Insert-Optimized Implementation of Streaming Data Sketches

Pascal Pfeil
Amazon Web Services, Germany
pfeip@amazon.de

Dominik Horn
Amazon Web Services, Germany
domhorn@amazon.de

Orestis Polychroniou
Amazon Web Services, USA
orestis@amazon.com

George Erickson
Amazon Web Services, USA
geoee@amazon.com

Zhe Heng Eng
Amazon Web Services, USA
zhee@amazon.com

Mengchu Cai
Amazon Web Services, USA
mengchu@amazon.com

Tim Kraska
Amazon Web Services, USA
timkrask@amazon.com

Abstract

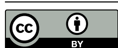
We present insert-optimized implementations of three fundamental data sketching algorithms: Count Sketch (CS), SpaceSaving (SS), and Karnin-Lang-Liberty (KLL). While these sketches are widely used for approximate query processing and stream analytics, their practical insert performance often falls short of their full potential. Through careful engineering and novel implementation strategies, we achieve substantial throughput improvements over both naïve and existing implementations. Our approach demonstrates speedups of up to 12.30x, 2.00x, and 1.52x for CS, SS, and KLL respectively when compared to the industry-standard Apache DataSketches library. When measured against naïve implementations, we achieve even more dramatic improvements: 8.63x for CS, 7.03x for SS, and 446.00x for KLL. We also measure against available open-source implementations used as baselines in other works and outperform them by a large margin. We detail the technical optimizations enabling these improvements, including fast hash range reduction and hash sharing for Count Sketch, SIMD vectorization for SpaceSaving, and preallocation and branching improvements for Karnin-Lang-Liberty. Our implementations maintain the theoretical guarantees of the original algorithms while providing substantially better practical insert performance, making them particularly valuable for high-throughput streaming applications where update speed is critical.

CCS Concepts

• **Information systems → Stream management**; • **Theory of computation → Sketching and sampling**; *Data structures design and analysis.*

Keywords

Data Sketches, Streaming Algorithms, Approximate Query Processing, Count Sketch, SpaceSaving, Karnin-Lang-Liberty, SIMD Vectorization, Performance Optimization



This work is licensed under a Creative Commons Attribution 4.0 International License.
DaMoN '25, Berlin, Germany
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1940-0/2025/06
https://doi.org/10.1145/3736227.3736238

ACM Reference Format:
Pascal Pfeil, Dominik Horn, Orestis Polychroniou, George Erickson, Zhe Heng Eng, Mengchu Cai, and Tim Kraska. 2025. Insert-Optimized Implementation of Streaming Data Sketches. In *21st International Workshop on Data Management on New Hardware (DaMoN '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3736227.3736238

1 Introduction

Data sketches have become ubiquitous in modern data processing systems, particularly in database management systems (DBMS) [1, 25, 29], click stream analysis [6, 23], and network monitoring applications [2, 28]. In DBMS, they are used for maintaining statistics for query optimization, or approximate query or stream processing.

These probabilistic data structures enable efficient analysis of massive data streams by maintaining compact summaries that support approximate query processing with theoretical accuracy guarantees. At their core, data sketches are specialized algorithms that process streaming data using sub-linear space while providing reliable statistical estimates. Their accuracy is characterized by two key parameters: ϵ (epsilon), which defines the maximum relative error of the estimate, and δ (delta), which bounds the probability of exceeding this error threshold. Non-probabilistic sketches never exceed their relative error bounds ($\delta = 0$). Common applications include frequency estimation for tracking item occurrences, quantile estimation for approximate percentile queries over large datasets, and heavy hitter detection for identifying frequently occurring items. The ability of sketches to process high-velocity data streams with minimal overhead makes them particularly valuable in scenarios where traditional exact algorithms would be prohibitively expensive or infeasible.

Sketches have been extended in various ways over the last decades. For example, mechanisms to allow for dynamic dataset modifications have been developed [10, 33, 34] and new sketch designs [7, 8, 14, 20, 30–32] were created to improve accuracy, adaptability, and/or space efficiency. Furthermore, research has focused on optimizing performance through techniques such as prefiltering [27, 36], hardware acceleration [27, 35], and refined update strategies [19, 21, 35].

In this paper, we focus on three widely-adopted sketch algorithms. First, we explore Count Sketch, a technique employed for frequency estimation, which provides approximate counts of individual elements. Next, we analyze SpaceSaving, an algorithm designed for

heavy hitters estimation, offering an approximation of the most frequently occurring elements. Finally, we investigate the Karnin-Lang-Liberty algorithm, utilized for quantile estimation, which returns an element approximately at a given quantile.

While recent research has introduced various modifications to classical sketch algorithms or entirely new designs, these approaches are typically evaluated against suboptimal implementations of classical algorithms. For instance, reported Count Sketch (or Count Min Sketch) insertion throughput varies widely: ~3M/s [32], ~13M/s [35], ~25M/s [36], ~25k/s [33], and ~6M/s [27]. Although these figures are not directly comparable due to hardware differences, our optimized implementation significantly outperforms these reported numbers, sometimes even by orders of magnitude. Analysis of available open-source implementations reveals commonalities with the naïve implementation used in our evaluation, frequently employing slow hash functions, utilizing modulo for range reduction, and/or separate hash functions for sign computation. We benchmark available open-source implementations against our best implementation and outperform them by a large margin. Similarly, reported SpaceSaving insertion throughput varies: ~6M/s [35], ~5-20M/s [36], and ~111k/s [33]. Again, while these figures are not directly comparable, our optimized implementation substantially surpasses these reported numbers. Existing implementations typically utilize a min heap but rely on inefficient hash maps or linear scans for item existence checks. Our implementation substantially exceeds the performance of available open-source implementations, as demonstrated by our benchmarks. Even for the well-optimized Karnin-Lang-Liberty algorithm implementation reported in [15], which reportedly achieves approximately 20M inserts/s, our optimizations yield further improvements, achieving up to 1.52x speedup over this implementation in a direct comparison.

Sketch algorithm performance is crucial for their adoption in real-time data processing systems, especially if they are integrated into the hot path. In DBMS, sketches allow continuous statistics collection during query processing or data ingestion without compromising performance. For instance, one of our large-scale database services is able to process hundreds of millions of items per second per thread. In network equipment, sketches enable real-time traffic analysis at line rate. Other applications include ad impression tracking, social media sentiment analysis, and financial fraud detection. Many applications require maintaining multiple sketches simultaneously over a single data stream. Since each additional sketch adds to the processing overhead, highly optimized implementations are essential. Our work demonstrates that an optimized implementation of classical sketch algorithms can yield performance improvements that exceed baseline measurements by one or even multiple orders of magnitude, opening new possibilities for their integration into various systems.

In summary, the contributions are: (a) We present implementation techniques that substantially improve insertion throughput for common sketch algorithms, (b) we do comprehensive performance comparisons against existing implementations from the Apache DataSketches library, other works where source code is available [32, 35, 36], and naïve implementations, and (c) we open-source our tuned sketch implementations for other researchers and system developers[1], ultimately creating a strong baseline for other researchers.

[1]https://github.com/amazon-science/Insert-Optimized-Data-Sketches
Note: Due to licensing restrictions, implementations from compared papers are not included in this repository.

2 Count Sketch (CS)

Count Sketch [3] is a probabilistic sketch used for frequency estimation. It consists of a two-dimensional array of counters with d rows and w columns. The parameter d balances error probability (δ) against memory consumption and insertion performance, while w establishes a trade-off between relative error (ϵ) and memory requirement. Initially, all array cells are set to zero. The sketch utilizes two hash functions per row: one maps items to column indices, while the other generates a sign (+1 or −1) for counter updates. For frequency estimation, the algorithm applies these same hash functions, selecting one counter from each row, multiplying it by its corresponding sign, and computing the median value. The structure requires $O(\frac{1}{\epsilon}\log\frac{1}{\delta})$ space and achieves $O(\frac{1}{\delta})$ update time.

CS is very similar to the Count Min Sketch (CMS)[4] which will always increment counters, and return the minimum counter when querying. This approach results in CMS producing inherently biased estimates, as it cannot underestimate frequencies.

Experiment Setup. All experiments were conducted on an `m5.12xlarge` AWS instance, featuring 48 Intel Xeon Platinum 8259CL CPUs @ 3.1 GHz, running Amazon Linux 2, unless specified otherwise. We performed single-threaded microbenchmarks using Google Benchmark v1.9.1 [12], with all code compiled using GCC 7.3.0.

To evaluate insertion speed, we generated 1 000 000 uniformly distributed values for insertion into the sketches. For string-based tests, we converted double-precision floating-point numbers to fixed-length string representations, each 320 characters long.

While our code can be easily adapted to generate data following different distributions, we focused on uniform distribution for this study. It's worth noting that skewed distributions with few distinct values do not impact the insertion performance of hash-based sketches like CS. However, they can improve insertion speeds for SS by up to 2x and for KLL by up to 1.33x. As the overall trends remain consistent across distributions, we have chosen to omit these additional results from this paper for clarity and conciseness.

Our performance comparison is presented in Figure 1. We configured the sketches with a width t of 2048, yielding a relative error bound ϵ of ~0.01%, and a depth d of 5, resulting in a theoretical error probability δ of ~0.67%. To ensure fair comparison, all of our own implementations utilize the same 128-bit MurmurHash3. Note that all implementations roughly use the same amount of memory as they use the same counter width and number of counters.

Apache DataSketches. The library provides a CMS implementation, which serves as one of our comparison baselines. Given CMS and CS are very similar, their performance characteristics are comparable. Their implementation employs modulo operations for hash range reduction and utilizes one hash function per row.

Naïve. This implementation employs a single hash function per row combined with modulo operations for range reduction. It incorporates an optimization technique from [5] that efficiently extracts both the sign and cell index from a single hash value, eliminating the need for a second hash computation. While matching the number of hashes required to compute of the Apache DataSketches CMS implementation, our approach differs in two key aspects: we interleave the hashing operations with counter updates, and we avoid allocating a separate vector for update locations. Through these modifications,

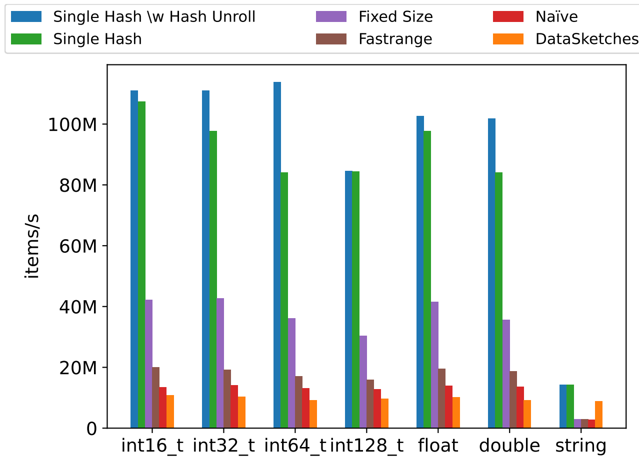


Figure 1: Count Sketch Insert. Comparison of insertion throughput for the CS implementations for different data types

the implementation achieves up to 1.48x higher throughput compared to DataSketches.

Fastrange. While modulo operations are traditionally used for range reduction, `fastrange` [17] provides a more efficient alternative while maintaining a fair, uniform mapping to the target range. The technique works by exploiting the fact that when multiplying a value x by the desired range N, the high bits of their product $(x*N)$ naturally distribute values uniformly across the range $[0, N-1]$, eliminating the need for expensive division instructions. This approach is approximately 4x faster than modulo operations when viewed in isolation [17]. A Count Sketch implementation using this range reduction technique achieves up to 1.48x higher throughput compared to our naïve implementation and is up to 2.03x faster than DataSketches.

Fixed-size and power-of-two range reduction. By restricting t to the next power of two (giving us better error bounds in exchange for higher memory consumption) and making the sketch size fixed, we can simplify range reduction to a single bitwise AND instruction. This optimization eliminates the need for more complex range reduction operations, resulting in throughput improvements of up to 3.11x compared to naïve and 4.14x compared to DataSketches.

Single Hash. The hash function provides 128 bits, but we use significantly fewer bits per row. For example, with $t = 2048$, we only need $\log_2(2048)+1 = 12$ bits per row to identify the cell and compute the sign. In total, for $d = 5$, we need $12*5 = 60$ bits per value, far less than a single call to the hash function can provide. This means we can use a single hash operation for any $t \leq 33554432$ when $d = 5$. Furthermore, for $t \leq 2048$, we can utilize just the lower 64 bits of the hash, allowing the compiler to generate only half the instructions. This optimization achieves throughput improvements of up to 7.94x compared to our naïve implementation and up to 9.87x faster than DataSketches.

Hash Function Unrolling. With previous optimizations in place, the hashing operation itself becomes the primary bottleneck. We optimized the 128-bit MurmurHash implementation by manually unrolling loops and eliminating code paths unnecessary for inputs smaller than 128 bits. As shown in Figure 2, these modifications yield up to 1.6x faster hashing operations. Notably, we observe no difference for `__int128_t` inputs, where the unrolling produces identical

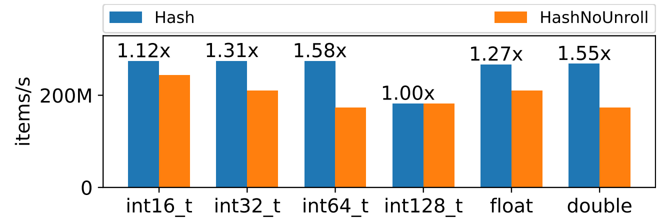


Figure 2: Hash Function Unrolling. Comparison of hashing throughput for the unrolled version of 128-bit MurmurHash versus the non-unrolled default implementation

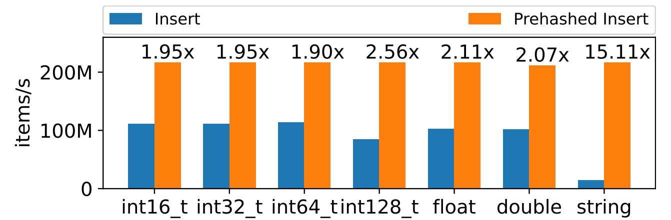


Figure 3: Count Sketch Prehashed Insertion. Comparison of insertion throughput for CS for prehashed values versus normal inserts

instructions. These hash function improvements further enhance our CS implementation, achieving throughput gains of up to 8.63x compared to naïve and up to 12.30x faster than DataSketches.

Prehashed Insertion. With hashing remaining the primary bottleneck, scenarios where hashes for values are already available offer significant performance opportunities. This optimization is particularly relevant in analytical database systems performing hash joins or hash aggregations, or in any system where values need to be inserted into multiple sketches requiring hash computations, such as when using both HyperLogLog [9] sketches and Count Sketches. By accepting pre-hashed values, our final CS implementation achieves substantial speedups that scale with input data size – up to 2.25x improvement for numeric types, and 13.24x improvement for strings of length 320.

Comparison with implementations from other papers. We compare our best implementation against available open-source implementations from other papers, with results shown in Figure 4. Our implementation achieves higher throughput, specifically: up to 1.92x over [35], up to 6.34x over [36], and up to 9.29x over [32], when not including strings, where the gap is even larger.

3 SpaceSaving (SS)

The SpaceSaving [22] sketch is a deterministic algorithm optimized for heavy hitter (frequent item) estimation in data streams. It maintains k counters, each storing an item-count pair, where k determines the space-accuracy trade-off. For each incoming item, the sketch increments its counter if the item is already monitored; otherwise, it replaces the item with the lowest count, assigning it that minimum count plus one. This replacement mechanism guarantees a maximum frequency estimation error of $\epsilon = N/k$, where N represents the stream size. The sketch guarantees retention of all items whose true frequency exceeds N/k, making it particularly effective for heavy hitter identification. It achieves $O(\frac{1}{\epsilon})$ space and update time complexity.

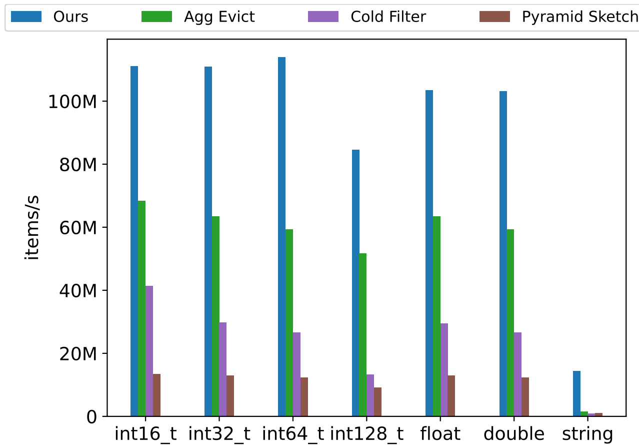


Figure 4: Count Sketch Insert vs. other papers. Comparison of insertion throughput for the CS implementations from [32, 35, 36] against ours

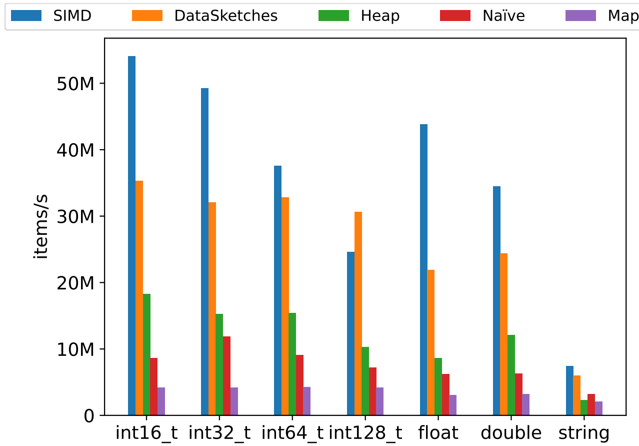


Figure 5: SpaceSaving Insert. Comparison of insertion throughput for the SS implementations for different data types

SpaceSaving represents a variant of the Misra-Gries (MG) [24] sketch, with both variants being readily convertible to each other [5].
Experiment Setup. We use the same experimental setup as described in Section 2. Figure 5 illustrates our performance comparison. The sketches are configured with $k = 96$, which ensures the inclusion of all elements with a frequency of approximately 1% or higher. For consistency, we employ the same 128-bit MurmurHash function used in previous experiments where hash computations are required.
Apache DataSketches. The DataSketches library implements a variant of the MG sketch. It features a specialized linear-probing hash map that efficiently manages memory while maintaining approximate item counts. The core structure tracks item-count pairs and incorporates a "reverse purge" mechanism for space management: When the map reaches capacity, it employs sampling to estimate the median frequency of stored items. This median value serves as a decrement factor applied across all counts, followed by a purge operation that removes items with zero or negative counts. This adaptive

```
size_t Find(const uint64_t& value) {
    size_t i = 0;
    const auto* data = values.data();
    const __m256i v = _mm256_set1_epi64x(value);
    for (; i + 32 <= K; i += 32) {
        __m256i x1 = _mm256_loadu_si256(
            reinterpret_cast<const __m256i*>(data+i));
        __m256i x2 = _mm256_loadu_si256(
            reinterpret_cast<const __m256i*>(data+i+4));
        // Analogous for x3, x4, x5, x6, x7, x8
        x1 = _mm256_cmpeq_epi64(x1, v);
        x2 = _mm256_cmpeq_epi64(x2, v);
        // Analogous for x3, x4, x5, x6, x7, x8
        uint64_t m = _mm256_movemask_pd(
            _mm256_castsi256_pd(x1));
        m |= _mm256_movemask_pd(
            _mm256_castsi256_pd(x2)) << 4;
        // Analogous for x3, x4, x5, x6, x7, x8
        if (m != 0) return i + __builtin_ctzll(m);
    }
    return i;
}
```

Figure 6: SIMD-optimized SpaceSaving Find function. AVX2-based parallel lookup that processes 32 values per iteration

compression approach preserves frequency estimates for significant items while gracefully degrading precision for less frequent ones.

The implementation has one constraint: the map size must be a power of two for efficient hashing operations, preventing direct k specification. We use a map size of 2^8, yielding $\epsilon \simeq 1/73$, slightly less precise than $1/96$. Increasing the map size to 2^9 for better accuracy ($\epsilon \simeq 1/146$) shows minimal impact on performance characteristics. For optimal insertion performance, we preallocate the hash map to its maximum size to avoid rehashing. This implementation demonstrates impressive baseline performance.
Naïve. This baseline implementation maintains two vectors: one for values and one for their corresponding counts. For each insertion, it performs a linear scan over the values vector. If the item is found, its count is incremented. Otherwise, the algorithm locates the element with the smallest count, replaces its value, and increments its count. This version is up to 4.17x slower than DataSketches.
Map. This implementation replaces the linear scan with a hash map to accelerate lookups, a strategy employed in some baseline implementations cited in the introduction. While theoretically reducing the complexity from $O(n)$ for linear scanning to $O(1)$ for hash map lookups, practical performance tells a different story. For smaller sketch sizes, the linear scan remains more efficient. Consequently, this version performs worse compared to our naïve implementation, with throughput up to 2.86x slower.
Heap. We employ a min heap data structure to efficiently identify the element with the smallest count, eliminating the need for a linear scan through counts. While this optimization achieves up to 2.12x higher throughput compared to our naïve implementation, DataSketches remains up to 3x faster.
SIMD. This implementation leverages SIMD (Single Instruction, Multiple Data) operations to speed up item lookups, accelerating the linear scan as shown in Figure 6. The comparison generates a bitmap, and, if non-zero, we use the count trailing zeros (ctz) instruction to identify the matching index.

```
1  size_t Find(const T& value, const uint64_t& hash) {
2    size_t i = 0;
3    const auto* data = hashes.data();
4    const __m256i v = _mm256_set1_epi64x(hash);
5    for (; i + 32 <= K; i += 32) {
6      uint64_t mask = // analogous to m in Figure 6
7      while (mask != 0) {
8        size_t j = i + __builtin_ctzll(mask);
9        if (LIKELY(values[j] == value)) return j;
10       mask &= mask - 1;
11     }
12   }
13   return i;
14 }
```

Figure 7: SIMD-accelerated hash-based lookup. AVX2-based parallel lookup using 64-bit hash comparisons, with efficient bitmap processing for potential match verification

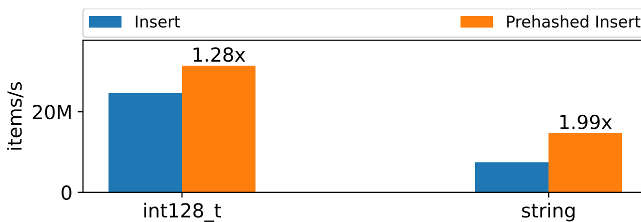


Figure 8: SpaceSaving Prehashed Insertion. Comparison of insertion throughput for SS for prehashed values versus normal inserts

Since SIMD operations are limited to 16, 32, or 64-bit types, we need a different approach for larger types like strings or 128-bit integers. We maintain 64-bit hashes alongside the actual values and perform the SIMD search on them, portrayed in Figure 7. When the hash comparison bitmap contains set bits, we verify potential matches against the actual values. Since hash collisions are expected to be rare, we use compiler hints to optimize for the likely case of a true equality comparison. We employ the technique from [18] for efficient bitmap iteration.

This optimization achieves up to 7x higher throughput compared to Naïve, and up to 2x faster than DataSketches. However, for 128-bit integers, DataSketches maintains a 1.25x performance advantage.

Similarly to Count Sketch, prehashed insertions provides additional speedups in case the hashes are already available: 1.12x for 128-bit integers and 1.99x for strings, as demonstrated in Figure 8. This enables us to outperform DataSketches for strings and be on par for 128-bit integers.

Comparison with implementations from other papers. We compare our best implementation against those from other papers, with results shown in Figure 9. Our implementation achieves higher throughput: up to 9.14x over [35], and 5.26x over [36].

4 Karnin-Lang-Liberty (KLL)

Karnin-Lang-Liberty [16] is a probabilistic sketch for computing approximate quantiles that offers practical improvements over deterministic approaches like Greenwald-Khanna (GK) [13]. KLL maintains a sequence of levels, each containing sorted items and operating at exponentially decreasing sampling rates. New items enter at the

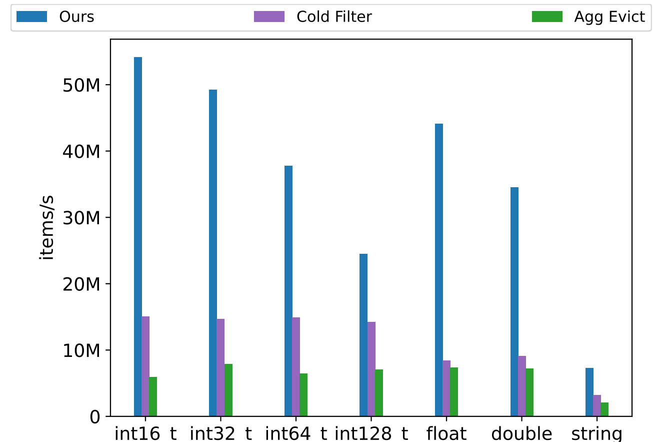


Figure 9: Space Saving Insert vs. other papers. Comparison of insertion throughput for the SS implementations from [35, 36] against ours

bottom level and are progressively merged upward through a randomized compaction process, which retains items with probability $\frac{1}{2}$ at each level transition. This mechanism ensures that higher levels contain increasingly sparse samples of the original stream. The optimized version of Ivkin et al. [15] reduces the worst case update time from $O(\frac{1}{\epsilon})$ down to $O(\log\frac{1}{\epsilon})$ while maintaining the same accuracy. The sketch parameter k sets the minimum level size. The sketch provides probabilistic accuracy guarantees, maintaining an error bound of ϵ with probability at least 1-δ while using $O(\frac{1}{\epsilon}\log\log(\frac{1}{\delta}))$ space. Unlike GK, KLL is fully mergeable, allowing sketches built from distinct subsets of the full dataset to be combined while maintaining the same accuracy guarantees as a sketch built over the entire dataset.

Experiment Setup. We use the same experimental setup as described in Section 2. Figure 10 presents our performance comparison. For our microbenchmark, we configure KLL with $k = 200$, yielding an average relative error ϵ of approximately 1.8%.

Naïve. Implementing KLL efficiently presents significant challenges. Our naïve implementation, based on the original paper [16], uses a vector of vectors to represent levels and performs compaction across all levels when additional space is required[2]. This approach results in extremely poor performance, being up to 446x slower than our most optimized implementation and up to 309.2x slower than the DataSketches implementation. Due to this large performance gap, we only use DataSketches as baseline in the coming paragraphs.

Apache DataSketches. The DataSketches implementation, widely adopted and highly optimized, is based on the efficient implementation strategies proposed by Ivkin et al. [15]. It employs a single packed contiguous array to store all levels, with level boundaries tracked using separate indices. This approach minimizes memory fragmentation and improves cache locality. The implementation uses a lazy compaction scheme, where compactions are triggered only when necessary, typically when inserting a new item would exceed the allocated space. Also, in contrast to the original version, levels may "overgrow", that is, they may exceed their space budget.

[2] A similar implementation can be found in the Android source code [11]

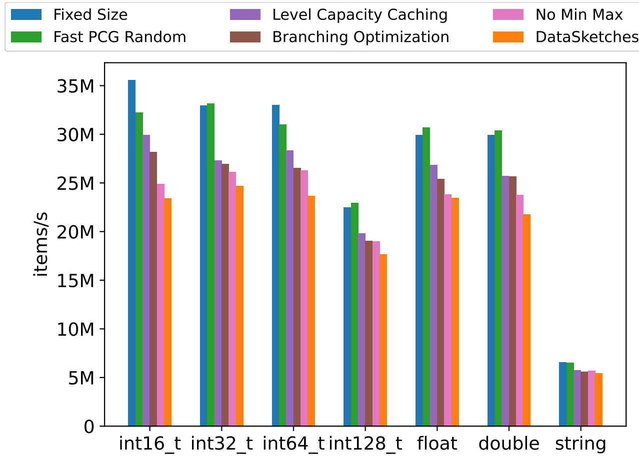


Figure 10: Karnin-Lang-Liberty Insert. Comparison of insertion throughput for the KLL implementations for different data types

Our implementations build upon this foundation, introducing further optimizations to enhance insertion throughput.

No Min Max Tracking. Tracking minimum and maximum values is not part of the KLL algorithm, and not used in approximating quantiles. By eliminating this non-essential feature, we achieve speedups of up to 1.11x compared to the DataSketches implementation.

Branching Optimizations. The DataSketches implementation is designed to be generic, supporting a wide range of data types. For non-fundamental types in C++, self-move operations[3] can lead to undefined behavior. To protect against this, the DataSketches implementation introduces numerous branches throughout the code. However, these safety checks are unnecessary for fundamental types. We implement a compile-time type check that allows the compiler to eliminate these branches for fundamental types, enabling unconditional overwrites. This optimization further increases our speedup over DataSketches, achieving up to 1.2x improvement in insertion throughput.

Level Capacity Caching. Performance profiling using `perf` revealed that the `level_capacity` function accounts for approximately 5% of the total insertion time due to its frequent invocation. These capacities are constant for a given sketch parameter k. By precomputing and caching these values, we eliminate redundant calculations during runtime. This optimization further enhances our performance, increasing the speedup over DataSketches to up to 1.28x.

Faster PCG Random. Profiling identified random number generation as the next significant bottleneck. To address this, we replaced the existing C++ standard library random number generator with PCG (Permuted Congruential Generator) [26], which offers both improved speed and higher quality randomness. This optimization further increases our speedup over DataSketches, achieving up to 1.4x improvement in insertion throughput.

Fixed Size / Memory Preallocation. The sketch typically grows its levels array somewhat frequently during initial insertions when the sketch is still small. We found that preallocating the maximum amount of memory the sketch would ever need is not excessively wasteful. For $k = 200$, the array size can never exceed 985 elements, as larger sizes would allow for more than 2^{64} insertions, exceeding the capacity of 64-bit counters. Even though we always allocate 985 elements (~8kb for 64-bit integers), the actual memory requirements are lower: a sketch handling 1 million insertions uses 617 elements (wasting only 37% of the allocated space), while one processing 10 billion insertions uses 649 elements (34% unused space). If a tighter bound on the number of insertions than 2^{64} is known in advance, the array size can be reduced accordingly. Table 1 details how the array size grows in relation to the number of levels and insertion count.

[3] https://en.cppreference.com/w/cpp/utility/move

Table 1: KLL Sketch Size and Capacity. This table shows the growth of the KLL sketch's array size and insertion capacity for selected levels as the number of levels increases, for $k = 200$

Number of levels	Array Size	Inserts	Number of levels	Array Size	Inserts
1	200	$200 = \sim 2^{7.64}$	10	593	$\sim 2^{17.23}$
2	333	$533 = \sim 2^{9.06}$	20	673	$\sim 2^{27.23}$
3	422	$1155 = \sim 2^{10.17}$	30	753	$\sim 2^{37.23}$
4	481	$2369 = \sim 2^{11.21}$	40	833	$\sim 2^{47.23}$
5	521	$4778 = \sim 2^{12.22}$	59	985	$\sim 2^{63.46}$

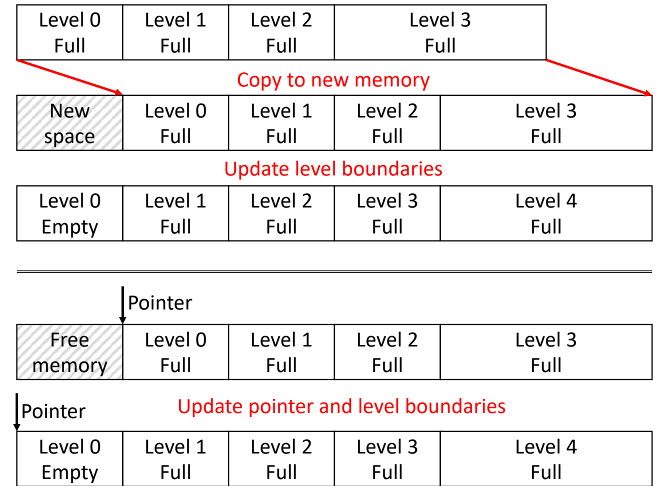


Figure 11: Adding a new level to a completely full KLL sketch. Comparison of level addition strategies for KLL sketch implementations using a packed contiguous array. **Top:** Original approach from [15] – When full, allocate a larger array to accommodate one additional level, copy existing data, and update level boundaries. **Bottom:** Our fixed-size implementation – Pre-allocated space eliminates memory allocation and data copying. Adding a level only requires updating the Level 0 pointer and level boundaries. No copying/moving of data or allocations are necessary.

This approach aligns perfectly with the optimized algorithm from [15], which grows the array from the "front" by allocating more memory and shifting elements backward. By preallocating, we eliminate the branching that checks for growth necessity, the costly element shifting/copying operations and memory allocations. We illustrate this optimization in Figure 11. Instead of growing and shifting, we maintain a pointer to the start of level 0 and simply update

this pointer when growth would normally occur. For 1 million insertions, this eliminates 13 growth operations, while for 10 billion insertions, it eliminates 17 growths. This optimization further increases our speedup over DataSketches to up to 1.52x – a significant improvement considering the already highly optimized nature of the DataSketches implementation.

5 Discussion

Thread Safety. In this work, we defer synchronization between multiple threads to the user, aligning with the approach that Apache DataSketches takes. This design choice is particularly beneficial for scenarios like analytical processing, where data can be naturally partitioned into disjoint subsets for parallel processing, followed by a final merge operation. Since all sketches presented in this work support full mergeability, we avoid imposing unnecessary synchronization overhead during insertions, allowing users to implement concurrency control strategies that best suit their specific requirements.
Sketch Parameters. The choice of sketch parameters influences performance characteristics. Generally, there exists a trade-off between accuracy (or error probability) and insertion speed: higher accuracy demands come at the cost of slower insertions. For both SS and KLL sketches, variations in their respective parameter k are expected to affect all implementations uniformly. Our selected parameters result in memory footprints of approximately 1KB for SS and 4KB for KLL sketches, both comfortably fitting within the L1 cache (32KB). Should these sizes exceed the L1 cache capacity, we anticipate performance degradation. The CS sketch, with its current 82KB size, resides in L2 cache; modifying its parameters to either fit in L1 or exceed L2 would alter performance metrics, though we expect such changes to impact all implementations proportionally.
Hash Function Choice. For consistency with Apache DataSketches and fair comparisons, we employ 128-bit MurmurHash3 as our hash function. While faster alternatives exist, such optimizations would uniformly benefit all implementations. Our key contributions to CS regarding range reduction and avoiding computation of more hashes than are necessary are hash function-agnostic and can be applied universally. In scenarios requiring only 64 hash bits, performance could be further enhanced by utilizing at 64 bit hash function such as MurmurHash2, which might reduce the computational overhead associated with generating additional unneeded bits.
More Modern Compilers. The compiler used in the benchmarks is not the latest version. We conducted additional experiments using the freshly released GCC version 15.1 on identical `m5.12xlarge` hardware. While performance differences between sketch implementations remain significant, the gaps narrow with the modern compiler, yielding several interesting insights. The newer compiler benefits all Count Sketch implementations, notably through automatic hash function unrolling without explicit implementation guidance. Our manually unrolled code for floating-point types now exhibits lower performance than the compiler-optimized version, primarily due to our additional logic ensuring consistent hash values for +0 and −0 which, despite being equal, have different binary encodings. For SpaceSaving, the modern compiler significantly slows down our handwritten SIMD code, showing performance degradation across all data types except 16- and 128-bit integers and strings. This implementation no longer consistently outperforms DataSketches, instead showing comparable performance. The heap-based SS variant, however, sees substantial improvements, achieving up to 3x speedup for floating-point types when compared to the older compiler version, likely due to auto-vectorization of the linear scan. For 16-, 32-, and 64-bit integers, performance now matches both the SIMD and DataSketches implementations. For KLL, the new compiler also speeds up all implementations across the board slightly. Examining the KLL sketch assembly code reveals that while the newer compiler still retains the redundant self-move protection check, our optimization continues to yield performance benefits.
ARM Processors. To evaluate the architecture dependence of our optimizations, we conducted additional experiments on an `r7g.16xlarge` AWS instance equipped with 64 ARM-based AWS Graviton3 CPUs @ 2.1 GHz running Amazon Linux 2. While the performance gaps between different implementations narrow on ARM architecture, they remain significant – there are some exceptions, however. The Count Sketch optimizations show consistent relative performance improvements across architectures. For SpaceSaving, we implemented ARM-specific SIMD code using Neon instructions to replace the x86 AVX code. On ARM, the heap-based variant demonstrates stronger relative performance compared to other implementations than on x86. DataSketches outperforms our SIMD implementation for 64-bit, 128-bit integers, and doubles. The KLL sketch optimizations remain effective but show more modest gains, with our best implementation achieving up to 1.15x speedup over DataSketches.

Using GCC 15.1 on ARM produces effects similar to those observed on x86. Notably, the newer compiler negatively impacts handwritten SIMD performance, with DataSketches outperforming our SIMD implementation across all data types. In some cases, our SIMD implementation even falls behind the paper implementations, while DataSketches consistently maintains superior performance, making it a more reliable baseline for ARM architectures.

6 Conclusion

In this paper we have presented highly insert-optimized implementations of three fundamental data sketching algorithms: Count Sketch, SpaceSaving, and Karnin-Lang-Liberty. Through careful engineering and novel implementation strategies, we achieved substantial performance improvements over naïve, Apache DataSketches, and other papers' implementations. Our optimizations include fast hash range reduction and hash sharing for CS, SIMD vectorization for SS, and memory preallocation and branching improvements for KLL.

These improvements demonstrate that significant performance gains are possible through implementation optimization, even for well-established algorithms. Our work establishes new performance baselines and shows that classical sketching algorithms can deliver exceptional performance without sacrificing their theoretical guarantees. By open-sourcing our implementations, we hope to contribute to the advancement of high-performance data sketching.

Acknowledgments

We extend our thanks to Magnus Müller, Mohammed Al-Kateb, and Roger Kim all of whom played a part in making this work possible. We also thank the anonymous reviewers for their valuable feedback.

References

[1] Amazon Web Services, Inc. or its affiliates. [n. d.]. *HyperLogLog sketches*. Retrieved Jan 14, 2025 from https://docs.aws.amazon.com/redshift/latest/dg/hyperloglog-overview.html

[2] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185.

[3] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.

[4] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[5] Graham Cormode and Ke Yi. 2020. *Small summaries for big data*. Cambridge University Press.

[6] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. 2009. Cots: A scalable framework for parallelizing frequency counting over data streams. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1323–1326.

[7] Otmar Ertl. 2023. UltraLogLog: a practical and more space-efficient alternative to HyperLogLog for approximate distinct counting. *arXiv preprint arXiv:2308.16862* (2023).

[8] Otmar Ertl. 2024. ExaLogLog: Space-Efficient and Practical Approximate Distinct Counting up to the Exa-Scale. *arXiv preprint arXiv:2402.13726* (2024).

[9] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science* Proceedings (2007).

[10] Michael Freitag and Thomas Neumann. 2019. Every row counts: Combining sketches and sampling for accurate group-by result estimates. *ratio* 1 (2019), 1–39.

[11] Google. [n. d.]. *Android Code Search - KLL implementation*. https://cs.android.com/android/platform/superproject/main/+/main:packages/modules/StatsD/lib/libkll/include/kll.h.

[12] Google. 2025. *Google Benchmark*. https://github.com/google/benchmark.

[13] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record* 30, 2 (2001), 58–66.

[14] He Huang, Jiakun Yu, Yang Du, Jia Liu, Haipeng Dai, and Yu-E Sun. 2023. Memory-efficient and flexible detection of heavy hitters in high-speed networks. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–24.

[15] Nikita Ivkin, Edo Liberty, Kevin Lang, Zohar Karnin, and Vladimir Braverman. 2019. Streaming quantiles algorithms with small space and update time. *arXiv preprint arXiv:1907.00236* (2019).

[16] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, 71–78.

[17] Daniel Lemire. 2016. *A fast alternative to the modulo reduction*. http://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/.

[18] Daniel Lemire. 2018. *Iterating over set bits quickly*. https://lemire.me/blog/2018/02/21/iterating-over-set-bits-quickly/.

[19] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. 2022. Stingy sketch: a sketch framework for accurate and fast frequency estimation. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1426–1438.

[20] Charles Masson, Jee E Rim, and Homin K Lee. 2019. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *arXiv preprint arXiv:1908.10693* (2019).

[21] Dimitrios Melissourgos, Haibo Wang, Shigang Chen, Chaoyi Ma, and Shiping Chen. 2023. Single Update Sketch with Variable Counter Structure. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4296–4309.

[22] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2006. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems (TODS)* 31, 3 (2006), 1095–1133.

[23] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*. Springer, 398–412.

[24] Jayadev Misra and David Gries. 1982. Finding repeated elements. *Science of computer programming* 2, 2 (1982), 143–152.

[25] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[26] Melissa E. O'Neill. 2014. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Technical Report HMC-CS-2014-0905. Harvey Mudd College, Claremont, CA.

[27] 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*. 1449–1463.

[28] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. 164–176.

[29] Daniel Ting. 2018. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*. 1129–1140.

[30] Pinghui Wang, Yitong Liu, Zhicheng Li, and Rundong Li. 2024. An LDP Compatible Sketch for Securely Approximating Set Intersection Cardinalities. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[31] 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*. 561–575.

[32] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. 2017. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1442–1453.

[33] Fuheng Zhao, Divyakant Agrawal, Amr El Abbadi, and Ahmed Metwally. 2021. SpaceSaving±: An Optimal Algorithm for Frequency Estimation and Frequent items in the Bounded Deletion Model. *arXiv preprint arXiv:2112.03462* (2021).

[34] Fuheng Zhao, Sujaya Maiyya, Ryan Wiener, Divyakant Agrawal, and Amr El Abbadi. 2021. KLL± Approximate Qantile Sketches over Dynamic Datasets. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1215–1227.

[35] Yang Zhou, Omid Alipourfard, Minlan Yu, and Tong Yang. 2018. Accelerating network measurement in software. *ACM SIGCOMM Computer Communication Review* 48, 3 (2018), 2–12.

[36] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. 2018. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*. 741–756.