Cooled-KLL: Enhancing Quantile Estimation by Filtering Hot Item

Anonymous Author(s)

Abstract

Quantile estimation in data streams is essential for applications such as database management and network traffic monitoring. Quantile sketches, which are probabilistic algorithms for estimating quantiles, have seen extensive use in practical scenarios. Among these, the KLL sketch, introduced in 2016, is acclaimed for its theoretically space-optimal solution. However, it overlooks the repetition of numbers within the data stream. In real-world applications, data streams are often highly skewed, with a few items appearing frequently, known as hot items. KLL processes these items without considering their frequency, leading to suboptimal space utilization due to repeated insertion and storage. To address this limitation, we propose Cooled-KLL, an enhanced version of the KLL sketch. Cooled-KLL introduces a structure called the Hot Filter, which filters out hot items and stores them as key-value pairs, allowing only cold items to be processed by the subsequent KLL. This method reduces memory usage with only a slight decrease in processing speed. Extensive experiments demonstrate that Cooled-KLL outperforms four other algorithms and improves accuracy by up to 1.5 orders of magnitude compared to the optimal KLL.

Keywords

Quantile estimation; Sketch; KLL; Hot items;

1 Introduction

Quantile estimation [13, 15, 17, 21, 36, 41, 51, 55] is a crucial task in big data mining. Formally, given a data stream $S = \{x_1, x_2, \ldots, x_N\}$ where items are numbers, the rank of an item x in S is denoted as $R(x, S)$, representing the number of items such that $x_i \leq x$. The δ-quantile of the data stream S is defined as the item with rank $\lceil \delta \cdot N \rceil$, denoted as $Q(\delta, S) = x$ such that $R(x, S) = \lceil \delta \cdot N \rceil$, where $\lceil \cdot \rceil$ is the ceiling function, and δ is within the interval $(0, 1]$. For example, in financial analysis, the median (0.5-quantile) of a stream of stock prices provides a measure of central tendency, helping analysts understand typical market behavior. In network traffic monitoring [16, 18, 23, 25, 56], estimating the 0.95-quantile of packet sizes can help identify the threshold beyond which only the largest packets exist, which is useful for anomaly detection and capacity planning. In web services [8, 22, 35, 40], the 0.99-quantile of response times ensures that the vast majority of user requests are handled within an acceptable time frame, thus maintaining quality of service.

In the big data era, the speed and volume of data streams make accurate item tracking challenging [12, 19, 26, 27, 42, 43, 50, 52]. Advanced algorithms that operate on individual items require fast, consistent updates, which necessitate avoiding slow DRAM in favor of limited-capacity SRAM [38, 39, 44, 46–48, 53]. Therefore, sketch-based algorithms are widely used for quantile estimation due to their effective space and time complexity, as well as acceptable error margins [11, 24, 30–33, 49]. These algorithms are referred to as "quantile sketches." In this paper's quantile estimation task, we focus on all-quantile estimation. Specifically, for a data stream S, we aim to provide an estimated rank $\hat{R}(x, S)$ for any value x, and an estimated $\hat{Q}(\delta, S)$ for any $\delta \in (0, 1]$, with the goal of achieving the highest possible accuracy.

Many quantile sketches have been proposed in recent years. The classic *GK-Sketch* [20] employs a summary data structure to efficiently approximate quantiles within a data stream, balancing memory usage with error bounds through intelligent merging of sampled points. Subsequent algorithms, such as the *t-digest* [14] based on clustered sampling, the *DD-Sketch* [35] focusing on tail quantile estimation, and the *Req-Sketch* [9, 10] based on multi-layer compactor sampling, are also commonly used in various fields.

In 2016, the compactor-based KLL sketch [28] was introduced, boasting theoretical optimality in space complexity for quantile estimation. The structure of KLL involves an exponentially increasing compactor that initially compresses data lightly to preserve accuracy, thus requiring minimal space to control error margins. The results in the experiments paper [17] also verify its superiority. In general, **KLL has the best theoretical and experimental results at present.** However, KLL's approach to treating each input independently—storing repeated instances of the same data item without aggregating their occurrences—poses a challenge. This method becomes particularly inefficient in the presence of skewed data distributions common in real-world applications [5–7, 29, 34, 37, 41, 45, 54], where certain data items, or "hot items," appear with high frequency. For instance, an item x appearing 30 times in a stream would be individually recorded in 30 instances rather than as a more space-efficient key-value pair $\langle x, 30 \rangle$. This raises the question: ***Can we enhance KLL to handle highly skewed data streams more effectively without sacrificing performance?***

Optimizing KLL directly is improbable, as its structure and code have already been fine-tuned extensively by previous researchers. Instead, we propose a new quantile sketch called **Cooled-KLL**, which integrates a novel component, the **Hot Filter**, to intercept hot items before they enter the KLL, thereby achieving the purpose of "**cooling**" the **KLL**. This work makes the following contributions:

(1) We designed an effective structure for Hot Filter so that it can effectively segregate hot items from the incoming data stream by storing them as key-value pairs. This approach not only conserves space but also enhances accuracy without impacting the processing speed of the standard KLL (§3).
(2) We provide a rigorous error bound for Cooled-KLL and theoretically prove its superior accuracy over the standard KLL sketch in highly skewed distributions (§4).
(3) We implement Cooled-KLL on the CPU and perform the all-quantile estimation task to evaluate its real-world performance. Our results indicate that Cooled-KLL significantly outperforms the current state-of-the-art, leading four other algorithms in terms of both average ranking error and quantile error, and showcases high throughput. In particular, Cooled-KLL achieves a 1.5 orders of magnitude accuracy advantage in real-world highly skewed datasets (§5).

∗ The source code is in the anonymous repository on GitHub [2].

The rest of the paper is organized as follows: Section 2 provides an overview of the background and state-of-the-art (SOTA) algorithms for quantile estimation. Section 3 details Cooled-KLL's data structure and algorithm. Section 4 offers a comprehensive mathematical analysis, and Section 5 presents the experimental results. Finally, Section 6 summarizes the paper. For reproducibility and further research, we have released our source code on Github [2].

2 Background

This section formally defines our target problem and then introduces the state-of-the-art quantile sketch: the KLL Sketch.

2.1 Problem Statement

DEFINITION 1: Data Stream Model.
In a data stream $S = \{x_1, x_2, \cdots, x_N\}$ (all sets mentioned in this section are multisets), we handle a continuous arrival of N data items, each appearing as a real number, where the value represents the metric we aim to process (e.g., network packet delay).
DEFINITION 2: Rank.
The rank of an item x in a data stream S is $R(x, S)$, which is the number of items in S that are $\leq x$. Formally, $R(x, S) = |\{x_i \mid x_i \in S \wedge x_i \leq x\}|$.
DEFINITION 3: δ-Quantile.
The δ-quantile of a data stream S is $Q(\delta, S)$, which is the item with rank $\lceil \delta \cdot N \rceil$. Formally, $Q(\delta, S) = x$ (where $x \in S \wedge R(x, S) = \lceil \delta \cdot N \rceil$).
DEFINITION 4: All-quantile Estimation.
Including the rank and δ-quantile estimation. For a data stream S, the algorithm must provide its estimated rank $\hat{R}(x, S)$ given a item x, and its estimated δ-quantile $\hat{Q}(\delta, S)$ given a $\delta \in (0, 1]$.

2.2 Related Work—Quantile Sketch

Quantile sketches for solving quantile estimation have a rich history of development, featuring significant contributions such as the GK-Sketch [20], DD-Sketch [35], Req-Sketch [9], and t-digest. We briefly introduce them here.

2.2.1 GK-Sketch. When selecting items to drop from a sketch, GK-Sketch tracks detailed information about each stored item, which achieves a space complexity of $O(\epsilon^{-1} \log \epsilon n)$. The GK algorithm is one of the most well-known deterministic algorithms for quantile approximation. It ensures that the error in quantile estimates is tightly bound, making it a robust choice when precise quantile tracking is necessary.

While GK is optimal regarding deterministic guarantees, its operational approach is also relatively idiosyncratic. Unlike probabilistic or heuristic-based methods that might trade off strict accuracy for speed or memory efficiency, GK focuses on maintaining precise control over the sketch's error bounds by meticulously tracking each item's rank and positional data within the stream. This approach can lead to higher computational overhead and slower performance, particularly in high-frequency data stream applications.

2.2.2 DD-Sketch: DD-Sketch is designed to estimate value distributions by partitioning the value range into segments, each monitored by a counter for values within the segment. The segment boundaries are determined by $\gamma := \frac{(1+\alpha)}{(1-\alpha)}$, with each segment's counter, C_i, tallying values x in the range $\gamma^{i-1} < x \leq \gamma^i$. The insertion of a value x is indexed by $\lceil \log_\gamma(x) \rceil$. DD-Sketch approximates values in a segment by $\hat{x} = \frac{2\gamma^i}{\gamma+1}$, maintaining a relative error within α. The trade-off between the range coverage and accuracy is governed by α: a higher α extends the range but reduces accuracy. To estimate the value at a certain percentile $p \in [0, 1]$, we sum counters up to the relevant segment i and use $\hat{x} = \frac{2\gamma^i}{\gamma+1}$ as the quantile estimate.

Algorithm 1: *KLL.Insert*(x)

Input: An item x.
1 $C[0] \leftarrow C[0] \cup \{x\}$;
2 *KLL.Compaction*();

2.2.3 Req-Sketch: Req-Sketch employs multiple levels of compactors to store item *values*, utilizing $O(\log N)$ compactors for a flow size N, each with a buffer of similar size. Items enter at level 0, and upon reaching a compactor's capacity, a sorted even-sized subset is compacted. Half its elements are elevated to the next level, preserving total item weight due to the weighting scheme where items at level h are assigned a weight of 2^h. This mechanism, known as compaction, ensures the integrity of distribution estimates. To estimate a value at a certain percentile p, item weights are cumulatively tallied from the smallest value until reaching p, with the corresponding item's value serving as the estimate.

2.3 Related Work—KLL Sketch

Quantile sketches for solving quantile estimation have a rich history of development, featuring significant contributions such as the GK-Sketch [20], DD-Sketch [35], Req-Sketch [9], and t-digest. In 2016, Karnin et al. introduced the KLL Sketch, which they demonstrated to be theoretically optimal in terms of space complexity for quantile estimation. Our proposed Cooled-KLL builds upon the KLL Sketch, so we will provide a detailed introduction below.

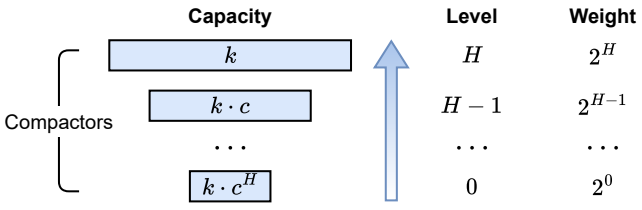


Figure 1: Data Structure of KLL Sketch.

2.3.1 Data Structure. As illustrated in Fig. 1, the KLL Sketch consists of a series of compactors C (essentially arrays) with varying capacities. Compactor $C[h]$ resides at level h, with all items in $C[h]$ carrying weights of 2^h. Essentially, each item in $C[h]$ represents 2^h original input items. The capacity of compactor $C[h]$ is defined as $k \cdot c^{H-h}$ (but at least 2), where H is the highest level containing a non-empty compactor, k is a value set in advance based on the desired accuracy, and $c \in (0.5, 1)$ is a constant. Notably, the compactor capacities diminish as we move down the levels.

2.3.2 Insert(x): The pseudo-code is presented in Algorithm 1. To insert an item x, it is initially added to $C[0]$. At this stage, no information is lost. When $C[0]$ reaches its capacity, a compaction operation is performed (as shown in Algorithm 2), which randomly transfers half of the items from $C[0]$ to $C[1]$ and resets $C[0]$ to empty. This may cause $C[1]$ to reach or exceed its capacity, triggering a series of compaction operations. Finally, if $C[H]$ also reaches capacity, $C[H+1]$ will be created, thereby increasing the maximum height by one.

Algorithm 2: *KLL.Compaction()*

```
for h = 0 to H do
    if |C[h]| ≥ max(k · c^{H−h}, 2) then
        Sort(C[h]);
        coin ← random_bit();
        if coin is 0 then
            C[h+1] ← C[h+1] ∪ {even-index items in C[h]};
        else
            C[h+1] ← C[h+1] ∪ {odd-index items in C[h]};
        Clear C[h];
if C[H + 1] has been created then
    H ← H + 1;
```

2.3.3 Query: KLL can provide two types of queries: $Rank(x)$ and $Quantile(\delta)$, which we will introduce below.

• $Rank(x)$ estimates the rank of item x in the data stream. The query is represented by the following equation:

$$Rank(x) = \sum_{h=0}^{H} \left(R(x, C[h]) \cdot 2^h \right)$$

In the summation, $R(x, C[h])$ represents the rank of item x in the h-th compactor, i.e., the number of items less than or equal to x; 2^h is the weight of the h-th compactor.

• $Quantile(\delta)$ estimates the δ-quantile of the data stream. First, all data from the 0-th to H-th compactors are sorted to form a data array D and a weight array W, where $W[i]$ is the cumulative weight of the item $D[i]$. The estimated value of $Quantile(\delta)$ is $D[j]$, where j satisfies the following condition:

$$j = \arg\max_j \left(\sum_{i=0}^{j} W[i], \sum_{i=0}^{j} W[i] \le \lceil \delta \cdot N \rceil \right)$$

2.3.4 Running Example: Here, we illustrate a running example of KLL insertion, assuming $k = 4$ and $c = \frac{1}{\sqrt{2}}$. As depicted in Fig. 2, we want to perform $Insert(34)$. We first insert 34 into $C[0]$, which will result in $C[0] = \{16, 34\}$, triggering a compaction. Assuming we opt to move the even-index items up, then $C[0] = \emptyset$ and $C[1] = \{34, 92, 92\}$, which will subsequently trigger a series of compactions:

- Compress $C[1]$: Suppose we choose the odd-index items. Then $C[1] = \emptyset$, $C[2] = \{23, 34, 81, 81, 92\}$.
- Compress $C[2]$: Suppose we choose the even-index items. Then $C[2] = \emptyset$, $C[3]$ is created and $C[3] = \{34, 81\}$.

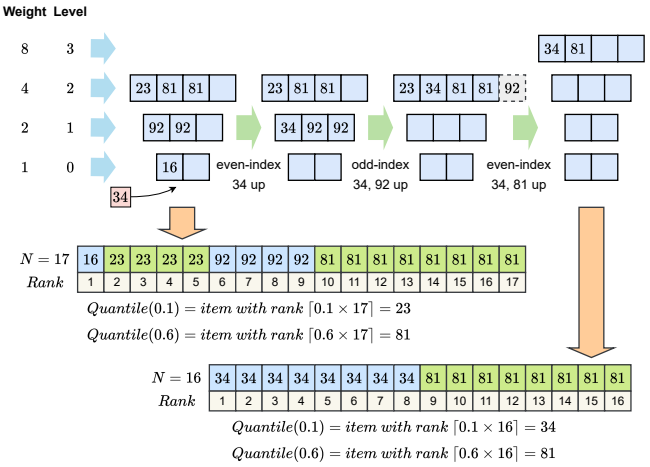


Figure 2: A running example of KLL.

Next, we introduce the query steps of KLL, as shown in the lower part of Fig. 2. Before insertion, if a query is needed, we take all the stored numbers in KLL, accumulate their frequencies, arrange them into an array, and sort them (with the index representing the rank). For example, the number "23" appears once in C[2], so its frequency is 1×2^2. We can see that there are a total of 17 numbers. If we query the 0.1-quantile, we return the number with the rank $\lceil 0.1 \times 17 \rceil = 2$, which is "23." The same method applies for querying the 0.6-quantile. After the insertion, if a query is needed, we arrange all the numbers in KLL into an array in the same way. The methods for querying rank and quantile are similar.

2.4 Motivation – Potential Weaknesses of KLL

From the previous sections, we can intuitively observe that the core mechanism of KLL involves using multiple layers of compactors to perform $\frac{1}{2}$ sampling on the data and assign a weight of 2^n (which can also be viewed as frequency) to the sampled items. For instance, on the right side of Fig. 2, KLL treats the data stream at this point as consisting of eight "34"s and eight "81"s. We can identify the following facts.

FACT 1: Compaction Will Introduce Errors. In Fig. 2, we observe that the KLL structure initially contained four items: "16", "23", "81", and "92", with weights of 1, 4, 8, and 4, respectively. After "34" was inserted, a series of compaction operations were triggered, and the final structure only contained two items: "34" and "81", with weights of 8 and 8, respectively. The information about items "16", "23", and "92" was completely lost, and the weight of item "34" was significantly overestimated (the correct value was 1). This greatly increased the error of KLL. Therefore, in order to improve accuracy, we aim to minimize the frequency of KLL's compaction.

FACT 2: Hot Items Will Cause More Compaction. Referring to the left side of Fig. 2, we notice some repeated items. For instance, "81" appears twice in $C[2]$, indicating it appears at least $4 + 4 = 8$ times in the original data stream. We can infer that its insertion must have triggered several compactions. If there is an item that appears very frequently in the data stream (referred to as a hot item), the number of compactions triggered by it will also be substantial.

These facts highlight that if we can devise a method to filter out the hot items in the data stream beforehand and prevent them from

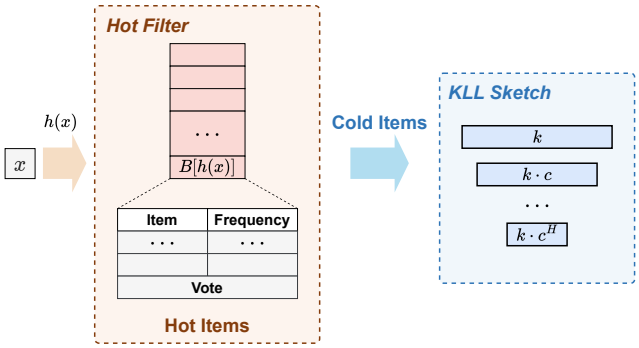


Figure 3: Data Structure of Cooled-KLL.

entering the KLL, the number of compaction operations will be greatly reduced, thereby improving the accuracy of the KLL. This realization forms the basis for our proposed Cooled-KLL.

3 Cooled-KLL Framework

In this section, we describe the data structure and algorithm of Cooled-KLL. As mentioned earlier, a quantile sketch algorithm should provide three fundamental interfaces: $Insert(x)$, $Rank(x)$, and $Quantile(\delta)$ to support all-quantile estimation for a data stream.

3.1 Data Structure

As depicted in Fig. 3, our Cooled-KLL framework consists of two main components: the Hot Filter and the standard KLL. The Hot Filter comprises a list of w buckets, with each item mapped to a corresponding bucket using the hash function $h(\cdot)$. Each bucket contains E entries and a $vote$ field. Each entry records an item and its frequency. The KLL part is used to process items coming from Hot Filter.

3.2 Algorithm and Operations

3.2.1 Insert(x): The pseudo-code is shown in Algorithm 3. When an item x arrives, we first use the hash function to locate it in bucket $B[h(x)]$. We then check all entries in $B[h(x)]$, encountering one of the following **four cases**:

Case 1: If x already exists in the bucket, we increment its frequency by 1.

Case 2: If x is not in the bucket, but there is an empty entry in the bucket, we simply place $\langle x, 1\rangle$ into this entry.

Case 3: If x is not in the bucket and there is no empty entry, we increment $vote$ by 1. Assume the entry with the smallest frequency in the bucket is $\langle x_{min}, f_{min}\rangle$. If $\frac{vote}{f_{min}} < \lambda$, we insert $\langle x, 1\rangle$ into the subsequent KLL.

Case 4: Continuing from Case 3, if $\frac{vote}{f_{min}} \geq \lambda$, we replace $\langle x_{min}, f_{min}\rangle$ with $\langle x, 1\rangle$ and reset $vote$ to 0. We then evict $\langle x_{min}, f_{min}\rangle$ and insert it into the subsequent KLL. We call λ the "eviction threshold."

3.2.2 Query. Like the ordinary KLL, our Cooled-KLL also provides two types of queries: $Rank(x)$ and $Quantile(\delta)$.

• $Rank(x)$: We simply sum the ranking of x in both the Hot Filter and KLL. Formally:

$$Rank(x) = \sum_{y \in \text{Hot Filter} \wedge y \leq x} (f_y) + \text{KLL.Rank}(x)$$

Algorithm 3: *Cooled-KLL.Insert*(x)

Input: An item x.

Assume $B[m][n]$ is the n-th entry in the m-th bucket, $\langle x_{min}, f_{min}\rangle$ is the least frequency entry in $B[h(x)]$;
$h(x) \in [1, w]$; $1 \leq j \leq E$; Eviction threshold λ;
if *x is in* $B[h(x)][j]$ **then**
  $B[h(x)][j].frequency++$; ▷ **Case 1**
else if *there is an empty entry* $B[h(x)][j]$ **then**
  put $\langle x, 1\rangle$ into $B[h(x)][j]$; ▷ **Case 2**
else
  $B[h(x)].vote++$;
  if $\frac{vote}{f_{min}} < \lambda$ **then**
    $KLL.Insert(x)$; ▷ **Case 3**
  else
    Replace $\langle x_{min}, f_{min}\rangle$ with $\langle x, 1\rangle$; ▷ **Case 4**
    $B[h(x)].vote = 0$;
    $KLL.Insert(\langle x_{min}, f_{min}\rangle)$;

where f_y refers to the frequency of y recorded in the Hot Filter. We need to traverse the items in the Hot Filter to get f_y.

• $Quantile(\delta)$: Similar to the original KLL, since we record the frequency of items in the Hot Filter (i.e., the weight in KLL), we only need to combine the items in the Hot Filter with those in the KLL to generate the data array D and the weight array W. If an item appears in both the Hot Filter and KLL, we simply sum their weights. The remaining operations are the same as in Section 2.3.3, and the formal equations are omitted here.

3.3 Example of Insertion in Hot Filter

Here, we demonstrate an insertion example of the Hot Filter in Cooled-KLL. As shown in Fig. 4, assume $\lambda = 4$. ***(1) To insert*** x_1, we use the hash function to locate its corresponding bucket $B[h(x_1)]$ and find x_1 in it, so we increment its frequency. ***(2) To insert*** x_4, we locate $B[h(x_4)]$, where x_4 is not found, but there is an empty entry, so we place $\langle x_4, 1\rangle$ in. ***(3) To insert*** x_9, we locate $B[h(x_9)]$, where x_9 is not found and the bucket is full, so we increment $vote$. At this point, we compare $\frac{vote}{f_{min}}$ and λ. Since $\frac{vote}{f_{min}} = \frac{33}{11} = 3 < \lambda$, x_5 cannot be evicted, and we insert x_9 into KLL. ***(4) To insert*** x_{10}, we locate $B[h(x_{10})]$, and similarly, x_{10} is not found and the bucket is full. We increment $vote$, and now $\frac{vote}{f_{min}} = \frac{48}{12} = 4 \geq \lambda$. Thus, we reset $vote$ to 0, evict x_7, insert $\langle x_7, 12\rangle$ into KLL, and place $\langle x_{10}, 1\rangle$ into entry.

3.4 Optimization: Binary Split Insertion

In Case 4 of Fig. 4, we need to insert $\langle x_7, 12\rangle$ into the subsequent KLL. A straightforward method is to execute $KLL.Insert(x_7)$ twelve times, that is, insert twelve x_7 into $C[0]$ of KLL sequentially, which tends to trigger compaction. As shown in Fig. 6, we found a more efficient way to insert $\langle x_7, 12\rangle$: since the weight of each layer of KLL increases exponentially by two, we can decompose 12 into $2^3 + 2^2$ (i.e., binary split), thereby directly inserting x_7 once into $C[3]$ and once into $C[2]$. This optimization significantly reduces the likelihood of triggering compaction.

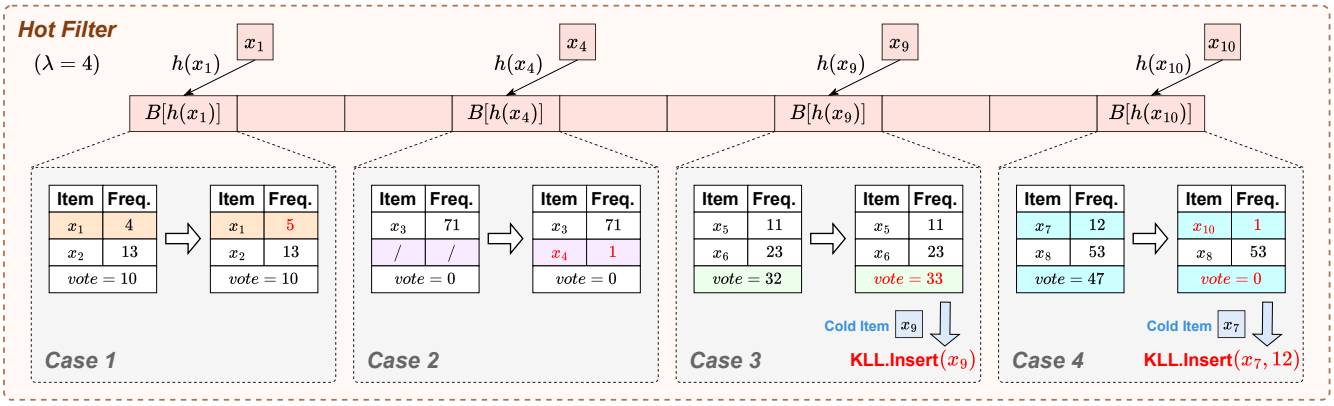


Figure 4: A running example of the Hot Filter in Cooled-KLL.

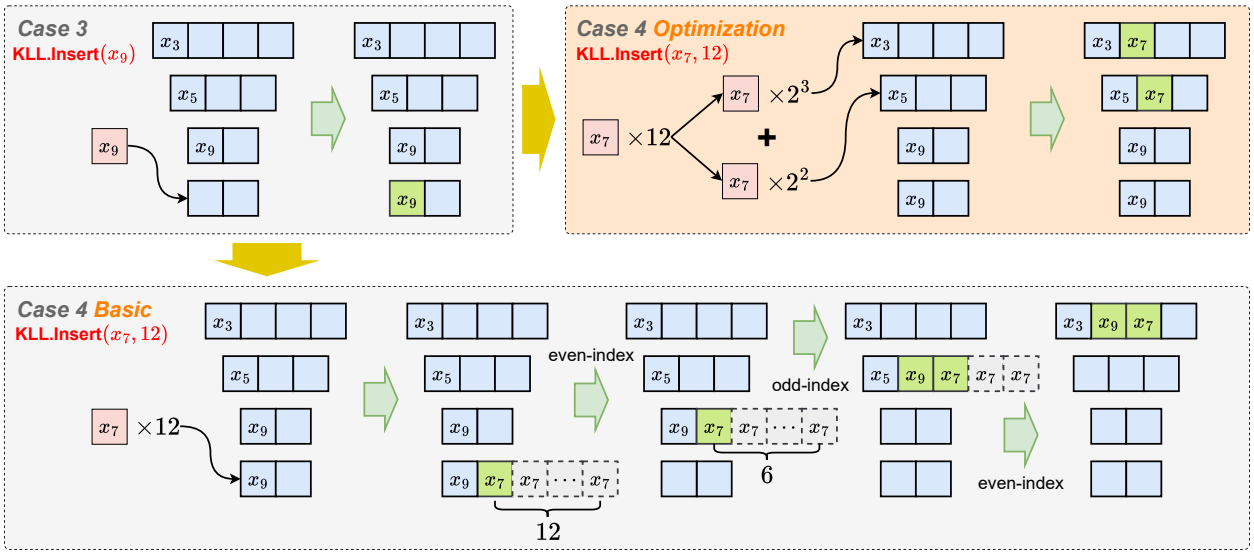


Figure 5: A running example of the basic and optimized version of KLL in Cooled-KLL.

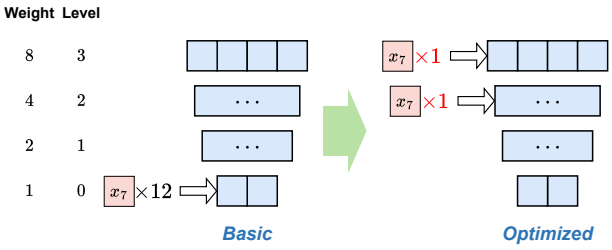
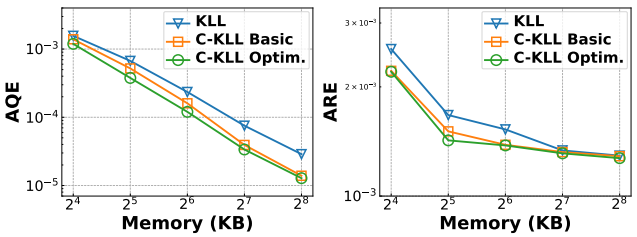


Figure 6: Example of binary split insertion.

Fig. 7 compares the basic and optimized versions of Cooled-KLL (abbreviated as C-KLL) with the original KLL at various memory sizes. We use the Price dataset (introduced in Section 5.1.2) and measure the average quantile error and average ranking error, which are metrics that assess the performance of $Quantile(\delta, S)$ and $Rank(x, S)$, respectively, as introduced in Section 5.1.4. Our optimized version of C-KLL consistently outperforms the basic version, and both surpass KLL, particularly when memory is limited. This demonstrates the effectiveness of binary split insertion.



(a) Average Quantile Error (b) Average Rank Error

Figure 7: Comparison of optimization effects.

3.5 Example of Insertion in KLL

Next, we present a running example of the interaction between the Hot Filter and KLL, as shown in Fig. 5. Note that their interaction is unidirectional, meaning that items only flow from the Hot Filter to KLL, but not the other way around. We continue with Case 3 and Case 4 from Fig. 4. When executing $KLL.Insert(x_9)$, the operation follows the same procedure as described in Fig. 2 for KLL. Assuming the current KLL structure is as shown in the top left corner, we simply insert x_9 into the lowest level, $C[0]$. When executing $KLL.Insert(x_7, 12)$, as described in the previous section, there are two versions: the basic version and the optimized version.

- In the basic version, we directly insert 12 instances of x_7 into $C[0]$, and then perform the compaction operation from Algorithm 2. The steps are similar to those in Figure 2, so we will not repeat them here. Due to compaction, errors are introduced; for example, the frequency of x_9 changes from 3 to 8, and the frequency of x_5 changes from 4 to 0 (which should not change).
- In the optimized version, we decompose 12 into $2^3 + 2^2$, so we can directly place one x_7 into $C[3]$ and one x_7 into $C[2]$. This avoids any compaction, and thus no errors are introduced.

4 Mathematical Analysis

Intuitively, using the Hot Filter to pre-filter hot items from the data streams enhances accuracy compared to using only KLL. We discuss the error bound of Cooled-KLL in Sections 4.1. Additionally, we estimate the space cost and time cost of Cooled-KLL in Section 4.2. Finally, we analyze the improvement achieved by the Binary Split Insertion method in Section 4.3. To facilitate understanding of our mathematical analysis, we will present the necessary parameters below.

Symbol	Description
N	Amount of data $\{x_1, ..., x_N\}$ in data stream
N_K	Amount of items in KLL part
N_H	Amount of items in Hot Filter part
H	Overall height of KLL
k_h	Capacity of compactor $C[h]$
m_h	Number of operations performed by compactor $C[h]$
$c(c < 1)$	Capacity ratio of adjacent compactors
ω_h	The weight of each item in compactor $C[h]$

4.1 Error Bound

Since the Cooled-KLL sketch comprises two parts, we need to calculate the error contributed by each part separately in the quantile estimation. Firstly, it's evident that items exclusively in the Hot Filter do not contribute to the quantile estimation error because the count of numbers stored in the Hot Filter is exact. Therefore, we only need to estimate the error from the KLL part.

In order to estimate $ERR(x)$, which indicates the total error caused by Cooled-KLL of x (actually, only the error of the KLL part needs to be calculated), we need to consider the following two values:

$r(x, h)$: The rank of x among the items yielded by the compactor $C[h]$ and all the items stored in the compactors with a height of no more than h after inserting.

$err(x, h)$: Defined as $r(x, h) - r(x, h-1)$,which indicates the influence of the compactor $C[h]$ on x.

Theorem 4.1. *For any constant $\epsilon > 0$, we get the estimate of the error bound in the KLL part:*

$$\mathbb{P}(|ERR(x)| > \epsilon N_K) \leq 2e^{-\frac{1}{8}(2c-1)\epsilon^2 k_H^2}$$

Proof. The proof is available in Appendix A. □

Based on the above conclusion, we can see that given a constant c, a smaller k_H is required to achieve higher accuracy, and the size of k_H is determined by the size of N_K. By filtering out some items using the Hot Filter, we can greatly reduce the size of k_H, thereby increasing the likelihood of obtaining more accurate results. In other words, the error in the KLL Sketch arises from its compacting operation. By reducing the number of compacting operations through "preprocessing", we can avoid larger errors.

To intuitively demonstrate the impact of the Hot Filter on k_H, we quantitatively calculated the changes in the k_H value before and after adding the Hot Filter under common distributions, which includes Exponential Distribution, Zipf Distribution, Uniform Distribution, Normal Distribution, Poisson Distribution. The details could be found in Appendix C.

4.2 Space and Time Cost

Theorem 4.2. *The space complexity of KLL is $O(\frac{1}{\epsilon}\sqrt{log_2\frac{1}{\delta}} + e)$, where ϵ, δ, e are all constants. The time complexity is $O(Nt_E + N_K t_K)$.*

Proof. Consider the Space complexity:

KLL Part: Consider adding the capacity k_h of each layer:

$$\sum_{h=1}^{H} k_h = \sum_{h=1}^{H} k_H c^{H-h} = k_H \sum_{h'=0}^{H-1} c^{h'} \leq k_H \frac{1-c^{H-1}}{1-c} \leq k_H \frac{1}{1-c}$$

Hot Filter Part: For each bucket,we need to store the items, their frequency and the value of votes. So for e buckets in total, the space cost is $O(e)$.

To ensure the accuracy

$$\mathbb{P}(|ERR(x)| > \epsilon N) \leq \mathbb{P}(|ERR(x)| > \epsilon N_K) \leq 2e^{-\frac{1}{8}(2c-1)\epsilon^2 k_H^2} \leq \delta$$

We can use the following k_H: $k_H = c_1 \cdot \frac{1}{\epsilon}\sqrt{log_2\frac{1}{\delta} + c_2}$, where c_1, c_2 are constants. So we get the space cost of the whole algorithm (under the specified accuracy requirements) to be $O(\frac{1}{\epsilon}\sqrt{log_2\frac{1}{\delta}} + e)$, where ϵ, δ, e are all constants. We can change the accuracy of the algorithm by setting e (increasing e will reduce k_H) and space cost. Without considering the particularity of data distribution, in order to ensure the same accuracy as KLL, we need to use more space.

Consider the Time cost:

The time cost in Hot Filter part is $O(Nt_E)$: Each item is input in turn, mapped and then judged (then output or retained), and the time cost for a single item is $O(t_E)$.

In KLL part, we consider that the time cost for a single operation of a single item is $O(t_K)$ and the number of items received by each layer is $\frac{1}{2}$ of the previous layer. The total number of operations performed is $\sum_{h=1}^{H} \frac{k_h}{2} \frac{N_K}{2^{h-1}k_h} = \sum_{h=1}^{H} \frac{N_K}{2^h} = O(N_K)$.

When inputting data, our total time cost is $O(Nt_E + N_K t_K)$. Compared with the total time cost of $O(Nt_K)$ using only KLL, the Cooled-KLL will be a little slower. □

4.3 Improvement of Binary Split Insertion

Theorem 4.3. *The error after using binary split insertion in Cooled-KLL is smaller than that of the basic version.*

Proof. The proof is available in Appendix D. □

5 Performance Evaluation

We have released our source code on the Github [2].

5.1 Experiment Setup

5.1.1 Test Platform: Our experiments were conducted on a machine featuring an Intel i7-9700 CPU @ 3.0GHz and 16GB of DRAM, operating on Ubuntu 20.04. To minimize CPU jitter errors, each evaluation was repeated five times, and the results were averaged.

5.1.2 Datasets: We employed three datasets for our experiments, all sourced from Kaggle. The eCom dataset [1] comprises purchase data from a leading online store specializing in home appliances and electronics. The Price dataset [3] contains behavioral data from a vast multi-category online retailer. Lastly, the Voltage dataset [4] includes real-time operational voltage readings from a permanent magnet synchronous motor used in automotive applications. Detailed descriptions of these datasets can be found in Table 1.

Dataset	CAIDA	Delay	eCom	Price	Voltage
Value	5.8M	7.3M	2.2M	30M	30M
Unique value	1.9M	1.6K	2.8K	54K	188K
Max freq.	353K	572K	52K	151K	36K

Table 1: Dataset details (M=10^6, K=10^3).

5.1.3 Comparison Algorithms: Our Cooled-KLL algorithm (C-KLL), implemented in C++, was benchmarked against four existing sketches: DD-Sketch [35] (DD), GK-Sketch [20] (GK), Req-Sketch [9] (Req), and KLL Sketch [28] (KLL). Each algorithm was implemented faithfully according to specifications found in their respective original publications. Detailed configurations for each algorithm are provided in Appendix E.

5.1.4 Tasks and Metrics. As detailed in Section 2.1, our primary task is all-quantile estimation. We evaluate the performance using the following metrics:

Throughput: This is calculated as N/T, where N represents the total number of items processed, and T is the total time taken to run the algorithm.

Average Quantile Error (AQE): AQE is defined as

$$\frac{1}{|\Delta|}\sum_{\delta\in\Delta}\frac{|Rank(\widehat{Quantile}(\delta,S),S)-\lceil\delta\cdot|S|\rceil|}{|S|}$$

where $\widehat{\text{Quantile}}()$ is the estimated value returned by the algorithms, S is the data stream, and Δ is the set of quantile queries. For our experiments, we use $\Delta=\{0.01\%,0.02\%,\ldots,99.98\%,99.99\%\}$.

Average Rank Error (ARE): ARE is defined as

$$\frac{1}{|X|}\sum_{x\in X}\frac{|\widehat{Rank}(x,S)-Rank(x,S)|}{|S|}$$

where $\widehat{\text{Rank}}()$ is the estimated rank, and X is the set of rank queries. For our experiments, we use $X=\bigcup_{\delta\in\Delta}\{\text{Quantile}(\delta,S)\}$, which includes items whose quantiles range from 0.01% to 99.99%.

5.2 Parameter Tuning

5.2.1 Eviction Threshold (λ). In Algorithm 3, the eviction threshold λ is a crucial parameter. If λ is set too low, evictions will occur more frequently, potentially causing the premature removal of hot items. Conversely, if λ is too high, it becomes challenging to evict items from the Hot Filter, giving early items an unfair advantage. Fig. 8(a) illustrates the AQE curve of C-KLL as λ varies from 1 to 20, with total memory capacities of 32KB, 64KB, 128KB, and 256KB. We observed that the overall AQE decreases as λ increases from 1 to 16 but rises again when λ reaches 20. Hence, we chose $\lambda=16$ for our subsequent experiments.

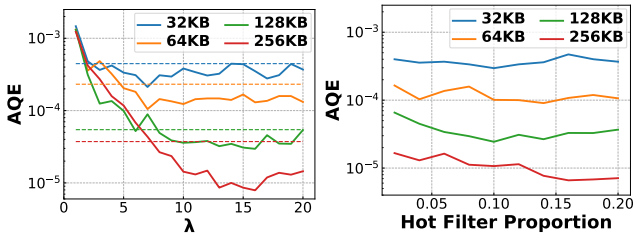
5.2.2 Hot Filter Proportion. The proportion of memory allocated to the Hot Filter is equally significant. If the Hot Filter's proportion is too small, it cannot accommodate all hot items, leading to accuracy loss. Conversely, a disproportionately large Hot Filter reduces query speed, as C-KLL needs to traverse it when estimating quantiles. Fig. 8(b) shows the AQE curve of C-KLL as the Hot Filter ratio varies from 0.02 to 0.2. The curve is relatively stable, with a slight decrease in AQE from 0.02 to 0.1. Consequently, we set the Hot Filter proportion to 0.1 in our subsequent experiments, allocating 10% of the memory to the Hot Filter and 90% to the KLL.

5.3 Processing Speed

5.3.1 Insertion Speed: Fig. 9 presents the insertion speed of each algorithm as memory usage varies across five datasets. Our C-KLL algorithm, along with KLL and DD, consistently demonstrates the highest insertion speeds, placing them in the top tier. In contrast, Req's speed decreases significantly as memory increases, while GK remains the slowest, with an average speed of less than 0.3Mps.

5.3.2 Query Quantile Time: Fig. 10 displays the average time taken by each algorithm to perform the "$Quantile(\delta,S)$" operation. Our C-KLL is as fast as GK, Req, and KLL across all five datasets, while DD is approximately ten times slower. Additionally, we observed that the running time for all algorithms increases with memory due to the sorting operations required in batch queries.

5.3.3 Query Rank Time: Query rank time measures the average duration each algorithm takes to perform the "$Rank(\delta,S)$" operation. The results are similar to the query quantile time, hence we omitted them here. For detailed results, please refer to Appendix F.



(a) AQE vs. λ (b) AQE vs. Hot Filter proportion

Figure 8: Parameter tuning.

Metrics \ Algorithm	Dataset	**Cooled-KLL**	KLL	DD-Sketch	GK-Sketch	Req-Sketch
Insertion Speed (Mps)	CAIDA	1.7213	1.9542	1.9531	0.0552	0.3982
	Delay	2.2083	2.1249	2.1738	0.0575	0.492
	eCom	2.2048	2.1208	2.1518	0.0708	0.5167
	Price	2.0009	2.097	2.0809	0.0654	0.5147
	Voltage	1.9918	2.1498	2.0077	0.0688	0.5093
Query Quantile Time	CAIDA	1.863×10^{-3}	1.3514×10^{-3}	5.7516×10^{-3}	8.244×10^{-4}	6.341×10^{-4}
	Delay	1.31×10^{-4}	8.078×10^{-4}	6.715×10^{-3}	7.212×10^{-4}	4.085×10^{-4}
	eCom	2.223×10^{-4}	5.35×10^{-4}	3.917×10^{-4}	6.868×10^{-4}	8.2694×10^{-3}
	Price	1.5647×10^{-3}	8.806×10^{-4}	8.9606×10^{-3}	5.986×10^{-4}	8.095×10^{-4}
	Voltage	1.1723×10^{-3}	1.0142×10^{-3}	7.4638×10^{-3}	8.188×10^{-4}	9.326×10^{-4}
Average Quantile Error	CAIDA	1.18×10^{-5}	1.63×10^{-5}	3.3×10^{-6}	1.78×10^{-5}	4.11×10^{-5}
	Delay	2.5×10^{-11}	5.07568×10^{-7}	2.5×10^{-11}	9.8×10^{-7}	2.50071×10^{-6}
	eCom	7.18×10^{-10}	2.14855×10^{-6}	4.680302×10^{-3}	2.57×10^{-6}	5.0459×10^{-6}
	Price	3.2×10^{-6}	1.27×10^{-5}	9.96×10^{-4}	2.59×10^{-5}	4.24×10^{-5}
	Voltage	3.2×10^{-6}	1.05×10^{-5}	1.41176×10^{-2}	2.14×10^{-5}	5.32×10^{-5}
Average Rank Error	CAIDA	2.25×10^{-5}	4.46×10^{-5}	1.56×10^{-5}	3.3×10^{-5}	5.81×10^{-5}
	Delay	1.9×10^{-6}	5.38×10^{-5}	1×10^{-7}	1×10^{-6}	3.98×10^{-5}
	eCom	1.98×10^{-5}	6.74×10^{-5}	1.035×10^{-3}	2×10^{-5}	4.68×10^{-5}
	Price	2.16×10^{-5}	3.6×10^{-5}	7.332×10^{-4}	2.74×10^{-5}	6.31×10^{-5}
	Voltage	2.49×10^{-5}	5.78×10^{-5}	1.39776×10^{-2}	2.49×10^{-5}	6.53×10^{-5}

Table 2: Performance summary under Memory=512KB. The best and worst results for each row are shown in green and orange.

5.4 δ-Quantile Estimation

5.4.1 Varying the Sketch Size: Fig. 11 illustrates how the AQE of each algorithm changes with varying memory sizes. In the two long-tail datasets, CAIDA and Delay, our C-KLL algorithm is only slightly behind DD and outperforms the remaining algorithms. In the three relatively uniform datasets—eCom, Price, and Voltage—C-KLL surpasses all other algorithms, while DD performs poorly, exhibiting almost a straight line. Notably, in the eCom dataset, C-KLL outperforms KLL by up to 1.5 orders of magnitude.

5.4.2 Varying the Queried Quantile (δ): Fig. 12 depicts the change in AQE with varying queried quantiles for each algorithm under 512KB of memory. Some vertical lines in the figure represent zero (since the y-axis uses logarithmic coordinates and cannot represent zero). Our C-KLL significantly outperforms all other algorithms in the eCom, Price, and Voltage datasets, demonstrating stable results at each quantile, while DD performs the worst. In the Delay dataset, C-KLL and DD show similar performance, with nearly zero error.

5.5 Rank Estimation

5.5.1 Varying the Sketch Size: Fig. 13 presents the ARE results of each algorithm as memory usage varies. Similar to the previous AQE results, our C-KLL algorithm outperforms all algorithms except DD in the CAIDA and Delay datasets and consistently leads over the ordinary KLL. In the eCom, Price, and Voltage datasets, both C-KLL and GK perform best, while DD performs poorly, as expected.

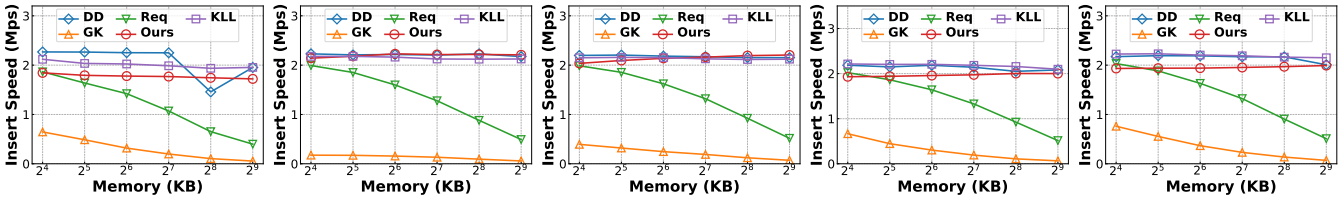
5.5.2 Varying the Queried Quantile (δ): Fig. 14 shows the ARE results of each algorithm as the queried quantile changes. Our C-KLL maintains the best and most stable performance in the eCom, Price, and Voltage datasets. Although GK's overall ARE is nearly as good as ours, it lacks stability. For instance, in the Voltage dataset (Fig. 14(e)), GK performs poorly outside 30% ~ 70%. In the CAIDA and Delay datasets, our algorithm is only surpassed by DD.

5.6 Experimental Performance Overview

Table 2 illustrates the comparative performance of the five algorithms under 512KB. Our Cooled-KLL excels in both insertion speed and error rate, which are the two most critical metrics for data stream algorithms. Insertion speed directly impacts the ability to handle data in real-time, which is crucial in applications such as network monitoring or financial data analysis, where data needs to be processed quickly without delay. The error determines the precision of the quantile estimates, which is vital in scenarios like decision-making based on real-time analytics. Although our query time has no significant advantage, it is still comparable to the fastest algorithms (within the same order of magnitude). Generally, query time is less critical. For example, frequent real-time data insertions are required in network traffic analysis, but quantile queries may only be performed periodically. Thus, optimizing for insertion speed and error is more important than query speed in most real-time systems. Notably, we outperform the KLL in almost all aspects.

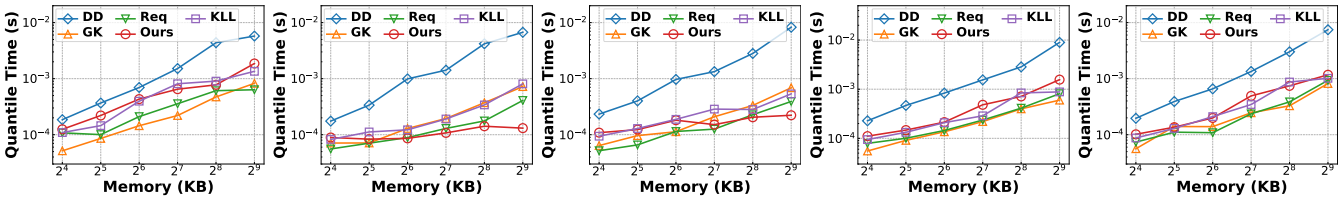
6 Conclusion

Quantile estimation is crucial in data stream processing, with diverse applications across the field. However, the current optimal algorithm, KLL, often results in significant memory waste when processing highly skewed data streams. To address this issue, we introduce Cooled-KLL, which employs a filtering mechanism for high-frequency (hot) items, effectively "cooling" KLL and thereby enhancing its accuracy. We present a comprehensive optimization and solid theoretical analysis to support our approach. Extensive experiments demonstrate that Cooled-KLL not only achieves superior accuracy compared to state-of-the-art algorithms but also maintains high speed, underscoring its real-world practicality.



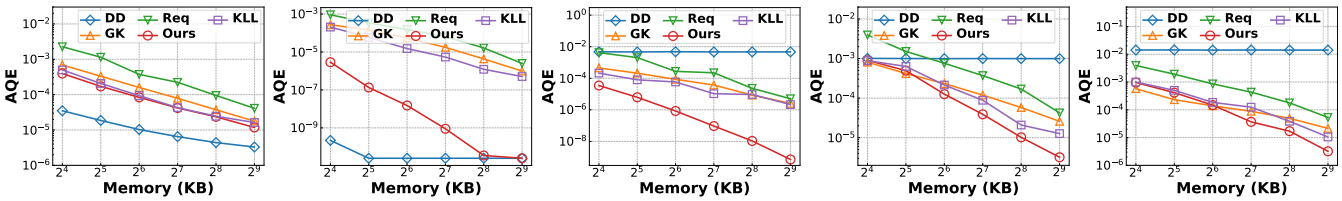
(a) CAIDA (b) Delay (c) eCom (d) Price (e) Voltage

Figure 9: Insertion Speed vs. Memory (KB).



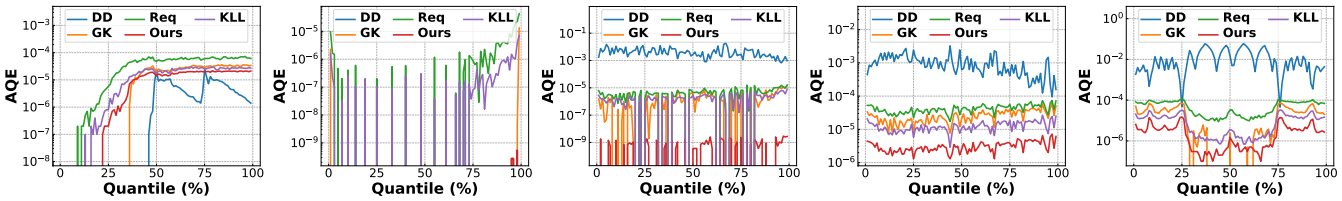
(a) CAIDA (b) Delay (c) eCom (d) Price (e) Voltage

Figure 10: Query Quantile Time (s) vs. Memory (KB).



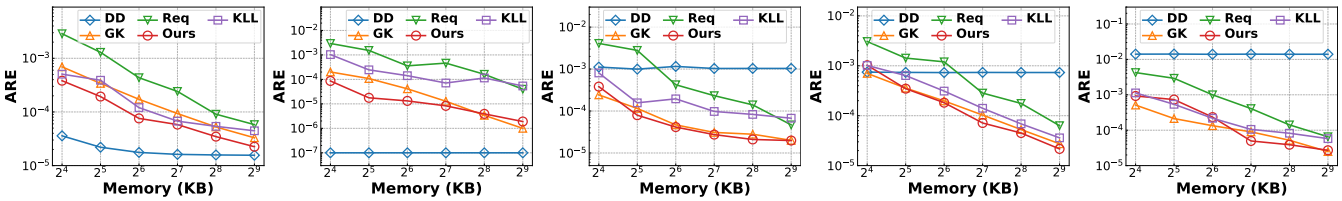
(a) CAIDA (b) Delay (c) eCom (d) Price (e) Voltage

Figure 11: AQE vs. Memory (KB).



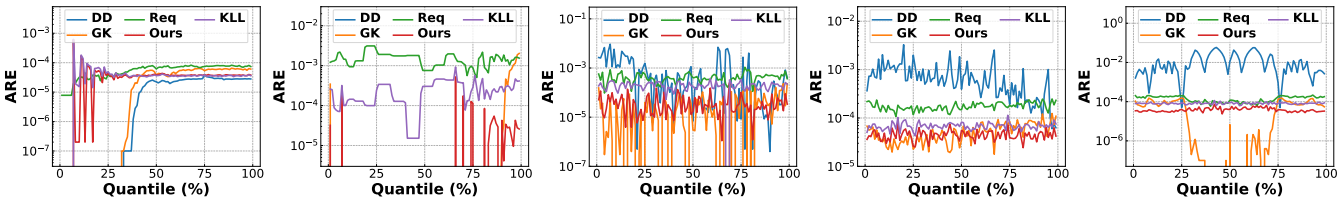
(a) CAIDA (b) Delay (c) eCom (d) Price (e) Voltage

Figure 12: AQE vs. Queried Quantile (%).



(a) CAIDA (b) Delay (c) eCom (d) Price (e) Voltage

Figure 13: ARE vs. Memory (KB).



(a) CAIDA (b) Delay (c) eCom (d) Price (e) Voltage

Figure 14: ARE vs. Queried Quantile (%).

References

[1] [n. d.]. eCommerce dataset. https://www.kaggle.com/datasets/mkechinov/ecommerce-purchase-history-from-electronics-store.

[2] [n. d.]. Our open source Github. https://github.com/Cooled-KLL/Cooled-KLL-Sketch.

[3] [n. d.]. Price dataset. https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store.

[4] [n. d.]. Voltage dataset. https://www.kaggle.com/datasets/graxlmaxl/identifying-the-physics-behind-an-electric-motor.

[5] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. 2022. {HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks. *in USENIX NSDI* (2022).

[6] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. 2021. SALSA: Self-Adjusting Lean Streaming Analytics. *in IEEE ICDE* (2021).

[7] Valerio Bruschi, Ran Ben Basat, Zaoxing Liu, Gianni Antichi, Giuseppe Bianchi, and Michael Mitzenmacher. 2020. DISCOvering the heavy hitters with disaggregated sketches. *in ACM CoNEXT* (2020).

[8] Ziling Chen, Haoquan Guan, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. 2024. Determining Exact Quantiles with Randomized Summaries. *in ACM SIGMOD* (2024).

[9] Graham Cormode, Zohar Karnin, Edo Liberty, Justin Thaler, and Pavel Veselỳ. 2021. Relative error streaming quantiles. *in ACM PODS* (2021).

[10] Graham Cormode, Abhinav Mishra, Joseph Ross, and Pavel Veselỳ. 2021. Theory meets practice at the median: A worst case comparison of relative error quantile algorithms. *in ACM SIGKDD* (2021).

[11] Shuang Cui, Kai Han, Jing Tang, He Huang, Xueying Li, and Zhiyu Li. 2023. Streaming algorithms for constrained submodular maximization. *in ACM SIGMETRICS* (2023).

[12] Haipeng Dai, Muhammad Shahzad, Alex X Liu, and Yuankun Zhong. 2016. Finding persistent items in data streams. *Proceedings of the VLDB Endowment* 10, 4 (2016), 289–300.

[13] Siyuan Dong, Zhuochen Fan, Tianyu Bai, Tong Yang, Hanyu Xue, Peiqing Chen, and Yuhan Wu. 2024. M4: A Framework for Per-Flow Quantile Estimation. *in IEEE ICDE* (2024).

[14] Ted Dunning. 2021. The t-digest: Efficient estimates of distributions. *Software Impacts* 7 (2021), 100049.

[15] Shaked Elias Zada, Arik Rinberg, and Idit Keidar. 2023. Quancurrent: A Concurrent Quantiles Sketch. *in ACM SPAA* (2023).

[16] Italo Epicoco, Catiuscia Melle, Massimo Cafaro, Marco Pulimeno, and Giuseppe Morleo. 2020. Uddsketch: Accurate tracking of quantiles in data streams. *IEEE Access* 8 (2020), 147604–147617.

[17] Lasantha Fernando, Harsh Bindra, and Khuzaima Daudjee. 2023. An Experimental Analysis of Quantile Sketches over Data Streams.. In *EDBT*. 424–436.

[18] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-based quantile sketches for efficient high cardinality aggregation queries. *arXiv preprint arXiv:1803.01969* (2018).

[19] Cormode Graham, Korn Flip, Muthukrishnan Shanmugavelayutham, and Srivastava Divesh. 2003. Finding hierarchical heavy hitters in data streams. *in ACM VLDB* (2003).

[20] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. *in ACM SIGMOD* (2001).

[21] Elena Gribelyuk, Pachara Sawettamalya, Hongxun Wu, and Huacheng Yu. 2024. Simple & Optimal Quantile Sketch: Combining Greenwald-Khanna with Khanna-Greenwald. *in ACM SIGMOD* (2024).

[22] Jiarui Guo, Yisen Hong, Yuhan Wu, Yunfei Liu, Tong Yang, and Bin Cui. 2023. Sketchpolymer: Estimate per-item tail quantile using one sketch. *in ACM SIGKDD* (2023).

[23] Meghal Gupta, Mihir Singhal, and Hongxun Wu. 2024. Optimal quantile estimation: beyond the comparison model. *arXiv preprint arXiv:2404.03847* (2024).

[24] 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. *in ACM SIGMOD* (2023).

[25] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. Qpipe: Quantiles sketch fully in the data plane. *in ACM CoNEXT* (2019), 285–291.

[26] Peng Jia, Pinghui Wang, Rundong Li, Junzhou Zhao, Junlan Feng, Xidian Wang, and Xiaohong Guan. 2024. A Compact and Accurate Sketch for Estimating a Large Range of Set Difference Cardinalities. *in IEEE ICDE* (2024).

[27] Peng Jia, Pinghui Wang, Junzhou Zhao, Shuo Zhang, Yiyan Qi, Min Hu, Chao Deng, and Xiaohong Guan. 2021. Bidirectionally densifying lsh sketches with empty bins. *in ACM SIGMOD* (2021).

[28] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. *in IEEE FOCS* (2016).

[29] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A self-designing range filter. *in ACM SIGMOD* (2022).

[30] Weihe Li and Paul Patras. 2023. P-Sketch: A Fast and Accurate Sketch for Persistent Item Lookup. *IEEE/ACM Transactions on Networking* (2023).

[31] Weihe Li and Paul Patras. 2023. Tight-sketch: A high-performance sketch for heavy item-oriented data stream mining with limited memory size. *in ACM CIKM* (2023).

[32] Weihe Li and Paul Patras. 2024. Stable-Sketch: A Versatile Sketch for Accurate, Fast, Web-Scale Data Stream Processing. *in ACM WWW* (2024).

[33] Jiaqian Liu, Ran Ben Basat, Louis De Wardt, Haipeng Dai, and Guihai Chen. 2024. DISCO: A Dynamically Configurable Sketch Framework in Skewed Data Streams. *in IEEE ICDE* (2024).

[34] Antonis Manousis, Zhuo Cheng, Ran Ben Basat, Zaoxing Liu, and Vyas Sekar. 2022. Enabling efficient and general subpopulation analytics in multidimensional data streams. *in ACM VLDB* (2022).

[35] 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).

[36] Rory Mitchell, Eibe Frank, and Geoffrey Holmes. 2021. An empirical study of moment estimators for quantile approximation. *ACM Transactions on Database Systems (TODS)* 46, 1 (2021), 1–21.

[37] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. 2022. {SketchLib}: Enabling efficient sketch-based monitoring on programmable switches. *in USENIX NSDI* (2022).

[38] Yiyan Qi, Rundong Li, Pinghui Wang, Yufang Sun, and Rui Xing. 2024. QSketch: An Efficient Sketch for Weighted Cardinality Estimation in Streams. *arXiv preprint arXiv:2406.19143* (2024).

[39] Yiyan Qi, Pinghui Wang, Yuanming Zhang, Qiaozhu Zhai, Chenxu Wang, Guangjian Tian, John CS Lui, and Xiaohong Guan. 2020. Streaming algorithms for estimating high set similarities in loglog space. *IEEE Transactions on Knowledge and Data Engineering* 33, 10 (2020), 3438–3452.

[40] Rana Shahout, Roy Friedman, and Ran Ben Basat. 2022. SQUAD: Combining sketching and sampling is better than either for per-item quantile estimation. *arXiv preprint arXiv:2201.01958* (2022).

[41] Rana Shahout, Roy Friedman, and Ran Ben Basat. 2023. Together is Better: Heavy Hitters Quantile Estimation. *in ACM SIGMOD* (2023).

[42] 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 IEEE ICDE* (2024).

[43] Qilong Shi, Yuchen Xu, Jiuhua Qi, Wenjun Li, Tong Yang, Yang Xu, and Yi Wang. 2023. Cuckoo Counter: Adaptive Structure of Counters for Accurate Frequency and Top-k Estimation. *IEEE/ACM Transactions on Networking* (2023).

[44] Lu Tang, Qun Huang, and Patrick PC Lee. 2019. Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. *IEEE INFOCOM* (2019).

[45] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *in ACM VLDB* (2022).

[46] Hancheng Wang, Haipeng Dai, Rong Gu, Youyou Lu, Jiaqi Zheng, Jingsong Dai, Shusen Chen, Zhiyuan Chen, Shuaituan Li, and Guihai Chen. 2024. Wormhole filters: Caching your hash on persistent memory. *in ACM EuroSys* (2024).

[47] Hancheng Wang, Haipeng Dai, Meng Li, Jun Yu, Rong Gu, Jiaqi Zheng, and Guihai Chen. 2022. Bamboo filters: Make resizing smooth. *in IEEE ICDE* (2022).

[48] Pinghui Wang, Yiyan Qi, Yuanming Zhang, Qiaozhu Zhai, Chenxu Wang, John CS Lui, and Xiaohong Guan. 2019. A memory-efficient sketch method for estimating high similarities in streaming sets. *in ACM SIGKDD* (2019).

[49] Xiaocan Wu, He Huang, Yang Du, Yu-E Sun, and Shigang Chen. 2023. Coupon filter: A universal and lightweight filter framework for more accurate data stream processing. *Computer Networks* 228 (2023), 109748.

[50] Yuhan Wu, Zhuochen Fan, Qilong Shi, Yixin Zhang, Tong Yang, Cheng Chen, Zheng Zhong, Junnan Li, Ariel Shtul, and Yaofeng Tu. 2022. She: A generic framework for data stream mining over sliding windows. *in ICPP* (2022).

[51] Yuhan Wu, Aomufei Yuan, Zhouran Shi, Yuanpeng Li, Yikai Zhao, Peiqing Chen, Tong Yang, and Bin Cui. 2024. Online Detection of Outstanding Quantiles with QuantileFilter. *in IEEE ICDE* (2024).

[52] Kaicheng Yang, Sheng Long, Qilong Shi, Yuanpeng Li, Zirui Liu, Yuhan Wu, Tong Yang, and Zhengyi Jia. 2023. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[53] Minlan Yu. 2019. Network telemetry: towards a top-down approach. *in ACM SIGCOMM Computer Communication Review* 49, 1 (2019), 11–17.

[54] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. 2021. DHS: Adaptive Memory Layout Organization of Sketch Slots for Fast and Accurate Data Stream Processing. *in ACM SIGKDD* (2021).

[55] Fuheng Zhao, Punnal Ismail Khan, Divyakant Agrawal, Amr El Abbadi, Arpit Gupta, and Zaoxing Liu. 2023. Panakos: Chasing the Tails for Multidimensional Data Streams. *in ACM VLDB* (2023).

[56] Fuheng Zhao, Sujaya Maiyya, Ryan Wiener, Divyakant Agrawal, and Amr El Abbadi. 2021. Kll±approximate quantile sketches over dynamic datasets. *in ACM VLDB* (2021).

Appendix

A Error Bound

Considering the impact of each layer of the compactor on x, after performing an operation on layer h, the estimate of the rank of x either remains unchanged or increases or decreases by ω_h with equal probability. This occurs because if x ranks odd in this layer, there will be an even number of items smaller than x in this layer, so whether we remove the odd or even items, our estimate of x will not change. Conversely, if x ranks even in this compactor, there will be an odd number of items smaller than x. In that case, if we remove the even ones, the estimate of the rank of x will increase by ω_h, and if we remove the odd ones, the estimated rank of x will decrease by ω_h.

Thus, we can consider the following random variable $X_h^{(i)}$, which represents the error concerning x generated during the i-th operation at layer h. (We define that if the estimate of the rank of x decreases or increases, X will be equal to -1 or 1; otherwise, X will be equal to 0.) Based on the previous analysis, we can derive the following properties:

(1) $\mathbb{E}[X_h^{(i)}] = 0; |X_h^{(i)}| \le 1$

(2) $ERR(x) = \Sigma_{h=1}^{H} err(x,h) = \Sigma_{h=1}^{H}\Sigma_{i=1}^{m_h}\omega_h X_h^{(i)}$

It is not difficult to consider using the Hoeffding inequality to estimate the $ERR(X)$ in (2) above.

Lemma A.1 (Hoeffding Inequality). *For independent random variables $X_1, X_2 ... X_n$, if $X_i \in [-\omega_i.\omega_i]$, $\mathbb{E}[X_i] = 0$,then for any t>0, we have:*

$$\mathbb{P}(|\sum_{i=1}^{n} X_i| > t) \le 2e^{-\frac{t^2}{2\Sigma_{i=1}^{n}\omega_i^2}}$$

Proof. The proof is available in Appendix B. □

Back to the estimate of $ERR(X)$, we regard $\omega_h X_h^{(i)}$ as a random variable whose size belongs to $[-\omega_i.\omega_i]$, $E(X_h^{(i)}) = 0$, which obviously meets the conditions for using the Hoeffding Inequality:

$$\mathbb{P}(|ERR(x)| > \epsilon N_K) \le 2e^{-\frac{\epsilon^2 N_K^2}{2\Sigma_{h=1}^{H}\Sigma_{i=1}^{m_h} w_h^2}} = 2e^{-\frac{\epsilon^2 N_K^2}{2\Sigma_{h=1}^{H} m_h w_h^2}} \quad (*)$$

We need to estimate the range of H, m_h:

(1) $H \le log_2 \frac{N_K}{ck_H} + 2$

(2) $m_h \le \frac{N_K}{k_h\omega_h} = \frac{N_K}{k_H c^{H-h} 2^{h-1}}$

(2) is obvious, and the reason why (1) is valid is that $N_K \ge k_{H-1}\omega_{H-1} = k_H c 2^{H-2}$. The above inequality is obviously true, otherwise we would not need the H-th compactor.

Then, we have:

$$\sum_{h=1}^{H} m_h w_h^2 \le \sum_{h=1}^{H} \frac{N_K}{k_H c^{H-h} 2^{h-1}} 2^{2h-2} \le \sum_{h=1}^{H} \frac{N_K}{k_H c^{H-h}} 2^{h-1}$$

Consider the sum of geometric sequence:

$$\sum_{h=1}^{H} m_h w_h^2 \le \frac{N_K}{2k_H c^H} \frac{(2c)^{H+1}}{2c-1}$$

After that, we substitute it into $(*)$:

$$\mathbb{P}(|ERR(x)| > \epsilon N_K) \le 2e^{-\frac{N_K k_H (2c-1)\epsilon^2}{2^{H+1}c}} \le 2e^{-\frac{(2c-1)\epsilon^2 k_H^2 c 2^{H-2}}{2^{H+1}c}}$$

Finally, we get the estimate of the error bound in the KLL part:

$$\mathbb{P}(|ERR(x)| > \epsilon N_K) \le 2e^{-\frac{1}{8}(2c-1)\epsilon^2 k_H^2}$$

B Proof of Lemma A.1

Lemma A.1. (Hoeffding Inequality) *For independent random variables $X_1, X_2 ... X_n$, if $X_i \in [-\omega_i.\omega_i]$, $\mathbb{E}[X_i] = 0$,then for any t>0, we have:*

$$\mathbb{P}(|\sum_{i=1}^{n} X_i| > t) \le 2e^{-\frac{t^2}{2\Sigma_{i=1}^{n}\omega_i^2}}$$

Proof. It's not difficult to find that:

$$\mathbb{P}\left\{\sum_{i=1}^{n} X_i \ge t\right\} = \mathbb{P}\left\{e^{\lambda\Sigma_{i=1}^{n}X_i} \ge e^{\lambda t}\right\}$$

According to Markov inequality:

$$\mathbb{P}\left\{e^{\lambda\Sigma_{i=1}^{n}X_i} \ge e^{\lambda t}\right\} \le \frac{\mathbb{E}[e^{\lambda\Sigma_{i=1}^{n}X_i}]}{e^{\lambda t}} = e^{-\lambda t}\prod_{i=1}^{n}\mathbb{E}[e^{\lambda X_i}]$$

Therefore, we consider estimating $\psi(X) = log\mathbb{E}[e^{\lambda X}]$:

$$\psi'(\lambda) = \frac{\mathbb{E}[Xe^{\lambda X}]}{\mathbb{E}[e^{\lambda X}]}, \quad \psi''(\lambda) = \frac{\mathbb{E}[X^2 e^{\lambda X}]}{\mathbb{E}[e^{\lambda X}]} - \left(\frac{\mathbb{E}[Xe^{\lambda X}]}{\mathbb{E}[e^{\lambda X}]}\right)^2$$

We define $\mathbb{Q}$, whose probability density function is $\frac{d\mathbb{Q}}{d\mathbb{P}} = \frac{e^{\lambda X}}{\mathbb{E}[e^{\lambda X}]}$, $\frac{d\mathbb{Q}}{d\mathbb{P}}$ is called Radon-Nikodym derivative. When two probability measures $\mathbb{P}$ and $\mathbb{Q}$ are absolutely continuous with each other, then there exists a nonnegative measurable function $\frac{d\mathbb{Q}}{d\mathbb{P}}$ such that for any measurable set A, $\mathbb{Q}(A) = \int_A \frac{d\mathbb{Q}}{d\mathbb{P}} d\mathbb{P}$. So for any measurable set A, we have:

$$\mathbb{E}_{\mathbb{Q}}(X) = \int_A X\frac{d\mathbb{Q}}{d\mathbb{P}}d\mathbb{P} = \frac{\mathbb{E}[Xe^{\lambda X}]}{\mathbb{E}[e^{\lambda X}]}, \quad \mathbb{E}_{\mathbb{Q}}(X^2) = \frac{\mathbb{E}[X^2 e^{\lambda X}]}{\mathbb{E}[e^{\lambda X}]}$$

Then we have $\psi''(\lambda) = \mathbb{E}_{\mathbb{Q}}[X^2] - (\mathbb{E}_{\mathbb{Q}}[X])^2 = Var_{\mathbb{Q}}[X]$.

For $a \le X \le b, Var_{\mathbb{Q}}[X] = \mathbb{E}_{\mathbb{Q}}[|X - \mathbb{E}_X|^2] \le \mathbb{E}_{\mathbb{Q}}[|\frac{b-a}{2}|^2] = \frac{(b-a)^2}{4}$.

From $\psi(0) = 0, \psi'(0) = 0, \psi''(\lambda) \le \frac{(b-a)^2}{4}$, we could get the estimation of $\psi(\lambda)$:

$$\psi(\lambda) = \psi(0) + \int_0^{\lambda}(\psi'(0) + \int_0^{s}\psi''(t)dt)ds \le \frac{(b-a)^2\lambda^2}{8}$$

Based on the above estimates and conditions, we have:

$$\mathbb{P}\left\{\sum_{i=1}^{n} X_i \ge t\right\} \le \inf_{\lambda>0} e^{-\lambda t + \frac{\lambda^2\Sigma_{i=1}^{n}(w_i-(-w_i))^2}{8}} \le e^{-\frac{t^2}{2\Sigma_{i=1}^{n}w_i^2}}$$

For any random variable we have $\mathbb{P}(X \ge \epsilon) = \mathbb{P}(-X \le -\epsilon)$, so:

$$\mathbb{P}(|\sum_{i=1}^{n} X_i| > t) = 2\mathbb{P}(\sum_{i=1}^{n} X_i > t) \le 2e^{-\frac{t^2}{2\Sigma_{i=1}^{n}\omega_i^2}}$$

□

C Analysis of Common Distributions

Using the inequality $N_K \geq k_{H'-1}\omega_{H'-1} = k_{H'}c2^{H'-2}$ (assuming that the same k_{min} and c are used when only using KLL Sketch, and its height is H, then $N \geq k_H c2^{H-2}$), we can estimate $(\frac{c}{2})^{H-H'} \sim \frac{N_K}{N}$.

Using the above estimation, we can estimate that

$$\frac{k_{H'}}{k_H} \sim c^{H-H'} = (\frac{N_K}{N})^{\frac{lnc}{lnc-ln2}}$$

C.1 Exponential Distribution

The probability density function of an exponential distribution ($X \sim Exp(\lambda)$) is:

$$p(x;\lambda) = \begin{cases} \frac{1}{\lambda}e^{-\frac{x}{\lambda}}, x > 0 \\ 0, x \leq 0 \end{cases}$$

If we consider that the data in the interval $(0, t)$ is retained in the Hot Filter as hot items,then the items we left behind accounted for:

$$\frac{\int_t^{+\infty} p(x;\lambda)dx}{\int_0^{+\infty} p(x;\lambda)dx} = \frac{\int_t^{+\infty} \frac{1}{\lambda}e^{-\frac{x}{\lambda}}dx}{\int_0^{+\infty} \frac{1}{\lambda}e^{-\frac{x}{\lambda}}dx} = \frac{-e^{-\frac{x}{\lambda}}|_t^{\infty}}{-e^{-\frac{x}{\lambda}}|_0^{+\infty}} = e^{-\frac{t}{\lambda}}$$

According to the above analysis: $N_K = Ne^{-\frac{t}{\lambda}}$

Compared with the basic KLL method, a smaller $k_{H'} \sim (e^{-\frac{t}{\lambda}})^{\frac{lnc}{lnc-ln2}} k_H$ is achieved. If $t = \lambda, c = \frac{2}{3}$ are selected,then $k_{H'} \sim 0.69k_H$.

C.2 Zipf Distribution

The probability mass function is:

$$P(X = x|\alpha, n) = \frac{1}{x^{\alpha}}/(\sum_{i=1}^{n}\frac{1}{i^{\alpha}}), x = 1, 2...n$$

If we consider that the items in the interval $(0, t)$ are retained in the Hot Filter as hot items, then the items we left behind accounted for(smaller x is "hotter"):

$$\frac{N_K}{N} = 1 - \frac{\sum_{i=1}^{t} P(X = t|\alpha, n)}{1} = 1 - (\sum_{i=1}^{t}\frac{1}{i^{\alpha}})/(\sum_{i=1}^{n}\frac{1}{i^{\alpha}})$$

If $n = 1000, \alpha = 1, t = 50, c = \frac{2}{3}$ are selected,then $k_{H'} \sim 0.71k_H$.

C.3 Uniform Distribution

The probability density function of an uniform distribution ($X \sim U[a, b]$) is:

$$p(x; a, b) = \frac{1}{b-a}I_{[a,b]}$$

$I[a, b]$ above means the indicator function of $[a, b]$.

If we leave t% of the items in the Hot Filter:$\frac{N_K}{N} = 1 - t\%$. If $t = 5, c = \frac{2}{3}$ are selected,then $k_{H'} \sim 0.98k_H$.

C.4 Normal Distribution

The probability density function of a Normal distribution ($X \sim N[\mu, \sigma^2]$) is:

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If we leave the items which belongs to $[\mu - \sigma, \mu + \sigma]$ in the Hot Filter, then we could got that:

$$\frac{N_K}{N} = 1 - \frac{\int_{\mu-\sigma}^{\mu+\sigma} \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}dx}{\int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}dx}$$

Based on well-known conclusions, $\frac{N_K}{N} = 1 - 68.3\% = 0.317$. Therefore, if we choose $c = \frac{2}{3}$, then $k_{H'} \sim 0.654k_H$. More conditions could be calculated by querying the normal distribution integral table.

C.5 Poisson Distribution

The probability mass function is:

$$P(X = k) = \frac{\lambda^k}{k!}e^{-\lambda}$$

For Poisson distribution, When $\lambda > 20$, it can be approximated as a normal distribution. So we just consider a small $\lambda = 10$. Then we select the top ten terms with the highest frequency in the Poisson distribution, which are left in the Hot Filter. From a simple calculation in Matlab, we could find that:

$$\frac{N_K}{N} = 1 - \sum_{k=5}^{14} P(X = k) = \sum_{k=5}^{14}\frac{10^k}{k!}e^{-10} = 0.113$$

Therefore, if we choose $c = \frac{2}{3}$, then $k_{H'} \sim 0.447k_H$.

Summary: The more skewed the data is, the greater the accuracy improvement brought by adding the Hot Filter on top of KLL.

D Improvement of Binary Split Insertion

In Section 3.4, we mention a better way to insert $< x, freq(x) >$. First we convert $freq(x)$ to a binary number $(\overline{a_{H-1}...a_2a_1a_0})_2$. Next, find all i for which a_i is 1, which means we need to insert an x in the $(i + 1)$th layer of the KLL part. For convenience, we rearrange the above i into a sequence in ascending order, represented by $i_1, i_2, ..., i_p \cdots$.

This method can be faster than inserting x consecutively from the bottom layer because it significantly reduces the number of compactions. Besides, it could also improve the accuracy of the quantile estimate. Just consider inserting x one by one. The estimate will be exact if the items in some layers are all the same. If some items in that layer are different from x, however, it is likely to cause errors because the weight of some discarded x may be added to other items.

E Configuration of Each Algorithm

This section details the parameter settings for our C-KLL algorithm and compares them with other sketch algorithms used in our experiments.

C-KLL Sketch:

- **Hot Filter Memory Proportion:** The memory allocation for the Hot Filter is set at a ratio of $ratio = 0.1$.
- **Eviction Threshold λ:** The threshold for replacing items in the Hot Filter is $\lambda = 16$.
- **Item ID size:** The items in the dataset are all double-precision floating point numbers, and we use 64 bits to store them.

- **Counter Size:** Each counter within the algorithm uses a 32-bit size.
- **KLL Parameters:** The KLL component within the C-KLL retains the same settings as the standalone KLL algorithm, where the memory for the i-th level is $c = 1.4142 \approx \sqrt{2}$ times that of the $(i-1)$-th level.

KLL Sketch:

- The standalone KLL algorithm also follows the memory allocation rule described above for its hierarchical levels.

Req-Sketch:

- **Layer Number:** Determined by the total available memory and the number of insertions. This algorithm does not require further parameter adjustments.

GK-Sketch:

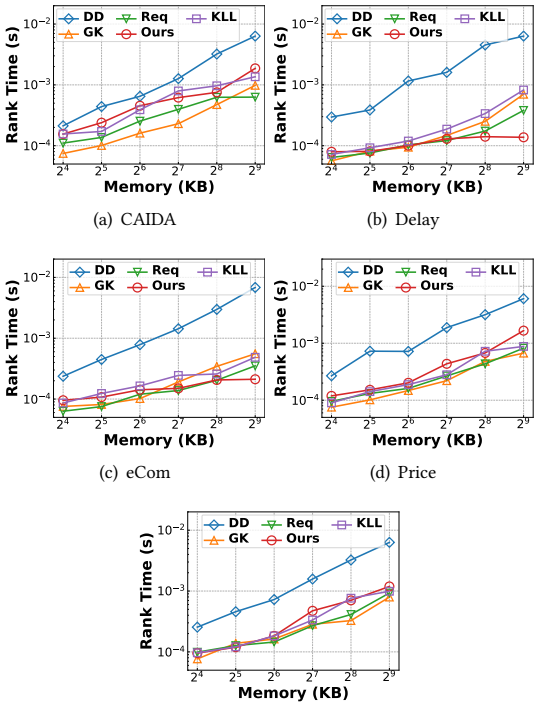
- **Epsilon ϵ:** Predetermined based on the total memory usage desired for each experiment round, ensuring consistent memory allocation across comparisons. Specific values are documented in the accompanying code.

DD-Sketch:

- **Alpha α:** Set based on available memory and insertion volume, with no additional parameters required.

These configurations ensure that each algorithm is optimized for performance while maintaining a fair basis for comparison in our experimental evaluations.

F Results of Query Rank Time



(a) CAIDA (b) Delay

(c) eCom (d) Price

(e) Voltage

Figure 15: Query Rank Time (s) vs. Memory (KB).