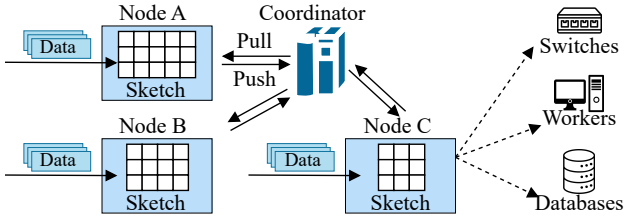


Figure 1: Distributed monitoring with heterogeneous nodes and dynamic workload maintaining sketches of varying sizes.

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 × 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 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 heterogeneous, there is no known method to merge them, and all sketches must be scanned. (iii) As

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

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 [38], others shrinkability [36], 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, 38] still logically stack multiple fixed-size sketches internally, so resizing does not make them compatible for merging.	• No support for merging heterogeneous-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 [16, 18, 20, 27, 32]. Known approaches rely on *static partitioning* defined at initialization.	Cannot dynamically rebalance partitions or migrate state (historical data) between sketches.

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* [21]. 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 [26], 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 [18, 20, 27, 32], 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. § 2 provides preliminaries on frequency estimation sketches and their mergeability. § 3 formalizes the three properties. § 4 presents the design of ReSketch. § 5 provides theoretical analysis. § 6 presents experimental results. Finally, § 7 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: $\widehat{f}(e) = \min_{i=1}^{d} \text{count}[i][h_i(e)]$.

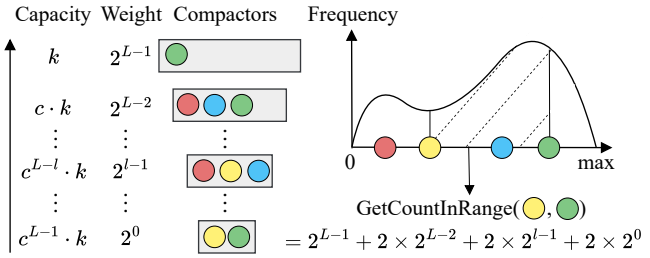


Figure 2: KLL sketch maintains multi-level compactors where items at level l have weight 2^{l-1}. The GetCountInRange operation estimates counts within any range, which generalizes both rank queries and single-item frequency estimation.

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)
```

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 [22] 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 $k = O((1/\varepsilon)\sqrt{\log(1/\delta)})$ guarantees that the estimated rank is within an additive error of εN with probability at least $1 - \delta$, using $O(k)$ 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 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 [24] and SpaceSaving [23] are mergeable [2]. For quantile estimation, KLL [21] and t-digest [13] 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 heterogeneous 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 $w' \neq w_1 + w_2$.*

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

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.
$\varepsilon_{KLL}, \delta_{KLL}$	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}.

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 than simple counters. As detailed in Table 2, each bucket contains: (1) A primary counter, *buckets*$[i][j]$*.count*,

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
2.1  Procedure FindBucket(e_hash, ring)
2.2      Search for first (p, id) in ring where p ≥ e_hash
2.3      if no such pair found then return first id in ring
2.4      else return id
2.5  Procedure Update(e)
2.6      for i ← 0 to d − 1 do
2.7          y ← Hash(e, seeds[i])
2.8          id ← FindBucket(y, rings[i])
2.9          buckets[i][id].count += 1
2.10         buckets[i][id].q_sketch.UpdateKLL(y)
2.11 Procedure Query(e)
2.12     min_count ← ∞; estimates[d] ← []
2.13     for i ← 0 to d − 1 do
2.14         y ← Hash(e, seeds[i])
2.15         id ← FindBucket(h, rings[i])
2.16         row_kll_est ← buckets[i][id].q_sketch.QueryFrequency(y)
2.17         estimates[i] ← row_kll_est
2.18     return median(estimates)    // Estimator: median of KLL estimates
```

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 [22] (with the same parameter k each) due to its analytical guarantees, and efficient algorithmic implementations in existing works [19].

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 mod$w$ serves as a *bucket-mapping function* that 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 [21]. 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). 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 § 5.

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

Algorithm 3: ReSketch – Redistribute Row

```
3.1  Procedure RedistRow(in_ring, in_buckets, out_ring)
3.2      out_buckets ← new array of empty buckets for out_ring
3.3      all_points ← sorted list of unique hash points from in_ring and out_ring
         // Iterate through the disjoint ranges
3.4      for i ← 0 to length of all_points − 2 do
3.5          start, end ← all_points[i], all_points[i + 1]
             // For each, find its source and destination
3.6          in_id ← FindBucket(start, in_ring)
3.7          out_id ← FindBucket(start, out_ring)
3.8          in_bucket ← in_buckets[in_id]
3.9          count ← in_bucket.q_sketch.QueryCountInRange(start, end)
3.10         if count > 0 then
                 // Move the data for this range to its destination
3.11             out_buckets[out_id].count += count
3.12             sub_sketch ← in_bucket.q_sketch.FilterKLL(start, end)
3.13             out_buckets[out_id].q_sketch.MergeKLL(sub_sketch)
3.14     return out_buckets
```

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

```
4.1  Procedure QueryFrequency(e_hash)
4.2      estimated_count ← 0
4.3      for each level l do
4.4          for each value in compactor at level l do
4.5              if value = e_hash then estimated_count += 2^l
4.6      return estimated_count
4.7  Procedure FilterKLL(start, end)
4.8      new_sketch ← new KLLSketch()
         // Filter values within specified range to build new KLL
4.9      for each level l of this sketch's Compactors do
4.10         for each value in compactor at level l do
4.11             if start < value ≤ end then
4.12                 Add value to compactor at level l in new_sketch
4.13                 new_sketch.total_count += 2^l
4.14     return new_sketch
```

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 new ring. The function then iterates through these intervals. For an interval (*start*, *end*), it identifies the source bucket in the old

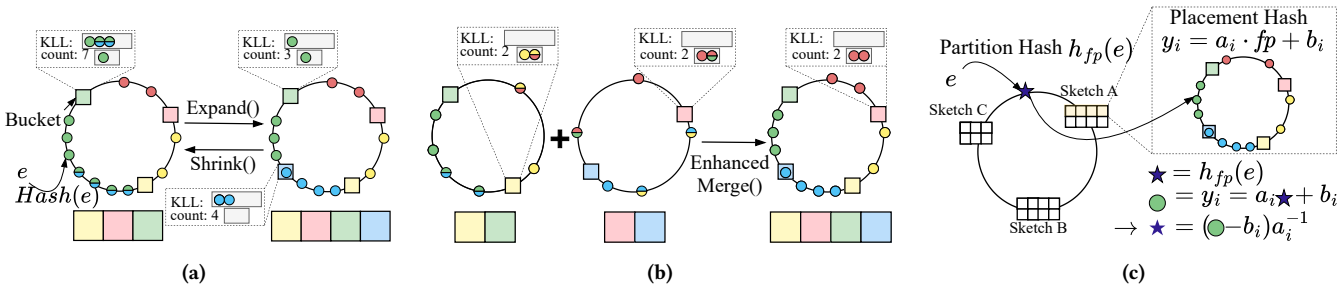


Figure 3: ReSketch's structure-defining operations: (a) Resize (in form of Expand and Shrink) operations add or remove boundary points on the consistent hash ring, redistributing counts using per-bucket KLL sketches. (b) EnhancedMerge creates a unified ring with a combined width and redistributes both input sketches to the new ring structure. (c) Partition-aware hashing: 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, therefore introducing no additional storage usage.

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.

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 4.7), 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 3a and Figure 3b.

Properties. The redistribution operation takes $O((w+w')(\log(w+w')+k))$ steps, which can be broken down as follows: First, creating the sorted list of boundary points takes $O(w + w')$ steps, where w' is the target width. Second, the algorithm iterates through the resulting $O(w + w')$ intervals; for each, identifying the source and destination buckets using binary search takes $O(\log(w+w'))$ steps, and the quantile operations (`FilterKLL` and `MergeKLL`) perform linear scans of compactors in $O(k)$ time. The formal guarantees on the associated approximation error are provided in § 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 § 4.2, regarding hashing). (Note that when `Partition` support is required, this is replaced by the partition-aware hashing detailed in § 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 2.16, 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 $O(d(\log w + \log k))$ steps, which accounts for d consistent hashings (i.e. $\log w$ each) and KLL updates ($O(\log k)$ each). The term $\log k$ is achieved by using the lazy compaction strategies from [19] (compared to $O(k)$ in the standard version [22]). Queries take $O(d(\log w + k))$ steps, which can be optimized to $O(d(\log w + \log k))$ for batch queries if the internal compactors are sorted beforehand. The approximation bounds for this estimator are derived in § 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 3a). 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

```
5.1 Procedure Resize(A, w')
5.2     buckets ← new bucket array of size A.d × w'; Δ ← w' − A.w
5.3     for i ← 0 to A.d − 1 do
5.4         out_ring ← A.rings[i]
5.5         if Δ > 0 then Add Δ random points to out_ring
5.6         else Remove |Δ| random points from out_ring
5.7         buckets[i] ← RedistRow(A.rings[i], A.buckets[i], out_ring)
5.8         A.rings[i] ← out_ring;
5.9     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 3b 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.

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
```

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, 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 3c), which replaces the standard pairwise-independent hashing (highlighted in red).

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 3c). 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.

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 (mod$p$), 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 [25]. 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.

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
```

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 3c 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 [33], 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 $fp = (y - b_i) \cdot a_i^{-1} \pmod{2^W}$.

Time Complexity. The cost of `Resize` is determined by the cost of `RedistRow` across d rows, which is $O(d(w + w')(\log(w + w') + k))$. 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 $O(k)$ [22]

steps, the additional merging cost is $O(d(w_1 + w_2)k)$. Adding this to the redistribution cost yields a total complexity of $O(d((w_1 + w_2)\log(w_1 + w_2) + k))$. 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 $O(k)$ items, and considering that re-inserting each item requires finding the target bucket in $O(\log w)$ and adding it as a weighted sample in $O(1)$, this operation has a total time complexity of $O(dwk \log w)$.

5 RESKETCH ANALYSIS

In this section, we analyze ReSketch's frequency estimation approximation bounds. We begin in § 5.1 by establishing the bound for a static single sketch and then extend to dynamic and distributed cases in § 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, $|\widehat{f}(e) - f(e)| \le 2\varepsilon_{KLL} \cdot N$ 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 $|\widehat{f}(e) - f(e)| \le 2\varepsilon_{KLL} \cdot N$ 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 $|\widehat{f}(i)e - f(e)| \le \varepsilon$ 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 $d = O(\log(1/\delta))$.*

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., $X \ge d/2$. We have $\mathbb{E}[X] = p \cdot d < d/2$.

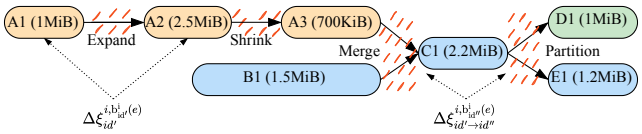


Figure 4: Instance provenance DAG illustrating error tracking across operations. Each node represents a sketch instance with its size, also represents data processing period with error reference $\xi^{i,b^i_{id'}}_{id'}(e)$; solid edges represent structure-defining operations with error reference $\xi^{i,b^i_{id'}}_{id'\to id''}(e)$. 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 $\mathcal{A}_{B1}$).

By the Chernoff bound, $\Pr[X \ge d/2] = \exp(-\Theta(d))$. Setting $d = O(\log(1/\delta))$ 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 $|\widehat{f}(e) - f(e)| \le \varepsilon N$ with probability $\ge 1 - \delta$ using space $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\right)$.*

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. $\le \delta_{KLL}$), or (2) the bucket load satisfies $2\varepsilon_{KLL} \cdot C^{i,j} > \varepsilon N$ (the factor 2 follows from Lemma 5). By Markov, $\Pr[(2)] \le \frac{2\varepsilon_{KLL} \cdot 2N/w}{\varepsilon N} = \frac{4\varepsilon_{KLL}}{\varepsilon w}$, so $P_{\text{row\_fail}} \le \delta_{KLL} + \frac{4\varepsilon_{KLL}}{\varepsilon w}$.

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 $P_{\text{row\_fail}} \le 1/3$, 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 $d = O(\log(1/\delta))$, the median fails with probability $\le \delta$.

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

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 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 4) 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* $\xi^{i,\mathrm{b}^i_{id}(e)}_{\cdot}$ 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 $\xi^{i,\mathrm{b}^i_{id}(e)}_{\cdot}$ *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:*

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

Remark. The following lemmas show that the *accuracy requirements in Definitions 1, 2, and 3 are met.* Due to space constraints, we provide their proofs in the extended version [26].

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:* $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id}(e)}_{id}] = 2\Delta N_{id}/w_{id}$.

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:* $\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'} = 0$.

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:* $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}] = \mathbb{E}[\Delta C^{i,\mathrm{b}^i_{id'}(e)}_{id\to id'}] = 2N_{id}(1/w' - 1/w)$.

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:* $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{(id_1,id_2)\to id'}] = 0$.

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:* $\mathbb{E}[\xi^{i,\mathrm{b}^i_{id'}(e)}_{id\to(id_1,id_2)}] = 0$.

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:*

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

Proof. For row i, by Lemma 5, the KLL sketch satisfies $|\widehat{f_i}(e) - f(e)| \le 2\varepsilon_{KLL}\cdot\Xi^{i,\mathrm{b}^i_{id}(e)}_{id}$ with probability $1-\delta_{KLL}$, where $\Xi^{i,\mathrm{b}^i_{id}(e)}_{id}$ serves as stream length for the KLL in that bucket. By Definition 9, and Lemmas 10–14, we have: $\mathbb{E}[\Xi^{i,\mathrm{b}^i_{id}(e)}_{id}] = \sum_{id'\in\mathcal{P}_{id}} \frac{2\Delta N_{id'}}{w_{id'}} + \sum_{(id'\to id'')\in\mathcal{Q}_{id}} \left(\frac{2N_{id'}}{w_{id''}} - \frac{2N_{id'}}{w_{id'}}\right)$. The row-$i$ estimate fails if either: (1) The KLL fails internally with probability $\le \delta_{KLL}$, or (2) The cumulative error $2\varepsilon_{KLL}\cdot\Xi^{i,\mathrm{b}^i_{id}(e)}_{id}$ exceeds εN_{id}. By Markov's inequality, $\Pr[2\varepsilon_{KLL}\cdot\Xi^{i,\mathrm{b}^i_{id}(e)}_{id} \ge \varepsilon N_{id}] \le \frac{2\varepsilon_{KLL}\cdot\mathbb{E}[\Xi^{i,\mathrm{b}^i_{id}(e)}_{id}]}{\varepsilon N_{id}}$. Let X be the left-hand side of the condition inequality. Then $\mathbb{E}[\Xi^{i,\mathrm{b}^i_{id}(e)}_{id}] = 2X$. 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 $d = O(\log(1/\delta))$ 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 § 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 [31, 36, 37]. 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 § 2, an efficient, established sketch, however lacking support for resizing and enhanced mergeability. (2) Dynamic Count-Min Sketch (DCMS) [38], 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 $h_{fp}(e)$ (§ 6.12), due to its high throughput on small inputs and good quality regarding collisions and random dispersion of hash values [34].

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* ($\frac{1}{|S|}\sum_{e\in S}(\frac{|f(e)-\widehat{f}(e)|}{f(e)}-\text{ARE})^2$) measures the dispersion of relative errors across items within a single run. (7) *AAE variance* ($\frac{1}{|S|}\sum_{e\in S}(|f(e)-\widehat{f}(e)|-\text{AAE})^2$) 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 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

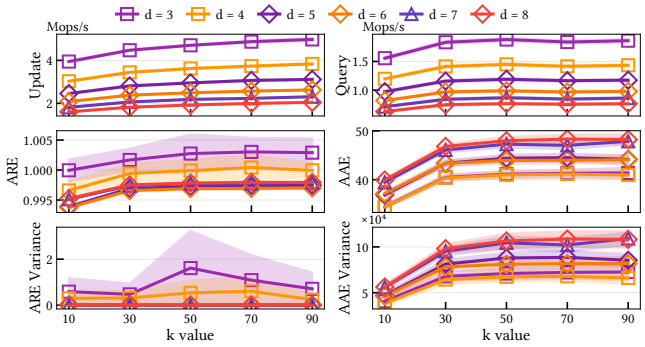


Figure 5: Varying k and d (rows) for ReSketch with fixed $M = 64$ KiB.

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 $M = 64$ KiB here, while the remaining figures can be found alongside the implementation, in the open repository [26]. 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 $154000 \cdot 24 \approx 1203$ 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 5. 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. 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.

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

[3]Open-source, available at https://github.com/vinhqngo5/ReSketch, [26]

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

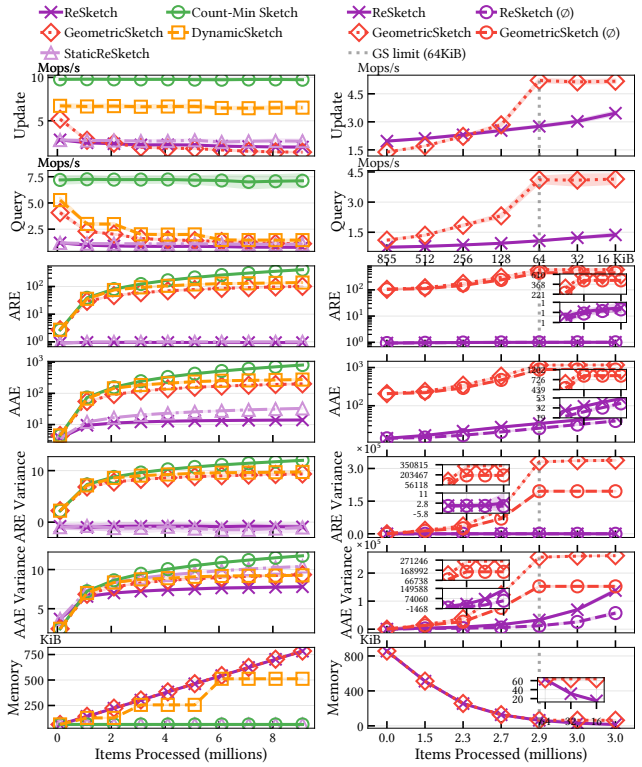


Figure 6: Comparison of performance subject to resizing operations compared to baselines (partially) supporting them; expand on the left and shrink on the right.

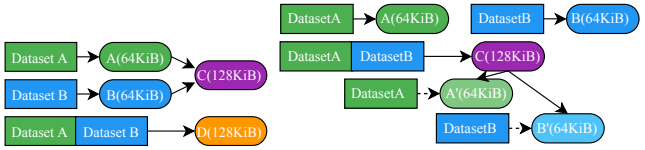
6.2.1 Resize. We evaluate the performance of ReSketch when resizing, in terms of throughput and accuracy. We compare with the resizable baselines as well as a static, non-resized CMS and ReSketch instance to see potential performance or accuracy changes due to resizing, while highlighting the benefit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

Method. The benchmark proceeds in two phases. First, all sketches are initialized at $M_0 = 64$ 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 $M_\Delta = 8$ KiB, and the process continues until 10 M items have been processed and the sketches have grown to $M_1 = 864$ 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 $M_2 = 16$ KiB. Accuracy is again recorded after each shrink. One copy of each sketch is kept pristine (denoted ∅), 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 6 shows results in all metrics for the expand and shrink 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 7: DAG of sketch instances for Merge & Partition experiments.

Method. To this end, Figure 7 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 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.

Results. The relative and absolute error of every unique item in the input dataset (ordered by rank) are shown in Figure 8. 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

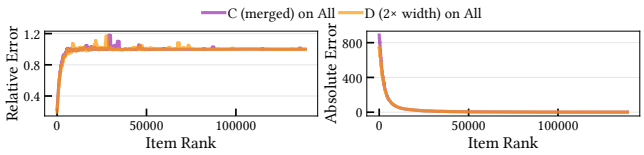


Figure 8: Accuracy after EnhancedMerge.

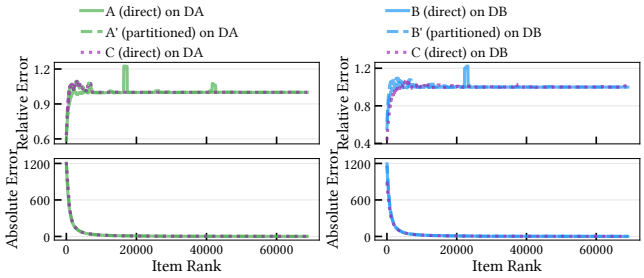


Figure 9: Accuracy after Partition.

Lemma 13. Similarly, for Partition, Figure 9 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).

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

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 4, which prior work could not support.

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

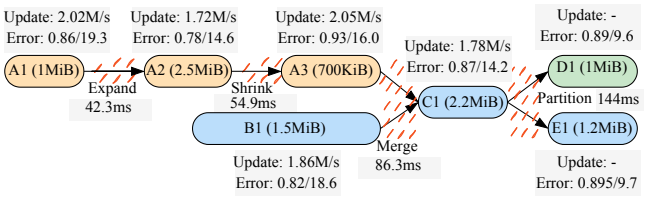


Figure 10: Application execution modelled as DAG (Figure 4. Gray boxes contain performance results; nodes (sketch instances) are annotated with achieved update throughput and accuracy (ARE/AAE), edges (structure-defining operations) with latency.

Results. The results are shown in Figure 10. 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 [18, 20, 27, 32] about partitioning leading to better memory-accuracy ratios. (3) Resize, EnhancedMerge, and Partition are a powerful set of operations for supporting elasticity; they maintain their theoretical accuracy and performance as per the lemmas in § 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.

7 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. By offering a blueprint for transforming rigid matrix-based sketches into (enhanced) mergeable, partitionable, and resizable structures, we believe ReSketch paves the way for a new generation of adaptive approximate query processing systems capable of seamless elastic scaling and rebalancing.

REFERENCES

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/304182.304207

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

[3] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. 1999. Tracking join and self-join sizes in limited storage. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '99)*. Association for Computing Machinery, New York, NY, USA, 10–20. https://doi.org/10.1145/303976.303978

[4] Dvir Biton, Roy Friedman, and Rana Shahout. 2025. Geometric Sketch: The Inflatable-Shrinkable Sketch. In *Advanced Information Networking and Applications*, Leonard Barolli (Ed.). Springer Nature Switzerland, Cham, 270–281. https://doi.org/10.1007/978-3-031-87766-7_24

[5] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[6] CAIDA. 2018. The CAIDA UCSD Anonymized Internet Traces - 2018-03-15. https://catalog.caida.org/dataset/passive_2018_pcap

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

[8] Yann Collet. 2026. xxHash. https://github.com/Cyan4973/xxHash

[9] Graham Cormode and Minos Garofalakis. 2005. Sketching streams through the net: distributed approximate query tracking. In *Proceedings of the 31st international conference on Very large data bases (VLDB '05)*. VLDB Endowment, Trondheim, Norway, 13–24. https://dl.acm.org/doi/abs/10.5555/1083592.1083598

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

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

[12] Graham Cormode and S. Muthukrishnan. 2005. Summarizing and Mining Skewed Data Streams. In *Proceedings of the 2005 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, Newport Beach, CA, USA, 44–55. https://doi.org/10.1137/1.9781611972757.5

[13] Ted Dunning. 2021. The t-digest: Efficient estimates of distributions. *Software Impacts* 7 (Feb. 2021), 100049. https://doi.org/10.1016/j.simpa.2020.100049

[14] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi (Eds.). 2016. *Data Stream Management: Processing High-Speed Data Streams*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-28608-0

[15] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. 2001. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 79–88.

[16] Liyuan Gu, Ye Tian, Wei Chen, Zhongxiang Wei, Cenman Wang, and Xinming Zhang. 2024. Per-Flow Network Measurement With Distributed Sketch. *IEEE/ACM Transactions on Networking* 32, 1 (Feb. 2024), 411–426. https://doi.org/10.1109/TNET.2023.3286879

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

[18] Martin Hilgendorf and Marina Papatriantafilou. 2025. LMQ-Sketch: Lagom Multi-Query Sketch for High-Rate Online Analytics. In *39th International Symposium on Distributed Computing (DISC 2025) (Leibniz International Proceedings in Informatics (LIPIcs))*, Dariusz R. Kowalski (Ed.), Vol. 356. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 36:1–36:24. https://doi.org/10.4230/LIPIcs.DISC.2025.36

[19] Nikita Ivkin, Edo Liberty, Kevin Lang, Zohar Karnin, and Vladimir Braverman. 2019. Streaming Quantiles Algorithms with Small Space and Update Time. https://doi.org/10.48550/arXiv.1907.00236 arXiv:1907.00236 [cs].

[20] Victor Jarlow, Charalampos Stylianopoulos, and Marina Papatriantafilou. 2025. QPOPSS: Query and Parallelism Optimized Space-Saving for finding frequent stream elements. *J. Parallel and Distrib. Comput.* 204 (Oct. 2025), 105134. https://doi.org/10.1016/j.jpdc.2025.105134

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC '97*. ACM Press, El Paso, Texas, United States, 654–663. https://doi.org/10.1145/258533.258660

[22] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal Quantile Approximation in Streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, New Brunswick, NJ, USA, 71–78. https://doi.org/10.1109/FOCS.2016.17

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

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

[25] Michael Mitzenmacher and Eli Upfal. 2012. *Probability and computing: randomized algorithms and probabilistic analysis*. Cambridge University Press, Cambridge.

[26] Vinh Quang Ngo and Martin Hilgendorf. 2025. ReSketch: A Mergeable, Redistributable, and Resizable Sketch. https://github.com/vinhqngo5/ReSketch

[27] Vinh Quang Ngo and Marina Papatriantafilou. 2025. Cuckoo Heavy Keeper and the Balancing Act of Maintaining Heavy Hitters in Stream Processing. *Proceedings of the VLDB Endowment* 18, 9 (May 2025), 3149–3161. https://doi.org/10.14778/3746405.3746434

[28] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. 2012. Sketch-based querying of distributed sliding-window data streams. *Proc. VLDB Endow.* 5, 10 (June 2012), 992–1003. https://doi.org/10.14778/2336664.2336672

[29] Wieger R. Punter, Odysseas Papapetrou, and Minos Garofalakis. 2023. OmniSketch: Efficient Multi-Dimensional High-Velocity Stream Analytics with Arbitrary Predicates. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 319–331. https://doi.org/10.14778/3632093.3632098

[30] Nicoló Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. 2015. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, Oslo Norway, 80–91. https://doi.org/10.1145/2675743.2771827

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

[32] Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. 2020. Delegation sketch: a parallel design with support for fast and accurate concurrent operations. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3342195.3387542

[33] Mikkel Thorup. 2020. High Speed Hashing for Integers and Strings. https://doi.org/10.48550/arXiv.1504.06804 arXiv:1504.06804 [cs] version: 9.

[34] xxHash Contributors. 2024. Performance comparison - Benchmarks concentrating on small data. https://github.com/Cyan4973/xxHash/wiki/Performance-comparison/c624714b9a0d455eb474c7ae3f92864451d7b4be#benchmarks-concentrating-on-small-data-

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

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

[37] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. *IEEE/ACM Transactions on Networking* 27, 5 (Oct. 2019), 1845–1858. https://doi.org/10.1109/TNET.2019.2933868

[38] Xiaobo Zhu, Guangjun Wu, Hong Zhang, Shupeng Wang, and Bingnan Ma. 2018. Dynamic Count-Min Sketch for Analytical Queries Over Continuous Data Streams. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE Computer Society, Bengaluru, India, 225–234. https://doi.org/10.1109/HiPC.2018.00033