Quantile Estimation with Duplicates

TIANRUI XIA, Tsinghua University, China
ZILING CHEN, Tsinghua University, China
SHAOXU SONG*, Tsinghua University, China

Quantile estimation in data streams is a fundamental task in data analysis. Data structures named quantile sketches are constructed to summarize data and estimate quantiles using limited storage. Duplicate data records often occur in real-world data, while there is no relevant design or analysis in existing quantile sketches, including the best known method KLL sketch. In this paper, we present DupliSketch, a quantile sketch with optimizations for duplicates. The sketch compresses arrival duplicates into elements and performs randomized compaction operations to meet the space limit. To better summarize heavy-tailed data, DupliSketch maintains values frequent enough dynamically and counts them separately, while summarizes other values with weighted elements obtained from compactions. Analyses show its error in rank estimation, space complexity to provide error guarantees under input with duplicates, and time complexity. The approach is deployed as a user-defined function in a database Apache IoTDB. Extensive experiments support the theoretical analyses and compare DupliSketch with existing methods on real and synthetic datasets. On average, the error of the DupliSketch is 44% smaller than that of the baseline methods under the same space limit.

CCS Concepts: • **Information systems → Data streams**; • **Theory of computation → Sketching and sampling**; *Data structures and algorithms for data management*.

Additional Key Words and Phrases: Quantile estimation, Data streams, Approximate query processing, Sketch algorithms, Duplicate data

ACM Reference Format:
Tianrui Xia, Ziling Chen, and Shaoxu Song. 2026. Quantile Estimation with Duplicates. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 69 (February 2026), 27 pages. https://doi.org/10.1145/3786683

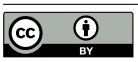
1 Introduction

Computing quantiles of data is a fundamental task in data science. As order statistics, typical quantiles such as the median, the 95th and the 99th percentiles are frequently used in downstream tasks, including data profiling [27, 33, 46], outlier detection [8] and monitoring data [20, 35, 39]. In practice, when monitoring real-time data or analyzing unordered data on disk, data arrive in a one-pass streaming fashion. In this one-pass streaming scenario, quantiles are impossible to compute exactly without loading all data into memory [38], which is often not affordable. Thus, the common solution is to summarize data with a size-limited structure in memory, named quantile sketch or summary [10, 12, 41], to provide estimated ranks and quantiles.

Formally, given a multiset of values $\{x_1, x_2, \ldots, x_N\}$ drawn from the universe U equipped with a total order, the rank of a value x, $R(x)$, is the number of values such that $x_i \leq x$, and the ϕ-quantile $Q(\phi)$ is the value x such that $R(x) = \phi N$. A vast body of work [3, 4, 21, 25, 31, 34], like ours,

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

Authors' Contact Information: Tianrui Xia, Tsinghua University, Haidian Qu, Beijing Shi, China, xiatr24@mails.tsinghua.edu.cn; Ziling Chen, Tsinghua University, Haidian Qu, Beijing Shi, China, chen-zl22@mails.tsinghua.edu.cn; Shaoxu Song, Tsinghua University, Haidian Qu, Beijing Shi, China, sxsong@tsinghua.edu.cn.



ACM 2836-6573/2026/2-ART69
https://doi.org/10.1145/3786683

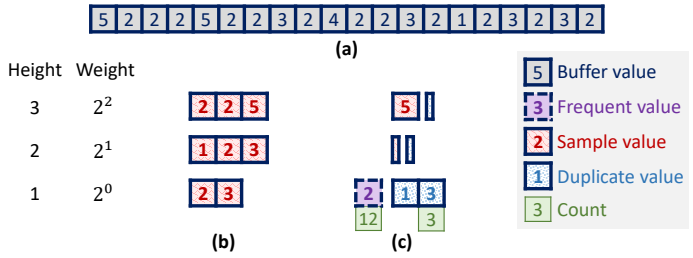


Fig. 1. An overview of summarizing a data stream with duplicates (a). When processed by a standard KLL sketch (b), the high-frequency value '2' is scattered across multiple levels, leading to inaccuracies. In contrast, our proposed *DupliSketch* (c) uses **duplicate compression** for general efficiency and employs **frequent-value tracking** to ensure accuracy for frequent values.

focuses on providing a uniform (additive) ϵN error guarantee: $|R(x, S) - R(x)| \leq \epsilon N$ for any $x \in U$. Research in this area has culminated in the KLL sketch [31], which is widely regarded for its near-optimal space complexity and excellent practical performance [12]. However, when faced with real-world data streams where duplicate values are ubiquitous, KLL's core compaction mechanism reveals significant limitations in both accuracy and efficiency, motivating the need for a new design paradigm.

1.1 Motivation and Intuition

Duplicate records are common in real-world data, arising from sources such as limited-precision sensors, default values, or heavy-tailed distributions inherent in many natural phenomena [24, 30, 32]. For instance, in the Price [16] dataset containing 6.7×10^7 purchase prices, the most frequent 1% of distinct values account for 45% of the total data. Despite this, existing quantile sketches, including KLL [31] and its enhancements like KLL± [45] and ReqSketch [11], do not have specialized designs for handling duplicates and their high frequencies.

To illustrate the core problem, consider the data stream shown in Figure 1(a), where the value '2' is a frequent value. When this stream is processed by a standard KLL sketch, the result is a structure like that depicted in Figure 1(b). The issues with this approach are twofold. First, it leads to redundant storage by storing multiple items with the same value individually. Second, and more critically, KLL's mechanism scatters instances of the value '2' across multiple levels, and the final estimated frequency is derived from high-weight *sample elements*. These samples are themselves lossy summaries of potentially heterogeneous items, introducing significant uncertainty and inaccuracy.

Our work, *DupliSketch*, directly addresses these limitations, as shown in Figure 1(c). It introduces two key ideas. First, to tackle redundant storage, it employs (1) **duplicate compression**, which efficiently represents repeating items (e.g., the three instances of value '3') as a single, compact (value, count) pair. Furthermore, to solve the critical issue of inaccuracy for frequent values, it introduces (2) **frequent-value tracking**. This mechanism identifies the globally frequent value '2', separates its pure instances from the lossy compaction process, and consolidates them into a dedicated, precise counter. This two-pronged strategy results in a more robust and reliable summary that is both space-efficient and accurate for distributions with frequent values.

1.2 Challenge

The primary challenge is to design a sketch that can dynamically distinguish between globally frequent values and locally repeating items without any prior knowledge of the data's duplication

patterns or frequency distribution. While storing the frequency of duplicate values is an intuitive idea, integrating it into a theoretically-guaranteed sketch like KLL presents three non-trivial challenges, which directly correspond to our main contributions:

(1) **Dynamic Frequent-Value Tracking:** Designing a low-overhead dynamic mechanism to accurately identify frequent values and adapt to distribution shifts on top of this new structure.

(2) **Generalized Sketch Maintenance:** Designing a generalized data structure and compaction process to handle `(value, count)` pairs instead of single items, without violating the core principles of KLL's leveled design.

(3) **Rigorous Theoretical Guarantees:** Providing a rigorous theoretical analysis for this complex, two-part system to prove that our innovations lead to provably tighter error and space bounds under duplication.

1.3 Contribution

To tackle the aforesaid challenges with duplicates, our major technique contributions in this study are as follows.

(1) We propose a novel sketch structure that efficiently represents duplicate data. To enable more accurate frequency estimation, we introduce two classes of weighted elements to distinguish between pure duplicates and mixed samples, and generalize the compaction operation to handle them (Section 3.1 and 3.3).

(2) Building upon this structure, we design a dynamic mechanism for frequent-value tracking. This includes methods for estimating global frequencies from the sketch, a dynamic threshold for identifying frequent values, and processes for upgrading or degrading items as the data distribution evolves over time (Section 3.4).

(3) We present the process of maintaining the size-limited sketch including compressing arrival data, performing compaction operations and maintaining frequent values in Sections 3.2, 3.3 and 3.4. The amortized time complexity of the sketch construction in Section 3.5 is shown in Proposition 4.6.

(4) We analyze the error bound and space cost of our sketch. The error bound always benefits from the compact representation of duplicates (Proposition 4.2). Furthermore, under our c-Batched Duplicates model, we provide theoretical results showing how the error bound and space cost improve as the degree of duplication increases (Propositions 4.3 and 4.4).

(5) We deploy the sketch in a time-series database Apache IoTDB [23] and conduct extensive experiments on real-world and synthetic data in Section 5. Experiment results support our theoretical analyses and demonstrate the superiority.

To ensure reproducibility, the code and data for our experiments are publicly available [9]. Furthermore, the core implementation of our proposal has been included in the GitHub repository of the deployed system [28].

2 Preliminary

In the domain of stream processing, quantile estimation is a fundamental task. Due to the inability to store an entire stream in memory, algorithms typically employ a compact data structure known as a "quantile sketch" to summarize the data within a limited space. This sketch is then used to provide high-accuracy approximate answers to quantile queries. Among the many quantile sketch algorithms, the KLL sketch [31] has become a gold standard and a crucial baseline in the field, owing to its strong theoretical guarantees and excellent practical performance. As the design of *DupliSketch* is deeply informed by KLL's architecture, we first review the key concepts of its leveled-compactor framework below.

Table 1. Notations

Symbol	Description
N	Amount of data $\{x_1, ...x_N\}$ in data stream
U	Universe of data
S	The quantile sketch summarizing data
M	Size limit of sketch S
$R(x)$	Actual rank of value x in data
$Q(\phi)$	Value of ϕ-quantile
$R(x, S)$	Estimated rank of value x in sketch S
$Q(\phi, S)$	Estimated ϕ-quantile in sketch S
H	Number of compactors in sketch S
$S_{[h]}$	Compactor at height h in sketch S, where $1 \leq h \leq H$
$S_{[1]}.E_F$	Set of frequent elements
$S_{[h]}.E_D$	Set of duplicate elements in compactor $S_{[h]}$
$S_{[h]}.E_S$	Set of sample elements in compactor $S_{[h]}$, $2 \leq h \leq H$
γ	Capacity decrement factor for compactors
B	Buffer for new arrival data at height 1
w_h	Weight of elements in compactor $S_{[h]}$
m_h	Number of compactions performed at height h
$TotC_h$	Total number of items in compactor $S_{[h]}$

The core data structure of a KLL sketch is a sequence of **leveled compactors**. The entire sketch consists of multiple levels, or compactors, indexed from $h = 1$ to H and denoted as $S_{[1]}, S_{[2]}, \ldots, S_{[H]}$. Each compactor at level h contains a set of items, all of which share the same **weight**, $w_h = 2^{h-1}$. This weight signifies that a single item at level h conceptually represents 2^{h-1} items from the original data stream. This hierarchical, weighted structure is key to KLL's efficiency, as it maintains higher precision for more recent data at lower levels while summarizing vast amounts of older data with less precision at higher levels.

To maintain its fixed memory footprint, KLL periodically performs a **compaction** operation. In practical implementations, new items from the stream are first accumulated in an input **buffer**. When this buffer becomes full, its contents are sorted and merged into the bottom-level compactor, $S_{[1]}$. A compaction is triggered whenever a compactor $S_{[h]}$ subsequently exceeds its capacity limit. The operation takes all items from the full compactor, sorts them, and then culls their number through a randomized and lossy process. A classic implementation of this process is to randomly choose, with equal probability, to keep either all even-indexed items or all odd-indexed items. For instance, after sorting a full compactor of 10 items, the algorithm might keep only the 5 items at odd-numbered positions. The selected subset is then "promoted" to the next level, $S_{[h+1]}$, where its weight is correspondingly doubled.

In summary, KLL's foundational design rests on two key pillars: (1) a hierarchical, weighted structure of leveled compactors to manage data summaries at different granularities, and (2) a randomized compaction process to maintain a fixed memory footprint.

Figure 2 contrasts the standard KLL architecture with that of DupliSketch, visually highlighting our two main new components: (1) the use of compressed (value, count) elements within the leveled compactors, and (2) the addition of a separate list for frequent-value tracking.

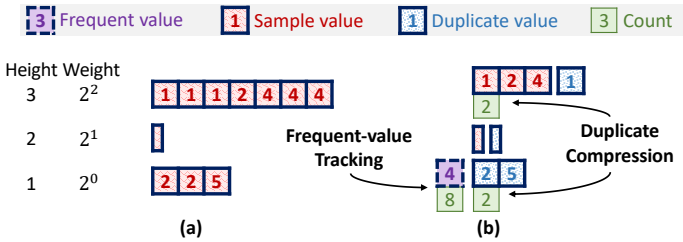


Fig. 2. A comparison of (a) the standard KLL sketch architecture, and (b) our proposed DupliSketch.

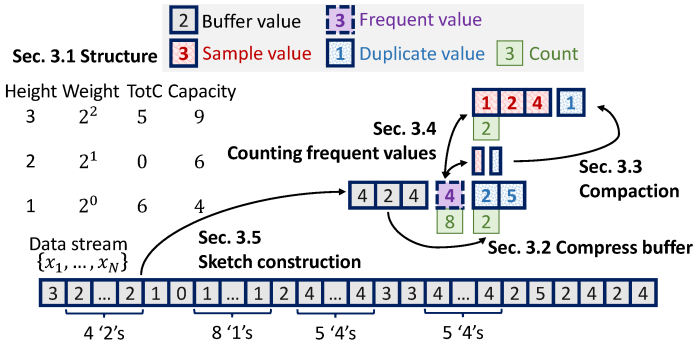


Fig. 3. Overview of techniques in the solution

3 Duplicate Sketch

This section introduces *DupliSketch*, a new quantile sketch specifically designed to efficiently process data streams with a high degree of duplication. At its core, DupliSketch introduces two main innovations: a compact representation for duplicate items and a dynamic tracking mechanism for globally frequent values. Figure 3 gives an overview of the method, summarizing the techniques in the following sections: Section 3.1 shows the sketch structure and definitions, Section 3.2 compresses new arrival data, Section 3.3 performs generalized compactions, Section 3.4 maintains frequent values, and Section 3.5 constructs the full sketch.

3.1 Sketch Structure

Our design builds upon KLL's foundational compaction method. Our key innovation is to generalize this process to operate on `(value, count)` pairs, and handle the creation of both sample and duplicate elements for the next level. Moreover, we introduce a separate mechanism for tracking frequent values.

3.1.1 Compressed Compactor. The architecture of *DupliSketch* is built upon the well-established foundation of leveled sketches [11, 31, 34, 45], as introduced in Section 2. Our sketch S consists of a sequence of H compactors, $S_{[1]} \dots S_{[H]}$, identified by their height. The bottom compactor $S_{[1]}$ is at height 1, and each compactor $S_{[h]}$ contains elements with a weight of $w_h = 2^{h-1}$.

To specifically handle duplicates, DupliSketch departs from the traditional design by incorporating two key features into this structure. First, it employs a compact (value,count) representation for repeating items. Second, it maintains a separate, lossless counter for globally frequent values. These features are formalized in the following definitions.

Fig. 4. Example of (a) a bottom compactor $S_{[1]}$ and (b) a compactor $S_{[h]}$ at some height $h \geq 2$.

Definition 3.1. The bottom compactor is $S_{[1]} = \{E_F, E_D\}$, where the frequent elements $E_F = \{(v_{Fi}, c_{Fi})\}$, $v_{Fi} < v_{Fi+1}$ are (value, count) pairs representing frequent data, $E_D = \{(v_{Di}, c_{Di})\}$, $v_{Di} < v_{Di+1}$ are (value, count) pairs representing non-frequent data.

Non-frequent data in the bottom level will become duplicate elements, after items from the buffer are first grouped by value into pristine `(value, count)` pairs. The frequent elements in $S_{[1]}$ represent data with a frequency known to exceed the threshold of 2^{H-1} . To keep the frequency of data while summarizing them, we introduce two types of element in high-level compactors and define them as follows:

Definition 3.2. The compactor at height $h \geq 2$ is $S_{[h]} = \{E_S, E_D\}$, where the sample elements $E_S = \{(v_{Si}, c_{Si})\}$, $v_{Si} < v_{Si+1}$ and the duplicate elements $E_D = \{(v_{Di}, c_{Di})\}$, $v_{Di} < v_{Di+1}$ are (value, count) pairs.

E_D in Definitions 3.1 and 3.2 are the same as non-frequent elements in $S_{[1]}$ are all duplicate elements with $w_1 = 1$. The two types of elements are the same in weight $w_h = 2^{h-1}$ but different in data representation. A sample element (v_{Si}, c_{Si}) summarizes $c_{Si} \cdot 2^{h-1}$ data whose values may be different, while a duplicate element (v_{Di}, c_{Di}) represents $c_{Di} \cdot 2^{h-1}$ data with the same value v_{Di}. Since sample elements do not help understand the frequency of values, duplicate elements are essential for determining frequent values.

Example 3.3. In DupliSketch, H denotes the total sketch height, and h is a compactor's level index. Figure 4 shows a bottom compactor and a compactor at some height $h \geq 2$. The frequency threshold is $2^{H-1} = 4$. The frequent elements $E_F = \{(3, 5), (4, 6)\}$ represent that the data with values 3 and 4 are known to occur 5 and 6 times, respectively. In Figure 4(b), the sample element $(5, 2)$ summarizes $2 \cdot 2^{h-1}$ data, and the duplicate element $(5, 3)$ represents $3 \cdot 2^{h-1}$ data with the same value of 5. Note that the counts equal to 1 are omitted in the figure, since they are not stored as introduced later.

Each compactor $S_{[h]}$ has a capacity of $\gamma^{H-h}K$, where $\gamma \in (\frac{1}{2}, 1)$ is the decrement factor and K is the capacity of the top compactor. That is, the capacity decreases exponentially at lower heights as that in [11, 31, 45]. Since compactors are compressed, the real number of elements in a compactor is the sum of the counts and is denoted as $TotC_h$. Thus, we set the capacity of the top compactor as $K = \sum_{h=1}^{h=H} TotC_h / \sum_{h=1}^{h=H} \gamma^{H-h}$.

Example 3.4. Figure 3 shows an example of the calculation of $TotC_h$. For the bottom compactor $S_{[1]}$, there are 3 items in the buffer, and $E_D = \{(2, 2), (5, 1)\}$, so $TotC_1 = 3 + 2 + 1 = 6$. Similarly, for the top compactor $S_{[3]}$, $TotC_3 = 2 + 1 + 1 + 1 = 5$.

3.1.2 Storage in Memory. Existing works [1, 29] propose to implement the leveled quantile sketch with a shared array and we extend the technique here. Compared to list-based data structures, a shared array can avoid some constant factor in space [29]. As shown in Figure 5, there is no gap between sub-arrays each representing a compressed compactor.

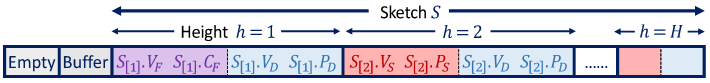


Fig. 5. DupliSketch stored in memory as an array

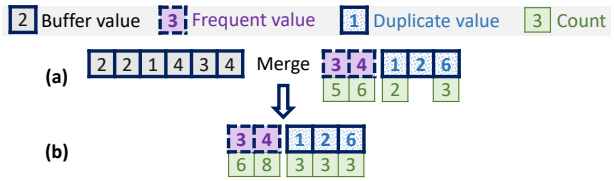


Fig. 6. Example of compressing buffered values into the bottom compressed compactor $S_{[1]}$

Note that we do not use the trivial form of (value, count) pairs to store elements except the frequent elements, since it takes more space than no pairs when the count is 1. Instead, we separately store the values and information about counts that are at least 2. That is, we store a (p_i, c_i) pair to represent that the p_i-th value has a count of $c_i \geq 2$, which is denoted as P_S and P_D in Figure 5. Thus, the proposed technique takes less space than no pairs in all cases.

3.1.3 Rank and Quantile Estimation. Estimating ranks and quantiles in the sketch are straightforward since all the elements are weighted (value, count) pairs. For any value x, its rank is estimated by the sum of the weight of elements with values smaller or equal than x. For example, the estimated rank of value 3 in the compactor $S_{[h]}$ in Figure 4(b) is $R(3, S_{[h]}) = (2+1) \cdot 2^{h-1} = 3 \cdot 2^{h-1}$, since the elements with values 2 and 3 occur $2+1=3$ times in total. The estimated rank of x in the sketch S is simply $R(x, S) = \sum_{h=1}^{h=H} R(x, S_{[h]})$. The estimated ϕ-quantile in S is $Q(\phi, S) = \min\{x | R(x, S) \geq \phi N\}$.

3.2 Compress Data to Compactor

The new arrivals in the input stream should be compressed as elements in $S_{[1]}$ before being summarized further. Inserting a single data takes $O(|S_{[1]}|)$ time in the worst case, therefore, we set a buffer B for new arrivals and compress all buffered data. When the buffer is full, the buffered data are sorted and merged. Our key innovation here is to generalize this merge step to dynamically check if an item belongs to either the frequent elements $S_{[1]}.E_F$ or the duplicate elements $S_{[1]}.E_D$. Since elements are ordered by value, the merge process is similar to a MergeSort and takes $O(|B| + |S_{[1]}|)$ time with temporary space.

Example 3.5. This example illustrates the entire process that transitions the sketch from state (a) to state (b) in Figure 6. It shows an example of compressing the buffered data $\{2, 2, 1, 4, 3, 4\}$ into $S_{[1]}$. The data 3 and two data 4 are found to be frequent and the corresponding counts are updated. Note that the two data 2 change the count of value 2 from 1 to 3, and the sketch does not store a count equal to 1 as in Section 3.1.2, so the corresponding count information is created in $S_{[1]}.E_D$.

As in Algorithm 1, buffered data are sorted and scanned to update $S_{[1]}$. Either $S_{[1]}.C_F$ or $S_{[1]}.V_D$, $S_{[1]}.P_D$ will be updated, depending on whether the data is found to be frequent enough or not. The whole update process in Lines 2 to 5 takes $O(|B| + S_{[1]})$ time, because the data scanning and items in $S_{[1]}$ are all ordered, which is similar to a MergeSort. For the method that sorts the buffer before merging, utilizing a duplicate-friendly sorter like Counting Sort [5], 3-way Quicksort [42], or dual-pivot quicksort [43] like Java's `Arrays.sort`, is beneficial. There is a trade-off that Counting Sort is fast but only works for small ranges of integers, 3-way Quicksort specifically divides the region of equal values, while dual-pivot quicksort utilizes the asymmetry of comparison [43].

Algorithm 1: Compress_Buffer

Input: The bottom compactor $S_{[1]}$, Buffer B
Output: $S_{[1]}$ merged with B

```
sort B;
for data b in B do
    if b ∈ S_[1].V_F then update S_[1].C_F;
    else update S_[1].V_D and S_[1].P_D ;
end
clear Buffer B;
```

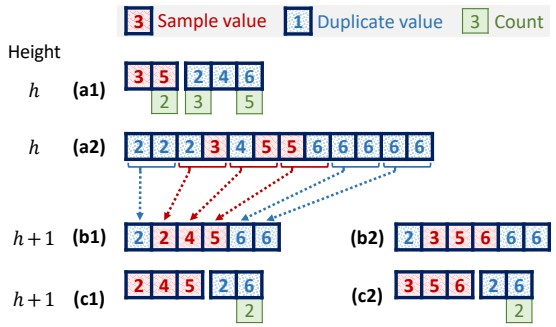


Fig. 7. A randomized compaction performed on $S_{[h]}$ with compressed elements in (a1) equivalent to (a2). It selects and compresses odd-indexed elements as in (b1)(c1) or even-indexed elements as in (b2)(c2) with equal probability.

3.3 Compressed Compaction

This section details the compaction operation, which summarizes elements from level h to $h+1$. Our process builds upon KLL's foundational randomized compaction method. Our key innovation is to generalize this process to operate on `(value, count)` pairs, which requires a mechanism to handle the creation of both sample and duplicate elements at the next level.

The compaction mechanism in DupliSketch generalizes the randomized selection process (introduced in Section 2) to accommodate our two distinct element types. The process still involves selecting either all even-indexed or all odd-indexed items for promotion. However, the type of the resulting element at the next level depends on the items being compacted: a new **duplicate element** is formed only if two duplicate elements with the same value are merged. In all other cases (such as merging a sample element with a duplicate element, or two duplicate elements with different values), a new **sample element** is created to reflect the summarized uncertainty. The resulting elements are then merged into $S_{[h+1]}$. Note that frequent elements stored in $S_{[1]}$ are never involved in this compaction process.

Example 3.6. This example provides a detailed, step-by-step walkthrough of the `CompressCompactor` operation. The selection process of odd/even indexed items in (a2) is the randomized process of keeping either all odd- or all even-indexed items from a sorted list to perform compaction. (b1) and (b2) show the details of keeping odd-indexed and even-indexed uncompressed elements, respectively. Randomness brings two possible results shown in (c1) and (c2) with equal probability.

Algorithm 2: Compaction

Input: A compressed compactor $S_{[h]}$
Output: $S'_{[h+1]}$ to be merged with $S_{[h+1]}$

1 $S'_{[h+1]}$, *index* $\leftarrow \emptyset$, Random(0,1) ;
2 **for** duplicate item (v,c) in $S_{[h]}$ ordered by v **do**
3 **if** $c > 1$ **then**
4 **if** (v,c) is from $S_{[h]}.V_D, S_{[h]}.P_D$ **then** update $S'_{[h+1]}.V_D, S'_{[h+1]}.P_D$ with $(v,\lfloor c/2 \rfloor)$;
5 **else** update $S'_{[h+1]}.V_S, S'_{[h+1]}.P_S$ with $(v,\lfloor c/2 \rfloor)$;
6 **if** c MOD 2 = 0 **then continue**;
7 **end**
8 **if** *index* = 1 **then** update $S'_{[h+1]}.V_S, S'_{[h+1]}.P_S$ with $(v,1)$;
9 *index* $\leftarrow$ *index* XOR 1;
10 **end**

Algorithm 2 shows the randomized compaction in a compactor $S_{[h]}$. In Line 1, *index*=0 or *index*=1 corresponds to keeping even-indexed or odd-indexed uncompressed items, respectively. Line 2 iterates each compressed item with value v occurring c times (c=1 if no count information), except the frequent values in $S_{[1]}.V_F, S_{[1]}.C_F$ since they do not involve in compactions. Lines 3 to 7 deal with duplicate items. Only duplicate items occurred multiple times in $S_{[h]}.V_D, S_{[h]}.P_D$ still represent duplicate data after the compaction. Though there are no sample items in the bottom compactor $S_{[1]}$, sample items at height $h+1$ are generated when compacting two items at height h with different values or types as in Line 8.

3.4 Counting Frequent Values

This section details our dynamic frequent-value tracking system, a core innovation of DupliSketch not present in the standard KLL. As introduced, frequent values are counted separately in $S_{[1]}$ and not influenced by compactions. The advantage is that new arrivals of those values will not introduce any error, while the disadvantage is a fixed space occupation in the sketch. Intuitively, values should be counted when frequent enough, otherwise they should be summarized by the leveled compactors.

3.4.1 Dynamic Threshold of Frequent Values. The first problem is the determination of frequent values. Recall that duplicate elements represent data with the same value, the sketch S provides an approximated frequency of any value x, denoted as $f_D(x,S)$. It is a lower bound of the real frequency according to the compressed compaction. DupliSketch considers value x to be frequent enough when $f_D(x,S)$ is above the threshold. Note that the maximum weight of elements in sketch with height H is 2^{H-1}, when $f_D(x,S) > 2^{H-1}$, replacing all duplicate elements with value x in the compactors with a frequent element in $S_{[1]}$ is more space-efficient. Thus, the threshold is set to be 2^{H-1}, which dynamically grows as more data are inserted into the sketch.

3.4.2 Upgrading Frequent Values. Though the threshold of frequent values is defined, it can be costly to check and upgrade new frequent values during the streaming input. Computing the estimated frequency $f_D(x,S)$ and turning duplicate elements into frequent elements need to scan or move all elements in the sketch. DupliSketch checks all values of duplicate elements to find new

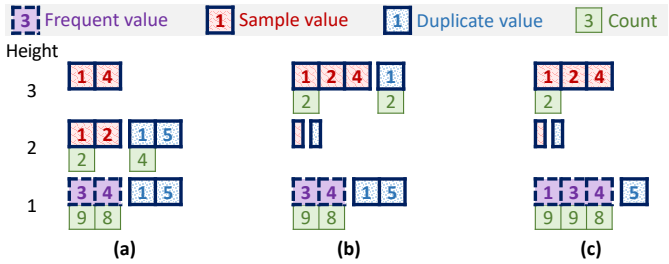


Fig. 8. Upgrading the value 1 with estimated frequency over the threshold 2^{H-1}

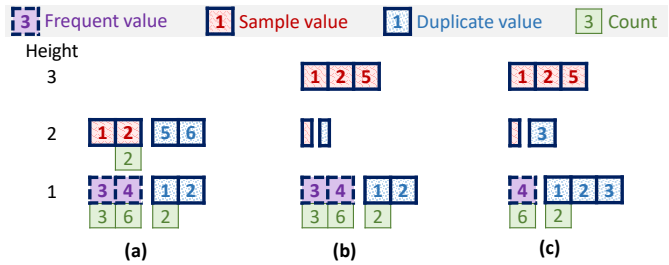


Fig. 9. Degrading the value 3 no longer frequent enough when the threshold 2^{H-1} grows

frequent values with temporary space when a compaction at height $H-1$ is triggered. In this way, the cost of generating new frequent elements is the same as that of the compaction at height $H-1$.

Example 3.7. Figure 8 shows the upgrade of frequent values in the sketch. The sketch in Figure 8(a) triggers a compaction at height $H-1=2$ and becomes Figure 8(b), where the four frequent elements with value 1 in $S_{[2]}$ are summarized as two frequent elements in $S_{[3]}$. After the compaction, the value 1 is found to be frequent enough since $f_D(1,S)= 2\times 2^2+1=9$ exceeds the threshold of $2^{H-1}=4$. Then all duplicate elements with value 1 are removed, and a new frequent element in $S_{[1]}$ is generated as in Figure 8(c).

3.4.3 Degrading Values No Longer Frequent. When the threshold 2^{H-1} grows, some frequent elements in $S_{[1]}$ might be not frequent enough. The threshold of 2^{H-1} changes only when the height of sketch increases, i.e., there is a compaction at height H. When the threshold increases, all frequent elements in $S_{[1]}$ are checked. Values no longer frequent are inserted into leveled compactors according to the binary representation of their counts. Again, the time cost is the same as the compaction at height H.

Example 3.8. Figure 9 shows processing data no longer frequent when the threshold 2^{H-1} grows. The sketch in Figure 9(a) triggers a compaction at height 2 and H increases to 3 in Figure 9(b). Then the value 3 is found to be not frequent enough. As in Figure 9(c), the frequent element is degraded into duplicate elements in $S_{[1]}$ and $S_{[2]}$ because the count is $3=2^0+2^1$.

3.5 DupliSketch Construction

Now we can construct a size-limited sketch from the streaming input with the techniques introduced above. When the new data arrives, it is simply appended to the buffer if there is empty space; otherwise the following process is repeated to create free space. Our process adopts KLL's foundational

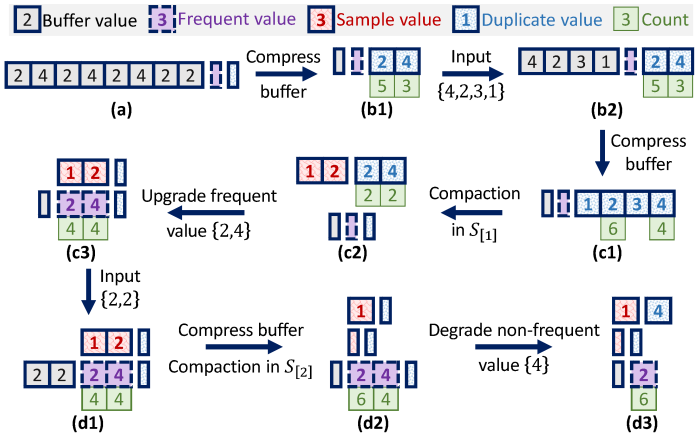


Fig. 10. Example of constructing a sketch with a size limit of 8. Subfigures are snapshots of the sketch. DupliSketch compresses buffer or performs compaction for free space. Data 2 and 4 are found frequent when $H = 2$ (c3). As more input and H becomes 3, 4 is not frequent enough (d3).

framework: appending new data to a buffer and triggering compactions on the lowest full compactor. Our key innovations are the integration of our generalized operations (`Compress_Buffer` and `Compaction`) and, crucially, the dynamic maintenance (upgrading/degrading) of frequent values.

The sketch compresses the buffer B into $S_{[1]}$, and if $|B| \log |B| \geq |S_{[1]}|$, i.e., not many data are compressed, it triggers a compaction. The compaction is performed on the lowest compactor $S_{[h]}$ that exceeds its capacity defined in Section 3.1.1. Note that the process may repeat several times because a single compaction may not create free space due to the (value, count) pairs. During data summation, the sketch maintains frequent values dynamically as in Sections 3.4.2 and 3.4.3.

Example 3.9. Figure 10 shows snapshots of the sketch with limited size $M = 8$ during input data stream. The first 8 data are simply in buffer as in Figure 10(a). The buffer is then compressed in (b1), creating 4 free spaces for the following arrival data $\{4, 2, 3, 1\}$ as in (b2). Then the buffer is compressed in (c1) and a compaction is triggered because the buffer is small, resulting in (c2). After that, the values $\{2, 4\}$ are found and upgraded to frequent elements in $S_{[1]}$ as in (c3). After the input of $\{2, 2\}$, the buffer is compressed and a compaction is triggered at height 2. The height of the sketch increases to $H = 3$ as in (d2) and the value 4 is found to be no longer frequent enough. 4 is degraded to a duplicate element in $S_{[3]}$ in (d3) because its count is $4 = 2^2$.

When the new data arrives, the algorithm simply appends the data to the buffer if there is empty space in memory, otherwise it creates free space by compressing buffered data or performing compactions as in Lines 4...12. In Line 5, $|B| \log |B|$ is the cost of sorting the buffer and $|S_{[1]}|$ is the cost of building the result compactor. This condition acts as an amortization heuristic: a compaction is triggered only if the previous buffer compression was not effective enough in reducing the number of items. Otherwise, the lowest compactor $S_{[h]}$ that exceeds its capacity is found and a compaction at height h is performed. Line 8 upgrades values found to be frequent enough as in Section 3.4.2. Line 11 degrades data with estimated frequency below the threshold as in Section 3.4.3.

Algorithm 3: Sketch_Construction

Input: Data stream X, sketch size limit M
Output: Compressed sketch S

$S_{[1]}, H, Buffer \leftarrow \emptyset, 1, \emptyset$;
for arriving data x in X **do**
  while $|S| \geq M$ **do**
    $B, S_{[1]} \leftarrow |Buffer|$, Compress_Buffer$(S_{[1]}, Buffer)$;
    if $B \log B > |S_{[1]}|$ **then** continue ;
    $h \leftarrow \text{MIN}\{i \mid TotC_i$ exceeds the capacity for $S_{[i]}\}$;
    merge $S_{[h+1]}$ with Compaction$(S_{[h]})$;
    if $h = H - 1$ **then** upgrade new frequent values ;
    if $h = H$ **then**
      $H \leftarrow H + 1$;
      degrade values no longer frequent;
    end
  end
  append x to $Buffer$;
end
$S_{[1]} \leftarrow$ Compress_Buffer$(S_{[1]}, Buffer)$;

4 Performance Analysis

This section provides a rigorous theoretical analysis of DupliSketch, summarizing our main results and proof strategy. Our analysis shows that DupliSketch's error is provably no worse than KLL's (Proposition 4.2), and that its error and space cost improve as data duplication increases (Propositions 4.3 and 4.4). Intuitively, our proofs are based on the insight that our method reduces the number of compactions (the primary source of error). We formalize this by first establishing a tighter upper bound on the number of compactions, and then applying a Chernoff-style tail bound to the resulting sum of random errors to derive the final guarantees.

4.1 Error Bound

To analyze the error in rank estimation $R(x, S) - R(x)$, we first discuss the error introduced by an individual compaction.

Lemma 4.1. *Let the random variable $X_{i,h}$ denote the error introduced by the i-th compressed compaction at height h, there is $X_{i,h} \in \{-2^{h-1}, 0, 2^{h-1}\}, \mathbb{E}[X_{i,h}] = 0, \text{Var}[X_{i,h}] \leq 4^{h-1}$.*

Since upgrading and degrading frequent elements do not introduce any error, the error of the sketch $R(x, S) - R(x)$ is the sum of the errors from individual compactions and is thus sub-Gaussian. A zero-mean variable X with variance σ^2 is sub-Gaussian if $\mathbb{E}[\exp(sX)] \leq \exp(-\frac{1}{2}s^2\sigma^2)$ for any $s \in \mathbb{R}$. We first establish a general proposition that, due to its more efficient representation of duplicates, DupliSketch always provides an error bound that is no worse than that of a standard KLL sketch.

Proposition 4.2. *For any $\delta > 0$ and sketch size limit, let the DupliSketch S guarantee $\Pr[|R(x, S) - R(x)| \geq \epsilon N] \leq \delta$ and the KLL sketch S' guarantee $\Pr[|R(x, S') - R(x)| \geq \epsilon' N] \leq \delta$, there is $\epsilon \leq \epsilon'$.*

To formally analyze how performance scales with the degree of data duplication, we introduce the c-**Batched Duplicates** model. This model serves as a theoretical framework in which we

analyze a stream under the more practical assumption that, within a single processing batch (i.e., the multiset of items arriving between two consecutive compaction events), any given value appears at least c times. Statistics on the Price dataset shows that a vast majority, 70%, appear with $c \geq 4$ (including batched duplicates and frequent items), and the average observed c is 10.

This controlled setting allows us to use c as a parameter to precisely study how the sketch's performance improves as the intensity of this batch-level duplication increases. Intuitively, the sketch's error bounds improve as this degree of duplication, c, increases, as follows.

Proposition 4.3. *Under the c-Batched Duplicates model, for any $\delta > 0$, let the DupliSketch S guarantee $\Pr[|R(x,S) - R(x)| \geq \epsilon N] \leq \delta$ and the KLL sketch S' guarantee $\Pr[|R(x,S') - R(x)| \geq \epsilon' N] \leq \delta$, there is*

$$\epsilon/\epsilon' \leq \begin{cases} \sqrt{1 - \frac{2}{3}(\frac{1}{2\gamma})^{H' - \log c}} & , \log c < H' \\ \sqrt{1/\left(3 \cdot 2^{\log c - H'}\right)} & , \log c \geq H' \end{cases}$$

where $\gamma \in (\frac{1}{2}, 1)$ is capacity decrement factor, H' is the height of S'.

Proposition 4.3 shows that as the degree of batched duplication c grows, the error ratio ϵ/ϵ' decreases. This indicates that DupliSketch's advantage over KLL becomes more significant with a higher intensity of batched duplication. Note that for $1 \leq c \leq 2^{H'}$, the ratio ϵ/ϵ' decreases more slowly than when $c \geq 2^{H'}$, implying the advantage is most pronounced when the degree of batched duplication is substantial.

4.2 Space Cost

The space complexity of the quantile sketches is the required sketch size to provide ϵN error guarantee. Conclusions about the space cost are similar to the previous error analyses. Under the c-Batched Duplicates model, the required space M of DupliSketch decreases as the degree of duplication c grows, and we show below the relationship between M and c.

Proposition 4.4. *Under the c-Batched Duplicates model, for a given stream size and any fixed ϵ, δ, let M be the space bound of DupliSketch to guarantee $\Pr[|R(x,S) - R(x)| \geq \epsilon N] \leq \delta$. Then, there is $M \propto 1/c^{\frac{1}{3}}$.*

4.3 Time Complexity

The proposed techniques like maintaining frequent values may introduce additional time cost. Based on the dynamic threshold of frequent values, the number of times that the frequent elements are updated is as follows.

Lemma 4.5. *In a DupliSketch with height H, the upgrade of new frequent values is executed at most $4H$ times, and the degrade of values no longer frequent is executed at most H times.*

Combining the cost of a single execution of updating frequent elements and conclusions about compactions, the amortized update time of the sketch is determined.

Proposition 4.6. *In the construction of a DupliSketch with $\Pr[|R(x,S) - R(x)| \geq \epsilon N] \leq \delta$, the amortized update time is $O(\log \frac{1}{\epsilon})$.*

5 Experimental Evaluations

In the experiments, we verify the theoretical results and compare our method with other quantile sketches. In brief, baselines include comparison-based sketches (GK sketch [25],KLL sketch [31], Req sketch [11]) and arithmetic-based sketches (DSS [44], t-digest [19], DDSketch [35]) introduced

in Section 6. Code and data are available for reproducibility [9]. All experiments are conducted on a machine with an AMD EPYC 7542 32-Core Processor @ 2.90GHz, 256 GB RAM, running Ubuntu 22.04 LTS. All methods are compared in Apache IoTDB [23].

5.1 System Deployment

The proposed DupliSketch is deployed in an LSM-tree based database, Apache IoTDB [23]. The code is available in the system repository [28] together with the corresponding document [2], maintained by system developers. The method is implemented as a built-in function. The query statement is as follows:

```
SELECT quantile(series0,'M'='64KB','q'='0.5')
FROM root.storagegroup0.device0
WHERE time >= 2025-01-01 00:00:00
```

In the system, the KV data are organized by their keys while quantile queries are posed on unordered values. The query executor applies I/O components to read data from the disk and consumes the data one by one, which is a streaming input for the quantile operator in memory. During the query process, a DupliSketch or any other quantile sketch is maintained in memory to summarize the arriving values with the size limit of M. After all queried data on disk are scanned and consumed, the operator returns the queried quantile estimated by the quantile sketch.

5.2 Datasets

The employed real-world and synthetic datasets are as follows.

• Voltage [6, 17] is a public dataset on Kaggle, recording the voltage of an automotive-typical permanent magnet synchronous motor.
• Price [16] is a public dataset on Kaggle, recording the float prices of purchased products in a large multi-category online store.
• Custom [15] is a public dataset on Kaggle, recording the values of Japan's 100 million customs trade statistics since 1988.
• Batched Duplicates (c): A synthetic dataset where each value appears c times consecutively. This idealized structure is used to create a clean testbed that satisfies the assumptions of our c-Batched Duplicates model regardless of the underlying buffer size, enabling a precise validation of our theory.
• Zipf α is a synthetic dataset consists of i.i.d. values with frequencies following a Zipf distribution with parameter α.

Figure 11 shows probability mass functions (PMF) and Table 2 lists statistics of datasets except for the Batched Duplicates data. These datasets are chosen to map their duplicate rates, skew, and key types to the target scenarios that the techniques in our paper address. For example, the floating-point Price dataset illustrated in Figure 11(b) exhibits a high overall duplicate rate, representing the target scenario of long-tail skew distributions. This makes it an ideal testbed for evaluating our dynamic frequent-value tracking technique (Section 3.4). Similarly, the Voltage dataset in Figure 11(a) provides a case with multiple distinct, frequent-value peaks, while the Custom dataset in Figure 11(d) represents a real-world, highly skewed distribution.

5.3 Baselines

Besides the proposed DupliSketch, other quantile sketches implemented for comparison are as follows:

• DSS is the Dyadic SpaceSaving [44] applying dyadic intervals [13] and SpaceSaving [37]. It estimates quantiles and ranks by summing up frequencies of $O(\log |U|)$ disjoint dyadic intervals.

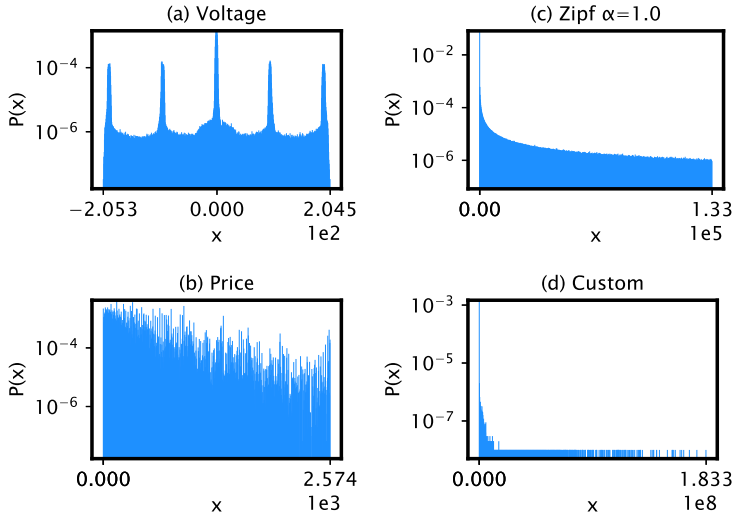


Fig. 11. PMF of datasets

Table 2. Characteristics of datasets

Name	Values	Unique Values	Skewness
Voltage	3.7×10^7	1.9×10^5	-7×10^{-4}
Price	6.7×10^7	6×10^4	1.06
Zipf $\alpha = 1.0$	3×10^7	1.3×10^5	-0.67
Custom	1.1×10^8	1.2×10^6	0.24

- DD is the DDSketch [35] designed for estimating extreme quantiles of heavy-tailed distributions.
- GK is the GK sketch [25], the optimal deterministic comparison-based sketch [14] with additive error guarantees for all quantiles.
- t-digest is the merging t-digest [19] with the scale function k_0 and performing linear interpolation in quantile estimation.
- KLL sketch [31] is the asymptotically optimal randomized sketch providing additive error guarantees for all quantiles.
- Req [11] is a randomized sketch providing multiplicative error guarantees, i.e., being more accurate for extreme quantiles.

5.4 Metrics

To measure the accuracy of quantile sketches, we report the average rank error of estimating 1×10^4 quantiles {0.01%, 0.02%, ..., 99.99%} on different parts of the dataset. To measure efficiency, we execute the query statement several times and report the average time cost.

5.5 Parameter Settings

In our experiments, we evaluate performance under two primary memory configurations tailored to our experimental goals. For the evaluations on real-world datasets, we use a sketch size of M = 128KB. This reflects a practical budget for many applications and is consistent with related

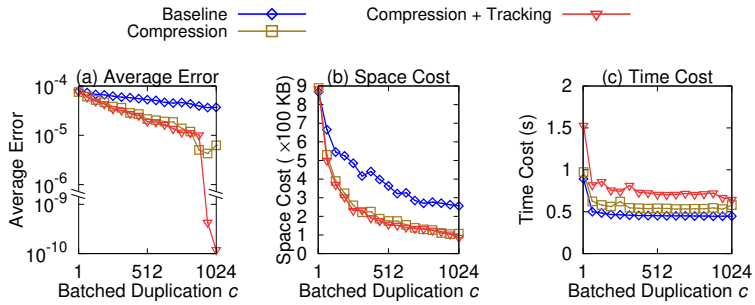


Fig. 12. Verification of theoretical analyses by varying the degree of batched duplication, c.

works [35, 45]. To probe the algorithms' robustness under more extreme conditions, we use a stricter limit of $M = 64$KB for the experiments on synthetic heavy-tailed datasets (i.e., Zipf and Lognormal). For all other parameters, we set a default stream size of $N{=}1 \times 10^7$, and the capacity decrement factor $\gamma = 2/3$ for KLL and DupliSketch, following common practice [1]. All queried data values are double-precision floating-point numbers.

5.6 Verification of Techniques

5.6.1 Ablation Study of Core Mechanisms. To dissect the impact of our two primary contributions, we conduct an ablation study on the synthetic Batched Duplicates (c) dataset, comparing three configurations: Baseline, Compression (with only duplicate compression), and Compression + Tracking (with duplicate compression and frequent-value tracking). The Baseline is DupliSketch with all the new components of the duplicate compression and the frequent-value tracking disabled. Without any new techniques applied, there is no difference on the results to the original KLL, i.e., functionally equivalent.

The results in Figure 12 reveal the distinct, synergistic roles of our innovations and the associated trade-offs. As shown in Figure 12(a) and (b), the duplicate compression provides a foundational improvement, consistently reducing both the average error and the required space cost (for a target error of 10^{-5}) as the duplication factor c increases. Building upon this, the Compression + Tracking configuration achieves a sharp error reduction for $c \geq 960$, suggesting our frequent-value tracking mechanism becomes highly effective once data duplication is substantial.

We also include $c = 1$ in the experiment, i.e., the scenarios without duplicates. It shows the comparable results to KLL, since we do not save `count` when it is 1.

Proposition 4.6 in Section 4.3 states that DupliSketch's amortized update time complexity remains the same as KLL, Figure 12(c) confirms this by showing the gap between the `Compression + Tracking` and `Baseline` lines is a small, proportional overhead.

5.6.2 Verifying Frequent Values. Here we verify the proposed technique of counting frequent values in real datasets. For brevity, we present the results on the Custom dataset, as other datasets exhibited similar trends.

Figure 13 measures the upgrade probability of a value versus its true frequency. It validates that our mechanism in Section 3.4 is effective in correctly identifying the truly frequent items. In Figure 13, the x-axis is the rank, which means that the most frequent values in the queried data are on the left side. The dark blue curve shows the actual counts of values and the purple curve shows their rates of being upgraded as frequent elements. All values with more occurrences than the threshold are displayed in the figure. As shown, values occur more frequently usually have a higher rate to

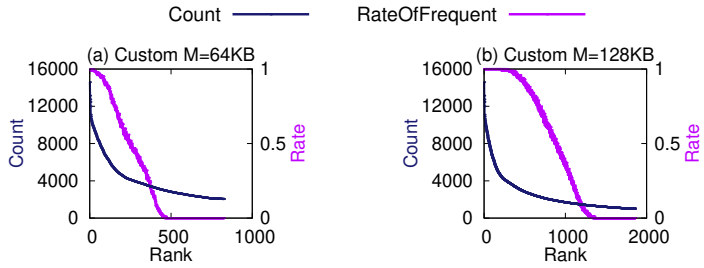


Fig. 13. The count of the most frequent values and the rate of being upgraded

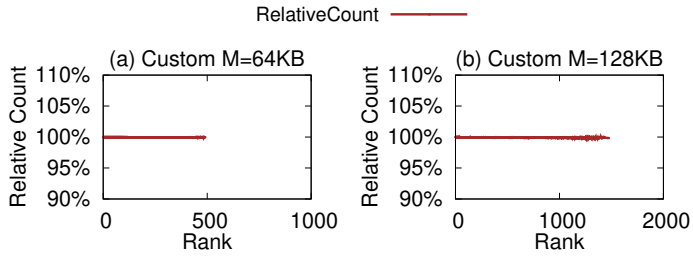


Fig. 14. The count in the upgraded frequent elements of DupliSketch, relative to the actual count.

Table 3. Duplicate rate vs. average error relative reduction

Name	Duplicate rate (↑)	Relative reduction (↓)
Custom	35.9%	-23.5%
Voltage	49.5%	-36.2%
Price	74.9%	-70.8%

be upgraded, but the curve is not strictly monotonic because it can be influenced by concentrated occurrences in the streaming input. As M grows from 64KB to 128KB, the frequency threshold decreases, more values have more occurrences than the frequency threshold, and the values are more likely to be upgraded.

Figure 14 measures the precision of a value's count after it has been upgraded. It validates that our mechanism in Section 3.4 is accurate in tracking those counts, which is crucial for our overall error guarantees. In Figure 14, the x-axis is the rank, and the red curve RelativeCount shows the count in the upgraded frequent elements of the DupliSketch, relative to the actual count of the values. As shown, the results are quite close to 100% except for the values with occurrences slightly above the threshold. Although the values may not be upgraded as just discussed in Figure 13, Figure 14 shows that once a value is upgraded, the upgraded element will have count information very close to the actual count, i.e., it summarizes most of the arriving data with that value.

5.6.3 Impact of Duplication on Real-World Data. To validate that performance gains correlate with data duplication in real-world settings, we define a duplicate rate as the proportion of non-unique items within a processing batch. As shown in Table 3, this rate directly correlates with the relative error reduction of DupliSketch over the baseline KLL. This demonstrates that more duplicates in the dataset lead to better performance improvement over KLL.

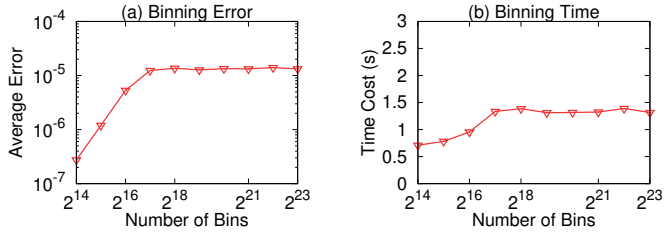


Fig. 15. Varying number of bins

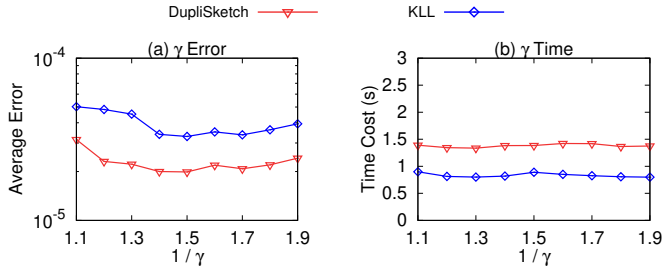


Fig. 16. Varying hyper-parameter γ

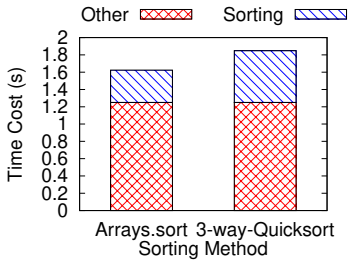


Fig. 17. Varying Sorting

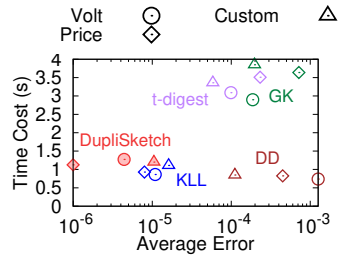


Fig. 18. Tradeoff

5.6.4 Extension with Entity Resolution by Binning. While our current work focuses on exact duplicates, we explore how binning [36] can serve as a practical form of entity resolution where the goal is to identify elements that may not be exactly same but still refer to the same entity. We conduct an experiment analyzing the impact of different numbers of bins on our method's performance using the Voltage dataset as an example, with results shown in Figure 15. As shown, the performance improves with fewer bins due to the increased data duplication by binning. The results are stable with a large number of bins, most different elements are in separate bins.

5.6.5 Sensitivity to Parameter γ. To evaluate the sensitivity to the parameter γ, we conduct an experiment using the Voltage dataset as an example. The results, presented in Figure 16, show that both DupliSketch and the KLL baseline achieve optimal error near the recommended value of $\gamma = 2/3$ ($1/\gamma = 1.5$), while the time cost remains stable. This similar behavior is expected, as duplicate element is treated as multiple items when considering capacity.

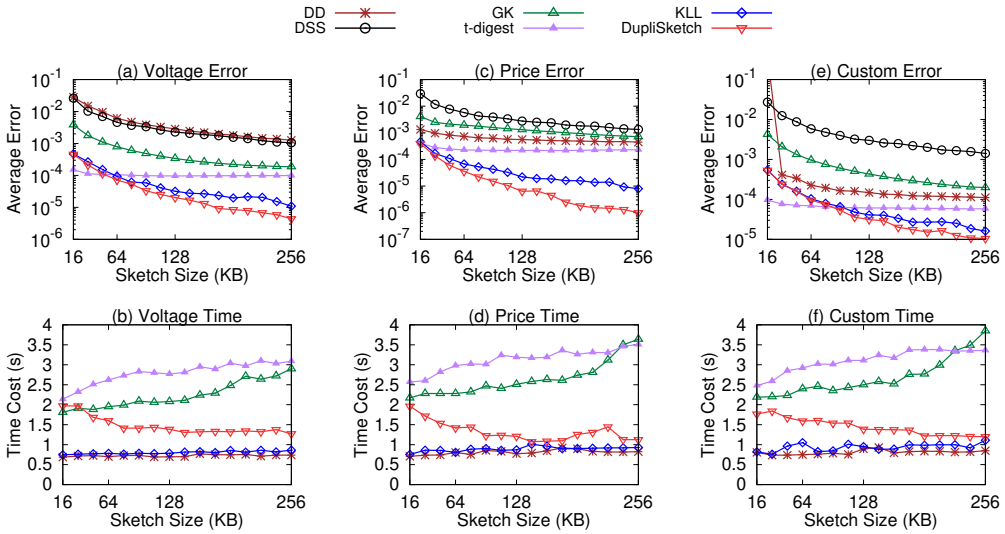


Fig. 19. Comparison by varying sketch size

5.6.6 Analysis of Sorting Cost. To provide an ablation evaluation that analyzes the cost of the buffer sorting step, we conduct a micro-benchmark presented in Figure 17 using the Voltage dataset as an example. The results show that sorting consumes about 20% of the total time. Since the default Java `Arrays.sort` (a dual-pivot quicksort) utilizes the asymmetry of comparison [43], it outperforms 3-way Quicksort.

5.7 Comparison on Real-World Datasets

We now evaluate the performance of DupliSketch against the baselines on three large-scale real-world datasets: Voltage, Price, and Custom. The following subsections present the comparisons in terms of average error and time cost while varying key parameters.

5.7.1 Varying Sketch Size. As an overall result, Figure 18 shows the time-versus-error plot at a memory budget of 256KB on the three real-world datasets, where each point shape represents a different dataset, and each color denotes an algorithm. It visualizes the performance trade-offs that DupliSketch achieves significantly lower error with a slightly higher time cost compared with KLL.

For more detailed insights, Figure 19 reports the results of varying the sketch size from 16KB to 256KB on our real-world datasets. A key finding is that our proposed DupliSketch consistently and significantly outperforms the state-of-the-art KLL sketch across nearly all tested settings, reducing the average rank error by 44%. While DupliSketch achieves the best overall accuracy in medium to large memory configurations, we note a performance trade-off in the small-memory regime ($M \leq 64$KB), where heuristic-based sketches like t-digest can be competitive on certain datasets. This is an expected outcome, reflecting the modest space overhead of DupliSketch's advanced mechanisms, which is more impactful relative to a smaller total memory budget and the relatively longer time required for compaction commensurate with the advanced mechanisms. In terms of efficiency, its time cost is higher than KLL but remains comparable to or faster than other baselines like GK and t-digest.

5.7.2 Varying Queried Data Size. We further evaluate performance as the number of processed data points, N, varies from 1×10^6 to 20×10^6, with results shown in Figure 20. As expected, while

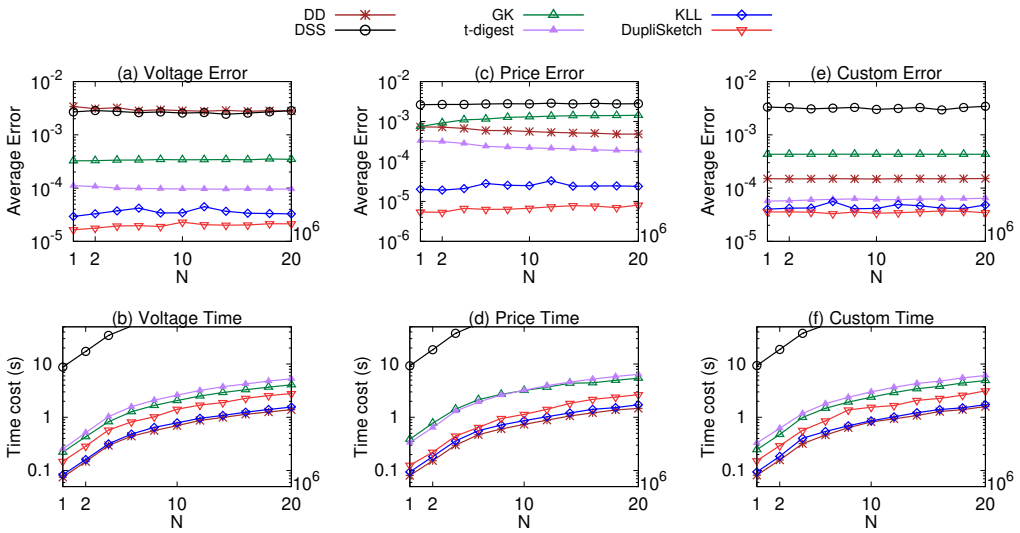


Fig. 20. Comparison by varying queried data size

the total processing time for all methods scales near-linearly with N, the relative accuracy of each sketch remains largely stable. In this setting, DupliSketch consistently demonstrates the best overall accuracy across all datasets. Specifically, compared to the KLL sketch, it reduces the average rank error by 46%. This significant accuracy gain is achieved with a moderate time cost, which is, on average, 43% higher than KLL but remains competitive against other baselines, confirming a favorable accuracy-performance trade-off.

5.7.3 Varying Queried Quantile. This experiment evaluates the performance of different sketches when estimating specific quantiles ranging from 1% to 99%. The results are presented in Figure 21. As expected, the query time for each method remains consistent across different quantiles, so we focus our analysis on the estimation error.

The observed error fluctuation in the figure is an inherent characteristic of quantile sketches. The root cause is the localized nature of compaction error: some value ranges undergo more lossy compactions than others. Similar phenomena have also been reported in the GK paper [25].

We explain how PMF for different datasets impacts the performance and error rate. As the PMF shown in Figure 11(d), the Custom dataset is highly skewed towards smaller values. It explains the results in the quantile query experiment in Figure 21. For low quantiles, <20%, where duplicates are dense, DupliSketch's error is orders of magnitude lower than baselines. On the Price dataset, DupliSketch demonstrates clear superiority across the entire quantile range, due to the high duplication shown in PMF Figure 11(b). On the Voltage dataset, its advantage is most pronounced in the dense middle-range of quantiles (30%-70%), where data duplication is most significant in Figure 11(a). In the sparser tails of this dataset, other heuristic-based sketches like t-digest and DDSketch become more competitive.

However, for quantiles in the upper tail (approximately >50%) of the Custom dataset, arithmetic-based sketches like DDSketch and t-digest, along with the relative-error-focused ReqSketch, begin to show a clear advantage. This suggests that while DupliSketch's optimizations are exceptionally effective for dense data regions, other algorithmic approaches can be more suitable for estimating

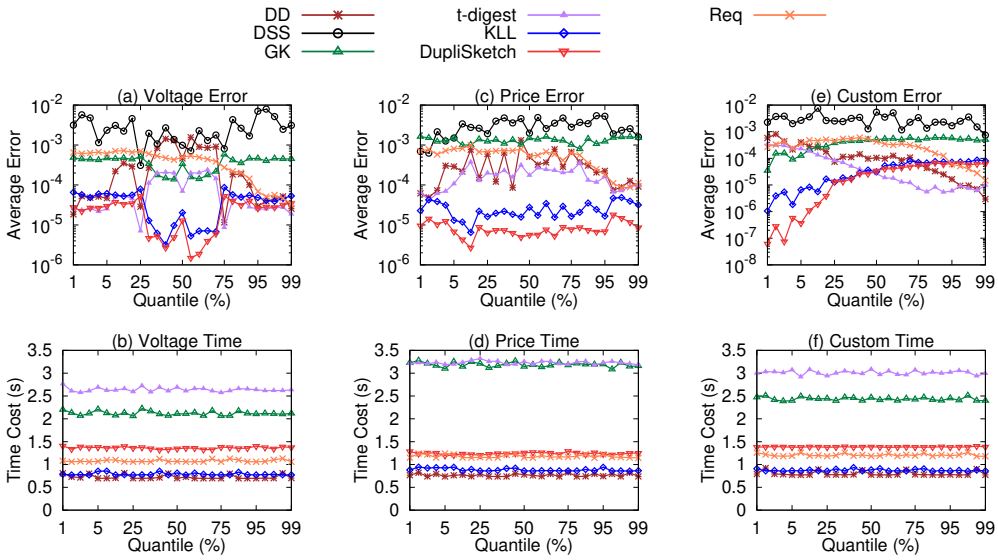


Fig. 21. Comparison by varying queried quantile

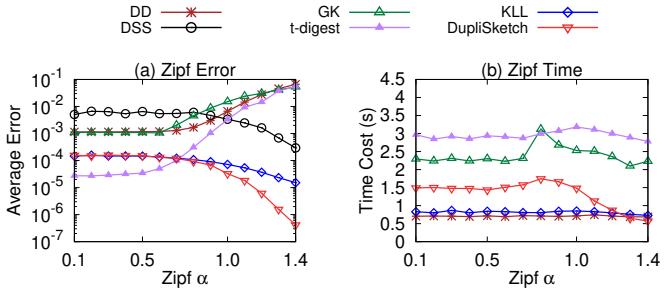


Fig. 22. Comparison by varying Zipf data

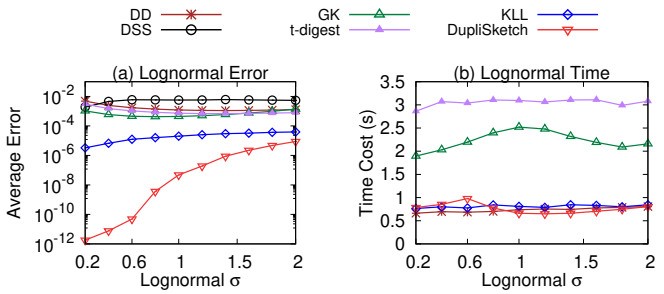


Fig. 23. Comparison by varying Lognormal data

ranks in certain sparse, heavy-tailed distributions. This highlights an important trade-off and the application boundaries for different types of quantile sketches.

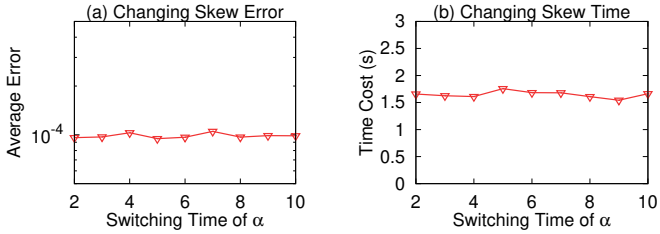


Fig. 24. Stress tests of changing skewness

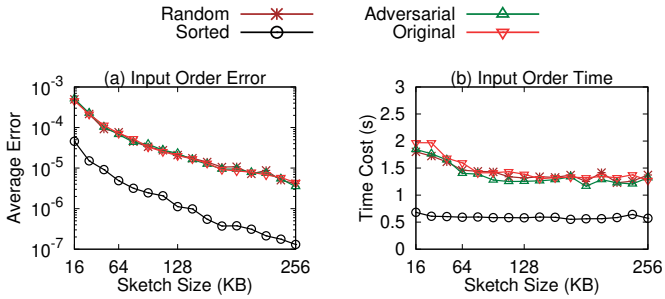


Fig. 25. Stress tests of input order

5.8 Performance on Synthetic Heavy Tails

We conclude our analysis with a focused evaluation on two synthetic datasets designed to model challenging real-world scenarios. Statistical articles [24, 32] show that both Zipf and Lognormal distributions are common models for heavy-tailed data. These experiments serve as stress tests to probe the algorithms' robustness under a particularly challenging combination of skewed data and resource-constrained environments.

Figure 22 shows the results under the varied exponent parameter α in Zipf distributions, generated according to [22]. Note that for small α where the distribution is closer to uniform, t-digest performs best. However, as α grows and the data becomes more skewed, the performance of several baselines degrades significantly. In contrast, DupliSketch and KLL excel in this high-skewness regime, and our method's advantage over KLL becomes particularly pronounced when $\alpha \geq 0.8$.

Figure 23 shows the performance under the varied shape parameter σ in discrete Lognormal distributions with a fixed scale parameter $\mu = 5$, which were generated as described in the statistics literature [7]. As σ grows and duplicates become less frequent, the errors of both KLL and DupliSketch increase as expected. When σ is close to 0, indicating an extremely high degree of duplication, DupliSketch is almost perfectly accurate. Across the entire range, it consistently maintains the lowest error, demonstrating its superiority on this type of data.

5.9 Limitations and Stress Tests

In this section, we analyze DupliSketch's trade-offs under several stress tests.

5.9.1 Rare Duplicates and Changing Skew. We analyze the performance in scenarios with few duplicates. Note that Figure 22 of Zipf varies the duplicate rate, as a larger α means a higher rate of duplication. Indeed, low-α represents a near-uniform distribution. We analyze the corresponding overhead on extra metadata and buffer merging. When duplicates are rare, i.e., low-α, the error

performance is at least no worse than KLL. The time overhead is bit higher due to extra metadata and buffer merging. We also include an experiment with changing skew data (e.g., switching between different α values over time). The results in Figure 24 show the number of skew switches does not impact performance, which is primarily determined by the average α.

5.9.2 Extreme Memory Constraints. Based on the results in Figure 19, we highlight how the relative cost of our metadata becomes more significant in such settings. As shown in the new Figure 19, the 16KB small-budget setting sees a smaller error improvement and a higher time cost compared to the 256KB setting. This is because our metadata overhead consumes a larger proportion of the tiny 16KB budget, and forces more frequent merging. It in turn increases the time cost and diminishes the error improvement.

5.9.3 Varying Input Order. We include a stress test on varying input order (random / sorted / adversarial), noting the general robustness due to randomized compaction but also considering potential adversarial scenarios. The results in Figure 25 verify our technical design on capturing local duplicates (Proposition 4.3), i.e., the `sorted` order yields significantly better performance. It also shows robustness of our method on handling variously distributed duplicates, since the `adversarial` and `random` orders have very similar performance as the original dataset order.

6 Related Work

The problem studied in this paper is the estimation of quantiles over a single, insertion-only data stream. Related works can be broadly categorized as follows.

6.1 Global Quantile Sketches

This body of work focuses on summarizing an entire stream to answer queries about the global distribution of values.

6.1.1 Comparison-based Sketches. This family of algorithms operates solely by comparing input data items and analyzes the error of estimated ranks. The problem of quantile computation with limited memory was first studied in [38]. Manku et al. [34] established the foundation for leveled sketches, proposing the MRL sketch. Greenwald and Khanna [25] later introduced the GK sketch, which was proven to be asymptotically optimal for deterministic comparison-based sketches [14]. Agarwal et al. [3] introduced randomized compaction, which forms the basis for modern probabilistic sketches. The KLL sketch [31] integrated these techniques, achieving optimal asymptotic space complexity. Practical implementations are available in libraries like Apache DataSketches [1]. Subsequently, ReqSketch [11] extended this line of work to provide relative error guarantees, making it particularly strong for tail quantiles. The conclusion of our experiment in Figure 21 is that while ReqSketch excels at tail quantiles due to its multiplicative error model, our method achieves lower average error across the full range.

6.1.2 Arithmetic-based Sketches. To improve practical performance, another line of work incorporates arithmetic operations. One family of methods, including Dyadic Count-Min [13] and Dyadic SpaceSaving (DSS) [44], estimates ranks by summing frequencies over dyadic intervals of the data universe, which naturally benefits from duplicate values. Other popular heuristic-based methods include the t-digest [19], which uses averaging and clustering, and DDSketch [35], which uses logarithmic mapping to provide better accuracy for tail quantiles. These methods are also included as important baselines.

6.2 Per-Key Quantile Estimation

A different but related problem is per-key (or per-flow) quantile estimation, where the stream consists of (key, value) pairs and the goal is to query quantiles for values associated with a specific key. Recent works in this area, such as M4 [18] and SQUAD [40], primarily focus on identifying frequent keys to perform this per-key analysis. This is a key distinction from our work, as DupliSketch focuses on identifying frequent values to improve global quantile estimation. The techniques are therefore not directly interchangeable and are instead orthogonal and complementary to our approach.

6.3 Orthogonal Works and Different Models

Other related research explores different problem dimensions or computational models. For instance, KLL± [45] extends the KLL sketch to handle data streams with both insertions and deletions, a problem orthogonal to ours. Its original paper confirms that it can keep the errors to the same level compare with errors from KLL with no deletions. Therefore, KLL is the more direct and canonical baseline for our setting.

Furthermore, a recent theoretical breakthrough by Gupta et al. [26] presented a deterministic sketch achieving $O(1/\epsilon)$ space. This significant result is achieved by operating in the highly-specialized word-RAM model, where algorithms can leverage the bitwise representation of integer data. In contrast, DupliSketch and the baselines we compare against (including arithmetic-based ones) operate on more general data types such as floating-point numbers. Due to these fundamental differences in the underlying computational model and data-type assumptions, a direct comparison is not applicable in our context.

7 Conclusion

Quantile estimation in data with duplicates is common in real-world systems. Existing quantile sketches like KLL do not handle well such duplicates with redundant items. In this paper, we propose DupliSketch to improve the performance with duplicates by better representing duplicate elements and frequent values. To our best knowledge, this is the first work to provide practical designs and analyses on utilizing duplicates in the quantile sketch. In DupliSketch, the basic building blocks, named compressed compactors, store distinct values and their multiple occurrences. DupliSketch counts frequent values separately to further improve the performance on heavy-tailed data. We introduce two types of elements to estimate frequency and design a mechanism to update the set of frequent values during streaming input. Furthermore, under our c-Batched Duplicates model, analyses on error bound and space complexity show how DupliSketch benefit from duplicates (Propositions 4.3 and 4.4). The amortized update time is provided. The proposed method is deployed in a database Apache IoTDB and compared with existing algorithms. Experiment results support the theoretical analyses and demonstrate the superiority of our proposal.

Acknowledgments

This work is supported in part by the National Key Research and Development Plan (2025ZD1601701, 2024YFB3311901, 2021YFB3300500), the National Natural Science Foundation of China (62232005, 92267203, 62021002), State Grid Ningxia Electric Power Co. Science and Technology Project (SGNXYX00SCJS2400058), Beijing National Research Center for Information Science and Technology (BNR2025RC01011), and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

References

[1] 2025. https://datasketches.apache.org/.

[2] 2025. https://iotdb.apache.org/UserGuide/latest/SQL-Manual/UDF-Libraries_apache.html#_3-12-quantile.

[3] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *ACM Trans. Database Syst.* 38, 4 (2013), 26. doi:10.1145/2500128

[4] Arvind Arasu and Gurmeet Singh Manku. 2004. Approximate Counts and Quantiles over Sliding Windows. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, Catriel Beeri and Alin Deutsch (Eds.). ACM, 286–296. doi:10.1145/1055558.1055598

[5] Paul E Black. 1998. Dictionary of algorithms and data structures. (1998).

[6] Anian Brosch, Oliver Wallscheid, and Joachim Böcker. 2021. Torque and Inductances Estimation for Finite Model Predictive Control of Highly Utilized Permanent Magnet Synchronous Motors. *IEEE Trans. Ind. Informatics* 17, 12 (2021), 8080–8091. doi:10.1109/TII.2021.3060469

[7] Subrata Chakraborty. 2015. Generating discrete analogues of continuous probability distributions-A survey of methods and constructions. *Journal of Statistical Distributions and Applications* 2, 1 (Aug 2015). doi:10.1186/s40488-015-0028-6

[8] Zhiwei Chen, Shaoxu Song, Ziheng Wei, Jingyun Fang, and Jiang Long. 2021. Approximating Median Absolute Deviation with Bounded Error. *Proc. VLDB Endow.* 14, 11 (2021), 2114–2126. doi:10.14778/3476249.3476266

[9] Experiment Code and Data. 2025. https://github.com/Mr-Spade/DupliSketch.

[10] Graham Cormode. 2021. Current Trends in Data Summaries. *SIGMOD Rec.* 50, 4 (2021), 6–15. doi:10.1145/3516431.3516433

[11] Graham Cormode, Zohar S. Karnin, Edo Liberty, Justin Thaler, and Pavel Veselý. 2021. Relative Error Streaming Quantiles. In *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021*, Leonid Libkin, Reinhard Pichler, and Paolo Guagliardo (Eds.). ACM, 96–108. doi:10.1145/3452021.3458323

[12] Graham Cormode, Abhinav Mishra, Joseph Ross, and Pavel Veselý. 2021. Theory meets Practice at the Median: A Worst Case Comparison of Relative Error Quantile Algorithms. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, Feida Zhu, Beng Chin Ooi, and Chunyan Miao (Eds.). ACM, 2722–2731. doi:10.1145/3447548.3467152

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

[14] Graham Cormode and Pavel Veselý. 2020. A Tight Lower Bound for Comparison-Based Quantile Summaries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, Dan Suciu, Yufei Tao, and Zhewei Wei (Eds.). ACM, 81–93. doi:10.1145/3375395.3387650

[15] Custom Dataset. 2021. https://www.kaggle.com/datasets/zanjibar/100-million-data-csv.

[16] Price Dataset. 2019. https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store.

[17] Voltage Dataset. 2020. https://www.kaggle.com/datasets/graxlmaxl/identifying-the-physics-behind-an-electric-motor.

[18] 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 *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 4787–4800. doi:10.1109/ICDE60146.2024.00364

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

[20] Ted Dunning and Otmar Ertl. 2019. Computing Extremely Accurate Quantiles Using t-Digests. *CoRR* abs/1902.04023 (2019). arXiv:1902.04023 http://arxiv.org/abs/1902.04023

[21] David Felber and Rafail Ostrovsky. 2015. A Randomized Online Quantile Summary in O(1/epsilon * log(1/epsilon)) Words. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2015, August 24-26, 2015, Princeton, NJ, USA (LIPIcs, Vol. 40)*, Naveen Garg, Klaus Jansen, Anup Rao, and José D. P. Rolim (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 775–785. doi:10.4230/LIPICS.APPROX-RANDOM.2015.775

[22] Apache Software Foundation. 2016. ZipfDistribution (Apache Commons Math 3.6.1 API). https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/distribution/ZipfDistribution.html.

[23] Apache Software Foundation. 2025. Apache IoTDB. https://iotdb.apache.org/.

[24] Neil Zhenqiang Gong, Wenchang Xu, Ling Huang, Prateek Mittal, Emil Stefanov, Vyas Sekar, and Dawn Song. 2012. Evolution of social-attribute networks: measurements, modeling, and implications using google+. In *Proceedings of the 12th ACM SIGCOMM Internet Measurement Conference, IMC '12, Boston, MA, USA, November 14-16, 2012*, John W. Byers, Jim Kurose, Ratul Mahajan, and Alex C. Snoeren (Eds.). ACM, 131–144. doi:10.1145/2398776.2398792

[25] Michael Greenwald and Sanjeev Khanna. 2001. Space-Efficient Online Computation of Quantile Summaries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, Sharad Mehrotra and Timos K. Sellis (Eds.). ACM, 58–66. doi:10.1145/375663.375670

[26] Meghal Gupta, Mihir Singhal, and Hongxun Wu. 2024. Optimal Quantile Estimation: Beyond the Comparison Model. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*. IEEE, 1137–1158. doi:10.1109/FOCS61266.2024.00075

[27] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711. doi:10.14778/2367502.2367510

[28] Implementation in Apache IoTDB. 2025. https://github.com/Mr-Spade/iotdb/tree/research/dupli-quantile.

[29] Nikita Ivkin, Edo Liberty, Kevin J. Lang, Zohar S. Karnin, and Vladimir Braverman. 2022. Streaming Quantiles Algorithms with Small Space and Update Time. *Sensors* 22, 24 (2022), 9612. doi:10.3390/S22249612

[30] Lianbao Jin, Na Lei, Zhongxuan Luo, Jin Wu, Chao Ai, and Xianfeng Gu. 2025. Semi-Discrete Optimal Transport for Long-Tailed Classification. *J. Comput. Sci. Technol.* 40, 1 (2025), 252–266. doi:10.1007/S11390-023-3086-0

[31] Zohar S. Karnin, Kevin J. Lang, and Edo Liberty. 2016. Optimal Quantile Approximation in Streams. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, Irit Dinur (Ed.). IEEE Computer Society, 71–78. doi:10.1109/FOCS.2016.17

[32] Illenin O. Kondo, Logan T. Lewis, and Andrea Stella. 2023. Heavy tailed but not Zipf: Firm and establishment size in the United States. *Journal of Applied Econometrics* 38, 5 (Apr 2023), 767–785. doi:10.1002/jae.2976

[33] Yuanzhen Li, Shengjie Zheng, Zi-Xin Tan, Tuo Cao, Fei Luo, and Chunxia Xiao. 2023. Self-Supervised Monocular Depth Estimation by Digging into Uncertainty Quantification. *J. Comput. Sci. Technol.* 38, 3 (2023), 510–525. doi:10.1007/S11390-023-3088-Y

[34] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. 1999. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh (Eds.). ACM Press, 251–262. doi:10.1145/304182.304204

[35] Charles Masson, Jee E. Rim, and Homin K. Lee. 2019. DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees. *Proc. VLDB Endow.* 12, 12 (2019), 2195–2205. doi:10.14778/3352063.3352135

[36] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*, Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa (Eds.). ACM, 169–178. doi:10.1145/347090.347123

[37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3363)*, Thomas Eiter and Leonid Libkin (Eds.). Springer, 398–412. doi:10.1007/978-3-540-30570-5_27

[38] J. Ian Munro and Mike Paterson. 1980. Selection and Sorting with Limited Storage. *Theor. Comput. Sci.* 12 (1980), 315–323. doi:10.1016/0304-3975(80)90061-4

[39] Feifan Pu and Jianqiu Xu. 2025. TSQ: An Optimized Framework for Efficiently Answering Time Series Queries. *Data Sci. Eng.* 10, 2 (2025), 296–313. doi:10.1007/S41019-024-00273-8

[40] Rana Shahout, Roy Friedman, and Ran Ben Basat. 2023. Together is Better: Heavy Hitters Quantile Estimation. *Proc. ACM Manag. Data* 1, 1 (2023), 83:1–83:25. doi:10.1145/3588937

[41] Daniel Ting, Jonathan Malkin, and Lee Rhodes. 2020. Data Sketching for Real Time Analytics: Theory and Practice. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 3567–3568. doi:10.1145/3394486.3406480

[42] Sebastian Wild. 2018. Quicksort Is Optimal For Many Equal Keys. In *Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2018, New Orleans, LA, USA, January 8-9, 2018*, Markus E. Nebel and Stephan G. Wagner (Eds.). SIAM, 8–22. doi:10.1137/1.9781611975062.2

[43] Sebastian Wild and Markus E. Nebel. 2012. Average Case Analysis of Java 7's Dual Pivot Quicksort. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7501)*, Leah Epstein and Paolo Ferragina (Eds.). Springer, 825–836. doi:10.1007/978-3-642-33090-2_71

[44] Fuheng Zhao, Divy Agrawal, Amr El Abbadi, and Ahmed Metwally. 2022. SpaceSaving± An Optimal Algorithm for Frequency Estimation and Frequent items in the Bounded Deletion Model. *Proc. VLDB Endow.* 15, 6 (2022), 1215–1227. doi:10.14778/3514061.3514068

[45] Fuheng Zhao, Sujaya Maiyya, Ryan Weiner, Divy Agrawal, and Amr El Abbadi. 2021. KLL±: Approximate Quantile Sketches over Dynamic Datasets. *Proc. VLDB Endow.* 14, 7 (2021), 1215–1227. doi:10.14778/3450980.3450990

[46] Kangfei Zhao, Zongyan He, Jeffrey Xu Yu, and Yu Rong. 2023. Learning with Small Data: Subgraph Counting Queries. *Data Sci. Eng.* 8, 3 (2023), 292–305. doi:10.1007/S41019-023-00223-W

Received July 2025; revised October 2025; accepted November 2025