Online Detection of Outstanding Quantiles with QuantileFilter

Yuhan Wu*†, Aomufei Yuan*‡, Zhouran Shi*, Yuanpeng Li*, Yikai Zhao*, Peiqing Chen§, Tong Yang*†, and Bin Cui*
*Peking University, Beijing, China, †Peng Cheng Laboratory, Shenzhen, China,
‡University of Edinburgh, Edinburgh, United Kingdom, §University of Maryland College Park, Maryland, United States

***Abstract*—In quantile estimation within a stream of key-value pairs, recent work has made significant progress in query flexibility, supporting quantile estimation for any key using a unified statistical structure. However, despite this flexibility, their query speed falls behind, unable to match the high speed of online data insertion. This "offline query + online insertion" model is not ideal for online quantile estimation. Our goal is to online detect keys whose quantiles exceed a user-queried threshold in real-time, such as identifying the user whose 95% latency exceeds 200ms in network data. These keys, termed "Quantile-Outstanding Keys," are vital for anomaly detection in streaming data. In this paper, we propose QuantileFilter, the first approximate algorithm specifically designed for detecting quantile-outstanding keys. QuantileFilter overcomes existing limitations by 1) enabling fast online computation, capable of handling streaming data in real-time with a constant processing time for each data item, accelerating the state-of-the-art (SOTA) by $10 \sim 100$ times, and 2) maintaining high space efficiency, saving $50 \sim 500$ times storage space compared to the SOTA while maintaining the same accuracy. All associated code is available on GitHub.**

I. Introduction

In data stream analysis, where each item presents itself as a key-value pair, quantile estimation [1] is a vital function applied in various fields, including network management [2], [3], security [4], [5], fault detection [6] and more [7]. Recent research has made significant progress in query capabilities. They allow querying arbitrary quantile (*e.g.*, 95%, 99%) of arbitrary key using a single data structure, eliminating the need to maintain separate structures for each key. For instance, in a network data stream, it's possible to query a user's 95% communication latency. Existing solutions typically follow a "online insertion + offline query" mode, enabling fast data insertion and slow estimation (*i.e.*, approximation) of answers upon query requests. However, the substantial difference in query and insertion speeds limits their application range. A "online insertion + online query" approach is more valuable, especially in anomaly detection scenarios. For example, monitors need to immediately identify the user (key) whose 95% latency (value) exceeds 200ms, if such a user exists. To address this, existing solutions rely on slow estimation of latencies over a large number of keys, then compare these with 200ms. This approach drastically slows down the algorithm to an offline pace. Hence, our research focuses on online estimation:

Problem: Online Detection of Quantile-Outstanding Keys. In a data stream composed of key-value pairs, picking out the keys whose value δ-quantiles[1] go beyond a threshold T in real-time, where δ and T are pre-defined. These keys are identified as "Quantile-Outstanding Keys" or outstanding keys, and the value quantiles that exceeds T are identified as "abnormal quantiles". We illustrate an example in Figure 1, and this functionality can be described with the following SQL statement:

```
SELECT key
FROM Key_Value_Stream
GROUP BY key
HAVING QUANTILE(value_set, delta) >= T
```

Detecting outstanding keys is useful and straightforward in many applications:

- In network tail latency monitoring, network monitors observe communication latencies in real-time, identifying outstanding user keys whose quantiles suggest latency anomalies (*e.g.*, 99% latency >200ms in SLA) and subsequently localize the faults [8].
- In system performance monitoring, monitors routinely check vital metrics like CPU usage, memory consumption, and wireless device channel use [9], [10]. If a CPU reaches 99% utilization for 50% of the time during what should be a light load period, it indicates a 0.5-quantile anomaly, suggesting a potential problem [11].
- In sensor data analytics [12], [13], devices located at the network edge, such as sensors, generate large volumes of stream data. When quantile anomalies occur, it signifies events that merit attention, such as the presence of animals triggering a sensor's response.

Within these contexts, detecting outstanding keys has two main requirements: **[R1] Fast online computation.** Traditional approaches are slow in querying, limiting the number of queries monitors can perform. Consequently, they often sample data less frequently, potentially missing brief anomalies or delaying warnings [11]. **[R2] Efficient space usage.** The solution should be as space-efficient as possible to fit within the limited storage capacities of network and sensor equipment.

Corresponding author: Tong Yang (yangtongemail@gmail.com).

[1]The complete formal definition can be found in Section II-A.

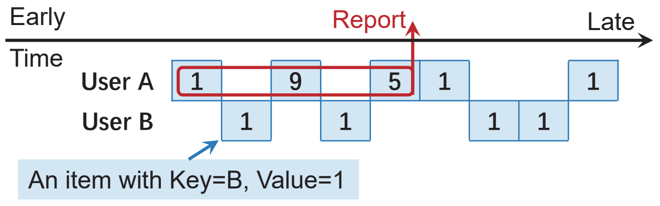


Fig. 1: An example containing two users, where each square represents an item, and the number on it signifies its value. Considering $\delta = 0.5$ and a value threshold $T = 3$. When user A's third item arrives, its δ-quantile, which is the second highest number in the value set $\{1, 5, 9\}$, is 5. This exceeds T, leading to reporting user A as an outstanding key. At the same moment, user B's value set is $\{1, 1\}$, not meeting the criteria for reporting.

Existing solutions are hindered by their slow offline queries and lack of space efficiency. They generally fit into two categories: the holistic approach and the per-key approach. The holistic approach treats the data stream as a single entity, without differentiating between keys. Examples of this method include algorithms like GK [14], KLL [15], [16], tdigest [17], and DD [18], which have been adapted for various scenarios. However, they require setting up a separate unit for every possible incoming key, leading to intolerable storage demands. On the other hand, the per-key approach calculates quantiles for each distinct key using a specific data structure. This granular approach has garnered significant research interest, as evidenced by three recent papers: SQUAD [19], Sketch-Polymer [20], and Histsketch [21]. Unfortunately, their can only estimate quantiles for a user-given key, which requires users to run complex and time-consuming offline queries for all keys. Here, offline query refers to queries where the query time cannot be considered a small constant[2]. Furthermore, they preserve statistical data for any δ, enabling support for flexible quantile queries. However, this is unnecessary for the online scenarios we focus on. First, we do not need excessive flexibility, as the δ for queries will not change rapidly for every item. Our solution will allow slower adjustments of δ. Second, this approach leads to space wastage due to the quantiles that are not of our interest.

In this paper, we propose the first estimation algorithm for online detection of quantile-outstanding key, named QuantileFilter. Its advantages are: **[R1] Fast online computation:** QuantileFilter processes each data item in constant time, and its efficient space enables quicker operations using fast storage mediums like CPU caches or SRAM on network switches and FPGAs. **[R2] Efficient space usage:** QuantileFilter saves up to 500 times space compared to SOTA solutions with the same high accuracy.

Technique I. For fast online computation, we integrate the insert of incoming data and the query. To this end, we devised a weight, Qweight, dedicated to assessing outstanding keys, transforming the process of quantile comparison into one of Qweight comparison, which is more suitable for real-time computation.

Technique 2. For efficient space usage, we devise a compact and high-accuracy sketch[3] to estimate Qweight, achieving efficient outstanding key report. Qweight significantly differs from the traditional statistical targets of sketches, such as the frequency of keys, because it can often be negative, whereas most sketches are designed only for positive statistics. Therefore, existing sketches are either inapplicable for Qweight statistics or, if forced into service, yield low accuracy. In response, we proposed a dual-part sketch technique: one structure for the vague filtering of potential outstanding keys, termed the vague part, and another for the precise tracking of candidate outstanding keys' Qweights, called the candidate part. Our replacement strategy endeavors to admit elements with positive, larger Qweights into the candidate part, while retaining negative, smaller Qweights within the vague part.

II. Preliminary

In this section, we first formalize the target problem and discuss our rationale. Subsequently, we introduce the advancements of related work.

A. Problem Formulation

Definition 1. (Stream Model [26]). In a key-value data stream $S = \{\langle x_1, v_1\rangle, \langle x_2, v_2\rangle, \dots\}$, we handle a continuous arrival of data items, each appearing as a pair of a key and a value. Items with the same key are grouped into a sequence, which for a specific key x is noted as $S_x = \{\langle x_i, v_i\rangle : x_i = x\}$, this includes all pairs where the key equals x. Additionally, the count of items in S_x, known as the frequency of x, is denoted by n, and the collection $V_x = \{v_i : \langle x_i, v_i\rangle \in S_x\}$ is referred to as the multi-set of values for x.

Definition 2. (δ-Quantile [1]). For a key x with a frequency of n, its values can be sorted in non-decreasing order, i.e., $v_0 \leq v_1 \leq \dots \leq v_{n-1}$. The δ-quantile of x is defined as the item at index $\lfloor \delta \cdot n \rfloor$, denoted as $q^x_\delta = v_{\lfloor \delta \cdot n \rfloor}$, where $\lfloor \cdot \rfloor$ denotes the floor function, and σ is bounded within interval $[0, 1)$.

Approximation algorithms allow for some error [1], so we introduce an allowable rank deviation ϵ based on the δ-quantile, leading to the definition of the (ϵ, δ)-Quantile:

Definition 3. ((ϵ, δ)-Quantile). The (ϵ, δ)-Quantile for a set of n values $v_0 \leq v_1 \leq \dots \leq v_{(n-1)}$ is the value at index $\lfloor \delta \cdot n - \epsilon \rfloor$, denoted as $q^x_{\epsilon,\delta} = v_{\lfloor \delta \cdot n - \epsilon \rfloor}$.

In particular, if $\lfloor \delta \cdot n - \epsilon \rfloor < 0$, then $q^x_{\epsilon,\delta}$ is considered to be $-\infty$.

We use the (ϵ, δ)-Quantile to define the Real-time Per-key Quantile Filtering problem as follows.

[2]SOTA requires a query time significantly longer than the insertion time, and cannot be considered a small constant. For example, GK and SQUAD, which use GK, require binary search during querying. SketchPolymer needs to query log (value range) number of counters, and HistSketch requires accessing a remote server to obtain results.

[3]Sketch is a category of approximate algorithms used for statistical purposes. [22]–[25]

Definition 4. (Online Detection of Quantile-Outstanding Keys). Based on the $\langle \epsilon, \delta, T \rangle$ criteria, this filtering process keeps a close watch on all items as they come in. If the (ϵ, δ)-quantile of the recent set of values V_x for any key goes over the set threshold T, that key x (termed as the outstanding key) should be immediately reported. After the report, the value set V_x is reset to empty, making room for new incoming values. Formally, for each new item $\langle x, v \rangle$, we do:

$$\begin{cases} \text{Report } x \text{ and Reset } V_x, & \text{if } q^x_{\epsilon,\delta} > T, \\ \text{Add } v \text{ to } V_x & \text{if } q^x_{\epsilon,\delta} \le T, \end{cases}$$

The flexibility of the $\langle \epsilon, \delta, T \rangle$ criteria: In broad applications, we hope to support different filtering criteria for different types of keys simultaneously, which will be feasible and be discussed in Section III-C.

We prefer the (ϵ, δ)-Quantile over the simple δ-Quantile for several reasons. It makes sure that removing ϵ values above T from V_x still leaves more than δ proportion of the values in V_x above T. It also offers two main benefits:

- Avoiding Premature Reporting. If a key x's value has only a 1% chance to exceed T but does so on the first try, V_x would falsely show a 100% rate of going over. Usual methods ($\epsilon = 0$) would report this, but to avoid such unlikely warnings, we wait for at least ϵ additional values to exceed T before making a report.
- Avoiding Reporting Infrequent Keys. We aim to ignore keys that don't come up often. For instance, tracking a $\delta = 0.95$ quantile is meaningful for a key if it has more than 20 values. If a key is observed only once and its value exceeds T, it could misleadingly seem like all its values are above T. Without a non-zero ϵ, such a key would be wrongly reported, which we wish to avoid.

In our definition, we reset the value set after a report. This practice, while not typical for static data, is standard and crucial in real-time data analysis to ensure the precision of reports and to minimize the chance of repeated alerts. When a key is reported, it signals to the user that the quantile $q^x_{\epsilon,\delta}$ for this key has exceeded the threshold T. Resetting the value set ensures that only recent data is considered from the moment after the last report, keeping the information up to date. Ongoing reports mean that the quantile $q^x_{\epsilon,\delta}$ is still above T. The epsilon parameter ensures that reports are not too frequent, as they will occur less often than every ϵ values. Without this reset mechanism, historical data could impact the current analysis, potentially leading to skewed outcomes.

Example. Let's consider a data stream that monitors noise levels (measured in decibels, dB) across various neighborhoods in a city. This stream updates every 5 minutes with new readings. We apply per-key quantile filtering to detect when a neighborhood's noise level consistently exceeds the threshold of 70 dB. Here, our quantile of interest, δ, is set to 0.8, and we will use an ϵ value of 1 to avoid reporting occasional noise spikes.

Data Stream Example (three keys):

- Neighborhood A: [65, 67, 72, 69, 74, 66, 68, 75]
- Neighborhood B: [60, 62, 64, 61, 63, 75, 80, 62]
- Neighborhood C: [55, 57, 59, 58, 76, 57, 56, 55]

Our Analysis:

- Neighborhood A: Out of 8 data points, three exceed the threshold (72, 74 and 75 dB). With $\delta = 0.8$, we consider the $\lfloor 0.8 * 8 \rfloor + 1 = 7$-th lowest value (with indices starting at 1), which is 74 dB. Then with $\epsilon = 1$, we need to consider $7 - 1 = 6$-th lowest value, 72 dB. Since this is above the threshold, we would report Neighborhood A.
- Neighborhood B: Two readings exceed 70 dB. However, the 6-th lowest value is 64 dB, which is below the threshold, so Neighborhood B is not reported.
- Neighborhood C: There is one spike at 76 dB, and the 6-th lowest value is 57 dB. Despite the spike, the (1, 0.8)-quantile doesn't exceed the threshold, so Neighborhood C is not reported.

B. Prior Art

We introduce three categories of related work based on algorithm functionality: (1) Single-key quantile estimation algorithms, (2) Multi-key quantile estimation algorithms, and (3) Exact quantile calculation algorithms.

Single-key quantile estimation algorithms are used to estimate the quantiles for an individual key. Notable works include GK [14] and KLL [15], [16]. GK 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. It's widely deployed in fields like real-time data processing and big data analytics. KLL's [15] main idea is to reduce storage requirements through hierarchical sampling while ensuring quantile accuracy. Other single-key quantile algorithms like the t -digest [17], DD [18] and Q-digest [12] use different data structures to reduce storage and quickly respond to quantile queries. However, they are usually not suited for multi-key scenarios as they require building and maintaining a separate data structure for each key, significantly increasing storage use. Additionally, they demand active querying by users to provide results, which leads to high real-time computational costs.

Multi-key quantile estimation algorithms aim to estimate quantiles across multiple keys. For example, SQUAD [19] uses a compound data structure to track quantiles for multiple distributions but suffers from slow query speeds and high storage costs due to the complexity of its data structures. SketchPolymer [20] improves storage and computational efficiency by mapping data to multiple lightweight structures. However, it is designed for batch processing and offline analysis, not real-time querying. HistSketch [21] employs a histogram-based approach, using hierarchical histograms to speed up queries. Yet, it still faces computational and storage efficiency issues when dealing with real-time data. Like single-key algorithms, multi-key methods require users to actively choose quantiles and undergo complex calculations to get results. These algorithms were originally designed for offline, not real-time queries, which is unsuitable for our problem.

In this context, the offline query denotes queries that do not have small constant query times. Examples include GK and SQUAD implementations that employ GK, requiring binary searches during the query process. SketchPolymer involves querying a logarithmic number of counters relative to the value range, while HistSketch demands accessing a remote server to retrieve the results.

Exact quantile calculation algorithms, such as histograms [27], balanced trees [12], and sorting features, can provide zero-error quantile calculations. For instance, SQL Server utilizes built-in functions based on sorting or histograms [28]. Sorting methods are effective for static datasets by constructing precise summaries to determine quantiles. However, these algorithms typically operate on static datasets. In highly dynamic streaming scenarios, storing many buckets or rebuilding when data distributions change incurs excessive update costs. Although they offer the advantage of zero-error, their speed limitations mean they may not be the best choice for streaming scenarios requiring real-time insertion and query.

C. Introduction to Practical Sketch Tools

Here we introduce basic sketch tools used for frequency estimation: the Count sketch [29], which are employed to estimate the frequency of keys in a data stream. We will construct a straightforward naive solution using it to preliminarily address our target problem. This solution will then be refined to present our final design. For other sketch algorithms, please refer to [24], [25], [30]–[47].

The Count Sketch (Csketch) [29] is a probabilistic data structure that serves as an approximate frequency table of keys in the stream data. It operates with d arrays, each consisting of w counters. Upon the arrival of a data item with key x , it computes d pairwise independent hash values using x. Each hash value points to a position within each of the d rows, and the counter is added by a value given by $S_i(x)$, where $S_i(\cdot)$ is a sign hash function returning either $+1$ or -1 with equal probability for the i-th row. During a query, for each of the d hash functions, Csketch retrieves the counter at the hashed index and multiplies it by the sign given by the hash function $S_i(x)$. The estimated frequency of x is then derived by the median of these d signed values.

The key idea of Csketch is that, the hash functions distribute keys evenly across the counters, and the statistical methods of choosing the median counter value help reduce the effect of collisions. This provides an estimation of the key's frequency, allowing for efficient handling of large-scale data streams with real-time updates and queries.

Csketch can be extended to support weighted sum, where each data item is assigned a weight value w, and the query is for the sum of the weights of all items for a certain key. To implement this functionality, one simply needs to change the increment value $S_i(x)$ of the Csketch counter to $wS_i(x)$.

D. Naive Solution

We propose a naive approach to address the Real-time Per-key Quantile Filtering problem employing a pair of Csketches, which may differ in size. The initialization of two Csketches, $Csketch_{\text{above}}$ and $Csketch_{\text{below}}$, is the first step where $Csketch_{\text{above}}$ is responsible for keeping the count of occurrences for each key with values exceeding threshold T, and $Csketch_{\text{below}}$ maintains the count of occurrences for values not exceeding T.

As we process each incoming item $\langle x, v\rangle$ in the stream S, the value v is compared against the threshold T to determine the update path for the sketches:

- Increment x in $Csketch_{\text{above}}$ if $v > T$.
- Increment x in $Csketch_{\text{below}}$ if $v \leq T$.

Following the updates for a key x, we query its frequency counts F_a from $Csketch_{\text{above}}$ and F_b from $Csketch_{\text{below}}$, by the median of the d signed values in each sketch. These frequency counts are used to decide if key k should be reported, which is checked by seeing if F_b is less than or equal to $\lfloor (F_a + F_b)\cdot\delta - \epsilon\rfloor$. If this condition is met, then the key is reported. Reporting the key triggers a reset in the frequency counts for x in both sketches. This reset is done by reducing the count F_a in every counter that x hashes to in $Csketch_{\text{above}}$ and decreasing the count F_b in $Csketch_{\text{below}}$ in the same way.

This naive dual-sketch method has notable limitations. First, the reset process after reporting introduces errors because it assumes that the subtracted minimum value exactly reflects the true frequency, which is often not the case due to hash collisions. Second, the accuracy of the naive solution is highly sensitive to the size of each sketch, demanding a larger memory footprint for higher accuracy.

III. QuantileFilter Design

We present the structure of our QuantileFilter, which uses two key techniques: Quantile weight (Qweight) and Candidate Election. Qweight allows a single action in one sketch to handle what would otherwise require three separate actions (one insert and two queries) in the naive solution involving two sketches, leading to a more efficient workflow and faster processing. The Candidate Election method identifies the keys with the highest chances of being reported by using a process of competitive selection. The keys that come out on top from this process are counted more accurately, which greatly lowers the chance of misreports due to hash collisions. In Table I, we present the key notations used in this paper.

Notation	Description
x, v	Item key, Item value
ϵ, δ, T	Rank Deviation, Quantile, Value Threshold
d, w	Depth (rows), Width (columns) in Csketch
$C_i[j]$	Counter at row i, column j
$h_i(x)$	Hash function mapping item x to a column
$S_i(x)$	Sign function returning $+1$ or -1

TABLE I: Key Notations Used in This Paper.

A. Quantile Weight Technique

Motivation (illustrated in Figure 2): In the naive solution, we employ two Csketches to count values that either exceed or do not meet the threshold T. This can lead to unnecessary

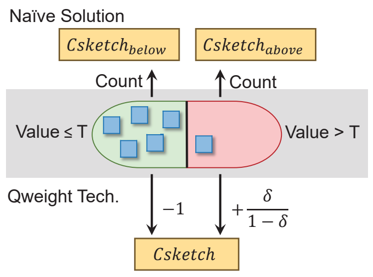


Fig. 2: A comparative illustration between the naive solution and the Quantile Weight Technique. Blue squares represent items. Items with a green background are those with a value less than or equal to T, and those with red are items with a value greater than T. The naive solution uses two separate Csketches to count these two types of items. The Quantile Weight Technique, however, employs only one Csketch.

complexity because, for reporting purposes, we only need to identify whether to report the key, not the count of values on either side of T.

Qweight Definition. We define Qweight as follows. Assign a weight of -1 to items less than or equal to T and a weight of $+\frac{\delta}{1-\delta}$ to items greater than T. The Qweight of a key is defined as the sum of the weights of all its items

$$Qw(key) := \sum_{v_i : v_i \leqslant T} -1 + \sum_{v_i : v_i > T} \frac{\delta}{1-\delta}.$$

Convert (ϵ,δ)-Quantile to Qweight. Considering the special case when $\epsilon = 0$, we find that $q^x_\delta > T$ is equivalent to $Qw(key) \geqslant 0$, which can be demonstrated as follows:

$$\begin{cases} \text{For } q^x_\delta > T, \text{ indicating at least } n - \lfloor n\delta \rfloor \text{ items exceed } T: \\ Qw(key) \geqslant \frac{\delta}{1-\delta}\cdot(n-\lfloor n\delta\rfloor) - \lfloor n\delta\rfloor = \frac{n\delta - \lfloor n\delta\rfloor}{1-\delta} \geqslant 0, \\ \\ \text{Conversely, for } q^x_\delta \leqslant T: \\ Qw(key) \leqslant \frac{\delta}{1-\delta}\cdot(n-\lfloor n\delta+1\rfloor) - \lfloor n\delta+1\rfloor = \frac{n\delta-\lfloor n\delta\rfloor-1}{1-\delta} < 0. \end{cases}$$

Similarly, for the general ϵ-biased quantile, $q^x_{\epsilon,\delta} > T$ is equivalent to $Qweight(key) \geqslant \frac{\epsilon}{1-\delta}$.

$$\begin{cases} \text{For } q^x_{\epsilon,\delta} > T, & \text{indicating at least } n - \lfloor n\delta - \epsilon\rfloor \text{ items exceed } T. \\ \quad Qw(key) & \geqslant \frac{\delta}{1-\delta}\cdot(n - \lfloor n\delta-\epsilon\rfloor) - \lfloor n\delta-\epsilon\rfloor \\ & = \frac{n\delta - \lfloor n\delta-\epsilon\rfloor}{1-\delta} \geqslant \frac{\epsilon}{1-\delta}, \\ \text{For } q^x_{\epsilon,\delta} \leqslant T, & \\ \quad Qw(key) & \leqslant \frac{\delta}{1-\delta}\cdot(n - \lfloor n\delta-\epsilon+1\rfloor) - \lfloor n\delta-\epsilon+1\rfloor \\ & = \frac{n\delta-\lfloor n\delta-\epsilon\rfloor-1}{1-\delta} = \frac{n\delta-\lfloor n\delta\rfloor-1}{1-\delta} + \frac{\epsilon}{1-\delta} < \frac{\epsilon}{1-\delta}. \end{cases}$$

Therefore, we can completely convert our problem to determining whether

$$Qweight(key) \geqslant \frac{\epsilon}{1-\delta}$$

is true. If it does, we report the key and reset $Qweight(key)$ to 0.

Specifically, we employ a Csketch data structure to maintain and calculate Qweights of keys in a data stream as follows.

Qweight Estimation with Csketch (See Pseudocode1). We use a Csketch with d rows and w columns, where each cell counter is denoted as $C_i[j]$, with i indicating the row and j the column. When a new item $\langle x, v\rangle$ arrives, we first compare its value v with T to calculate the item's weight Qw: Qw is $\frac{\delta}{1-\delta}$ if the value v is above a threshold T; otherwise, $Qw = -1$. We then insert Qw into the count sketch by updating the sign function's Qw times, which is done by adding $S_i(x) \times Qw$ to $C_i[h_i(x)]$ for each row i. To estimate the Qweight for x, we compute an estimate value for each row by $S_i(x) \times C_i[h_i(x)]$, and then take the median of these values as the final estimate for the Qweight, denoted as $\widehat{Qw}(x)$. This method, using the median, helps improve the accuracy of the estimation by reducing the impact of outliers due to collisions. If $\widehat{Qw}(x)$ is greater than or equal to $\frac{\epsilon}{1-\delta}$, we report the key x, and then delete its $\widehat{Qw}(x)$. The deletion operation for $\widehat{Qw}(x)$ of key x involves decrementing the mapped counter $C_i[h_i(x)]$ by $S_i(x)\widehat{Qw}(x)$ in each row i of the Csketch.

Algorithm 1: Qweight Estimation with One Csketch.

```
Input: A stream of items $\langle x, v\rangle$
Output: Report keys with $q^x_{\epsilon,\delta} > T$.
1 Initialize Csketch counters $C_i[j]$ to 0 for all $i$ and $j$.
2 for each incoming item $\langle x, v\rangle$ do
3     Compute item Qweight: $Qw \leftarrow \begin{cases} \frac{\delta}{1-\delta}, & \text{if } v > T \\ -1, & \text{otherwise.} \end{cases}$
4     Update counters:
        $C_i[h_i(x)] \leftarrow C_i[h_i(x)] + S_i(x)Qw, \forall i \in 1,\dots,d$
5     Estimate key Qweight:
        $\widehat{Qw}(x) \leftarrow \mathtt{Median}_{i=1}^{d}\{S_i(x)C_i[h_i(x)]\}$
6     if $\widehat{Qw}(x) \geqslant \frac{\epsilon}{1-\delta}$ then
7         Report key $x$ and decrease $C_i[h_i(x)]$ by
          $S_i(x)\widehat{Qw}(x)$.
```

Technical Details. For high space efficiency, sketch counters are typically integers instead of floating-point numbers. However, the item weight, $Qw = \frac{\delta}{1-\delta}$, might not be an integer. A straightforward solution is to use floating-point numbers for the sketch's counters. But we found another approach that works well: (1) First, add the integer part of Qw (*i.e.*, $\lfloor Qw\rfloor$) to the counter. (2) Then, with a probability equal to the fractional part of Qw, that is $Qw - \lfloor Qw\rfloor$, which falls within the interval [0,1), increment the counter by one. This method ensures that the counter's expected increase is Qw, allowing for an unbiased addition of Qw. The variance of this scheme is given by $(Qw - \lfloor Qw\rfloor)\cdot(1-(Qw-\lfloor Qw\rfloor))$, which is smaller than 0.25.

B. Candidate Election Technique

Our new optimization, called Candidate Election, attempts to track keys (candidate keys) that are most likely to exceed the threshold T, providing them with Qweight statistics that are not only more accurate but also less susceptible to hash collisions, thus boosting the overall reporting precision. Expanding upon the aforementioned Csketch, which we refer to as the vague part, Candidate Election incorporates a novel data

Algorithm 2: QuantileFilter with Two Parts.

```
Input: A stream of items $\langle x, v\rangle$.
Output: Report keys with $q^x_{\epsilon,\delta} > T$.
For each incoming item $\langle x, v\rangle$ do:
Compute the fingerprint $fp$ and item Qweight $Qw$.
Locate the bucket $B$ by hashing $h_b(x)$.
if $fp$ matches an entry $\langle fp, Qw_{fp}\rangle$ in bucket $B$ then
    Update $Qw_{fp}$ depending on $v$.
    Report key $x$ and reset $Qw_{fp}$ if $Qw_{fp}$ exceeds $T$.
else if bucket $B$ has space then
    Add entry $\langle fp, w\rangle$.
else
    Insert item into the vague part.
    Estimate Qweight $\widehat{Qw}(x)$ from the vague part.
    if $\widehat{Qw}(x)$ exceeds $T$ then
        Report key $x$ and reset its Qweight in the vague part
          to zero.
    Identify the entry $\langle fp', MinQw\rangle$ with the smallest
      Qweight in $B$.
    if $\widehat{Qw}(x)$ is greater than $MinQw$ then
        Remove $\langle fp', MinQw\rangle$ from candidate part and
          insert into vague part.
        Insert new entry $\langle h_{fp}(x), \widehat{Qw}(x)\rangle$ into candidate
          part and remove from vague part.
```

structure (named the candidate part) to find candidate keys and accommodate them. When an item cannot be accommodated in the candidate part, we fall back to the vague part, ensuring that no key is left untracked. Our illustration is in Figure 3.

Data Structure. The candidate part is an array composed of m buckets, with each capable of holding b entries. Each entry $\langle fp, Qw\rangle$ comprises a key fingerprint — a 16-bit hash value of the key generated by the hash function h_{fp} — alongside an integer counter that records the Qweight for that key. A separate hash function $h_b(x)$ randomly allocates each key x to one of the m buckets.

Reason for Fingerprint Use. *We store a candidate key's fingerprint instead of the complete key for space efficiency and structural neatness. Firstly, fingerprints take up less space, with controlled length, unlike the potentially long and variable keys. Secondly, fingerprints suffice for real-time reporting and do not face the offline issue of reporting a fingerprint without the full key being known. Errors from fingerprints are small and typically constitute a negligible part of the overall error. Using 16-bit fingerprints ensures a collision probability under 0.01%, and using longer fingerprints can further reduce the likelihood of hash collisions.*

Item Insertion (see Pseudocode 2). When a new item $\langle x, v\rangle$ arrives, it is initially hashed to a bucket B within the candidate part. Then, one of the following three scenarios occurs:

1) If the key's fingerprint matches any entry $\langle fp, Qw_{fp}\rangle$ in bucket B, the Qweight counter Qw_{fp} is updated based on v, by either adding $\delta/(1-\delta)$, or subtracting 1. If Qw_{fp} exceeds the threshold T, the key is reported, and Qw_{fp} is reset to zero.
2) If the bucket has space and the fingerprint is not matched, the key's fingerprint and Qweight are added to it.

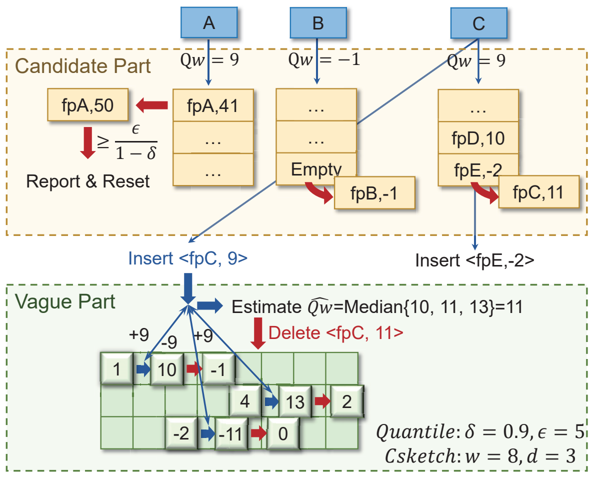


Fig. 3: Overview of Quantile Filter, comprising two parts: candidate and vague. In the figure, the candidate part includes 3 buckets, each capable of holding up to 3 entries. The vague part is a Csketch with a size of 8*3 counters. We demonstrate three instances under $\delta = 0.9, \epsilon = 5$.
Case A: When key A arrives with a +9 Qw, it matches a fingerprint in the candidate part and its Qw is directly updated. Since it reaches the threshold $\frac{\epsilon}{1-\delta} = 50$, A is reported and reset.
Case B: When key B arrives, there is a vacant spot in the candidate part, so it is directly stored.
Case C: When key C arrives with a +9 Qw and finds no vacant spot and no fingerprint match, its +9 Qw is inserted into the vague part. Meanwhile, its Qw is estimated to be $\widehat{Qw} = 11$. As this is greater than the smallest fingerprint E's corresponding -2 in the bucket, a swap is made between $\langle fpC, 11\rangle$ and $\langle fpE, -2\rangle$.

3) If the bucket is full, the item is inserted into the vague part as introduced previously. If its new Qweight $\widehat{Qw}(x)$ exceeds the threshold T, we report the key, and its Qweight in the vague part is reset to zero. Otherwise, if the new Qweight $\widehat{Qw}(x)$ is greater than the smallest Qweight in bucket B, an exchange occurs between the candidate and vague parts. In the bucket B, we locate the entry $\langle fp', MinQw\rangle$ with the smallest Qweight, remove its contents from the candidate part and insert them into the vague part; then we fill the vacant position in the candidate part with the new entry $\langle h_{fp}(x), \widehat{Qw}(x)\rangle$ and remove it from the vague part.

Beyond the standard item insertion process, we also support three additional operations: query, delete, and reset.

To query the Qweight for key x, we first locate the bucket B in the candidate part using hash h_b and calculate the fingerprint fp for x. If a matching fingerprint is found in the bucket, then the Qweight for x is directly available. There is no need to consult the vague part for this information. Conversely, if there is no fingerprint match in the bucket, we proceed to estimate the Qweight using Csketch method in the vague part.

To delete the Qweight for key x, we locate which part x is in using the query method and then clear it using the corresponding part's method. If it's in the candidate part, we zero the Qweight counter directly; if it's in the vague part, we perform Csketch's deletion operation.

To reset the data structure, a fixed-size QuantileFilter needs to be periodically cleared. This is partly due to real-time considerations, as outdated data should not be included, and partly due to accuracy, as it cannot maintain precision with an unlimited number of insertions. The reset operation is straightforward: all data structures are reset to empty. If it is necessary to adjust the size of the data structures, this can be done at this time.

Technical Details. To ensure the algorithm functions properly, some detailed issues must be addressed:

- Handling the missing key for vague part hashing. Since the candidate part stores only fingerprints, the hash location for the vague part is determined by combining the fingerprint with the bucket index from $h_b(x)$, thus modifying the original $h_i(x)$ to $h_i(fp + h_b(x))$. This method allows item insertion based on fingerprints, maintaining accuracy comparable to hashing the original keys.
- Handling the overflow of counters. The algorithm keeps keys with higher Qweights in the candidate part, so most vague part's Qweight stay small or negative. When numerous distinct keys map to the same counter, creating a hash collision, these keys are allocated +1 or -1 by the sign function with equal probability. This approach allows keys with similar weights to offset each other, thereby reducing the risk of overflow. Consequently, we can adopt 16-bit or even 8-bit counters to conserve space while maintaining close to 100% accuracy. Yet, it is crucial to prevent counters from naturally rolling over due to overflow, such as turning from the maximum 16-bit value of 32767 to -32768 with an increment of 1. Operations must prevent overflow reversals, ignoring any addition or subtraction that would cause it.

C. Flexibility of QuantileFilter Functions

We expand the flexibility of the QuantileFilter's filtering criteria $\langle \epsilon, \delta, T \rangle$ to support different criteria for different keys, dynamic modification of criteria, and the coexistence of multiple criteria for the same key.

First, QuantileFilter supports specifying different $\langle \epsilon, \delta, T \rangle$ reporting criteria for different keys. For example, when monitoring network data flows, it is necessary to pay special attention to the abnormal delays in UDP data flows used for audio and video calls (such as Zoom and WeChat calls) and set tighter reporting criteria for them. In this scenario, we assume that the users of the algorithm will specify clear criteria in advance, i.e., input the criteria $\langle \epsilon_x, \delta_x, T_x \rangle$ for the algorithm along with each item $\langle x, v \rangle$. This is not difficult for users; in the example mentioned, they can accurately know the application type through packet header analysis and set criteria accordingly. Implementing this functionality in QuantileFilter is not difficult; we simply substitute the criteria $\langle \epsilon_x, \delta_x, T_x \rangle$ for the existing algorithm parameters $\langle \epsilon, \delta, T \rangle$ during operation.

Second, QuantileFilter supports modifying the $\langle \epsilon, \delta, T \rangle$ reporting criteria for a specific key. For example, if certain keys have unique characteristics that have been confirmed by users, the reporting criteria for these keys can be relaxed. To modify key x, we remove its Qweight via the deletion operation, then will insert under new criteria. Following criteria change, V_x resets to empty. Experimental results indicate that the impact of modifications is quite complex; for detailed discussion, please refer to V-D.

Third, QuantileFilter supports setting multiple criteria for the same key and reporting when any criteria are met. For example, we may be interested in both the 99th percentile and the 95th percentile of delta for a key. However, the original QuantileFilter approach cannot be directly applied here because, in QuantileFilter, we only record one Qweight for a key, and one Qweight cannot support two different criteria (unless only ϵ differs). Therefore, we assign the monitoring of multiple criteria for a key to multiple keys, i.e., we combine the original data key with the criterion number to form a new key tuple, which is then inserted into QuantileFilter for processing. If a key supports r monitoring criteria, it will result in r new keys and r insertions into QuantileFilter. The overhead of this scheme increases with r, but it performs well when r is small.

D. Ours Design Choices and Innovative Techniques

Here, we discuss other design choices in the QuantileFilter algorithm and their impacts, as well as the innovative aspects of QuantileFilter's algorithmic techniques with their inspirational significance to other approximation algorithms.

Choice 1. Strategy in candidate election. Currently, if a key's Qweight $\widehat{Qw}(x)$ in the vague part exceeds the smallest Qweight $MinQw$ in the bucket, we opt to replace it into the candidate part (Comparative Replacement). In reality, various replacement strategies could be considered, such as switching the key into the candidate with a probability of $Max\left(\frac{\widehat{Qw}(x)}{\widehat{Qw}(x)+MinQw}, 0\right)$ (Probabilistic Replacement), or ensuring its replacement with a 100% certainty regardless of the Qweight size (Forceful Replacement). We have experimented with these strategies and found that they do not significantly affect the overall performance.

Choice 2. Sketch type of the vague part. Our vague part is designed based on Csketch, but it could also be replaced with other sketches, like the Count-Min Sketch (CMS) [25]. Our experiments indicate that using CMS does not improve the accuracy. However, this does not entirely eliminate the possibility that among the existing dozens of sketches, there might be ones more suitable for the vague part. We leave this question for future work.

We believe that the primary contribution of this paper is studying a new problem from the perspective of approximation algorithms and providing a feasible solution. However, the technical contributions of QuantileFilter's design is also noteworthy. We discuss this in hopes that it will inspire the design of other algorithms.

Technique 1. Mitigating the negative impact of fingerprints. Fingerprints are a common technique in filters, like cuckoo filters. Their downside is the loss of the original key, which prevents existing sketches from locating the correct counter based on hashing the key. As we pointed out, this issue can be addressed by changing the hash computation from the key to the fingerprint and its bucket index. As long as "the number of buckets" $\times$ "2 to the power of fingerprint length" is much greater than the number of counters in the sketch, this approach will not lose visible accuracy. All sketch methods that involve storing keys or fingerprints could potentially consider this technique to reduce the space occupied by keys and fingerprints, such as JoinSketch [48] and ElasticSketch [24].

Technique 2. Designing specific dual part solutions for specific problems. Our approach uses a dual-part structure to differentiate large Qweights, contrasting with traditional sketches that separate keys based on frequency. Our strategy, which assigns a key's Qweight exclusively to one part (either candidate or vague), simplifies calculations and sharpens the algorithm's logic. For more specific estimation objectives like gradients [49] or itemsets [50], employing a dual-part structure may be beneficial.

IV. Mathematical Analysis

In this section, we analyze the error bound of QuantileFilter, its time complexity, and its space complexity. First, we analyze the performance of QuantileFilter without using the candidate part (Theorem 1), then we show its performance after removing the top-k keys from QuantileFilter (Theorem 2), and finally, we demonstrate how the candidate part can reduce the error (Theorem 3). After completing the error analysis, we discuss the time and space complexity of QuantileFilter.

We begin by analyzing the error bound of the vague part. The vague part is structured according to the existing Csketch algorithm. Originally, Csketch was designed for weights equal to 1, mainly for counting key occurrences. However, it can be extended to handle weights, including negative ones. By following the proof methodology of Csketch, we can deduce the following theorem:

Theorem 1. *In a model comprising solely the vague part (characterized by size parameters d and w) and assuming no integer overflow, consider a data stream with n keys, labeled 1, 2, ..., n. Let Q_i be the true Qweight of the ith key, and Q'_i its estimated value. The following conclusions can be drawn:*

$$\textit{Unbiasedness: } E(Q'_i) = Q_i$$

Error bound with relative error ε and failure probability γ:

$$\Pr\left[|Q'_i - Q_i| \geqslant \varepsilon L_2\right] \leqslant \gamma \quad (1)$$

where $w = \lceil 4\varepsilon^{-2}\rceil, d = \lceil 8ln(\gamma^{-1})\rceil, L_2 = \sqrt{\sum_{i=1}^{n} Q_i^2}$.

Proof. For each array t, every key i has a signed value $S_t(i)$ which is randomly 1 or -1. Let Q_i^* denote the estimated value of Q_i in array t. The expectation in array t is unbiased: $E(Q_i^*) = E\left(Q_i + \sum_{j\neq i}(Q_j S_t(i) S_t(j) X_j\right) = Q_i$, where $X_j = \begin{cases} 1 & \text{iff } h_t(i) = h_t(j), \\ 0 & \text{otherwise.}\end{cases}$ indicates the collision. The error in Q'_i is symmetric due to the symmetry of $S_t(j)$. By taking the median of the estimates across t rows, we can still ensure that the estimated quantity is unbiased. That is, $E(Q'_i) = Q_i$.

We can calculate the variance as follows:

$$\begin{aligned} Var(Q_i^*) &= Var\left(Q_i + \sum_{j\neq i} Q_j S_t(i) S_t(j) X_j\right) \\ &\leqslant \sum_{j\neq i}\left[E((Q_j S_t(i) S_t(j) X_j)^2) - E(Q_j S_t(i) S_t(j) X_j)^2\right] \\ &= \frac{1}{w}\sum_{j\neq i} Q_j^2 \leqslant \frac{1}{w} L_2^2, \text{where } L_2 = \sqrt{\sum_{i=1}^{n} Q_i^2} \end{aligned}$$

With Chebyshev's inequality, we have $\Pr\left[|Q_i^* - Q_i| \geqslant \varepsilon L_2\right] \leqslant \frac{1}{w\varepsilon^2}$

By using the Chernoff bound, and choosing $w = \lceil 4\varepsilon^{-2}\rceil$ and $d = \lceil 8\ln(\gamma^{-1})\rceil$, we obtain:

$$\Pr\left[|Q'_i - Q_i| \geqslant \varepsilon L_2\right] \leqslant \gamma$$

□

Theorem 2. *With the absolute value of Qweight follow a Zipf distribution with parameter α, after removing the top k keys with the highest Qweight from the vague part, the error bound generated by Theorem 1 will be reduced by a factor of $\frac{1}{k^{\alpha-0.5}}$, i.e., replacing L_2 with $\frac{L_2}{k^{\alpha-0.5}}$.*

Proof. In a Zipf distribution, the k-th largest key in Qweight $\frac{1}{k^{\alpha}*\zeta(\alpha)}$ probability of occurrence. So, we can estimate that: $Q_k^2 \approx \left(\frac{1}{k^{\alpha}*\zeta(\alpha)}\right)^2 (L_2)^2$ and $\sum_{i=1}^{n}(Q_i^2) \approx \sum_{i=1}^{n}(i^{-2\alpha})\frac{1}{\zeta(\alpha)^2}(L_2)^2$

After filtering out top-k keys, we have $\sum_{i=k+1}^{n}(Q_i^2) \approx \sum_{i=k+1}^{n}(i^{-2\alpha})\frac{1}{\zeta(\alpha)^2}(L_2)^2$. Thus, $\frac{\sum_{i=k+1}^{n}(Q_i^2)}{\sum_{i=1}^{n}(Q_i^2)} \approx \frac{\sum_{i=k+1}^{n}(i^{-2\alpha})}{\sum_{i=1}^{n}(i^{-2\alpha})} \leqslant \frac{1}{k^{2\alpha-1}}$. □

Since keys in the candidate part do not affect the vague part, if we can use the candidate part to pick out the ideal keys, Theorem 2 tells us that this could potentially reduce the error of the vague part by a significant multiple. Since we consider the probability of fingerprint collision to be very small, it is not considered here.

Theorem 3. *For a key i in the candidate part with estimation Q'_i, suppose that the last time it enters into the candidate part is T. We use Q_k^T to represent the real Qweight of key k at time T. m_k^T indicates whether key k has ever entered the vague part before time T, with $m_k^T = 1$ representing 'yes', and otherwise $m_k^T = 0$. The L_2 in Formula 2 can be replaced with much smaller $\sqrt{\sum_{i=1}^{n}(m_i^T(Q_i^T)^2)}$:*

$$\Pr\left[|Q'_i - Q_i| \geqslant \varepsilon\sqrt{\sum_{i=1}^{n}(m_i^T(Q_i^T)^2)}\right] \leqslant \gamma \quad (2)$$

We analyze the time and space complexity of QuantileFilter. Since the conclusion of Theorem 3 is dependent on dataset characteristics, in the most conservative case, we can only estimate the complexity based on Theorem 1.

In terms of time, processing each item involves at most one visit to the candidate part and two visits to the vague part, so the time complexity per item is $O(b+d)$, where b is the bucket size of the candidate part, and d is the number of arrays in the vague part. To achieve error control under (ε and γ), the time complexity is $O(\log(\gamma^{-2}))$.

In terms of space, we can maintain the candidate part's size as a constant multiple of the vague part, hence directly analyze the space of the vague part, which is $O(w \times d)$, where w is the width of each array. Therefore, the space complexity is $O(\varepsilon^{-2}\log(\gamma^{-1}))$.

V. Evaluation

In this section, we present the experiment results of QuantileFilter, which include four main sections: (1) Experiment setup and methods. (2) Accuracy-Space comparison of QuantileFilter with state-of-the-art (SOTA) schemes, including SQUAD, SketchPolymer, and HistSketch. (3) Speed comparison of QuantileFilter with SOTA. (4) More detailed experiment results of QuantileFilter, including the effects of adjusting its parameters and design. The codes for QuantileFilter are open-sourced on Github [51].

We have two key results:

1) **Speed.** For the same volume of data in real datasets and with an accuracy above 50%, the processing speed (insertion + querying) of QuantileFilter for each item has improved by 10 to 100 times compared to SOTA schemes.
2) **Space.** QuantileFilter saves 50 to 500 times more space than SOTA schemes at the same level of accuracy, applicable across all accuracy ranges from 0 to 100%.

A. Experiment Setup and Methods

Datasets. We use two large-scale real datasets and one synthetic dataset, which are respectively the internet network data from CAIDA, cloud network data from Yahoo, and a synthetic dataset following a Zipf distribution.

1) Internet dataset. Derived from CAIDA's anonymized high-speed internet traffic data [52], this dataset uses a five-tuple key (source/destination IP addresses, port numbers, protocol numbers) with time intervals as values. It consists of a 23-second stream containing 26.1M items and approximately 0.64M unique keys.
2) Cloud dataset: From Yahoo stream data [53], capturing patterns between Internet users and Yahoo servers on AWS. Each item includes start/end times, source/destination IPs, ports, protocol, using a five-tuple as the key and time duration as value, excluding content. The dataset encompasses 20.5M items with 16.9M unique keys.
3) Zipf dataset. A synthetic dataset modeled on the Zipf distribution, with item occurrence frequencies following Zipf's law with parameter α. Each value is derived by summing two components: one that adheres to a fixed-parameter Zipf distribution, and another that is constant given a key and varies with the key according to a normal distribution with fixed mean and standard deviation. Adjusting α varies key distributions, resulting in two datasets of 25M items each, with distinct key counts of 4.2M and 120K.

Platform and Implementation. All our experiments were carried out on a server featuring an Intel CPU i9-10980XE (18 cores, 36 threads, 3.00 GHz, 64KB L1 cache per core, 1MB L2 cache per core, 24.75MB L3 cache shared by all cores) and 128GB of DRAM. The implementation of QuantileFilter was done in C++, and the SOTA code was obtained from their public repositories. The executable files were compiled with the O2 optimization setting activated. Based on the analysis in Section V-D, the **default parameters** selected for QuantileFilter are: a maximum of b=6 entries in the candidate part's bucket, and d=3 arrays in the vague part. The storage space allocation between the candidate part and the vague part is 4:1. The parameters for SOTA are derived from the recommendations in their papers. The default quantile parameters are set as $\epsilon = 30$, $\delta = 95\%$. The threshold (T) is 300ms adjusted to ensure the proportion of abnormal items is around 5%. Specifically, $T = 300$ms for the Internet dataset, 20s for the Cloud dataset, and 300ms for the Zipf dataset.

B. Experiments on Accuracy-Space

In this section, we compare our solution's space and accuracy performance against existing SOTA algorithms, highlighting our superior efficiency.

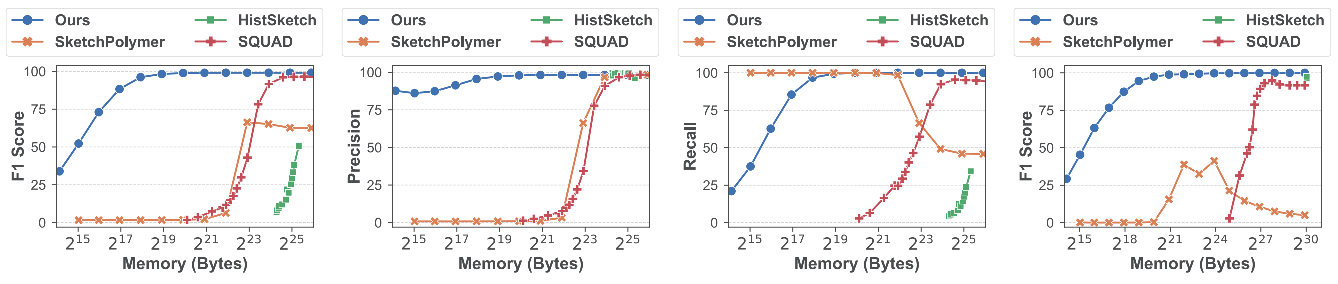
Metrics: We measure algorithm accuracy by streaming the entire dataset into each algorithm, which in real-time reports the outstanding keys. After the insertion process is complete, these keys are deduplicated and compared with the actual set of outstanding keys. A True Positive (TP) is a reported key correctly identified as outstanding, while a False Positive (FP) is a key incorrectly identified as outstanding. The following three metrics are defined:

- **Precision** $= \frac{TP}{TP+FP}$, indicating the accuracy of positive predictions.
- **Recall** $= \frac{TP}{TP+FN}$, reflecting the ability to find all outstanding keys.
- **F1 score** $= \frac{2\cdot\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}$, providing a balanced measure of both Precision and Recall.

They are the most basic test metrics, not yet including any constraints on reporting timeliness. However, even for this simple accuracy goal, the overhead of SOTA is far greater than ours.

Key results. Overall, QuantileFilter achieves a significant reduction in storage requirements, saving between 50 to 500 times more space than the SOTA algorithms at equivalent levels of accuracy. Furthermore, under constrained space conditions, such as when limited to 1MB, our solution attains an F1 accuracy of 99.77%, markedly surpassing the SOTA's F1 accuracy which falls below 25%.

As shown in Fig. 4, and 5, excluding our proposed method, SQUAD generally performs the best in most scenarios, with its metrics converging towards 100% as space limitations increase. HistSketch incorporates various data structures in its design, which allows for unbounded and unpredictable



(a) F1 score on Internet dataset. (b) Precision on Internet dataset. (c) Recall on Internet dataset (d) F1 score on Cloud dataset

Fig. 4: Comparison of accuracy.

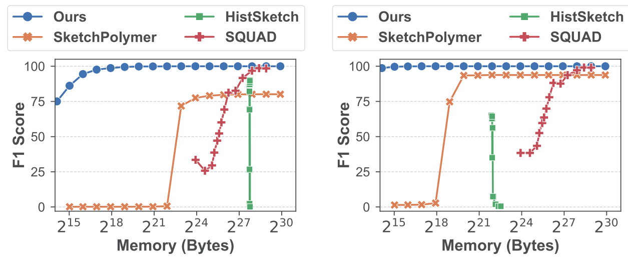
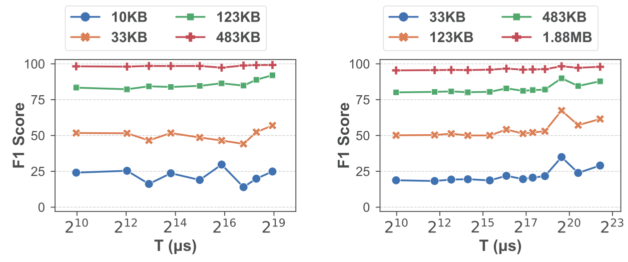
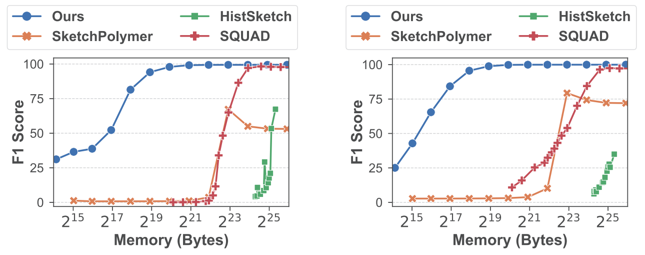


Fig. 5: Comparison of accuracy on Zipf dataset.



(a) Zipf_1 dataset. (b) Zipf_2 dataset.

Fig. 6: Accuracy under different values of T.



(a) 80%. (b) 99%.

Fig. 7: Accuracy *v.s.* Quantile Parameter δ.

space usage. Particularly in the Cloud dataset, characterized by its extensive number of keys, HistSketch typically demands around 1GB of space, irrespective of the parameter configurations. SketchPolymer demonstrates reasonable performance within certain space limitations (approximately 2^{23} bytes on the Internet). However, due to its design principle of discarding the earliest arriving values for each unique key, SketchPolymer is prone to inherent systematic recall errors, hindering accuracy improvements even with ample space. Additionally, when its space consumption drops below a certain threshold, SketchPolymer becomes inefficient, broadly misidentifying keys as outliers, which leads to very low precision but high recall.

Our algorithm, however, maintains a consistently high level of precision irrespective of the space constraints, with recall improving as more space becomes available and eventually converging to 100%. This indicates that our algorithm possesses a degree of unilaterality, making it particularly suitable for scenarios where stringent requirements are placed on reducing false positives. QuantileFilter efficiently utilizes our 1MB L2 cache even with minimal space occupation. In practical tests, when its space usage was 0.24, 0.47, 0.93, and 1.88MB, the hit rates achieved were 97%, 80%, 56%, and 40%, respectively. This demonstrates the value of its efficient memory usage. In Internet and cloud datasets, T is set to 300ms and 20s respectively, resulting in anomaly item proportions of 7.6% and 4.6%, respectively.

Effects of T. Fig. 6 shows that within the broad range of 1ms to 500ms for Internet Data and 1ms to 4096ms for Cloud, we can maintain accuracy relatively stable across various memory settings, demonstrating our ability to meet diverse T requirements set by users. The specific T value is determined based on the users' objectives. The resilience of T can be attributed to the implementation of the vague part in our algorithm, where a random +1/-1 coefficient is assigned for each key. This design ensures that changes in the proportion of abnormal items do not substantially alter the counter state in our solution.

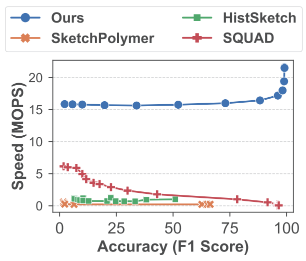
Effects of quantile δ. In Fig. 7, altering the queried quantile δ does not significantly diminish QuantileFilter's advantages. When the percentage is increased, it implies that the same key is more readily identified as an anomaly. This adjustment to some extent mitigates the issue of lower recall in SketchPolymer, but it cannot bridge the substantial performance gap between it and our approach.

C. Experiments on Speed

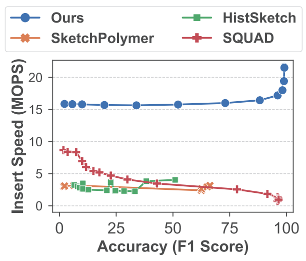
Metrics. We use **throughput** to evaluation the speed, in million operations per second (MOPS).

Key results. Overall, when accuracy exceeds 50%, our algorithm achieves a speed enhancement of 10 to 100 times compared to SOTA.

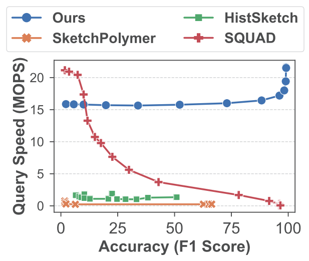
A significant advantage of QuantileFilter is its inherent capability for real-time monitoring. Unlike SOTA, which re-



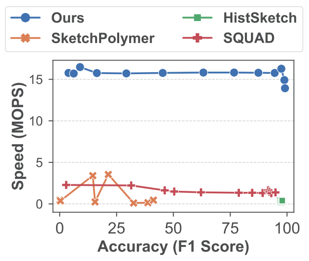
(a) Overall on Internet dataset.



(b) Insertion on Internet dataset.

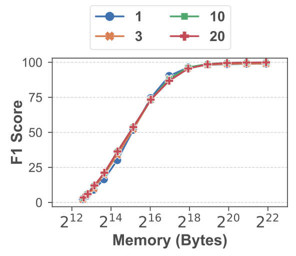


(c) Query on Internet dataset.

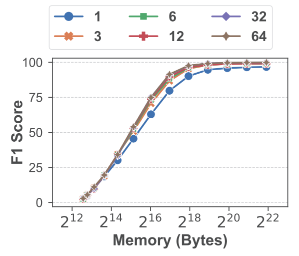


(d) Overall on Cloud dataset.

Fig. 8: Comparison of throughput.

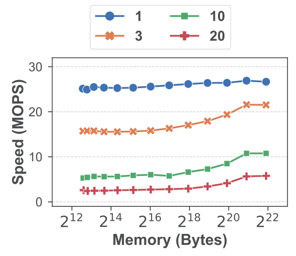


(a) Sketch number on Internet dataset.

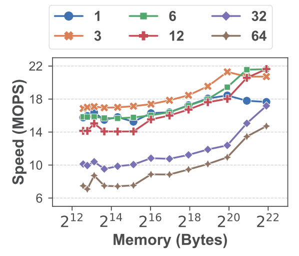


(b) Block length on Internet dataset.

Fig. 9: Accuracy on different parameter settings.



(a) Sketch number on Internet dataset.



(b) Block length on Internet dataset.

Fig. 10: Throughput on different parameter settings.

quires explicit querying after each item insertion, our approach eliminates this need. In fact, even if we only consider a part of SOTA's insertion and querying process, QuantileFilter still exhibits superior performance. For instance, on the Internet dataset, when our solution attains a 50% F1 Score, its speed reaches 15.76 MOPS, while SOTA's query and insert operations only manage 3.30 MOPS each at the same F1 Score, totaling just 1.65 MOPS.

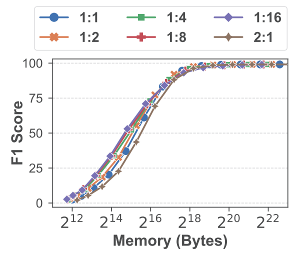
Additionally, our approach employs a strategy of initially querying the candidate part followed by the vague part, enhancing the hit rate of the candidate part when memory limitations are expanded. This avoids repetitive querying of the vague part, thereby further accelerating our solution's speed as space increases and precision improves. Conversely, SOTA's time for querying sharply rises with the expansion of space and improved accuracy, dropping to just 0.22 MOPS at a 96% F1 Score. This is 77.0 times slower than our algorithm at comparable levels of accuracy. The performance gap widens further as it approaches a 100% F1 Score. As illustrated in Fig. 8(d), this advantage is consistently sustained in a more stable manner on the Cloud dataset.

D. *In-Depth Exploration of QuantileFilter*

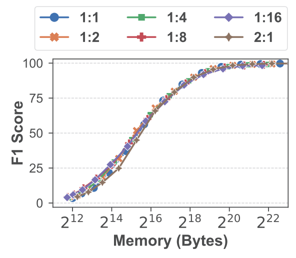
In this section, we present the impact of various algorithmic parameters and versions on QuantileFilter.

Effects of Array Number d **(Fig. 9(a) & 10(a)):** This parameter signifies the number of different hash functions in the vague part, and it also represents the number of items that need to be traversed during each query of the vague part. After enumerating this parameter from 1 to 20 and conducting corresponding tests, we discovered that it has a negligible impact on accuracy while exerting a certain influence on throughput. Considering that the median's characteristics are more advantageous when the sketch number is odd, thereby being less prone to disruption by extreme data, we selected 3 for its higher throughput.

Effects of Block Length (Fig. 9(b) & 10(b)): This parameter defines the maximum number of items each bucket in the candidate part can hold, and it is also the number of items that need to be traversed during each query of the candidate part. As shown in the figure, after exploring different values for this parameter, we found its behavior to be quite similar to that of the Sketch number, having only a minor impact on accuracy. Considering the potential memory waste associated with too small a block length, and its limited contribution to throughput enhancement, we therefore opted for the more prudent value of 6.



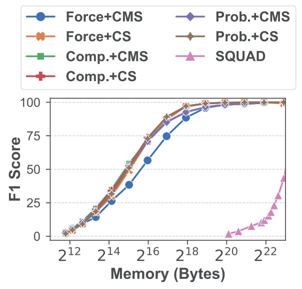
(a) Internet dataset.



(b) Cloud dataset.

Fig. 11: F1 score on different memory proportion.

Effects of Memory Proportion (Fig. 11): This parameter indicates the ratio of available space allocated to the vague and candidate parts of the data structure. The results show that the impact of memory ratio allocation is relatively minor when the differences on both sides are not significant. However, extreme allocations can lead to considerable fluctuations in accuracy. Therefore, for a more cautious approach, we chose the more


(a) F1 score on Internet dataset.

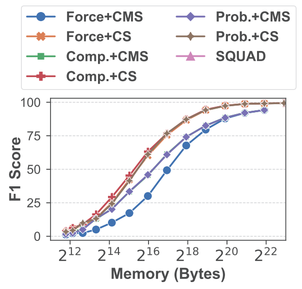

(b) F1 score on Yahoo.

Fig. 12: F1 score on different variants of QuantileFilter.

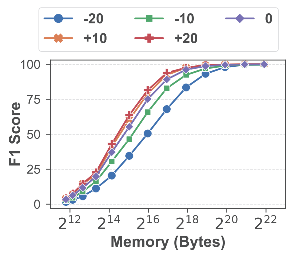
stable ratio of 1:4, where the vague approximately occupies 20% of the total space, and the candidate about 80%.

Effects of Algorithm Variants (Fig. 12): We explore three candidate election strategies (Comparative, Probabilistic, and Forceful) and two vague part types (CMS and Csketch (CS)), resulting in six distinct variants. Experimental results show that versions using CS perform better and are less affected by the candidate election strategies. For versions using CMS, effectiveness decreases in the order of Comparative, Probabilistic, and Forceful strategies. The throughput of the six variants—Force+CMS, Comp.+CMS, Prob.+CMS, Force+CS, Comp.+CS, and Prob.+CS–at a space of 245KB are 25.1, 24.3, 23.4, 18.9, 17.3, and 17.2 MOPS, respectively, showing minimal differences. Consequently, we select the Larger+CS combination, which demonstrates the highest accuracy. We also include SQUAD as a benchmark, but in figure 12(b), SQUAD, consuming about 30MB of memory, does not appear in the figure.

Effects of Dynamic Modification of Criteria. We modify the criteria parameters ϵ, δ, and T one at a time. For a chosen parameter, we modify it for half the keys, then compare the accuracy of modified and unmodified keys against the baseline scenario without modifications. The choice and timing of modifications are randomized to minimize strategy bias. We continue using F1 score and throughput as our accuracy and speed metrics. We utilized the Internet dataset. When the space is set to 128MB, introducing these modifications reduces QuantileFilter's throughput from 16MOPS to approximately 13MOPS, regardless of whether the modifications are to ϵ, δ, or T. Below are the specific impacts of the modifications on accuracy. Modifying ϵ (Fig. 13): For modified keys, making ϵ larger increases accuracy because labeling these keys as outliers becomes more challenging, and the algorithm's error has less impact on them. For the remaining unmodified keys, their accuracy is largely unaffected since changing ϵ does not influence the modification method of Qweight during insertion.

When modifying δ (Fig. 14): For modified keys, making δ smaller increases the error, as our algorithm is better suited for deltas close to 100%. Thus, the smaller the δ, the greater the error. For unmodified keys, a smaller δ means that the number added during the Qweight update of modified keys is smaller, thereby reducing the error for unmodified keys.

When modifying T (Fig. 15): For modified keys, making T smaller increases the error, because decreasing T leads to more abnormal quantiles, making detection more challenging. Therefore, the smaller the T, the greater the error. For unmodified keys, the smaller T is, the larger the number added during the Qweight update of modified keys, thus increasing the error for unmodified keys.


(a) Modified keys

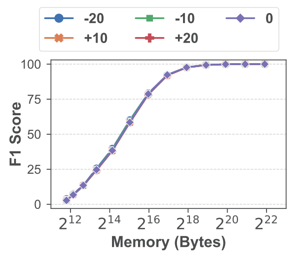
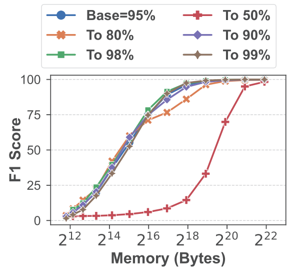

(b) Unmodified keys

Fig. 13: The effects of modifying ϵ.


(a) Modified keys

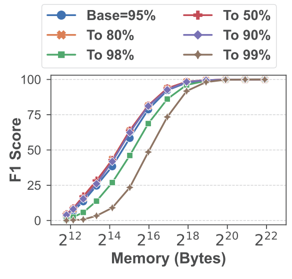
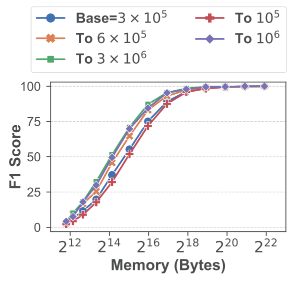

(b) Unmodified keys

Fig. 14: The effects of modifying δ.


(a) Modified keys

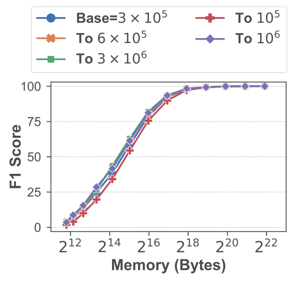

(b) Unmodified keys

Fig. 15: The effects of modifying T.

VI. Conclusion

While recent advances in quantile estimation for key-value streams have improved flexibility, they lag in query speed compared to the rapid data insertion. Our study introduces QuantileFilter, a novel algorithm designed for real-time detection of keys with quantiles above a threshold, crucial for spotting anomalies in streaming data. QuantileFilter significantly outperforms SOTA, offering 10 to 100 times faster processing and requiring 50 to 500 times less storage space, all while maintaining accuracy.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive feedback. This work is supported by National Key R&D Program of China (No. 2022YFB2901504), and National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009).

References

[1] Michael B Greenwald and Sanjeev Khanna. Quantiles and equi-depth histograms over streams. In *Data Stream Management: Processing High-Speed Data Streams*, pages 45–86. Springer, 2016.

[2] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 285–291, 2019.

[3] Daniel Ludtke, Dietmar Tutsch, and Matthias Kuhm. Quantile estimation for performance measures in network simulations with cinsim. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*, pages 111–112, 2007.

[4] Minje Park, Jaeseon Kim, Sungchul Shin, Cheolwoo Park, Jong-June Jeon, SoonSun Kwon, and Hosik Choi. Quantile estimation for encrypted data. *Applied Intelligence*, pages 1–10, 2023.

[5] David Umsonst, Justin Ruths, and Henrik Sandberg. Finite sample guarantees for quantile estimation: An application to detector threshold tuning. *IEEE Transactions on Control Systems Technology*, 31(2):921–928, 2022.

[6] Elena M Cimpoesu, Bogdan D Ciubotaru, and Dan Stefanoiu. Fault detection and diagnosis using parameter estimation with recursive least squares. In *2013 19th International Conference on Control Systems and Computer Science*, pages 18–23. IEEE, 2013.

[7] Naga K Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, 2005.

[8] Carla Sauvanaud, Kahina Lazri, Mohamed Kaâniche, and Karama Kanoun. Anomaly detection and root cause localization in virtual network functions. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 196–206. IEEE, 2016.

[9] Brian J Watson, Manish Marwah, Daniel Gmach, Yuan Chen, Martin Arlitt, and Zhikui Wang. Probabilistic performance modeling of virtualized resource allocation. In *Proceedings of the 7th international conference on Autonomic computing*, pages 99–108, 2010.

[10] Chanaka Ganewattha, Zaheer Khan, Janne J Lehtomäki, and Marja Matinmikko-Blue. Real-time quantile-based estimation of resource utilization on an fpga platform using hls. *IEEE Access*, 8:43301–43313, 2020.

[11] Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Shobha Venkataraman, and He Yan. Prometheus: Toward quality-of-experience estimation for mobile apps from passive network measurements. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, pages 1–6, 2014.

[12] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249, 2004.

[13] Fuheng Zhao, Punnal Ismail Khan, Divyakant Agrawal, Amr El Abbadi, Arpit Gupta, and Zaoxing Liu. Panakos: Chasing the tails for multidimensional data streams. *Proceedings of the VLDB Endowment*, 16(6):1291–1304, 2023.

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

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

[16] Fuheng Zhao, Sujaya Maiyya, Ryan Wiener, Divyakant Agrawal, and Amr El Abbadi. Kll$\pm$approximate quantile sketches over dynamic datasets. *Proceedings of the VLDB Endowment*, 14(7):1215–1227, 2021.

[17] Ted Dunning and Otmar Ertl. Computing extremely accurate quantiles using t-digests. *arXiv preprint arXiv:1902.04023*, 2019.

[18] Charles Masson, Jee E Rim, and Homin K Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment*, 12(12).

[19] Rana Shahout, Roy Friedman, and Ran Ben Basat. Together is better: Heavy hitters quantile estimation. *Proceedings of the ACM on Management of Data*, 1(1):1–25, 2023.

[20] Jiarui Guo, Yisen Hong, Yuhan Wu, Yunfei Liu, Tong Yang, and Bin Cui. Sketchpolymer: Estimate per-item tail quantile using one sketch. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 590–601, 2023.

[21] Jintao He, Jiaqi Zhu, and Qun Huang. Histsketch: A compact data structure for accurate per-key distribution monitoring. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 2008–2021. IEEE, 2023.

[22] Yinda Zhang, Peiqing Chen, and Zaoxing Liu. Octosketch: Enabling real-time, continuous network monitoring over multiple cores.

[23] Peiqing Chen, Dong Chen, Lingxiao Zheng, Jizhou Li, and Tong Yang. Out of many we are one: Measuring item batch with clock-sketch. In *Proceedings of the 2021 International Conference on Management of Data*, pages 261–273, 2021.

[24] Tong Yang, Jie Jiang, Peng Liu, and etal. Elastic sketch: Adaptive and fast network-wide measurements. In *SIGCOMM Conference*, 2018.

[25] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 2005.

[26] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. Quantiles over data streams: An experimental study. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 737–748, 2013.

[27] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings 2003 VLDB Conference*, pages 19–30. Elsevier, 2003.

[28] SQL Server Document. https://learn.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-db-stats-histogram-transact-sql?view=sql-server-ver16.

[29] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. Springer, 2002.

[30] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *SIGMCOMM Conference*, 2002.

[31] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[32] R. Pratanu, K. Arijit, and A. Gustavo. Augmented sketch: Faster and more accurate stream processing. In *Proc. ACM SIGMOD*, 2016.

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

[34] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *SIGMOD Conference*, 2018.

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

[36] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590, 2018.

[37] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019.

[38] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1574–1584, 2020.

[39] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. Salsa: self-adjusting lean streaming analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 864–875. IEEE, 2021.

[40] Peng Jia, Pinghui Wang, Junzhou Zhao, Ye Yuan, Jing Tao, and Xiaohong Guan. Loglog filter: Filtering cold items within a large range over high speed data streams. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 804–815. IEEE, 2021.

[41] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2285–2293, 2021.

[42] Kaicheng Yang, Sheng Long, Qilong Shi, Yuanpeng Li, Zirui Liu, Yuhan Wu, Tong Yang, and Zhengyi Jia. Sketchint: Empowering int with

towersketch for per-flow per-switch measurement. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

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

[44] Yuanpeng Li, Feiyu Wang, Xiang Yu, Yilong Yang, Kaicheng Yang, Tong Yang, Zhuo Ma, Bin Cui, and Steve Uhlig. Ladderfilter: Filtering infrequent items with small memory and time overhead. *Proceedings of the ACM on Management of Data*, 1(1):1–21, 2023.

[45] Yuhan Wu, Shiqi Jiang, Siyuan Dong, Zheng Zhong, Jiale Chen, Yutong Hu, Tong Yang, Steve Uhlig, and Bin Cui. Microscopesketch: Accurate sliding estimation using adaptive zooming. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2660–2671, 2023.

[46] Kaicheng Yang, Yuhan Wu, Ruijie Miao, Tong Yang, Zirui Liu, Zicang Xu, Rui Qiu, Yikai Zhao, Hanglong Lv, Zhigang Ji, and Gaogang Xie. Chamelemon: Shifting measurement attention as network state changes. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 881–903, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Rui Ding, Shibo Yang, Xiang Chen, and Qun Huang. Bitsense: Universal and nearly zero-error optimization for sketch counters with compressive sensing. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 220–238, 2023.

[48] Feiyu Wang, Qizhi Chen, Yuanpeng Li, Tong Yang, Yaofeng Tu, Lian Yu, and Bin Cui. Joinsketch: A sketch algorithm for accurate and unbiased inner-product estimation. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[49] Yikai Zhao, Wenchen Han, Zheng Zhong, Yinda Zhang, Tong Yang, and Bin Cui. Double-anonymous sketch: Achieving top-k-fairness for finding global top-k frequent items. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[50] José María Luna, Philippe Fournier-Viger, and Sebastián Ventura. Frequent itemset mining: A 25 years review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(6):e1329, 2019.

[51] Our open-sourced code. https://github.com/QuantileFilterGroup/QuantileFilter.

[52] The CAIDA UCSD Anonymized Internet Traces Dataset - 2018.03.15. http://www.caida.org/data/passive/passive_dataset.xml.

[53] Yahoo! Webscope dataset G4 - Yahoo! Network Flows Data, version 1.0. http://research.yahoo.com/Academic_Relations.