Sublime: Sublinear Error & Space for Unbounded Skewed Streams

NAVID ESLAMI, University of Toronto, Canada
IOANA O. BERCEA, KTH Royal Institute of Technology, Sweden
RASMUS PAGH, BARC, University of Copenhagen, Denmark
NIV DAYAN, University of Toronto, Canada

Modern stream processing systems often need to track the frequency of distinct keys in a data stream in real-time. Since maintaining exact counts can require a prohibitive amount of memory, many applications rely on compact, probabilistic data structures known as frequency estimation sketches to approximate them. However, mainstream frequency estimation sketches fall short in two critical aspects. First, they are memory-inefficient under skewed workloads because they use uniformly-sized counters to count the keys, thus wasting memory on storing the leading zeros of many small counts. Second, their estimation error deteriorates at least linearly with the length of the stream—which may grow indefinitely—because they rely on a fixed number of counters.

We present Sublime, a framework that generalizes frequency estimation sketches to address these challenges. To reduce memory footprint under skew, Sublime begins with short counters and dynamically elongates them as they overflow, storing their extensions within the same cache line. It employs efficient bit manipulation routines to quickly locate and access a counter's extensions. To maintain accuracy as the stream grows, Sublime also expands its number of counters at a configurable rate, exposing a new spectrum of accuracy-memory tradeoffs that applications can tune to their needs. We apply Sublime to both Count-Min Sketch and Count Sketch. Through theoretical analysis and empirical evaluation, we show that Sublime significantly improves accuracy and memory over the state of the art while maintaining competitive or superior performance.

CCS Concepts: • **Theory of computation → Sketching and sampling**; **Bloom filters and hashing**; **Lower bounds and information complexity**; • **Information systems → Data stream mining**.

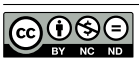
Additional Key Words and Phrases: Frequency Estimation Sketch, Data Growth, Scalability.

ACM Reference Format:
Navid Eslami, Ioana O. Bercea, Rasmus Pagh, and Niv Dayan. 2026. Sublime: Sublinear Error & Space for Unbounded Skewed Streams. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 239 (June 2026), 29 pages. https://doi.org/10.1145/3802116

1 Introduction

Data Streams. With data volumes growing exponentially, it has become infeasible for many applications to compute statistics by scanning entire datasets. At the same time, many modern systems must monitor and analyze data streams that grow at high speeds in real time, where storing the full stream (e.g., all packets transmitted over a network) is impractical. Nevertheless, users rely on these statistics to make timely decisions and extract insights from the data (e.g., to detect anomalies in network traffic [39, 84], identify trending content on social media networks [34], or monitor system performance [47, 94]). One way to address this challenge is to incrementally

Authors' Contact Information: Navid Eslami, navideslami@cs.toronto.edu, University of Toronto, Toronto, Canada; Ioana O. Bercea, bercea@kth.se, KTH Royal Institute of Technology, Stockholm, Sweden; Rasmus Pagh, pagh@di.ku.dk, BARC, University of Copenhagen, Copenhagen, Denmark; Niv Dayan, nivdayan@cs.toronto.edu, University of Toronto, Toronto, Canada.



ACM 2836-6573/2026/6-ART239
https://doi.org/10.1145/3802116

maintain statistics as new data arrives or old data expires. This approach works well for simple metrics such as sums or means, which can be updated exactly using basic arithmetic operations [47]. However, many other important statistics, such as cardinalities, quantiles, or item frequencies, cannot be maintained so easily. To address this challenge, data sketches have emerged as a popular approach. A data sketch is a compact data structure maintaining a small amount of information about the data that approximates a given statistic while providing accuracy guarantees.

Frequency Estimation Sketches. One of the most fundamental statistics in data stream analysis is frequency—the number of times each key appears in the stream. Tracking these counts exactly requires maintaining a counter for every distinct key, which leads to a substantial memory overhead if the stream contains many unique keys. To overcome this issue, a *Frequency Estimation Sketch* approximates the counts using a compact, fixed-size structure. At a high level, this structure maps the key domain to a smaller codomain of counters (e.g., via hashing) and uses the counter a key maps to as an estimate of that key's frequency. In this way, a frequency estimation sketch significantly reduces memory usage, though it introduces estimation errors due to the possibility of multiple keys mapping to the same counter. Frequency estimation sketches have become key components across many systems, including query optimizers [40, 43, 44], network monitors [9, 20, 22, 41, 55, 60, 78], streaming SQL engines [13], sensor data analyzers [2, 54], and genome analysis pipelines [80, 93].

Different Types of Error. Frequency estimation sketches differ in the nature of the errors they introduce. The Count-Min Sketch [21] either matches or overestimates a key's true count. This one-sided error makes the Count-Min Sketch suitable for security-critical applications such as detecting denial-of-service [87] and hardware row hammer [12] attacks, where misclassifying malicious behavior is unacceptable. The Count Sketch [14] can both overestimate and underestimate a key's count. Yet, these errors cancel out on average, yielding unbiased estimates. This property prevents the error from compounding when combining or multiplying estimates, making the Count Sketch useful for tasks such as query optimization [40, 43, 44] and covariance estimation [25]. The Misra-Gries structure [69] either matches or underestimates a key's true count. This makes Misra-Gries effective in identifying the most frequent keys (i.e., the heavy hitters) [39, 84], since the keys it tracks are guaranteed to have a relatively high frequency. Many variants of these data structures have been proposed, each offering distinct tradeoffs between accuracy, space, and performance [5, 7, 15, 28, 29, 35, 36, 58, 59, 62–64, 68, 77, 79, 81, 86, 88–92].

Goals. Ideally, a frequency estimation sketch should provide high accuracy using a modest memory footprint. At the same time, it should attain consistently high performance as a data stream grows. This paper shows that all existing frequency estimation sketches fall short of at least one of these goals due to two core problems:

Problem 1: Skew. In many real-world data streams, key frequencies are highly skewed, with a small number of heavy hitters appearing far more than other keys (e.g., following a Zipfian or, more generally, a heavy-tailed distribution) [8, 10, 58]. As a result, most counters within a frequency estimation sketch hold small values while only a few hold large values. Existing frequency estimation sketches use uniformly-sized counters, which must be long enough to represent the maximum counter value. Consequently, they waste a significant amount of memory storing the unused higher-order bits of small counter values. Several prior works attempt to address this problem by utilizing smaller counters and merging or sharing those that overflow [7, 15, 29, 35, 36, 58, 64, 91]. We show that these approaches compromise either accuracy or performance. Therefore, minimizing memory wastage under skew remains a challenge.

Problem 2: Unbounded Data Growth. In most applications, data streams continuously grow over time [22, 40, 43, 44, 55, 60]. Yet, all existing frequency estimation sketches are allocated with a fixed size from the get-go. This causes their estimation error to scale at least linearly with the stream's unknown length. One cannot construct a larger frequency estimation sketch to control the

error without losing all prior information, as the stream is too large to be stored, rescanned, and reinserted into a new sketch. Thus, applications are unable to maintain tight error bounds as the stream grows. This problem is critical when deploying frequency estimation sketches at scale, as recently reported by the creators of the Apache Data Sketches library [57, 75]. Recent works have tackled the challenge of accommodating unbounded data growth for filter data structures [6, 10, 17, 26, 27, 31, 52, 71, 73]. This paper is the first to address this problem for frequency estimation sketches. In fact, to the best of our knowledge, this problem has not previously been studied in the context of any type of sketch. We expect this work to open up interesting directions for sketches supporting other statistics, such as cardinalities [30, 33, 49], quantiles [38, 48, 66], graph analytics [3, 46].

Solution: Sublime. We introduce Sublime[1], a framework that generalizes frequency estimation sketches to address the above two challenges. To address Problem 1 (Skew), Sublime allocates short counters upfront and dynamically extends them as they overflow. It colocates each counter and its extension within the same cache line and employs specialized bit manipulation routines to decode them in constant time. To resolve Problem 2 (Unbounded Data Growth), Sublime expands the overall number of counters as the stream grows. In doing so, it achieves sublinear error bounds and memory footprint with respect to the stream's length. It further introduces a new Pareto frontier between these metrics, allowing applications to pick the tradeoff that best suits their needs.

Additional Contributions.

(1) We show that while traditional Count-Min and Count Sketches exhibit an estimation error that grows linearly with the stream size, variants designed to handle skew [7, 15, 29, 35, 36, 58, 64, 91] cause the error to grow super-linearly, raising the question of how to address Problem 1 without exacerbating Problem 2 (Section 3.1).
(2) We are the first to identify the problem of designing frequency estimation sketches that maintain both error and memory usage sublinear in the stream length (Section 3.2).
(3) We introduce a constant-time variable-length data encoding scheme (VALE), applicable beyond frequency estimation sketches, and use it to encode variable-length counters (Section 4.1).
(4) We describe how to expand and contract a frequency estimation sketch based on the stream's length while supporting constant-time queries and enabling fine-grained control over the accuracy-memory tradeoff. We also characterize this new tradeoff space (Section 4.2).
(5) To demonstrate Sublime's generality, we apply it to three representative frequency estimation sketches: Count-Min Sketch (Sections 4.1 and 4.2), Count Sketch (Section 4.3), and Misra-Gries (the Appendix [32]).
(6) We prove a lower bound on the minimum space required for any expandable frequency estimation sketch to achieve a target error. We show that Sublime's memory footprint approximately meets this bound (Section 5).
(7) We evaluate Sublime against state-of-the-art frequency estimation sketches and show that it simultaneously 1) reduces the memory footprint under skew, 2) tightly bounds error as the stream grows, and 3) maintains equal or better performance. We also apply Sublime to estimate the size of a join between two tables and show that it produces more accurate results than other frequency estimation sketches (Section 6).

2 Background

A stream is a sequence of key-value pairs $\langle (x_1, \Delta_1), (x_2, \Delta_2), \dots \rangle$, where x_i is the i-th key that appears in the stream and $\Delta_i = \pm 1$ represents whether its count was incremented due to an

[1]Sublime is a play on words referring to the **subli**ne**e**arly scaling error and memory footprint of our framework. It also describes, in the philosophical tradition of Immanuel Kant, the feeling of awe experienced when contemplating something vast or infinite. This notion of infinity resonates with the theme of our work, which addresses the challenge of accommodating indefinite data growth.

Symbol	Definition
N	The sum of the counts of all keys in the stream.
x, y	Example keys from the stream.
$f(x)$	The ground-truth count of key x.
$\hat{f}(x)$	The estimated count of key x.
w	Number of counters/slots.
d	Number of independent arrays.
$h_i(x)$	Hash function mapping key x to a counter.
$\sigma_i(x)$	Hash function defining key x's update direction (± 1).
c	Number of counters in each chunk.
s	Length of each stub in bits.
$W(\cdot)$	Size function.

Table 1. Definitions of terms and symbols.

insertion ($\Delta_i = 1$) or decremented due to a deletion ($\Delta_i = -1$). This is known as the *General Turnstile Model* of data streams [37]. We denote key x's current count as $f(x)$, accounting for its insertions minus its deletions. We represent the current total count of the keys in the stream with $N = \sum_x f(x) = \sum_i \Delta_i$. Table 1 lists the terms and symbols used throughout the paper.

Exhaustive Approach. The simplest approach to tracking frequencies is to store each key and its count in a hash table. Although this approach is fully accurate, it entails a high memory overhead for three reasons: (i) It stores all keys in the stream. (ii) It stores one counter for each unique key. (iii) The hash table can be as much as half empty to support collision resolution and expansions. These different overheads multiply with the number of streams an application monitors. These problems imply a memory footprint that is linear in the number of unique keys in the stream, which can be of the same magnitude as the total key count N.

Frequency estimation (FE) sketches address these problems by trading accuracy for space. That is, for any key x, an FE sketch returns an estimate $\hat{f}(x)$ of key x's count $f(x)$. We focus on Count-Min Sketch [21] for now, as it is the simplest and most widely used FE sketch [61, 67]. Later in Section 4.3, we describe Count Sketch [14] and apply our techniques to it. We do the same for Misra-Gries [69] in the Appendix [32].

Count-Min Sketch (CMS). The core design element of CMS is an array consisting of w counters, each initialized to zero. We insert a key x into this array by hashing it to one counter and incrementing it. We delete key x by decrementing its corresponding counter.[2] The counter that key x hashes to provides an estimate of x's count. Due to hash collisions, however, multiple keys can map to the same counter. Because of this, CMS may overestimate a key's true count. This error is at most N/w in expectation since the N keys are randomly hashed into w counters. Increasing the array's size makes collisions less likely, thereby improving accuracy. Even then, the error may deviate from its expectation. The probability that errors do not exceed the expectation "too much" is known as the sketch's *Confidence*. An application of Markov's inequality shows that for any key x, the error does not exceed $e \cdot N/w$ with a confidence of at least $1 - e^{-1}$, i.e., $\Pr(|\hat{f}(x) - f(x)| \leq e \cdot N/w) \geq 1 - e^{-1}$. Here, we use Euler's constant e due to convention. Any other constant would also work and would replace e in both the error bound and the confidence. This implies that the smaller the constant becomes, the more accurate we expect the FE sketch to be and the lower confidence drops.

Yet, with a single array, confidence is low. The reason is that all keys hashing to the same counter as a frequent key will suffer from inflated estimates. Moreover, since there are many keys with low or moderate counts, there is a non-negligible chance that enough of them map to the same

[2]Since CMS does not store the actual keys, users must ensure that deletions only target previously inserted keys.

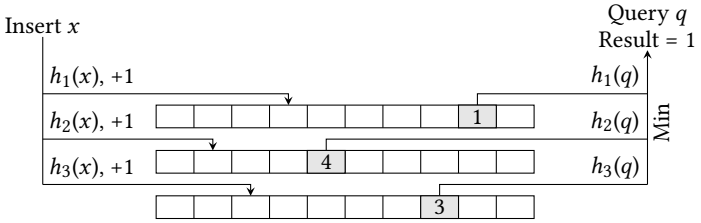


Fig. 1. CMS maintains d counter arrays of size w and hashes keys into them to estimate their frequencies. Insertions are illustrated on the left and queries on the right.

counter and inflate estimates beyond the error bound. CMS addresses these issues by allocating d independent counter arrays, each with a dedicated hash function h_i. CMS inserts or deletes a key by applying the operation to each array, and it answers a query by querying all arrays and taking the minimum of the estimates.[3] Returning the minimum for queries boosts confidence exponentially with respect to the number of arrays d, since the minimum estimate only carries severe errors when all estimates err significantly. Formally, with d arrays of length w, CMS guarantees an error of at most $e \cdot N/w$ with a confidence of $1 - e^{-d}$, i.e., $\Pr(|\hat{f}(x) - f(x)| \leq e \cdot N/w) \geq 1 - e^{-d}$ for any key x. Figure 1 shows an example of insertions and queries applied to three counter arrays.

Since CMS uses a fixed number of tightly packed counters and does not store the stream's keys, it significantly improves the memory footprint compared to the exhaustive approach of tracking each key's exact count. Nevertheless, CMS still suffers from several memory inefficiencies, which we explore in the following section.

3 Problem Analysis

We show that mainstream FE sketches such as CMS suffer from two core limitations: 1) they waste a significant amount of memory under skew—memory that could otherwise be used to improve accuracy or confidence—and 2) their estimation error grows rapidly with the stream length. These shortcomings prevent current FE sketches from achieving accuracy at scale.

3.1 Challenge 1: Skew

Most real-world workloads are skewed, meaning that a few heavy hitters receives the majority of insertions, while all other keys have low counts [8, 10, 58]. In CMS, however, all counters are uniformly sized, so each must be provisioned to accommodate the largest possible value. This leads to substantial memory waste. Ideally, each counter should be sized just large enough to fit its value, thereby saving space or enabling more counters to improve accuracy or confidence. This issue has been explored in recent work; we now review existing solutions and their limitations.

Hybrid Methods. A common strategy for tackling skew is to track the heavy hitters separately in a hash table while leaving the rest of the keys to CMS [59, 77, 81, 88–90]. Such hybrid structures aim to cap the maximum counter value in CMS, thereby permitting shorter counters. Yet, this approach does not fundamentally resolve the issue. The reasons are twofold: The keys tracked by CMS may still follow a skewed distribution, leading to significant memory wastage. Moreover, the heavy hitters can vary over time. Identifying the next heavy hitter and "promoting" it from CMS into the hash table is error-prone, since collisions in CMS can cause many infrequent keys to be misclassified as heavy hitters.

Compression-based Methods. Other approaches compress the counters using algorithms such as compressive sensing [29, 64] or hypergraph peeling [16]. Yet, these methods impose heavy decompression overheads on queries.

[3] CMS uses pairwise-independent hash functions to bound the probability of collisions and overestimation [21].

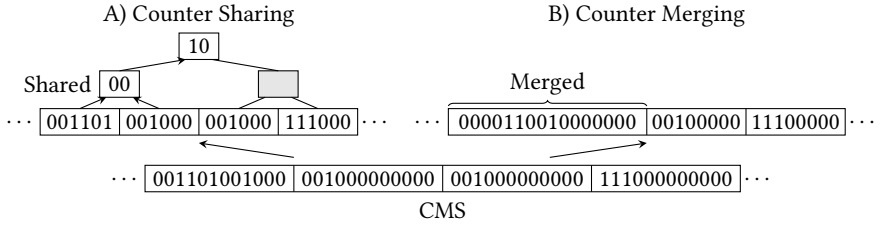


Fig. 2. Counter sharing and counter merging encode CMS's counters in less space in exchange for blowing up some counter values. Here, the bits in each counter are in increasing order of significance from left to right.

Counter Sharing. Counter sharing approaches store the counter values in a hierarchy of counters [15]. The bottom level employs short counters to store the lower-order bits of each counter, while the upper levels handle overflows using shared counters. For example, Figure 2-A) encodes four adjacent counters within CMS using 6-bit counters in the bottom level and 2-bit counters in the upper levels. Whenever a counter in this hierarchy overflows, it increments its parent and resets to zero. A key's count is retrieved by concatenating its corresponding bottom-level counter with the shared counters along its path to the root or until reaching an internal node to which no overflow has occurred. Such an internal node is identified using additional metadata. The core issue with this approach is that the error rapidly deteriorates as the stream grows. In the worst-case scenario, after N insertions into an array of size w, each counter overflows into $\Theta(\log(N/w))$ levels above it. Since the counters in the higher levels are shared, this blows up the values of $\Theta(N/w)$ other counters, increasing the estimation error by the same factor. To mitigate this issue, prior works experiment with varying the counter sizes and the hierarchy's fanout [29, 35, 36, 58, 64, 91]. Yet, they do not fundamentally resolve the problem.

Counter Merging. Counter merging is an alternative approach that starts with 8-bit counters and merges an overflowing counter with an adjacent counter into a longer counter that sums their values [7]. Figure 2-B) shows an example applied to CMS. After N insertions into an array of w counters, each counter will be merged with $\Theta(\log(N/w))$ other counters, increasing the error by the same factor. Merging only continues until each counter becomes four bytes long, at which point error degradation stops. Counter merging has a higher metadata overhead than counter sharing for indicating which counters are merged, leaving less space for storing counters. Parsing this metadata adds more complexity than alternative methods like counter sharing, reducing performance.

Deletions. Deletions deteriorate the accuracy of both counter sharing and counter merging methods since these techniques ambiguate how counters should be "separated" as they decrease.

Comparison. Prior work has shown counter sharing and counter merging to dominate hybrid and compression-based methods in terms of both accuracy and performance [7, 36]. Thus, in Figure 3, we qualitatively compare counter sharing and counter merging to a CMS with the same memory footprint to illustrate their error degradation with data growth. While these methods perform well for short streams, their error exceeds that of CMS as the stream grows. We empirically verify this behavior in Experiment 4 of Section 6.

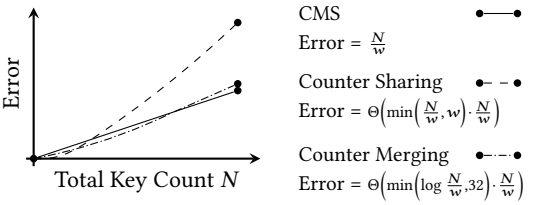


Fig. 3. Under the same memory budget as a CMS instance with 32-bit counters, counter sharing and counter merging can lead to lower accuracy when processing a growing stream.

3.2 Challenge 2: Unbounded Growth

In many streaming applications, the data stream grows continuously, and it is imperative to estimate the count of each key from the stream's beginning. For instance, query optimizers often track the number of times each key occurs in a table column to estimate the size of a join on that column [40, 43, 44]. As new data is inserted, the total number of keys N increases. Yet, the full count history of each key is still needed, since every occurrence contributes equally to the join size regardless of when it arrived.

Unbounded stream growth is also a challenge for applications that only require key counts over recent time windows. For example, network management systems detect anomalous or malicious activity by counting packets from each source within fixed time windows [22, 55, 60]. For bursty streams such as network traffic, the number of keys N in each window is unpredictable. Bounding N by defining windows based on insertion count fails to resolve the issue, as the notion of time is lost.

In existing FE sketches, the number of counters w is fixed at allocation time. Consequently, the expected error degrades linearly with N (e.g., N/w in the case of CMS). In hope of balancing accuracy and memory, practitioners can tune w upfront. This tuning is non-trivial, however, as having too few counters leads to significant errors, while having too many wastes memory. Deletions make the problem more challenging by causing N to fluctuate over time. Worse yet, even if the ultimate stream length is known upfront, allocating a large CMS from the start wastes memory during the early stages when the stream is short.

Summary. An ideal FE sketch should: 1) compactly encode counter values without incurring memory waste from skew, 2) scale its size with the stream's length to control both the memory footprint and the error rate, 3) support deletions without sacrificing accuracy, and 4) provide high-performance queries, insertions, and deletions. Is it possible to achieve these goals simultaneously?

4 Sublime

We present Sublime, a framework for generalizing an FE sketch to optimize its memory footprint and error rate with respect to the stream's skew and its length. Section 4.1 shows how to adapt to skew by compactly encoding each counter as a short integer and extending it as it overflows. Section 4.2 shows how to control both accuracy and space for a stream of unknown length by expanding and contracting the FE sketch. At the same time, Sublime supports deletions and maintains high performance. We first describe Sublime as applied to the Count-Min Sketch (CMS), denoted as $\text{Sublime}_{\text{CMS}}$. We apply Sublime to Count Sketch in Section 4.3 and to Misra-Gries in the Appendix [32].

4.1 Accommodating Skew

Under skew, many counter values in CMS are small and do not use all the bits in their counters. To avoid wasting space under any workload, $\text{Sublime}_{\text{CMS}}$ employs a general-purpose variable-length counter (VALE) encoding. Upfront, VALE stores each counter as a fixed-size integer called a *Stub*. When a stub overflows, VALE elongates it using a dedicated variable-length *Extension*. We adaptively tune the stub length based on the workload's skew to control the rate of stub overflows, thereby optimizing the overall memory footprint. To improve cache behavior and quickly reconstruct a counter, VALE colocates a stub and its corresponding extension in the same cache line. It quickly finds a stub's extension using a bit manipulation workflow that leverages rank and select operations.

Chunking. To colocate stubs and their extensions in 512-bit cache lines, VALE partitions each counter array into contiguous *Chunks*, each occupying a cache line. A chunk stores its counters' stubs in a contiguous array and tracks the counters that overflow their stubs in an *Overflows Bitmap* using one bit per counter. VALE stores the extensions of these overflowing counters contiguously

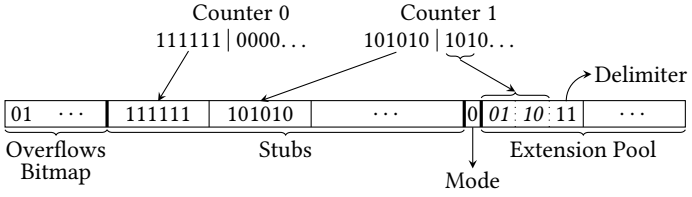


Fig. 4. VALE encodes a chunk of c counters in a cache line. It encodes the s lower-order bits of each counter in a stub and stores its remaining higher-order bits in extensions comprised of 2-bit fragments representing base-3 digits. Here, we use a stub length of $s = 6$.

in the remaining space towards the end of the chunk, called the *Extension Pool*. Counter 0 and Counter 1 in Figure 4 are examples of a non-overflowing and an overflowing counter, respectively.

Parameters. VALE has two global parameters: the stub length in bits s and the number of counters per chunk c. With more skewed workloads, having shorter stubs and more counters per chunk reduces space consumption. The reason is that skew shrinks most counter values, enabling the use of shorter stubs for storing them. Skew also reduces the average length of the extensions, leaving more space in a chunk to store more counters.

Later in this section, we discuss how to adapt the tuning of these parameters to the workload as the stream evolves. When the workload is uniform, this procedure tunes the stub length so that all counter values fit in stubs, allowing it to remove the overflows bitmaps and yield the configuration of a standard CMS. We illustrate the impact of this procedure on the memory footprint in Section 6.

Extensions. An extension consists of one or more 2-bit *Fragments* that encode the higher-order bits that exceed the length of the corresponding stub. We use 2-bit fragments to encode these bits at the finest possible granularity. We use the fragment 11 as a delimiter at the end of each extension. The other three combinations of bits are used to encode each extension's value in base 3. Specifically, the base-3 digits $(0)_3$, $(1)_3$, and $(2)_3$ are represented as the fragments *00*, *10*, and *01*, respectively. We store fragments in increasing order of their digits' significance from left to right. Formally, a value v is encoded in an extension by representing each of its $\lfloor \log_3 v \rfloor + 1$ digits in a fragment, with the j-th digit being $v_j = \lfloor v/3^j \rfloor \mod 3$. We decode v back into a 32-bit binary integer by computing the sum $3^0 \cdot v_0 + 3^1 \cdot v_1 + \ldots$. At the end of this section, we show how to efficiently compute this sum without multiplication operations.

For example, Counter 1 in Figure 4, has the value 21 in its lower-order bits and the value 5 in its higher-order bits. Since 5 has two digits in base 3, it is encoded as the extension $\langle (2)_3, (1)_3, \text{delimiter} \rangle = \langle 01, 10, 11 \rangle$. Decoding this extension yields $3^0 \cdot 2 + 3^1 \cdot 1 = 5$, as expected.

Forming a chunk's extensions from 2-bit fragments allows them to fit in 1-2 machine words in most cases. We prove in Section 5 that using these extensions in conjunction with short stubs enables VALE to encode each counter's value in space close to the length of its binary encoding. Even still, since extensions are slightly less space-efficient than stubs due to their base-3 encoding and their delimiters, we adapt the stub length at runtime to optimize the memory footprint.

Locating an Extension. VALE keeps the extensions in each extension pool in the same order as that of the counters. It locates the i-th counter's extension by counting the number of preceding bits set to 1s in the overflows bitmap. This is equivalent to the number of overflowing counters preceding the i-th counter, denoted by m. Skipping over the first m extensions in the extension pool leads to the position of the target extension. From this position, VALE traverses the extension backwards to decode its value.

Figure 5 illustrates how to identify the 3rd counter's extension in a chunk. Since the 3rd bit in the overflows bitmap is 1, the 3rd counter has an extension. As there is one bit set to 1 before the 3rd bit in the overflows bitmap, we skip one extension to locate the 3rd counter's extension, shown in gray.

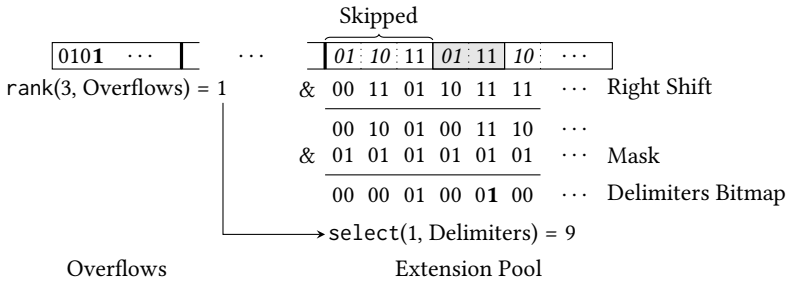


Fig. 5. Sublime$_{CMS}$ applies rank and select operations to locate an overflowing counter's extension.

Although one can locate an overflowing counter's extension by looping over the fragments in the extension pool, doing so incurs high CPU overheads. Instead, VALE quickly locates an extension using rank and select operations [18, 56, 70, 72, 95]. Given a bitmap B, the rank(i, B) operation counts the number of 1s strictly before the i-th bit, and the select(i, B) operation returns the position of the i-th 1. Later in this section, we show how to use specialized CPU instructions to implement these operations efficiently.

The first step is to determine m, the number of overflowing counters to the left of the i-th counter. As this is the number of 1s before the i-th bit in the overflows bitmap, VALE applies the rank operation to evaluate m = rank$(i$, overflows$)$.

The second step is to identify the target extension's delimiter by skipping over the m delimiters preceding it. We do this using several bit manipulation techniques. First, we convert the extension pool into a so-called *Delimiters Bitmap*. This bitmap contains one bit set to 1 at the position of every delimiter and 0s everywhere else, as illustrated at the bottom right of Figure 5. To derive this bitmap, we take the bitwise-and of the extension pool with a version of itself shifted to the right by one bit. The result is a bitmap with a 1 at the end of any two consecutive 1s in the original extension pool, as illustrated in Figure 5. We mask out all bits corresponding to the first bit of a fragment to ensure only the end of each delimiter is a 1. In sum, the delimiters bitmap is derived as

$$\text{delimiters} = \text{extensions} \;\&\; (\text{extensions} >> 1) \;\&\; 0101\ldots0101.$$

Finally, we use the select operation to evaluate select$(m$, delimiters$)$, which skips over the first m delimiters and returns the position of the m-th delimiter. We sequentially apply the bitwise operations above to each machine word comprising the chunk's extensions. Locating extensions in this way is efficient, as a chunk's extensions typically fit in 1-2 words.

Figure 5 shows an example of locating the 3rd counter's extension using this workflow. Here, rank$(3$, overflows$)$ = 1. After deriving the delimiters bitmap by manipulating the extension pool, VALE evaluates select$(1$, delimiters$)$ = 9 to conclude that the target extension ends at the 9th bit.

Tails. In the rare event that many heavy hitters hash to the same chunk, many of the chunk's counters will overflow, potentially growing the cumulative length of the extensions beyond the size of the chunk. VALE handles this case by allocating an external array of 32-bit integers called *Tails* for the chunk, with one integer per counter. It stores the i-th counter's higher-order bits in the i-th tail in binary. Each chunk contains a mode flag indicating if it has a tails array, as depicted in Figure 6. If it does, we repurpose the space taken up by the extension pool to store a pointer to the tails array. We ensure that the extension pool is large enough to accommodate this pointer (i.e., at least 48 bits[4]) when tuning the number of counters per chunk and the stub size. Any space after the pointer is unused.

[4]The upper 16 bits in a 64-bit pointer are unused on many operating systems due to their use of 4-level paging [24]. As such, we only store the lower 48 bits of the pointer. That said, modern machines and operating systems have moved towards using 5-level paging [23], which uses 57-bit addresses. Despite this shift, operating systems like Linux still default to using 48-bit addresses for compatibility reasons [50].

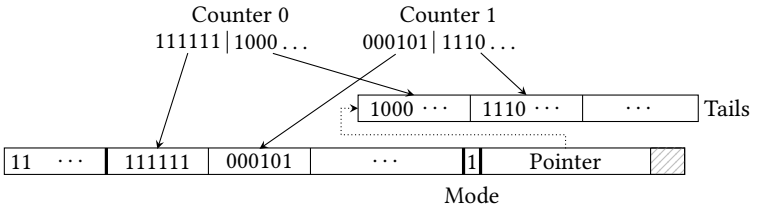


Fig. 6. When a chunk's extensions outgrow its space, it stores a pointer to a dedicated external array of tails to store the counters' higher-order bits.

Incrementing Counters. When an insertion maps to the i-th counter within a chunk, Sublime$_{\text{CMS}}$ increments it by adding one to the lower-order bits stored in the i-th stub, i.e., stub$[i]$. Most of the time, stub$[i]$'s increment does not carry over to the counter's higher-order bits, allowing the insertion to terminate early and achieve high throughput.

Whenever stub$[i]$'s increment carries over, Sublime$_{\text{CMS}}$ adds one to the counter's extension or tail. For an extension, this entails incrementing its first digit and propagating a carry to the other digits as necessary. This carry may propagate beyond the end of the extension, leading to the addition of a new digit. Sublime$_{\text{CMS}}$ makes room for the fragment representing this digit by shifting subsequent extensions to the right. If the i-th counter did not have an extension, a new extension encoding the value 1 is created for it. We use rank and select operations to determine the appropriate position of this extension in the extension pool. Updating a counter's higher-order bits is much simpler if they are stored within a tail's binary integer. For the heavy hitters (which comprise most of the stream), this yields faster insertions, as their counters often use tails.

Decrementing Counters. Analogously, when a deletion maps to the i-th counter in a chunk, Sublime$_{\text{CMS}}$ decrements the lower-order bits in stub$[i]$. If stub$[i]$ borrows from the counter's higher-order bits as a result, Sublime$_{\text{CMS}}$ decrements the corresponding extension or tail. This may trigger the removal of a digit from the counter's extension, in which case subsequent extensions are shifted to the left to keep the extension pool tightly packed.

Adaptive Tuning. To minimize space usage and ensure that a small fraction of the extension pools are unused at any time, Sublime$_{\text{CMS}}$ adaptively retunes VALE's number of counters per chunk c and its stub length s. Retuning is triggered whenever more than 1% of the chunks use tails arrays (due to insertions growing the extensions), or there are more than two unused bits per counter (due to deletions shrinking the extensions).[5] During tuning, Sublime$_{\text{CMS}}$ scans the arrays and computes a coarse histogram of the counter values based on their bit length. It uses this histogram to calculate the expected total length of a chunk's extensions for a given combination of parameters c and s. For each combination, it applies Chebyshev's inequality to the total extension length within a chunk to derive a conservative bound on the proportion of chunks that will use tails arrays. It uses this bound to estimate Sublime$_{\text{CMS}}$'s memory footprint when using the corresponding tuning, and picks the tuning that minimizes the memory footprint. The result sets the stub length s to be slightly longer than the length of the average counter value. This bounds the number of extensions and makes the number of stored tails arrays and their memory overhead negligible. The amortized cost of retuning VALE is small, as many more updates than the number of counters are required to trigger retuning. We demonstrate this under Experiment 3 of Section 6.

Impact of Cache Line Size. The aforementioned tuning procedure also allows Sublime$_{\text{CMS}}$ to support different cache line sizes. Having smaller cache lines leads to the tuning procedure storing fewer counters per chunk. This reduction increases the variance of the total size of a chunk's extensions, causing the tuning procedure to slightly increase the overprovisioned space for the

[5]These conditions lead to at most 20% overprovisioned space upfront when counters are short (1 byte). This fraction decreases quickly as the counters grow.

chunk's extension pool.[6] Conversely, having larger cache lines does not increase the required overprovisioned space, though it may degrade query performance if we increase the chunk size to match the cache line. This is because locating a counter's extension becomes more expensive as extension pools grow, as we process them word by word. Sublime$_{\text{CMS}}$ easily addresses this problem by storing multiple chunks per cache line (e.g., two 512-bit chunks within a 1024-bit cache line).

General Applicability. VALE is able to compactly encode arbitrary variable-length data by treating the binary representation of each data item as the value of a counter. Thus, it can be used in other applications operating on such data to improve memory efficiency.

Optimization 1: Hardware Accelerated Rank and Select. Sublime$_{\text{CMS}}$ utilizes specialized CPU instructions to implement rank and select operations. It implements `rank`(i, B) by applying the `POPCNT` instruction to the first $\lceil i/64 \rceil$ words of the bitmap B to count the 1s before the i-th bit.[7] It also implements `select`(i, B) by counting the 1s in B's words to locate the word containing the i-th 1. Once found, it applies the x86 BMI instructions `PDEP` and `TZCNT` to evaluate the target bit's position within that word, as described in [72].[8] Each of these commands is applied to an average of 1-2 machine words, as the bitmaps Sublime$_{\text{CMS}}$ operates over are 1-2 words long.

Sublime$_{\text{CMS}}$ includes efficient fallback implementations of rank and select for processors that lack the above instructions (e.g., ARM). We employ two lookup tables structured as direct-access arrays: one for counting the number of 1s in a byte to compute rank, and one for applying select within a byte. The entry corresponding to the value of a byte and the parameter of the operation in each table indicates the result of the relevant operation with these inputs. Sublime$_{\text{CMS}}$ iteratively applies these tables to the bytes within a bitmap to evaluate rank and select. Doing so slightly degrades query performance, as queries must always retrieve the extension of a counter. Nevertheless, insertions and deletions remain unaffected since they rarely access a counter's extension.

Optimization 2: Updating Stubs with Lookup Tables. Since stubs are not byte-aligned, modifying them can be costly. This is because extracting and writing back a stub to the machine words containing it entails five extra bitwise operations on top of the addition operation. Sublime$_{\text{CMS}}$ bypasses this cost by directly updating the stub's words using a single addition. It achieves this using two precomputed lookup tables. The first table stores the position of the word each stub lies in, and the second table stores the stub's position within its word. Sublime$_{\text{CMS}}$ updates a stub by adding or subtracting one from the bit indicated by the second table in the word indicated by the first. For example, in the context of Figure 4, the first lookup table indicates that the 1st stub lies in the 1st word, which has a binary representation of 111111101010 . . . . The second lookup table indicates that this stub starts at the 6th bit in the 1st word, allowing us to increment it by adding 000000100000 . . . , thus yielding 111111011010 . . . as the updated word.

Optimization 3: Prefetching Chunks. We allow multiple chunks to be read in parallel within a single operation as well as across multiple operations using prefetching techniques similar to that of [58]. Specifically, we maintain a prefetching queue for each array in Sublime$_{\text{CMS}}$ that records the eight most recent requests for chunks. Each time a chunk is requested from an array, a software prefetch instruction is issued for it, and the request is pushed into the array's queue. When a queue fills up, the chunk of its oldest request will be present in the L1 cache, allowing it to be read efficiently while other chunks are being loaded into the cache.

[6] We have observed that reducing the cache line size from 512 bits to 256 and 128 bits increases the proportion of overprovisioned memory by 5% and 6%, respectively.

[7] Formally, denoting the j-th word in B as B_j, this process corresponds to computing $\texttt{POPCNT}\big(B_{\lfloor i/64 \rfloor} \,\&\, ((1 << (i \mod 64)) - 1)\big) + \sum_{j=0}^{\lfloor i/64 \rfloor - 1} \texttt{POPCNT}(B_j)$.

[8] If the j-th word B_j contains the i-th 1 and there are r 1s in the preceding words, the result is computed as $64 \cdot j + \texttt{TZCNT}(\texttt{PDEP}((1 << (i - r)) - 1, B_j))$, where the first operand of `PDEP` is the payload and the second operand is the mask [19].

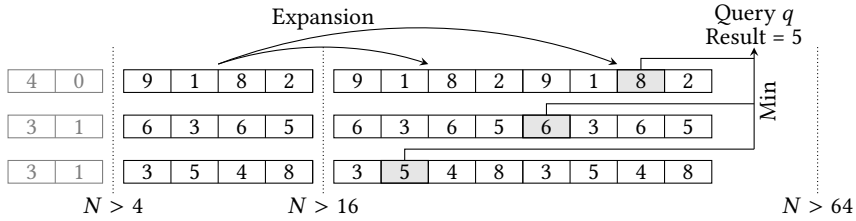


Fig. 7. Sublime$_{\text{CMS}}$ expands by concatenating each array with a copy of itself. It maps a key to a counter in an array by taking the appropriate number of bits from its hash.

Optimization 4: Hash Splicing. Recall that a key is hashed into each of the d arrays during an update or query. Instead of applying d separate hash functions and incurring their CPU overheads, Sublime$_{\text{CMS}}$ applies a single hash function to the key to compute a long hash. It splices the result into d sub-hashes of equal length and uses each as the hash value of an array.

Optimization 5: Fast Division. Determining the chunk of a counter at offset i in an array can be expensive if done by divinding i by the number of counters per chunk c, as c may not be a power of two. We avoid this cost by simulating the division through fixed-point multiplication [85]. Specifically, we compute the expression $\left\lfloor \left(i/2^{\lceil \log_2 c \rceil}\right) \cdot \left(2^{\lceil \log_2 c \rceil}/c\right) \right\rfloor = \lfloor i/c \rfloor$ by approximating the multiplicands on the left-hand side as fractional binary numbers with a fixed number of precision bits. To this end, we precompute the latter multiplicand's fixed-point representation with $\lceil \log_2 w \rceil$ precision bits, where w is the size of an array. This many bits of precision ensures that, when multiplying by a counter offset i smaller than w, the result has an error of at most one, allowing Sublime$_{\text{CMS}}$ to safely truncate it to compute the chunk's (integral) offset. For the former multiplicand, we treat i's binary representation as a fixed-point fractional value with $\lceil \log_2 c \rceil$ precision bits. Finally, we multiply these values using standard integer multiplication and shift the result $\lceil \log_2 w \rceil + \lceil \log_2 c \rceil$ bits to the right to remove the precision bits and derive the answer.

Optimization 6: Horner's Method with Shifts and Adds. Sublime$_{\text{CMS}}$ decodes a counter's extension with digits $v_0, v_1, \cdots, v_k$ by computing the sum $3^0 \cdot v_0 + 3^1 \cdot v_1 + \cdots + 3^k \cdot v_k$. We speed up this process using Horner's method [42, 53] and implementing multiplication by 3 using shift and add operations [85]. Specifically, we compute a running sum starting from the last term (i.e., initializing the sum to $3^0 \cdot v_k$) and move towards the first term, handling them one by one. That is, we handle v_{k-1} by multiplying the running sum by 3 and adding v_{k-1} to derive the sum $3^0 \cdot v_{k-1} + 3^1 \cdot v_k$, and so on. We multiply the running sum by 3 by adding it to a copy of itself shifted to the left by one bit (which is equivalent to multiplication by 2).

4.2 Accommodating Unbounded Growth

We now address the problem of maintaining bounded estimation error for frequency estimation sketches over growing streams. Recall from Section 3 that processing growing streams with existing FE sketches (e.g., CMS) yields an estimation error that increases at least linearly with the total key count N. Since N is typically unknown in advance, bounding this error requires overprovisioning memory upfront, leading to significant waste. Sublime addresses this limitation by generalizing an FE sketch to increase its number of counters as the stream grows. We also show how to support deletions and contractions without deteriorating accuracy.

In the case of CMS, Sublime$_{\text{CMS}}$ expands whenever N exceeds a tunable threshold. An expansion concatenates the current CMS with a copy of itself, doubling the size of each array. This operation is inexpensive since it copies counters without needing to rehash the keys. These larger arrays reduce the collision probability of future insertions. Figure 7 illustrates Sublime$_{\text{CMS}}$ just after the second expansion took place (on the right), as well as its state just before the first expansion (left)

and the second (middle). Each half of an array in the largest sketch is a copy of the corresponding array just before the second expansion. The counters in each array before this expansion sum to the expansion threshold of 16, plus the total value of the counters copied from the previous expansion, i.e., 4. Queries follow the same algorithm as CMS, targeting the newly expanded, largest sketch. Sublime$_{\text{CMS}}$ maintains d arrays across expansions to ensure a stable confidence bound on the estimation errors while retaining the same $O(d)$ time complexity as CMS.

Consistent Hash Functions. Sublime$_{\text{CMS}}$ uses the same hash function for each of its arrays across expansions. It hashes a key to a counter within an array of size w by taking the $\log_2 w$ lower-order bits of its hash. This ensures that after an expansion, a key hashes to either the same counter it would have hashed to before or its copy.[9] For example, the queried key q in Figure 7 has $h_1(q) = 011\ldots$ as its hash. Since the size of the array before expansion was 4, we only needed the first $\log_2 4 = 2$ bits of q's hash, i.e., 01, to map it to the 2nd counter in the first array. After expansion, we use the first $\log_2 8 = 3$ bits of the hash, i.e., 011, to map it to the 6th counter, which is a copy of the 2nd counter before expansion.

Initial Size. By default, Sublime$_{\text{CMS}}$ initializes each of its d arrays to a single chunk (i.e., a cache line) containing c counters, as described in Section 4.1. Thus, it avoids wasting memory from the onset in case the stream grows slowly.

Expansion Thresholds. By default, Sublime$_{\text{CMS}}$ expands whenever N quadruples. As shown at the bottom of Figure 7, expansions occur when the number of keys N reaches 4, 16, and 64. These thresholds cause the number of counters in each array to grow as $\sqrt{N}$. As a consequence, we achieve a sublinear memory footprint and sublinear errors. To see why, recall from Section 2 that a CMS allocated with arrays of w counters has an expected error of N/w. By making w grow as $\sqrt{N}$, Sublime$_{\text{CMS}}$ instead obtains an expected error of $O(N/\sqrt{N}) = O(\sqrt{N})$ even when the stream's length is unknown. Although keys inserted when the sketch was smaller contribute disproportionally to the error due to their counts duplicating with expansions, they are exponentially fewer in number. Hence, such keys only increase the error by a small constant factor. Theorem 4.1, proven in the Appendix [32], formalizes these intuitions. In sum, compared to a standard CMS with linear errors and a fixed memory footprint, Sublime$_{\text{CMS}}$ achieves significantly lower errors at a slightly higher space cost. As such, it is a better fit for applications that require maintaining low errors as the stream grows indefinitely. We shortly show how to set the expansion thresholds to open a new accuracy-memory tradeoff space.

Deletions and Contractions. Sublime$_{\text{CMS}}$ handles deletions using the same procedure as CMS. It contracts after many deletions to save memory. The simplest way to contract is to revert an expansion by folding the halves of each array onto each other and adding up the counters. Yet, this approach would inflate the error since it adds the counters copied during the last expansion to the originals, thereby doubling them. Instead, Sublime$_{\text{CMS}}$ keeps a record of the sketch's state just before each expansion. It extracts the recent updates made to the current sketch by subtracting the state of the sketch before the last expansion from its left and right halves. We illustrate this for a single array in Figure 8. The result is an array of deltas indicating how each counter from the earlier sketch and its copy have changed. Sublime$_{\text{CMS}}$ applies these deltas to the original counters in the earlier sketch. Finally, it discards the current sketch and uses the earlier sketch to handle future operations.

The contraction threshold is set to the average of the two previous expansion thresholds to avoid the problem of updates frequently resizing the structure. As such, a contraction is triggered when the majority of the stream's keys are deleted. Therefore, Sublime$_{\text{CMS}}$ always handles most of the

[9]Consistent hashing also grants Sublime$_{\text{CMS}}$ mergeability: the ability to merge sketches constructed on distinct streams to represent their union without needing to rescan them.

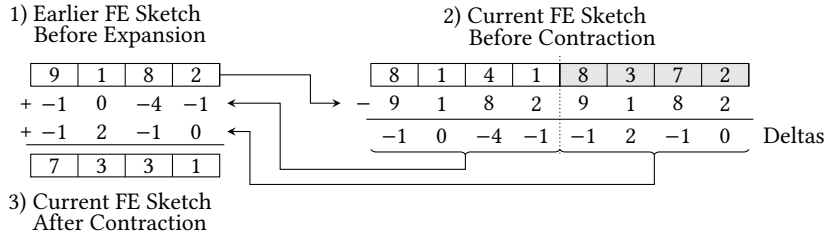


Fig. 8. During contractions, Sublime$_{\text{CMS}}$ extracts the recent updates by subtracting the previous FE sketch. It adds the results to the previous sketch to transfer the updates.

stream using arrays that are at most twice as large as $\sqrt{N}$. Thus, the expected error and the size of each array remain the same as before, despite the state of the sketch depending on the sequence of insertions and deletions.

Maintaining the sketch record increases the memory footprint by at most a factor of 2 compared to allocating and expanding a single sketch. When the total key count fluctuates, the memory saved by contracting outweighs this cost.

Rich Space of Tradeoffs. By varying the expansion thresholds, we expose a rich spectrum of accuracy-memory tradeoffs. We encode these policies using a *Size Function*, denoted as $W(N)$, which captures the growth rate of the number of counters in each array with respect to N. The faster the size function grows with respect to N, the sooner Sublime$_{\text{CMS}}$ expands, thereby tracking more of the stream's keys accurately in exchange for a larger memory footprint. By default, the size function is set to $W(N) = \sqrt{N}$. We generalize it to a *Sublinear Power* of the form $W(N) = N^{\alpha}/\epsilon$, with constants $\alpha \in [0, 1)$ and $\epsilon > 0$. This corresponds to having $1/\epsilon$ chunks in each array at the beginning and expanding when N grows by a factor of $2^{1/\alpha}$. Note that this factor is greater than two.

When using a sublinear power as the size function, Sublime$_{\text{CMS}}$ achieves an expected error of $O(N/W(N)) = O(\epsilon \cdot N^{1-\alpha})$.[10] One may hope to achieve a constant expected error by growing the number of counters linearly with the data size, i.e., by using a linear size function of the form $W(N) = N/\epsilon$. Yet, Sublime$_{\text{CMS}}$ obtains an error bound of $\log_2 N \cdot \epsilon$ in this case. Section 5 shows that without knowing the ultimate length of the stream, these expected error bounds are the best achievable by any FE sketch of (approximately) the same size as Sublime$_{\text{CMS}}$. The following theorem, proven in the Appendix [32], formalizes the above error bounds:[11]

Theorem 4.1. *When using a size function $W(\cdot)$ and a single array on a stream with a total key count of N, Sublime$_{CMS}$ provides an estimation error $E(N)$ satisfying*

$$\mathbb{E}\left[E(N)\right] \leq \begin{cases} O(1) \cdot \epsilon N^{1-\alpha} & \text{if } W(N) = N^{\alpha}/\epsilon, \text{ for } \alpha \in [0,1), \\ \log_2 N \cdot \epsilon & \text{if } W(N) = N/\epsilon. \end{cases}$$

Compared to a fixed-size CMS allocated upfront with an array of $W(N)$ counters with knowledge of N, the expected error terms above are higher by at most a constant and a logarithmic factor.

Applying Markov's inequality to the expected error bounds above yields the same confidence bounds as CMS. Moreover, employing d independent arrays in Sublime$_{\text{CMS}}$ raises this confidence to the power of d. These confidence bounds remain stable as the stream grows.

[10]The constant factor hidden by the asymptotics is $(2^{1/\alpha} - 1) \cdot \frac{1-2^{(\alpha-1)\cdot\log_2 N}}{2^{1/\alpha-1}-1}$, which is at most $\frac{2^{1/\alpha}-1}{2^{1/\alpha-1}-1}$ and converges to $\log_2 N$ as α tends to 1.

[11]In the Appendix [32], we show that similar bounds hold for Sublime when it is applied to Misra-Gries.

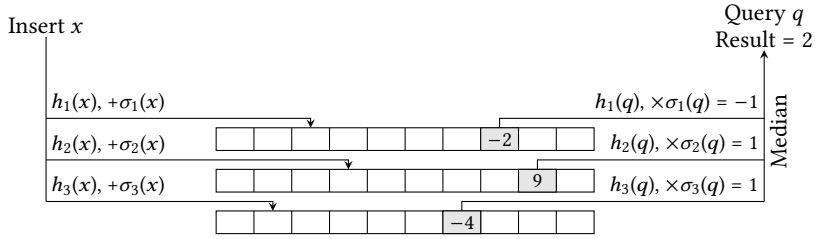


Fig. 9. CS assigns a random update direction to each key and uses it to insert them into d counter arrays of size w. The random update directions enable unbiased estimation.

FE Sketch	Bias	Error Bound	Confidence	Insert	Delete	Query
CMS	Over	$\frac{e \cdot N}{w}$	$1 - e^{-d}$	$O(d)$	$O(d)$	$O(d)$
CS	Unbiased	$e \cdot \sqrt{\frac{\sum_y (f(y))^2}{w}}$	$\geq 1 - e^{-d}$	$O(d)$	$O(d)$	$O(d)$

Table 2. CMS and CS strike different tradeoffs.

4.3 Applying Sublime to Count Sketch

As discussed in Section 1, many FE sketches differ from CMS in their error guarantees and use cases. We now demonstrate the generality of Sublime by applying it to Count Sketch to derive Sublime$_{\text{CS}}$. We also discuss applying Sublime to any kind of FE sketch, whether existing or yet to be proposed.

Count Sketch (CS). Recall from Section 1 that unbiased estimates equal the ground-truth frequency on average. This allows for multiplying estimates, as required in join size [40, 43, 44] and covariance [25] estimation, without the risk of compounding errors. CS provides such unbiased estimates and reaps its benefits.[12] Table 2 summarizes the properties of CS and compares it to CMS.

Similarly to CMS, the core component of CS is a counter array of size w coupled with a hash function h that maps each key to a counter. Unlike CMS, CS further associates each key with a random "update direction" from the set $\{-1, +1\}$ by hashing it with another hash function σ. It inserts a new key x by incrementing or decrementing its counter according to $\sigma(x)$, as shown in Figure 9. CS deletes x by subtracting $\sigma(x)$ from its counter to undo its insertion. Since each key's update direction is random, the expected interference on x's counter from any other key is zero due to the positive and negative possibilities cancelling out. Thus, the value of the counter along the update direction is an estimate of x's frequency. The zero expected interference removes bias, though it introduces two-sided errors (both under and overestimations).

Although the estimates are unbiased and accurate in expectation for any workload, they may suffer from large deviations. This is because many keys hashing to the same counter may have the same update direction, despite the zero expected interference. Skew exacerbates this phenomenon since a heavy hitter drastically changes the estimates of the keys hashing to the same counter, no matter its update direction. Therefore, to bound the probability of severe errors, we compute the estimation variance.[13] Denoting the frequency of key y by $f(y)$, this variance can be expressed as $V = \sum_y (f(y))^2 / w$. This lines up with the intuition that larger arrays make hash collisions less likely, reducing the variance. A straight-forward application of Chebyshev's inequality shows that the error does not exceed $e \cdot \sqrt{V}$ with a confidence of at least $1 - e^{-2}$.[14]

[12] One can also achieve unbiased estimation using CMS by, in each array, subtracting the average of the counter values a query key does not hash to from its counter [28]. Yet, the resulting estimate has a slightly higher variance than that of CS.
[13] CS uses 4-wise independent hash functions σ to bound the variance and the probability of large errors [14].
[14] CS's error bound is $\sqrt{V} = O(\sqrt{N/w})$ when the workload is more uniform due to the squared summands being small. This bound is smaller than CMS's error, i.e., N/w.

Similarly to CMS, CS boosts confidence by employing d independent arrays, each with its own hash functions h_i and σ_i. It answers a query by returning the median estimate, as shown in Figure 9. The median is returned, as it only has large errors when most estimates err significantly, which becomes exponentially less likely given more arrays. Formally, by a Chernoff bound, CS guarantees an error bound of at most $e \cdot \sqrt{V}$ with a confidence of at least $1 - e^{-d}$.[15]

Accommodating Skew. To avoid the space wastage of uniformly-sized counters, Sublime$_{\text{CS}}$ applies VALE to CS. It stores the absolute value of each counter using VALE as before, while representing the sign of each counter by appending a sign bit to its stub.

Accommodating Unbounded Growth. Since the estimation variance V determines CS's error bound, Sublime$_{\text{CS}}$ controls the error by expanding based on the growth of the variance. That is, it expands its number of counters, i.e., the denominator w of the variance, to balance out the numerator $F = \sum_y (f(y))^2$. This translates to expanding when the numerator F more than doubles, which is in contrast with how Sublime$_{\text{CMS}}$ expands when N (the numerator of CMS's error bound) more than doubles. We encode different expansion policies by defining the size function based on the numerator F, i.e., as $W(F)$. The quantity F is also known as the squared ℓ_2-norm of the stream.

In the Appendix [32], we prove through a similar analysis to that of Sublime$_{\text{CMS}}$, the following variance bounds for Sublime$_{\text{CS}}$:

Theorem 4.2. *When using a size function $W(\cdot)$ and a single array on a stream with a squared ℓ_2-norm of $F = \sum_y f(y)^2$, Sublime$_{CS}$ provides unbiased estimates with a variance $V(F)$ satisfying*

$$V(F) \leq \begin{cases} O(1) \cdot \epsilon F^{1-\alpha} & \text{if } W(F) = F^{\alpha}/\epsilon, \text{ for } \alpha \in [0, 1), \\ \log_2 F \cdot \epsilon & \text{if } W(F) = F/\epsilon. \end{cases}$$

Compared to a fixed-size CS allocated upfront with an array of $W(N)$ counters with knowledge of F, the variance terms above are higher by at most a constant and a logarithmic[16] factor.

Applying Chebyshev's inequality and employing d independent arrays yields stable confidence bounds that match those of CS.

Sublime$_{\text{CS}}$ maps keys to consistent counters and update directions by employing the same hash functions $h_1, \ldots, h_d$ and $\sigma_1, \ldots, \sigma_d$ across expansions. It handles insertions, deletions, and queries as in the case of a standard CS.

Tracking the Squared ℓ_2-norm F. Similarly to Sublime$_{\text{CMS}}$, where we have to track the total number of keys N to determine when to expand, Sublime$_{\text{CS}}$ must track the squared ℓ_2-norm F. Yet, one cannot simply increment a value with each insertion to track F. This is because, due to the squared summands, inserting a frequent key grows F more than inserting an infrequent key. While it is possible to use the counter arrays within Sublime$_{\text{CS}}$ itself to estimate F, the resulting estimate has a confidence that is determined by the number of arrays d. For very long streams, this estimate eventually errs significantly, leading to mistimed expansions and contractions that adversely affect the accuracy and the memory footprint. We would like to have an estimate of the squared ℓ_2-norm F with very high confidence to support the entire stream.

To this end, we employ AMS sketches, which are sketches tailored to approximating the ℓ_2-norm of the stream [4]. Each AMS sketch is a single counter that is updated with random update directions derived by hashing the stream's keys, similarly to CS. Squaring the value of the counter yields an estimate of F. It is known that by employing $4 \cdot \log_2 N$ AMS sketches, one can estimate F within a factor of 2 with a high confidence of $1 - 1/N$ [4], which allows for supporting the entire stream. Since N is at most the maximum value representable in a 64-bit machine word in practice,

[15]The precise confidence given by the Chernoff bound is $1 - e^{-(e^4/8)\cdot d}$, which is higher than the text's simplified expression.
[16]This logarithmic term can be bounded in terms of the total key count N, as $F \leq N^2$ and thus $\log_2 F \leq 2 \cdot \log_2 N$.

$\text{Sublime}_{\text{CS}}$ employs 64 AMS sketches to track F. It uses SIMD to quickly update[17] and query these sketches after each insertion or deletion.

Applying Sublime to Other FE Sketches. Using the same strategy as above, Sublime can be adapted to any FE sketch with an error bound of the form X/w, where X is a measurable parameter of the workload that the error depends on, and w is the FE sketch's size. For example, in the case of CMS and Misra-Gries, X is the stream's total key count N, whereas in the case of CS, it is the stream's squared ℓ_2-norm F. In all these cases, w is the number of counters within the FE sketch. By expanding when the quantity X more than doubles and defining the size function based on X (i.e., as $W(X)$), one obtains Sublime's sublinear scaling of error and memory.

5 Memory Analysis

So far in Section 4, we characterized Sublime's error with respect to the number of counters it uses. We now incorporate VALE into the analysis to prove an upper bound on Sublime's memory footprint in bits (Theorem 5.3). We also prove a memory footprint lower bound for any FE sketch that processes a growing stream without knowledge of its ultimate length (Theorems 5.4 and 5.5). This lower bound is parameterized by the sketch's accuracy. We observe that Sublime's upper bound is within a small logarithmic factor of this lower bound, demonstrating Sublime's ability to give rise to FE sketches that attain a desired level of accuracy with a close-to-optimal memory footprint.

Upper Bound. As described in Section 4.1, Sublime adaptively tunes VALE to reclaim memory and ensure that a small fraction of its bits are unused at any time. This implies that the memory footprint is dominated by the space occupied by the stubs and the extensions. We quantify this cost for a single counter based on its count as:

Lemma 5.1. *When working with unsigned counters and stubs of length s bits, VALE encodes a counter value v in s bits if $v < 2^s$ and in at most $4 + 1.26 \cdot \log_2 v$ bits otherwise. Signed counters use one more bit for encoding the sign in both cases.*

Proof. If $v < 2^s$, it fits in its stub and occupies s bits. Otherwise, its s lower-order bits are stored in its stub and its higher-order bits encoding $\lfloor v/2^s \rfloor$ are stored in an extension. This extension has $\lfloor \log_3 \lfloor v/2^s \rfloor \rfloor + 2$ fragments to represent the digits and the delimiter. Thus, it occupies

$$2 \cdot (\lfloor \log_3 \lfloor v/2^s \rfloor \rfloor + 2) \leq 2 \cdot (\log_3 2 \cdot \log_2 \lfloor v/2^s \rfloor + 2) \leq 4 - s + 1.26 \cdot \log_2 v$$

bits. This, added to the s bits of space consumed by the stub, yields the result. □

Applying Lemma 5.1 to a counter array while accounting for VALE's tuning yields the following lemma. Intuitively, it shows that the worst-case scenario is when the counter values are similar, i.e., their distribution is uniform. This is because skew gives VALE more opportunities to save space.

Lemma 5.2. *Consider a counter array of size w and let k be the sum of its counters' absolute values. When the counters are unsigned, VALE encodes the array in $w \cdot \left\lceil 4.41 + 1.26 \cdot \log_2(k/w) \right\rceil$ bits. When the counters are signed, it uses one extra bit per counter to encode their signs.*

Proof. Applying Lemma 5.1 to every counter within the array and summing results in a total memory footprint of $m \leq \sum_i \max(s, 4 + 1.26 \cdot \log_2 |v_i|)$ bits. To simplify this expression, we bound the stub length s based on the total absolute counter values. Recalling VALE's tuning mechanics from Section 4.1, Sublime uses stubs that are slightly longer than the length of the average counter. In fact, it never sets s to be more than 2 bits longer than the average counter length. Formally,

[17] More generally, it is possible to maintain k AMS sketches, where k is the machine word size, in constant time with high probability to approximate F. Since $k = \Omega(\log n)$ is standard in both theory and practice, it is possible to maintain a high confidence estimate of F in constant time without the use of SIMD. We omit the corresponding algorithm and its proof of correctness due to space constraints.

this corresponds to $s \leq 2 + \frac{1}{w} \cdot \sum_i \log_2|v_i|$, where v_i denote the signed value of the i-th counter in the array. Since the logarithm is a concave function, we use Jensen's inequality [45] to bound the sum on the right-hand side as $\log_2(\sum_i|v_i|/w)$. Noting that the absolute counter values $|v_i|$ sum to k yields $s \leq 2 + \log_2(k/w) = 4 + \log_2(k/(4w))$. Plugging into the memory bound from before yields $m \leq \sum_i 4 + \max(\log_2(k/(4w)), 1.26 \cdot \log_2|v_i|)$. We further bound each individual max term by

$$\max\left(1.26 \cdot \log_2 \frac{k}{4w}, 1.26 \cdot \log_2|v_i|\right) \leq 1.26 \cdot \max\left(\log_2 \frac{k}{4w}, \log_2|v_i|\right) \leq 1.26 \cdot \log_2\left(\frac{k}{4w} + |v_i|\right),$$

thus implying $m \leq \sum_i 4 + 1.26 \cdot \log(k/(4w) + |v_i|)$. Rearranging the sum and once again using the concavity of the logarithm to apply Jensen's inequality yields the desired result:

$$m \leq w \cdot \left[4 + 1.26 \cdot \frac{1}{w} \cdot \sum_i \log_2(k/(4w) + k/w)\right] \leq w \cdot \left[4.41 + 1.26 \cdot \log_2(k/w)\right].$$

□

We cannot simply set k to the stream length N and multiply the space bound of Lemma 5.2 by the number of arrays d to derive Sublime's memory footprint. This is because Sublime increases the sum of the absolute values of the counters during expansions by copying them. By accounting for this, we show the following space bound for Sublime:

Theorem 5.3. *When using a sublinear power size as the function $W(N)$, Sublime$_{CMS}$, has a memory footprint of $\approx d \cdot W(N) \cdot \left[4.41 + 1.26 \cdot \log_2 N/W(N)\right]$. With a linear size function, Sublime$_{CMS}$ uses $\approx d \cdot W(N) \cdot \left[4.41 + 1.26 \cdot \log_2(N \log_2 N/W(N))\right]$ bits of memory. The same bounds hold for Sublime$_{CS}$ with $W(F)$ instead of $W(N)$ and with an additive overhead of one bit per counter.*

Proof. By a similar analysis to that of Theorem 4.1, the sum of the absolute counter values within each array in Sublime when using a sublinear power as the size function is $\approx N$, modulo a small constant factor. This constant factor become especially negligible in the space bound, since it will appear within a logarithm. Similarly, when the size function is linear, we have $k \approx N \log N$. Plugging these values into the bound of Lemma 5.2 and multiplying by d proves the result. □

Lower Bound. We prove a memory footprint lower bound for FE sketches by reducing them to filters. That is, we use an FE sketch to implement filter functionality, which allows us to inherit known memory lower bounds for filtering. A filter is a compact data structure that answers whether a query key exists in a given set of keys. A filter never returns a false negative, but it may return a false positive with a probability called the *False Positive Rate (FPR)*. The lower the desired FPR is, i.e., the higher the target accuracy level, the larger the filter's memory footprint must be. It is known that a filter that supports a set of a growing size of n with an FPR of δ must use at least $n \cdot [\Omega(\log 1/\delta + \log\log n) - \Theta(1)]$ bits of memory at some point in time [71]. Thus, an FE sketch that can implement such a filter must also use at least this much memory.

Our reduction works as follows. Consider a FE sketch (e.g., CMS). Suppose this sketch never underestimates a key's count and guarantees an error of at most $E(N)$ with a confidence of $1 - \delta$ for streams of length N. We use it to implement a filter with an FPR of δ over a set of size $n = N/(E(N)+1)$. Specifically, we insert each key in the set $E(N) + 1$ times into the FE sketch. This causes the FE sketch to return an estimate of at least $E(N) + 1$ during a query to an existing key within the set. In contrast, the FE sketch's accuracy guarantees imply that a query to a non-existent key returns an estimate of at most $E(N)$ with a probability of $1 - \delta$. Thus, by thresholding the estimate by $E(N) + 1$, we return a true positive for existing keys and a false positive for non-existent keys with an FPR of at most δ. The following theorem formalizes these intuitions:

Theorem 5.4. *Consider an FE sketch with only overestimation errors that processes growing streams with an error bound of $E(N)$ and a confidence of $1-\delta$. Such a sketch must use at least $N/E(N) \cdot [\Omega(\log 1/\delta + \log\log(N/E(N))) - \Theta(1)]$ bits of memory at some point in time while processing a stream.*

Theorem 5.4 only holds for FE sketches that do not underestimate (e.g., CMS). It is non-trivial to adapt the same argument to FE sketches that can underestimate (e.g., CS), since underestimations can introduce false negatives and violate filtering semantics. Nevertheless, by accounting for the estimation variance, we reason about the false negatives and prove in our Appendix [32] that the same lower bound holds for these sketches as well:

Theorem 5.5. *Consider an FE sketch providing estimates with an expected absolute error of $E(N)$ and a variance of $\delta \cdot (E(N))^2$ for any key when processing a stream of size N. Such an FE sketch must use at least $N/E(N) \cdot [\Omega(\log 1/\delta + \log\log(N/E(N))) - \Theta(1)]$ bits of memory at some point in time while processing a stream.*

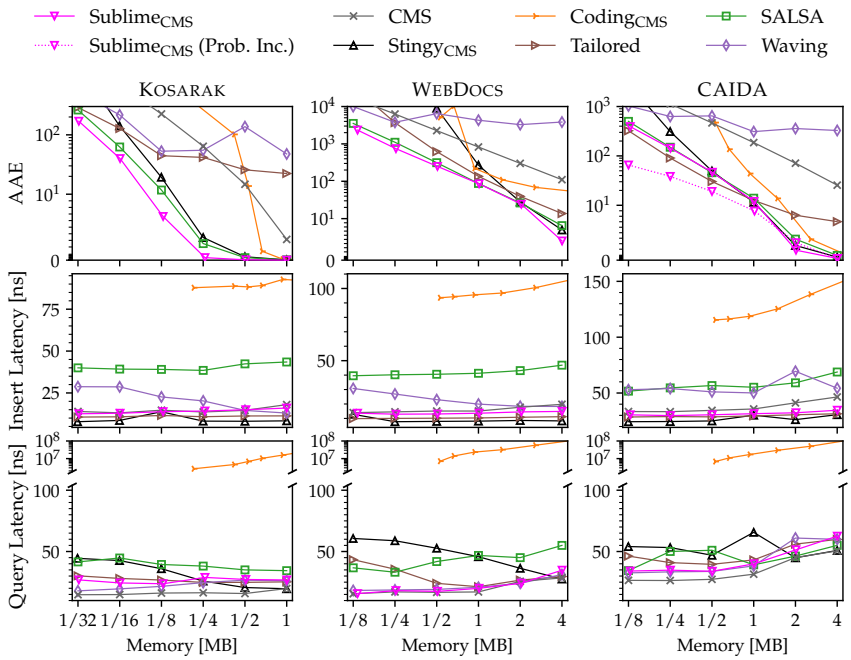
One can verify from Theorem 5.3 that, when fixing the confidence using a constant number of arrays d while using a sublinear power (resp. linear) size function $W(N)$ to achieve an error bound of $E(N)$, Sublime uses a memory of $O(N/E(N) \cdot \log(N/E(N)))$ (resp. $O(N/E(N) \cdot \log N \cdot \log\log N)$) bits. This almost matches the memory lower bound of Theorem 5.5 with a slight difference in the logarithmic terms. One can extend Sublime to match these lower bounds by compressing the counter arrays via coarsely quantizing their counters and applying run-length encoding. Doing so can potentially reduce space consumption by as much as 5× for long streams in practice. These compression techniques may compromise update and query performance, though we expect to be able to resolve this degradation using techniques based on rank and select [18, 56, 70, 74, 95].

6 Evaluation

We empirically evaluate Sublime. Experiments 1, 2, 3, and 6 focus on accommodating skew, while Experiments 4, 5, and 7 deal with unbounded data growth. Experiments 1-5 all employ $\text{Sublime}_{\text{CMS}}$. Experiments 6 and 7 showcase $\text{Sublime}_{\text{CS}}$. Experiment 8 applies both variants to join size estimation.

Platform. We run the experiments on a Fedora 41 machine with an Intel Xeon w7-2495X processor (4.8 GHz) with 24 cores. Our machine features an 80 KB L1 cache and a 2 MB L2 cache for each core, a 45 MB shared L3 cache, and 64 GB of RAM.

Baselines. We compare $\text{Sublime}_{\text{CMS}}$ to a traditional CMS, as well as the variants described in Section 3 that cater to skew. We use SALSA [7] to represent counter merging methods and Waving sketch [59] to represent hybrid methods, as they are the most accurate and the fastest in their category. For counter sharing methods, we employ the following three baselines: Stingy sketch [58] is the state of the art in terms of throughput since it implements the classic counter sharing design with a hierarchy of shared byte-aligned counters that prefetches the counters, as described in Section 3. Coding sketch [16] is the most accurate counter sharing method that compresses its hierarchy of counters, trading performance for memory. Tailored sketch [36] is the most space-efficient (and thus accurate) counter sharing method, as it probabilistically increments each counter in the classic hierarchy, thereby shortening the counters' length in exchange for introducing two-sided errors. Among these six baselines, only Coding sketch is tunable, though its tuning is manual and complex. This is because its compression algorithm obscures the level of skew in the workload to which the tuning should be adapted. We implement Sublime in `C++` and use the open-source `C++` implementations of the baselines. The aforementioned implementations are all single-threaded. We construct all FE sketches with $d = 3$ arrays. This parameter setting is common in practice since it yields a confidence bound of $\approx 95\%$, which is significantly higher than the confidence bounds resulting from $d = 1$ ($\approx 63\%$) and $d = 2$ ($\approx 86\%$) while being comparable to $d = 4$ ($\approx 98\%$).



Memory [MB]	KOSARAK		WEBDOCS		CAIDA	
	c	s	c	s	c	s
1/32	39	10	-	-	-	-
1/16	42	9	-	-	-	-
1/8	43	7	33	12	39	10
1/4	68	5	34	12	42	9
1/2	75	5	39	10	45	8
1	81	4	38	10	51	7
2	-	-	50	7	64	5
4	-	-	70	5	75	4

Table 3. Tuning VALE's parameters for each plot point to the left according to the table above enables high performance and accuracy for Sublime$_{CMS}$.

Fig. 10. Sublime$_{CMS}$ achieves the highest accuracy under real-world workloads while maintaining equal or superior insertion and query performance. We use the optimal tuning of the baselines at each point of the curves. The table above details the number of counters per chunk c and the stub length s used by Sublime$_{CMS}$.

Datasets and Workloads. Following prior work [5, 7, 15, 28, 29, 35, 36, 58, 59, 62–64, 68, 77, 79, 81, 86, 88–92], we compare the baselines in a single-threaded setting by employing three real-world streams:

- KOSARAK: 8M clicks from a news portal [11], anonymized as 8-byte integers. This dataset is highly skewed, resembling a ZIPFIAN distribution with an exponent parameter of ≈ 3.5.
- WEBDOCS: 300M words extracted from 1.7M online English documents [65]. Each word in this dataset is encoded as an 8-byte integer. This dataset follows a skewed distribution with a ZIPFIAN exponent of ≈ 1.0.
- CAIDA: 27M anonymized 5-tuples describing the network connection of packets from 10 traces taken from backbone Internet links in 2018 [1]. Each 5-tuple is represented as a 13-byte string. This dataset has a ZIPFIAN exponent of ≈ 2.0.

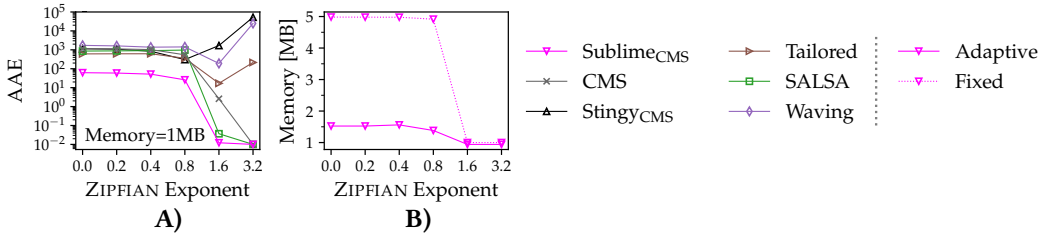


Fig. 11. By adapting VALE to skew, Sublime$_{CMS}$ achieves the highest accuracy among all baselines (Part A) and a smaller memory footprint compared to using a fixed tuning (Part B).

We evaluate the accuracy of each baseline by querying it with each distinct key in the stream and compute the *Average Absolute Error (AAE)* of the estimates from the ground-truth. We focus on absolute errors rather than relative errors to keep our presentation consistent with the theoretical error bounds studied in the literature. We have found the relative errors to behave similarly to the absolute errors.

Experiment 1: Accuracy and Speed vs. Memory. In Figure 10, we feed each workload to each baseline while varying the allotted memory budget. This experiment features bounded data growth, allowing us to focus on how well each baseline accommodates skew. Each point on the curves represents a complete run of the experiment starting from scratch. We tune Coding sketch and Sublime$_{CMS}$ at each point to minimize their error and showcase their highest accuracy. The table to the right of Figure 10 details the tuned parameters of Sublime$_{CMS}$—the number of counters per chunk c and the stub length s. We increase the memory budget by factors of 2 on the x-axis up to 4 MBs, though for the KOSARAK dataset, we use smaller budgets due to the workload trace being shorter.

Figure 10 shows that Sublime$_{CMS}$ has a lower estimation error than most other baselines by as much as an order of magnitude while providing similar or superior insertion and query performance. This is because it extends overflowing counters without affecting other counter values. Compared to the baseline with the closest errors, i.e., SALSA, Sublime$_{CMS}$ has lower errors by 40% across the board while providing faster insertions and queries by as much as 3× and 2.4×. This performance improvement is due to Sublime$_{CMS}$'s use of rank and select primitives, as well as Optimizations 1 to 6 described in Section 4.1. Coding sketch exhibits slower insertion and query performance by 2 and 7 orders of magnitude compared to the other baselines due to its decompression overheads. As such, we henceforth exclude it from the experiments that evaluate Sublime$_{CMS}$. When the memory budget is low under the CAIDA dataset, Tailored sketch is more accurate than Sublime$_{CMS}$ due to its probabilistic incrementation strategy enabling it to store shorter counters in exchange for introducing underestimations. Applying the same technique to Sublime$_{CMS}$ yields lower errors than Tailored sketch by 2.38×, as shown by the dotted curve.

Experiment 2: Accuracy under Varying Levels of Skew. Figure 11-A) compares the estimation errors of the baselines under the same memory budget as the amount of skew in the data varies. Here, we generate streams of 100M keys from a ZIPFIAN distribution over 100K distinct keys. We vary the exponent parameter, i.e., the skewness, on the x-axis from 0.0 to 3.2, which correspond to a uniform and a highly skewed dataset, respectively. Recall that, aside from Coding sketch (which we exclude from the experiment due to its poor performance), Sublime$_{CMS}$ is the only tunable FE sketch. Thus, we allow it to adapt VALE's tuning to the workload's skew.

Figure 11-A) shows that Sublime$_{CMS}$ provides the lowest error under all degrees of skew. It outperforms all other baselines by an order of magnitude for more uniform workloads and by 3× for more skewed workloads. This is because it exposes a tuning space that allows VALE to save memory by employing longer and shorter stubs for more uniform and skewed workloads, respectively.

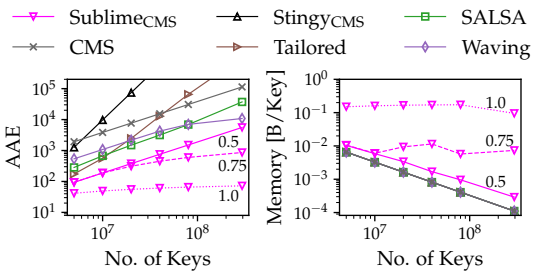


Fig. 12. Sublime$_{CMS}$ achieves both a low error rate and a small memory footprint when processing growing streams.

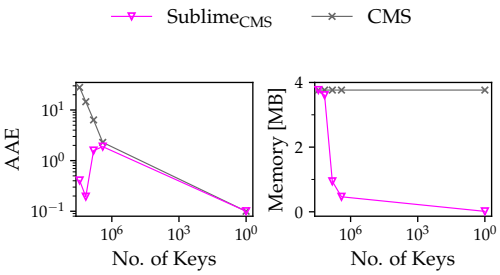


Fig. 13. Sublime$_{CMS}$ saves a significant amount of memory by contracting when many keys are deleted.

Experiment 3: Memory Savings of Adaptively Tuning VALE. We demonstrate the need for adapting VALE's tuning as the stream grows by comparing Sublime$_{CMS}$'s memory footprint when using a fixed tuning to when it adapts the tuning. We employ a total of 1M counters for both versions and use $c = 64$ counters per chunk and stubs of length $s = 6$ bits for the fixed tuning version. Figure 11-B) considers the Zipfian workloads from Experiment 2 and varies the skew on the x-axis. As shown, adaptively tuning VALE bounds the number of allocated tails arrays and improves the memory footprint over using a fixed tuning by as much as 5×. With enough skew, the average extension length decreases, and the fixed tuning version of Sublime$_{CMS}$ avoids allocating many tails arrays. Nevertheless, it does not optimally leverage the shorter extension lengths and consumes ≈ 10% more memory than the adaptive version.

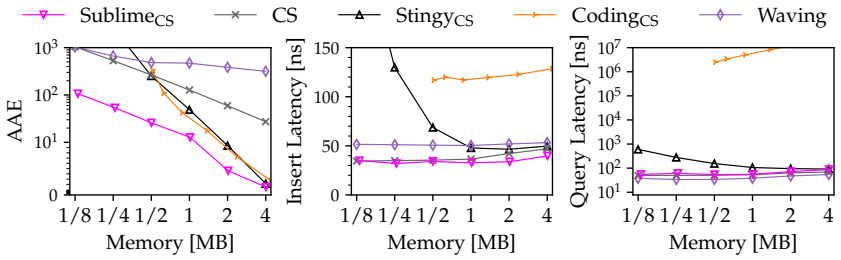
In this experiment, adaptations comprise less than 3% of the processing time, as their overhead is amortized. One can expect this overhead in mixed workloads with both insertions and deletions to be similar to our insertion-only workload. This is because our workload triggers just as many adaptations as mixed workloads by growing the counters and causing many chunks to overflow.

Experiment 4: Accuracy and Memory under Data Growth. We now compare the accuracy and memory footprint of the baselines when the stream grows indefinitely. Here, we allocate each baseline an initial memory footprint of 32KB and insert the WebDocs dataset. Figure 12 measures, for each baseline, its error and the ratio of its memory footprint to the stream's length. We use the default configuration for Sublime$_{CMS}$, which has a sublinear power size function of the form $W(N) = N^{\alpha}/\epsilon$ with $\alpha = 0.5$ and $\epsilon = 1$, as introduced in Section 4.2. We also compare versions of Sublime$_{CMS}$ with the exponent α of the size function set to 0.75 and 1.0. Figure 12 differentiates these versions by annotating each curve with the value of α used.

Figure 12 shows that Sublime$_{CMS}$ is the only FE sketch to exhibit sublinearly scaling estimation errors. It achieves this by expanding its number of counters along with the stream. This culminates in the three versions of Sublime$_{CMS}$ (with exponents α of 0.5, 0.75, and 1.0) improving the error over the best baseline by one, two, and three orders of magnitude. The right subfigure in Figure 12 shows that, despite expanding the number of counters, Sublime$_{CMS}$ maintains a modest memory footprint relative to the stream's length. Furthermore, we have found that expansions take up only 0.5% of the processing time, as they simply copy the counters sequentially.

Experiment 5: Contractions. Figure 13 showcases deletions and contractions in Sublime$_{CMS}$. Here, we insert the CAIDA dataset and subsequently delete keys in a random order while measuring the error and the memory footprint. We only compare Sublime$_{CMS}$ to CMS since all other baselines do not provide a deletion API.

As shown, Sublime$_{CMS}$ maintains lower errors than a CMS of the same size by as much as two orders of magnitude. This is because VALE encodes the counter values in less space, allowing the



Memory [MB]	c	s
1/8	39	10
1/4	42	9
1/2	45	8
1	51	7
2	64	5
4	75	4

Table 4. Tuning VALE as above leads to high performance and accuracy.

Fig. 14. Sublime$_{CS}$ has the lowest errors and fastest insertions compared to all other unbiased baselines. The table to the right presents the tuning of VALE's parameters, i.e., the number of counters per chunk c and the stub length s.

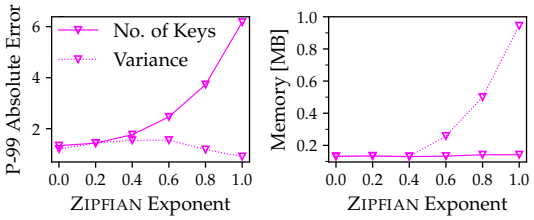


Fig. 15. By expanding based on the estimation variance, Sublime$_{CS}$ maintains a stable confidence bound.

storage of more counters. As keys are deleted, Sublime$_{CMS}$ contracts to save memory, while CMS wastes memory by remaining at the same size. This enables achieving a lower memory footprint than CMS by 2-3 orders of magnitude while still being more accurate, despite the reduced number of counters slightly increasing the error. Moreover, since contractions sequentially scan the FE sketch, similarly to expansions, they take up at most 0.5% of the total execution time.

Experiment 6: Unbiased Accuracy and Speed vs. Memory. We now turn to Sublime$_{CS}$ to demonstrate the applicability of Sublime beyond CMS. We compare it to a traditional CS and Waving sketch [59], which is representative of unbiased hybrid methods. We also compare to Stingy sketch [58] and Coding sketch [16] applied to CS, representing counter sharing methods with unbiased estimates. We exclude SALSA [7] as its library does not provide a variant with unbiased estimates. We employ the CAIDA dataset and optimally tune Coding sketch and Sublime$_{CS}$.

Figure 14 shows that, by leveraging the memory savings of VALE to store more counters, Sublime$_{CS}$ achieves lower errors than the next best baseline by as much as an order of magnitude. It also has the fastest insertions while closely matching the query speed of the most performant sketch due to its use of Optimizations 1 to 6 from Section 4.1.

Experiment 7: Expansion Based on Variance. Figure 15 illustrates how expanding Sublime$_{CS}$ based on the growth of the estimation variance (as described in Section 4.3) instead of the number of keys enables a stable confidence bound. We demonstrate this improvement by inserting 100M keys with increasing levels of skew. We measure the P-99 absolute error of the estimates instead of the average, since it showcases the errors that exceed their confidence bounds, whereas averages obscure this information. We also report the final memory footprint after ingesting the stream.

As shown in Figure 15, expanding based on the estimation variance ensures stable tail errors in exchange for a higher memory footprint. This is in contrast to expanding Sublime$_{CS}$ based on the number of keys, which yields tail errors that increase with skew. The reason is that a more skewed dataset grows the variance more quickly, necessitating more frequent expansions to maintain a stable confidence bound. As skew increases further, expanding based on the variance

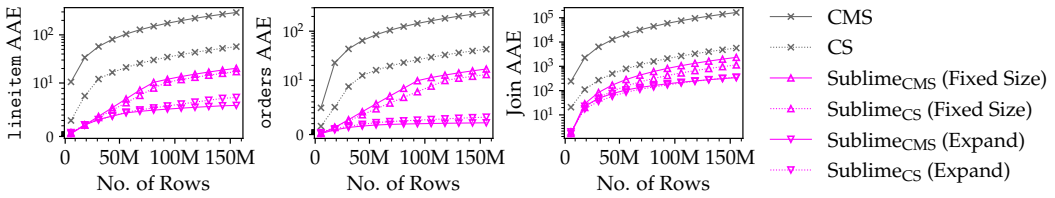


Fig. 16. Sublime provides the most accurate frequency estimates of table entries and the size of their join.

yields diminishing returns. This is due to the stream's distinct keys dropping in number with skew, making keys less likely to map to the same counter and create severe errors.

Experiment 8: Join Size Estimation. Figure 16 applies both Sublime$_{CMS}$ and Sublime$_{CS}$ to join size estimation. Here, we employ a 100 GB TPC-H workload [82]. We consider the `lineitem` and `orders` tables, incrementally insert rows into each, and periodically join them on their shared `ORDERKEY` column. Following [83], each row in the orders table is duplicated 1-3 times to simulate a many-to-many join. We construct FE sketches tracking the count of each `ORDERKEY` value in these tables and predict their counts in the join by multiplying the per-table estimates. We compare CMS and CS, as well as fixed-size and expandable versions of Sublime$_{CMS}$ and Sublime$_{CS}$. We set the initial memory budget to 8MB ($\approx$ 0.1% of the data size) and distribute it to each sketch in proportion to the initial size of its table. The expandable Sublime$_{CMS}$ and Sublime$_{CS}$ grow as a sublinear power (with a power of 0.75) of the tables. Figure 16 reports accuracy measurements for each table and the join result.

As shown in Figure 16, Sublime attains higher accuracy than CMS and CS by multiple orders of magnitude due to their use of variable-length counters. Moreover, the expandable Sublime$_{CMS}$ and Sublime$_{CS}$ outperform their fixed-size counterparts by as much as an order of magnitude. The fixed-size version of Sublime$_{CS}$ is more accurate than the fixed-size version of Sublime$_{CMS}$ on each separate table. This is because the workload in this experiment is almost uniform, leading to the positive and negative counts of colliding keys in Sublime$_{CS}$ to cancel out. When allowed to expand, Sublime$_{CMS}$ is more accurate than Sublime$_{CS}$ on individual tables, as collisions reduce with expansions, and Sublime$_{CMS}$ does not store a sign bit for each counter. Nevertheless, both versions of Sublime$_{CS}$ have more accurate join estimates than the corresponding version of Sublime$_{CMS}$. This is due to Sublime$_{CS}$ returning unbiased estimates, which allows for multiplying them without compounding errors.

7 Conclusion

We introduced Sublime, the first framework for generalizing a frequency estimation sketch to adapt to the skew and length of the stream. We showed that Sublime improves the accuracy and memory of FE sketches beyond the state of the art while maintaining high performance. Although Sublime is currently single-threaded, we expect significant performance gains from parallelization (e.g., through employing thread-local FE sketches to ingest subsets of the stream and periodically merging them into a unified FE sketch for answering queries [51, 76]). More broadly, Sublime opens a new research avenue on expandable data sketches applicable to other core statistics, such as quantiles and cardinalities, which exhibit similar error growth under workloads with growing datasets. This marks a fundamental shift from fixed-size sketches with degrading accuracy to adaptive structures that maintain stable accuracy for always-on analytics.

Acknowledgments

We thank the reviewers for their insightful comments. This research was supported by the NSERC grant #RGPIN-2023-03580.

References

[1] [n. d.]. Anonymized Internet Traces 2018. https://catalog.caida.org/dataset/passive_2018_pcap. doi:dataset/passive_2018_pcap Dates used: 2025-10-11. Accessed: 2025-10-11..

[2] Charu C. Aggarwal. 2013. *An Introduction to Sensor Data Analytics*. Springer US, Boston, MA, 1–8. doi:10.1007/978-1-4614-6309-2_1

[3] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms* (Kyoto, Japan) *(SODA '12)*. Society for Industrial and Applied Mathematics, USA, 459–467.

[4] Noga Alon, Yossi Matias, and Mario Szegedy. 1999. The space complexity of approximating the frequency moments. *J. Comput. System Sci.* 58, 1 (1999), 137–147. doi:10.1006/jcss.1998.1627

[5] Daniel Anderson, Pryce Bevan, Kevin Lang, Edo Liberty, Lee Rhodes, and Justin Thaler. 2017. A high-performance algorithm for identifying frequent items in data streams. In *Proceedings of the 2017 Internet Measurement Conference* (London, United Kingdom) *(IMC '17)*. ACM, New York, NY, USA, 268–282. doi:10.1145/3131365.3131407

[6] Jim Apple. 2022. Stretching your data with taffy filters. *Software: Practice and Experience* (2022).

[7] Ran Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. 2021. SALSA: Self-Adjusting Lean Streaming Analytics. 864–875. doi:10.1109/ICDE51399.2021.00080

[8] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. 2020. Faster and More Accurate Measurement through Additive-Error Counters. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 1251–1260. doi:10.1109/INFOCOM41043.2020.9155340

[9] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Roy Friedman, and Yaron Kassner. 2019. Randomized Admission Policy for Efficient Top-k, Frequency, and Volume Estimation. *IEEE/ACM Trans. Netw.* 27, 4 (Aug. 2019), 1432–1445. doi:10.1109/TNET.2019.2918929

[10] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. 2022. Memento: Making Sliding Windows Efficient for Heavy Hitters. *IEEE/ACM Transactions on Networking* 30, 4 (2022), 1440–1453. doi:10.1109/TNET.2021.3132385

[11] Ferenc Bodon. 2012. Frequent Itemset Mining Dataset Repository, Kosarak. http://fimi.uantwerpen.be/data/

[12] F. Bostanci, Ismail Yüksel, Ataberk Olgun, Konstantinos Kanellopoulos, Yahya Tuğrul, Abdullah Giray Yaglikçi, Mohammad Sadrosadati, and Onur Mutlu. 2024. CoMeT: Count-Min-Sketch-based Row Tracking to Mitigate RowHammer at Low Cost. 593–612. doi:10.1109/HPCA57654.2024.00050

[13] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

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

[15] Min Chen, Shigang Chen, and Zhiping Cai. 2017. Counter Tree: A Scalable Counter Architecture for Per-Flow Traffic Measurement. *IEEE/ACM Trans. Netw.* 25, 2 (April 2017), 1249–1262. doi:10.1109/TNET.2016.2621159

[16] Qizhi Chen, Yisen Hong, Yuhan Wu, Tong Yang, and Bin Cui. 2024. CodingSketch: A Hierarchical Sketch with Efficient Encoding and Recursive Decoding. In *ICDE*. 1592–1605. https://doi.org/10.1109/ICDE60146.2024.00130

[17] Yuvaraj Chesetti, Navid Eslami, Huanchen Zhang, Niv Dayan, and Prashant Pandey. 2026. Aeris Filter: A Strongly and Monotonically Adaptive Range Filter. In *Proceedings of the 2026 International Conference on Management of Data* (Bangalore, India) *(SIGMOD '26)*. ACM, New York, NY, USA, 1670–1684.

[18] David Clark. 1997. *Compact PAT trees*. Ph. D. Dissertation. http://hdl.handle.net/10012/64

[19] Felix Cloutier. 2023. *PDEP — Parallel Bits Deposit*. https://www.felixcloutier.com/x86/pdep

[20] 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* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 13–24.

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

[22] G. Cormode and S. Muthukrishnan. 2005. What's new: finding significant differences in network data streams. *IEEE/ACM Transactions on Networking* 13, 6 (2005), 1219–1232. doi:10.1109/TNET.2005.860096

[23] Intel Corporation. 2018. *5-Level Paging and 5-Level EPT White Paper*. Technical Report 335252-002. Intel Corporation. https://www.intel.com/content/www/us/en/content-details/671442/5-level-paging-and-5-level-ept-white-paper.html Revision 1.1.

[24] Intel Corporation. 2025. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*. Intel Corporation. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html Order Number 253668.

[25] Zhenwei Dai, Aditya Desai, Reinhard Heckel, and Anshumali Shrivastava. 2021. Active Sampling Count Sketch (ASCS) for Online Sparse Estimation of a Trillion Scale Covariance Matrix. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. ACM, New York, NY, USA, 352–364. doi:10.1145/3448016.3457327

[26] Niv Dayan, Ioana O. Bercea, and Rasmus Pagh. 2024. Aleph Filter: To Infinity in Constant Time. *Proc. VLDB Endow.* 17, 11 (July 2024), 3644–3656. doi:10.14778/3681954.3682027

[27] Niv Dayan, Ioana O. Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. In *Proceedings of the 2023 International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '23, Vol. 1)*. ACM, New York, NY, USA, Article 140, 27 pages. doi:10.1145/3589285

[28] Fan Deng and Davood Rafiei. [n. d.]. New Estimation Algorithms for Streaming Data: Count-min Can Do More. https://webdocs.cs.ualberta.ca/~drafiei/papers/cmm.pdf

[29] Rui Ding, Shibo Yang, Xiang Chen, and Qun Huang. 2023. BitSense: Universal and Nearly Zero-Error Optimization for Sketch Counters with Compressive Sensing. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. ACM, New York, NY, USA, 220–238. doi:10.1145/3603269.3604865

[30] Otmar Ertl. 2024. UltraLogLog: A Practical and More Space-Efficient Alternative to HyperLogLog for Approximate Distinct Counting. *Proc. VLDB Endow.* 17, 7 (March 2024), 1655–1668. doi:10.14778/3654621.3654632

[31] Navid Eslami, Ioana O. Bercea, and Niv Dayan. 2025. Diva: Dynamic Range Filter for Var-Length Keys and Queries. *Proc. VLDB Endow.* 18, 11 (July 2025), 3923–3936. doi:10.14778/3749646.3749664

[32] Navid Eslami, Ioana O. Bercea, Rasmus Pagh, and Niv Dayan. 2026. Sublime: Sublinear Error & Space for Unbounded Skewed Streams. arXiv:2603.14190 [cs.DS] https://arxiv.org/abs/2603.14190

[33] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms 2007 (AofA07)*, Philippe Jacquet (Ed.). Juan les pins, France, 127–146. https://hal.science/hal-00406166

[34] Guoju Gao, Tianyu Ma, He Huang, Yu-E Sun, Haibo Wang, Yang Du, and Shigang Chen. 2024. Scout Sketch+: Finding Both Promising and Damping Items Simultaneously in Data Streams. *IEEE/ACM Trans. Netw.* 32, 6 (Oct. 2024), 5491–5506. doi:10.1109/TNET.2024.3469196

[35] Guoju Gao, Zhaorong Qian, He Huang, and Yang Du. 2023. An Adaptive Counter-Splicing-Based Sketch for Efficient Per-Flow Size Measurement. 1–4. doi:10.1109/IWQoS57198.2023.10188733

[36] Guoju Gao, Zhaorong Qian, He Huang, Yu-E Sun, and Yang Du. 2025. TailoredSketch: A Fast and Adaptive Sketch for Efficient Per-Flow Size Measurement. *IEEE Transactions on Network Science and Engineering* 12, 1 (2025), 505–517. doi:10.1109/TNSE.2024.3503904

[37] Lukasz Golab. 2009. *Stream Models.* Springer US, Boston, MA, 2834–2836. doi:10.1007/978-0-387-39940-9_370

[38] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (Santa Barbara, California, USA) *(SIGMOD '01)*. Association for Computing Machinery, New York, NY, USA, 58–66. doi:10.1145/375663.375670

[39] Yu Gu, Andrew McCallum, and Don Towsley. 2005. Detecting anomalies in network traffic using maximum entropy estimation. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement* (Berkeley, CA) *(IMC '05)*. USENIX Association, USA, 32.

[40] Mike Heddes, Igor Nunes, Tony Givargis, and Alex Nicolau. 2024. Convolution and Cross-Correlation of Count Sketches Enables Fast Cardinality Estimation of Multi-Join Queries. *Proc. ACM Manag. Data* 2, 3, Article 129 (May 2024), 26 pages. doi:10.1145/3654932

[41] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: fast connectivity recovery entirely in the data plane. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 161–176.

[42] William George Horner. 1819. IX. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London* 109 (1819), 308–335.

[43] Yesdaulet Izenov. 2023. *Query Optimization using Sketches in Relational Database Systems.* PhD thesis. University of California, Merced, Merced, CA. Available at https://escholarship.org/uc/item/5k8334fs.

[44] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. 2021. COMPASS: Online Sketch-based Query Optimization for In-Memory Databases. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. ACM, New York, NY, USA, 804–816. doi:10.1145/3448016.3452840

[45] J. L. W. V. Jensen. 1906. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Mathematica* 30 (1906), 175–193. doi:10.1007/bf02418571

[46] Hossein Jowhari, Mert Sağlam, and Gábor Tardos. 2011. Tight bounds for Lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Athens, Greece) *(PODS '11)*. Association for Computing Machinery, New York, NY, USA, 49–58. doi:10.1145/1989284.1989289

[47] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 34–50. doi:10.1145/3132747.3132749

[48] Zohar S. Karnin, Kevin J. Lang, and Edo Liberty. 2016. Optimal Quantile Approximation in Streams. *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)* (2016), 71–78. https://api.semanticscholar.org/CorpusID:15640305

[49] Matti Karppa and Rasmus Pagh. 2022. HyperLogLogLog: Cardinality Estimation With One Log More. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 753–761. doi:10.1145/3534678.3539246

[50] The kernel development community. 2024. 5-level paging. Linux Kernel Documentation. https://docs.kernel.org/arch/x86/x86_64/5level-paging.html Accessed: 2026-01-19.

[51] Martin Kiefer, Ilias Poulakis, Eleni Tzirita Zacharatou, and Volker Markl. 2023. Optimistic Data Parallelism for FPGA-Accelerated Sketching. *Proc. VLDB Endow.* 16, 5 (Jan. 2023), 1113–1125. doi:10.14778/3579075.3579085

[52] Hyuhng Min Kim, Navid Eslami, and Niv Dayan. 2026. Zeno Filter: To Infinity in Tiny Steps. In *Proceedings of the 2026 International Conference on Management of Data* (Bangalore, India) *(SIGMOD '26)*. ACM, New York, NY, USA, 24 pages.

[53] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Professional, USA.

[54] George Kollios, John Byers, Jeffrey Considine, Marios Hadjieleftheriou, and Feifei Li. 2005. Robust Aggregation in Sensor Networks. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2005). https://www2.cs.arizona.edu/classes/cs645/fall05/cs645-papers/DataAggregation/dataC.pdf Special Issue on Data Management in Sensor Networks.

[55] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. 2003. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement* (Miami Beach, FL, USA) *(IMC '03)*. ACM, New York, NY, USA, 234–247. doi:10.1145/948205.948236

[56] Florian Kurpicz. 2022. Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors. In *String Processing and Information Retrieval – 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8–10, 2022, Proceedings. Ed.: D. Arroyuelo (Lecture Notes in Computer Science, Vol. 13617)*. Springer International Publishing, 257–272. doi:10.1007/978-3-031-20643-6_19

[57] Rhodes Lee, Alaxander Lang, Kevin Saydakov, Justin Thaler, Edo Liberty, and Jon Malkin. 2025. *DataSketches: A Java software library of stochastic streaming algorithms*. https://datasketches.apache.org/

[58] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. 2022. Stingy sketch: a sketch framework for accurate and fast frequency estimation. *Proc. VLDB Endow.* 15, 7 (March 2022), 1426–1438. doi:10.14778/3523210.3523220

[59] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 2020. WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. ACM, New York, NY, USA, 1574–1584. doi:10.1145/3394486.3403208

[60] Shangsen Li, Lailong Luo, Deke Guo, Qianzhen Zhang, and Pengtao Fu. 2020. A survey of sketches in traffic measurement: Design, Optimization, Application and Implementation. https://api.semanticscholar.org/CorpusID:236140679

[61] Shangsen Li, Lailong Luo, Deke Guo, Qianzhen Zhang, and Pengtao Fu. 2020. A survey of sketches in traffic measurement: Design, Optimization, Application and Implementation. https://api.semanticscholar.org/CorpusID:236140679

[62] Weihe Li and Paul Patras. 2023. P-Sketch: A Fast and Accurate Sketch for Persistent Item Lookup. *IEEE/ACM Trans. Netw.* 32, 2 (Aug. 2023), 987–1002. doi:10.1109/TNET.2023.3306897

[63] Weihe Li and Paul Patras. 2023. Tight-Sketch: A High-Performance Sketch for Heavy Item-Oriented Data Stream Mining with Limited Memory Size. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management* (Birmingham, United Kingdom) *(CIKM '23)*. ACM, New York, NY, USA, 1328–1337. doi:10.1145/3583780.3615080

[64] Zirui Liu, Yixin Zhang, Yifan Zhu, Ruwen Zhang, Tong Yang, Kun Xie, Sha Wang, Tao Li, and Bin Cui. 2023. TreeSensing: Linearly Compressing Sketches with Flexibility. *Proc. ACM Manag. Data* 1, 1, Article 56 (May 2023), 28 pages. doi:10.1145/3588910

[65] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. 2012. Frequent Itemset Mining Dataset Repository, Webdocs. http://fimi.uantwerpen.be/data/

[66] Aleksander Łukasiewicz, Jakub Těthek, and Pavel Veselý. 2025. SplineSketch: Even More Accurate Quantiles with Error Guarantees. *Proc. ACM Manag. Data* 3, 6, Article 362 (Dec. 2025), 26 pages. doi:10.1145/3769827

[67] Dzejla Medjedovic, Emin Tahirovic, and Ines Dedovic. 2025. *Algorithms and data structures for massive datasets*. Manning Publications Co., Chapter 4.

[68] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory* (Edinburgh, UK) *(ICDT'05)*. Springer-Verlag, Berlin, Heidelberg, 398–412. doi:10.1007/978-3-540-30570-5_27

[69] Jayadev Misra and David Gries. 1982. *Finding Repeated Elements.* Technical Report. USA.

[70] Daisuke Okanohara and Kunihiko Sadakane. 2007. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments* (New Orleans, Louisiana). Society for Industrial and Applied Mathematics, USA, 60–70.

[71] Rasmus Pagh, Gil Segev, and Udi Wieder. 2013. How to Approximate a Set without Knowing Its Size in Advance. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science* (2013), 80–89. https://api.semanticscholar.org/CorpusID:10365891

[72] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 775–787. doi:10.1145/3035918.3035963

[73] Prashant Pandey, Martín Farach-Colton, Niv Dayan, and Huanchen Zhang. 2024. Beyond Bloom: A Tutorial on Future Feature-Rich Filters. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. ACM, New York, NY, USA, 636–644. doi:10.1145/3626246.3654681

[74] Mihai Patrascu. 2008. Succincter. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*. IEEE Computer Society, USA, 305–313. doi:10.1109/FOCS.2008.83

[75] Lee Rhodes. 2023. *Insights from Engineering Sketches for Production and Using Sketches at Scale.* https://simons.berkeley.edu/talks/lee-rhodes-yahoo-inc-2023-10-12

[76] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. 2022. Fast Concurrent Data Sketches. *ACM Trans. Parallel Comput.* 9, 2, Article 6 (April 2022), 35 pages. doi:10.1145/3512758

[77] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented Sketch: Faster and More Accurate Stream Processing. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1449–1463. doi:10.1145/2882903.2882948

[78] Simon Scherrer, Che-Yu Wu, Yu-Hsi Chiang, Benjamin Rothenberger, Daniele Asoni, Arish Sateesan, Jo Vliegen, Nele Mentens, Hsu-Chun Hsiao, and Adrian Perrig. 2021. Low-Rate Overuse Flow Tracer (LOFT): An Efficient and Scalable Algorithm for Detecting Overuse Flows. 265–276. doi:10.1109/SRDS53918.2021.00034

[79] Qilong Shi, Chengjun Jia, Wenjun Li, Zaoxing Liu, Tong Yang, Jianan Ji, Gaogang Xie, Weizhe Zhang, and Minlan Yu. 2024. BitMatcher: Bit-level Counter Adjustment for Sketches. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 4815–4827. doi:10.1109/ICDE60146.2024.00366

[80] Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. 2021. Set-Min sketch: a probabilistic map for power-law distributions with application to k-mer annotation. *bioRxiv* (2021). arXiv:https://www.biorxiv.org/content/early/2021/02/25/2020.11.14.382713.full.pdf doi:10.1101/2020.11.14.382713

[81] Lu Tang, Qun Huang, and Patrick P. C. Lee. 2019. MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications* (Paris, France). IEEE Press, 2026–2034. doi:10.1109/INFOCOM.2019.8737499

[82] Transaction Processing Performance Council (TPC). 2024. TPC Benchmark™ H (Decision Support). https://www.tpc.org/tpch/. Accessed: 2025-12-03.

[83] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P. Chakkappen. 2015. Join size estimation subject to filter conditions. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1530–1541. doi:10.14778/2824032.2824051

[84] A. Wagner and B. Plattner. 2005. Entropy based worm and anomaly detection in fast IP networks. In *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05)*. 172–177. doi:10.1109/WETICE.2005.35

[85] Henry S. Warren, Jr. 2012. *Hacker's Delight* (2nd ed.). Addison-Wesley Professional, Upper Saddle River, NJ.

[86] Yuhan Wu, Shiqi Jiang, Siyuan Dong, Zheng Zhong, Jiale Chen, Yutong Hu, Tong Yang, Steve Uhlig, and Bin Cui. 2023. MicroscopeSketch: Accurate Sliding Estimation Using Adaptive Zooming. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. ACM, New York, NY, USA, 2660–2671. doi:10.1145/3580305.3599432

[87] Jiqiang Xia, Le Tian, Yuxiang Hu, Ziyong Li, Penghao Sun, and Jianhua Peng. 2025. CoDDoS: Detecting and mitigating diverse DDoS attacks with programmable switches. *Comput. Commun.* 240, C (Aug. 2025), 13 pages. doi:10.1016/j.comcom.2025.108215

[88] 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* (London, United Kingdom) *(KDD '18)*. ACM, New York, NY, USA, 2584–2593. doi:10.1145/3219819.3219978

[89] 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* (Budapest, Hungary) *(SIGCOMM '18)*. ACM, New York, NY, USA, 561–575. doi:10.1145/3230543.3230544

[90] 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 Trans. Netw.* 27, 5 (Oct. 2019), 1845–1858. doi:10.1109/TNET.2019.2933868

[91] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. 2017. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1442–1453. doi:10.14778/3137628.3137652

[92] Boyu Zhang, He Huang, Yu-E. Sun, Yang Du, and Dan Wang. 2024. Jigsaw-Sketch: a fast and accurate algorithm for finding top-k elephant flows in high-speed networks. *Science China Information Sciences* 67, 4 (20 Mar 2024), 142101. doi:10.1007/s11432-022-3794-1

[93] Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C Titus Brown. 2014. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS one* 9, 7 (2014), e101271.

[94] Yinda Zhang, Peiqing Chen, and Zaoxing Liu. 2024. OctoSketch: Enabling Real-Time, Continuous Network Monitoring over Multiple Cores. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1621–1639. https://www.usenix.org/conference/nsdi24/presentation/zhang-yinda

[95] Dong Zhou, David G. Andersen, and Michael Kaminsky. 2013. Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences. In *Experimental Algorithms*, Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–163.

Received 17 October 2025; revised 5 February 2026; accepted 11 March 2026