MagnifierSketch: Quantile Estimation Centered at One Point

Jiarui Guo*, Qiushi Lyu*, Yuhan Wu*, Haoyu Li†, Zhaoqian Yao*,
Yuqi Dong*, Xiaolin Wang*, Bin Cui*, and Tong Yang*
*Peking University †University of Texas at Austin

***Abstract*—In this paper, we take into consideration quantile estimation in data stream models, where every item in the data stream is a key-value pair. Researchers sometimes aim to estimate per-key quantiles (*i.e.* quantile estimation for every distinct key), and some popular use cases, such as tail latency measurement, recline on a predefined single quantile (*e.g.* 0.95- or 0.99-quantile) rather than demanding arbitrary quantile estimation. However, existing algorithms are not specially designed for per-key estimation centered at one point. They cannot achieve high accuracy in our problem setting, and their throughput are not satisfactory to handle high-speed items in data streams. To solve this problem, we propose MagnifierSketch for point-quantile estimation. MagnifierSketch supports both single-key and per-key quantile estimation, and its key techniques are named Value Focus, Distribution Calibration and Double Filtration. We provide strict mathematical derivations to prove the unbiasedness of MagnifierSketch and show its space and time complexity. Our experimental results show that the Average Error (AE) of MagnifierSketch is significantly lower than the state-of-the-art in both single-key and per-key situations. We also implement MagnifierSketch on RocksDB database to reduce quantile query latency in real databases. All related codes of MagnifierSketch are open-sourced and available at GitHub.**

I. Introduction

A. Background and Motivation

Quantile estimation is an important topic in many fields. In the field of statistics, researchers can have a better knowledge of the unknown parameters in the probability distribution function (pdf) by quantile estimation, which helps predict further sampling in the future [1]. In theoretical computer science, researchers make efforts to design elegant algorithms to estimate the quantile of a multiset with size N within $o(N)$ space complexity [2]–[5]. Quantile estimation also plays an important role in application, such as databases [6], meteorology [7], and finance [8].

When it comes to data stream models, it is important to estimate the quantile of distribution. Yet, few work targets at *per-key* quantile estimation. In a data stream, every item is a key-value pair (k, v), and we have to independently estimating quantile for every distinct key rather than viewing different keys as a whole. With the above preliminaries, single-key and per-key quantile estimation can be respectively summarized as follows:

```
-- Single-key quantile estimation
SELECT Quantile(value, w)
FROM DataStream
-- Per-key quantile estimation
SELECT Quantile(item.value, w)
FROM DataStream
WHERE item.key = key
```

where the function `Quantile(s, w)` returns the w-quantile of the multiset s. In fact, per-key quantile is more useful in many situations, and below we show some use cases.

Case 1: Latency Measurement. Latency is an important topic in network scenarios [9], [10]. It is defined as the delay in network communication, or the time from request to response. Users usually expect low latency when they are visiting a website, but the overall low latency cannot guarantee low latency for every user. Therefore, to attract as many visitors as possible, website administrators shall make efforts to achieve low latency quantile for every visitor [11], [12]. Also, the sudden increase of latency can imply potential cyber attacks or offending activities [13], [14]. However, the aggregated latency quantile can remain unchanged, as millions of packets with low latency can conceal this phenomenon. As a result, we can only detect the attack by per-key latency quantile estimation.

Case 2: Data Preprocessing. Data preprocessing is necessary in machine learning models [15], [16]. An effective method for data cleaning is to estimate per-key quantile. Since dirty data with extreme values usually have extremely high (low) percentage quantiles, we can detect and delete these values quickly [17], [18]. However, to obtain high-quality datasets for machine learning, traditional single-key algorithms either have to read the dataset multiple times, or have to build a data structure for every distinct key, resulting in a waste of time or space. Designing a per-key quantile estimation algorithm can solve this tough problem efficiently.

Also, while most prior work mainly focuses on all-quantile estimation (*i.e.* quantile estimation for arbitrary point), researchers sometimes are more interested in the quantile function at one point. For example, network administrators usually pay more attention to tail latency (*e.g.* 0.95 or 0.99-quantile of latency) in network scenarios, as many applications must achieve low tail latency to meet business objectives [19], [20]. In addition, the median (*e.g.* 0.5-quantile) can be used for estimation of income levels within population [21] and analysis of large amounts of transaction data in finance [22]. Finally, constructing $\lceil \frac{1}{\varepsilon} \rceil$ data structures for point-quantile estimation can solve the problem of all-quantile estimation within at most ε error, so point-quantile estimation also lays solid foundation for all-quantile estimation.

However, per-key point-quantile estimation can be challenging and costly in data stream models. We have to keep the

information of every item, but the high-speed items in data streams require us to deal with every item in $O(1)$ time. Also, to ensure that the data structure is small enough to be placed on caches, we have to use as little memory as possible. As a result, the goal of this paper is to design a compact sketch algorithm which can accurately estimate per-key point-quantile with small time and space complexity.

B. Prior Art and Their Limitations

We divide existing quantile estimation algorithms into two categories: *single-key* algorithms and *per-key* algorithms. Single-key algorithms can be used to estimate *aggregated* quantile, which means they view different keys as a whole and estimate the quantile of all values. Many algorithms belong to this category [3]–[5], [23]–[26]. However, although attractive in theory, they cannot be directly applied to per-key situation. In data stream models, a large number of items are mixed together, and these algorithms require several copies to realize per-key estimation, which is unacceptable in practice.

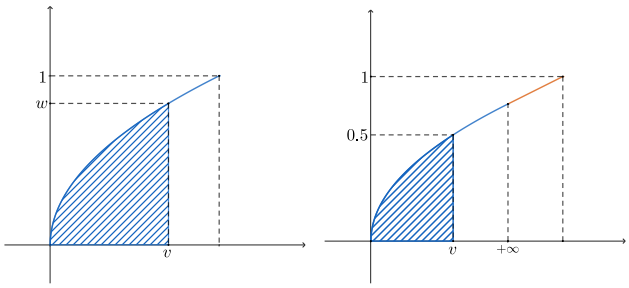
Per-key algorithms can be directly used for per-key quantile estimation. SQUAD [27], SketchPolymer [28] and M4 [29] are three of these per-key algorithms. These per-key algorithms are mainly designed for all-quantile estimation, so they can be applied for per-key point-quantile estimation as well. Nevertheless, in order to realize all-quantile estimation, these algorithms record too many value samples far from the target quantile, which is a waste of space in point-quantile situation. In addition, these algorithms fail to filter infrequent items efficiently, and infrequent items can have detrimental influence on the accuracy of frequent items.

C. Our Proposed Solution

Toward the design goal, we propose **MagnifierSketch** for per-key point-quantile estimation. MagnifierSketch is space-efficient: it is small enough to be placed in CPU caches. MagnifierSketch is fast: it deals with every item in constant time. MagnifierSketch is accurate: it achieves much smaller error compared to the state-of-the-art.

MagnifierSketch includes two stages: Stage 1 is a Tower Sketch [30] and Stage 2 is a hash table named Value Sketch. Tower Sketch is used to filter infrequent items in advance, and Value Sketch keeps samples of value for every frequent item. The techniques used in Tower Sketch and Value Sketch are named **Value Focus**, **Distribution Calibration** and **Double Filtration**. We now introduce these three techniques below:

Key Technique 1: Value Focus (§III-A). In point-quantile query, values close to the query quantile w shall be prioritized. Therefore, we design a special processing mechanism to keep as many values close to the target quantile as possible. MagnifierSketch maintains two data structures named the Candidate and the Representative in single-key situation. The value of every incoming item will be inserted into the Candidate, and only two median values can be inserted into the Representative after the Candidate is full when $w = 0.5$. The Representative further applies replacement strategy to evict unqualified values.



(a) before (b) after

Fig. 1: Idea of Distribution Calibration

In this way, MagnifierSketch can estimate quantile accurately by reporting the median value in the Representative.

Key Technique 2: Distribution Calibration (§III-B). We study data streams where the item values remain a stable distribution F in a certain interval [31], [32]. We find that this model is applicable in many scenarios, *e.g.* database [33], network monitoring [34] and commerce [35]. In addition, the performance of MagnifierSketch peaks when we set the query quantile $w = 0.5$, and merely querying 0.5-quantile is simpler than querying an arbitrary quantile. Consequently, we apply probability method to "calibrate" the original distribution F and construct a new distribution F', so that the w-quantile of the original distribution F is just the 0.5-quantile of F' (See in Figure 1). We prove that the expected quantile function at the query result *w.r.t.* F' is just 0.5, and MagnifierSketch still achieves high throughput after Distribution Calibration.

Key Technique 3: Double Filtration (§III-C-III-D). The majority of items in data streams are infrequent items [36], [37]. However, their quantile can hardly be estimated precisely due to their low frequency. As a result, it is necessary to filter infrequent items to save memory for frequent items. In MagnifierSketch, we filter infrequent items in both Tower Sketch (Stage 1) and Value Sketch (Stage 2). In Tower Sketch, MagnifierSketch leverages the idea of **Early Screening**, which originates from Cold Filter [38]. In Value Sketch, MagnifierSketch borrows the idea of **Ostracism** from Elastic Sketch [39] to evict infrequent items. In this way, we drop infrequent items and keep the information of frequent items as accurately as possible, which can be used for quantile estimation.

D. Key Contributions

This paper makes the following contributions:

- We propose a two-stage data structure, namely the Candidate and the Representative, to estimate point-quantile in single-key situation.
- We extend the algorithm to per-key situation and propose a novel data structure, namely MagnifierSketch, to estimate per-key point-quantile in data streams.
- We show the error bound and time complexity of MagnifierSketch by strict derivation. Mathematical analysis shows the superiority of MagnifierSketch in both single-key and per-key situations.
- We conduct extensive experiments on different datasets. Experimental results show that MagnifierSketch outperforms

other algorithms in terms of error and still maintains high throughput in both single-key and per-key situations.

II. Problem Statement And Related Work

A. Problem Statement

The symbols frequently used in this paper are shown in Table I.

Definition 1. ***Data Stream.*** *A data stream S is a series of items $\{e_1, e_2, \cdots, e_n, \cdots\}$ appearing in sequence. In this paper, every item e is a key-value pair (k, v). Items with different keys are called distinct items.*

Definition 2. ***Quantile.*** *Given a multiset of numbers $\mathcal{S} = \{a_1, a_2, \cdots, a_n\}$ and a percentage w, where $a_1 \leq a_2 \leq \cdots \leq a_n$ and $0 \leq w \leq 1$, the w-quantile of multiset $\mathcal{S}$ is defined as $a_{\lfloor w(n-1) \rfloor + 1}$. The quantile function at a_k is defined as $\frac{k-1}{n-1}$.*

TABLE I: Symbols frequently used in this paper.

Notation	Meaning
e	A distinct item in data streams
k	Key of a certain item
v	Value of a certain item
w	An arbitrary quantile
r	Maximum value samples in the Candidate
s	Maximum value samples in the Representative
T	Threshold for Tower Sketch
λ	Threshold for ratio in Value Sketch
$B[i][j]$	j^{th} cell in i^{th} bucket in Value Sketch
$h(.)$	The hash function for Value Sketch
d	Number of cells in each bucket
u	Number of buckets in Value Sketch
$vote^+$	The positive vote field
$vote^-$	The negative vote field

We now officially state the definition of point-quantile estimation and all-quantile estimation respectively.

Definition 3. ***Point-quantile Estimation.*** *Given a quantile w in advance, for n numbers $t_1, \cdots, t_n$ in data streams, construct a data structure which estimates the w-quantile of all these numbers.*

Definition 4. ***All-quantile Estimation.*** *For n numbers $t_1, \cdots, t_n$ in data streams, construct a data structure which, for an arbitrary quantile w, estimates the w-quantile of all these numbers.*

The difference between point-quantile and all-quantile estimation is that in point-quantile situation, we know the estimated quantile w in advance, so we can specially design an algorithm which solves w-quantile estimation. Finally, we state the design goal of this paper.

Definition 5. ***Per-key Point-quantile Estimation.*** *Given a fixed quantile w, for an arbitrary key k, the design goal of MagnifierSketch is to estimate the w-quantile of value for all items with key k.*

B. Related Work

1) Quantile Estimation:

Given a multiset S with N items, a quantile $0 < w < 1$ and an error parameter $0 < \varepsilon < \frac{1}{2}$, the design goal of (single-key) quantile estimation is to return an item $x \in S$, *s.t.* the quantile function at x (suppose it is $\hat{w}$) satisfies $|\hat{w} - w| < \varepsilon$. The most famous algorithm for this problem is GK algorithm [4]. It solves this problem within $O(\frac{1}{\varepsilon}\log(\varepsilon N))$ space complexity. Other algorithms for quantile estimation uses randomization to guarantee $|\hat{w} - w| < \varepsilon$ with probability at least $1 - \delta$. KLL algorithm [5] applies this technique and achieves $O(\frac{1}{\varepsilon}\log\log\frac{1}{\delta})$ space complexity for the first time. ReqSketch [24], [25] later guarantees $|\hat{w} - w| < \varepsilon w$ using $O(\frac{1}{\varepsilon}\log^{1.5}(\varepsilon N)\sqrt{\log\frac{1}{\delta}})$ space.

Other sketching algorithms are also designed for single-key quantile estimation problem. DDSketch [23] uses logarithm to divide all positive numbers into several discrete intervals, and records the frequency of each interval in the buckets. Moreover, DDSketch will merge adjacent buckets to save memory if it has too many buckets. Some algorithms utilize the fact that the data stream is often random-ordered (*i.e.* i.i.d.) to save memory. Guha and Mcgregor [40] first provide a method that achieves $O\left(\log\frac{1}{\varepsilon} + \log\log\frac{1}{\delta}\right)$ space complexity for random-ordered streams. t-digest [26] is another work for random-ordered data streams. It divides the data into small clusters and aggregates and compresses the data within each cluster. The size of each cluster is adaptively determined based on the density of the data distribution. Moreover, t-digest employs a technique called "centroids" to group similar data points together and achieves accurate quantile estimation for w close to 0 and 1.

Some algorithms are specifically designed for per-key quantile estimation. SQUAD [27] combines both sketching and sampling methods to estimate per-key quantile for heavy-hitters. It uses Reservoir Sampling to sample several items from the data stream, and employs Space Saving to construct GK arrays for these items, which can be further used for quantile estimation. SketchPolymer [28] is designed for estimating per-key tail quantile in data streams. It filters infrequent items in advance and applies Value Splitting and Sharing to count the frequency of every key in each interval for tail quantile estimation. M4 [29] is another framework in this field. For every single-key quantile estimation algorithm META, M4 constructs several layers of buckets with a META in each layer. Every distinct item will be mapped into several buckets by hash functions, and M4 aggregates information within different buckets to achieve per-key quantile estimation.

However, none of these algorithms can be directly used for per-key point-quantile estimation in our problem setting. As to single-key algorithms, they achieve low space complexity in theory, but they have to make several copies of their data structures to support per-key point-quantile estimation, which is unacceptable in terms of memory. In addition, even if per-key algorithms (SQUAD, SketchPolymer, M4) can be directly utilized for our problem, SQUAD does not filter infrequent items in advance, and these infrequent items have a detrimental influence on overall accuracy; SketchPolymer is designed for

estimating tail quantile, so it cannot be used for arbitrary point-quantile estimation. Finally, our experiments show that M4 cannot run within tight memory constraint, so it still has room for improvement in per-key quantile estimation.

2) Frequency Estimation:

Sketches are compact data structures which support approximate query with limited error, and they are widely used for frequency estimation in data streams. CM Sketch [41] is the simplest sketch for frequency estimation. It is composed of d counter arrays and each array is associated with a hash function. For every incoming item, CM Sketch uses d hash functions to map it into d counters and increments all these counters by 1. To report the frequency of an item, CM Sketch returns the minimum value of these d counters.

Tower Sketch [30] is a novel counting sketch originating from CM Sketch but uses different-sized counters for different arrays. Since Tower Sketch still allocates the same memory for different arrays, it has more small counters for infrequent items, which can keep the frequency of these items more accurately. Also, Tower Sketch has fewer counters for frequent items, and their frequencies are more likely to overflow in small counters. Therefore, frequent items suffer from over-estimation error in practice, and Tower Sketch can separate frequent items from infrequent items more effectively in data streams compared to other algorithms [42].

Elastic Sketch [39] consists of a hash table with several buckets. Each bucket records the key, positive votes $vote^+$ (equal to its frequency) and negative votes $vote^-$ (equal to the number of items colliding with it). To insert an item e, Elastic Sketch uses a hash function to map it to a bucket. If the item e matches the key in the bucket, the positive votes $vote^+$ will be incremented by 1; otherwise, the negative votes $vote^-$ will be incremented by 1. Moreover, Elastic Sketch applies a technique called Ostracism to evict infrequent items: If $\frac{vote^-}{vote^+}$ exceeds a predefined threshold, the item will be viewed as an infrequent item and will be evicted from the bucket.

III. MagnifierSketch Algorithm

In this section, we propose the data structure of MagnifierSketch. We start from the simplest situation when all items in the data stream share the same key and $w = 0.5$. Then we introduce the idea of Distribution Calibration and support an arbitrary w in single-key situation. We adapt the MagnifierSketch to per-key situation and introduce the Value Sketch. Finally we present an optimized version of MagnifierSketch to filter infrequent items in advance for higher accuracy. Due to space constraint, we put relevant pseudo-code of MagnifierSketch in Appendix A.

A. Single-key Situation when $w = 0.5$: Value Focus

Value Focus Rationale: The main idea of **Value Focus** is to maintain both the Candidate and the Representative in our data structure. Items in the data stream will be inserted into the Candidate first. When the Candidate is full, we select two value samples from the Candidate and insert them into the Representative. The Representative further evicts value samples far from the 0.5-quantile when it is full.

The Candidate Operation: The design goal of the Candidate is to keep several value samples from the data stream, and only values which have the potential to be the 0.5-quantile can be selected into the Representative. Specifically, the Candidate can keep r value samples in the data structure (we assume that r is an even number). For every item $e = (k, v)$ in the data stream, we first insert its value into the Candidate. Then we check whether the Candidate is full after insertion. If the number of value samples in the Candidate is smaller than r, we just finish insertion procedure and return; Otherwise, we select two median value samples in the Candidate and send them to the Representative. Finally, we clear all value samples in the Candidate to make room for further items in the data stream.

The Representative Operation: The design goal of the Representative is to record value samples close to the target quantile. The Representative can keep s value samples (we assume that s is also an even number). To insert two value samples v' and v'' (which are selected from the Candidate) into the Representative, we first check whether the Representative is full. If the Representative still has space for value samples, we just insert v' and v'' into the Representative; Otherwise, we traverse the Representative and find the smallest and largest value samples from s value samples in the Representative and the value samples to be inserted (suppose they are v_m and v_M respectively). The two smallest and largest value samples v_M and v_m will be evicted from the bucket and v' and v'' will be inserted into the Representative. Finally, to query for the 0.5-quantile, we regard value samples in the Representative and return the median in the Representative as the 0.5-quantile of all items.

Running Examples: For simplicity, we choose $r = s = 4$. Figure 2 shows three running examples in single-key situation. In the first example, to insert the value $v = 7$, we insert it into the Candidate. Since the Candidate is not full, no value will be inserted into the Representative and we finish the insertion procedure. In the second example, to insert the value $v = 11$, we insert 11 into the Candidate, and the Candidate will be full after insertion. Consequently, we select two median value samples from the Candidate (42 and 133 here). As the Representative still has space for these two value samples, 42 and 133 will be inserted into the Representative. In the last example, to insert the value $v = 35$, we insert 35 into the Candidate. The Candidate is full after insertion, so we select two median value samples (56 and 35 here) and insert them into the Representative. Since the Representative is full, we select the maximum and the minimum values from these six numbers (18 and 109 here). 18 and 109 will be evicted from the Representative, and 56 and 35 will take their place in the Representative.

B. Idea of Distribution Calibration

Suppose the value of all items in the data stream follows an arbitrary distribution F. To support arbitrary quantile estima-

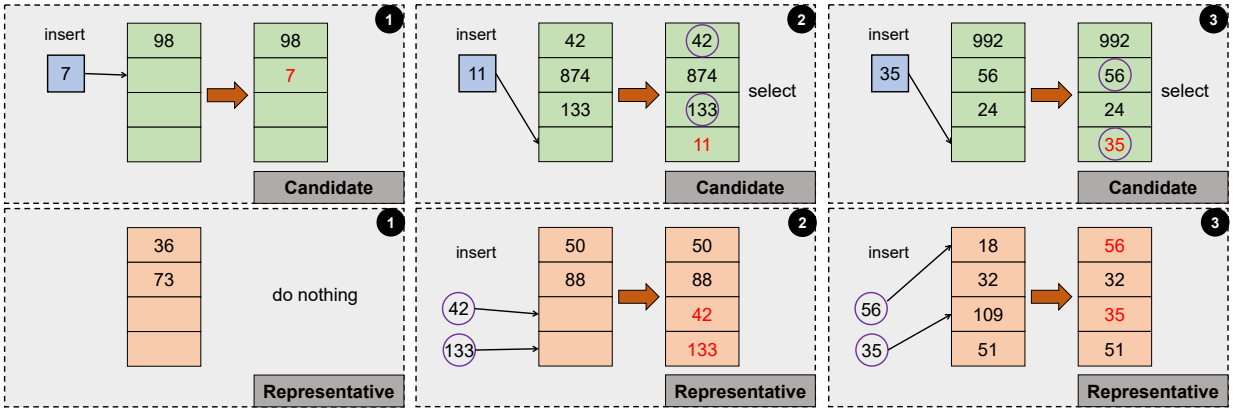


Fig. 2: Examples in Single-key Situation

tion in single-key situation, we apply **Distribution Calibration** technique. Its key idea is to construct a new distribution F', *s.t.* the w-quantile of the original distribution F is just the 0.5-quantile of the new distribution F'. Specifically, to "calibrate" the distribution, we apply probability method to generate several positive (negative) infinities and insert them into the data structure. When $w > 0.5$, we set a random variable Z following geometric distribution with parameter $\frac{1}{2w}$. We insert $Z-1$ positive infinities and v into MagnifierSketch. In this way, every value sample taken from the distribution F' follows the distribution F with probability $\frac{1}{2w}$, and the value sample will be positive infinity with probability $\frac{2w-1}{2w}$. The situation when $w < 0.5$ is similar, except that Z follows geometric distribution with probability $\frac{1}{2-2w}$, and we insert v together with $Z-1$ negative infinities instead of positive infinities into MagnifierSketch. We will prove that the expectation of quantile function at the query result is 0.5 *w.r.t.* F' in Section IV, hence MagnifierSketch gives an unbiased estimation of quantiles *w.r.t.* F'.

C. Per-key Situation: Value Sketch

Value Sketch Data Structure: We find that the technique called Ostracism, which is first proposed by Elastic Sketch [39] for frequency estimation, fits well for our problem setting in per-key quantile estimation. To cater for per-key situation in data streams, MagnifierSketch maintains a hash table with u buckets in the Value Sketch. Each bucket has d cells, and each cell records three fields: the key k, the positive votes $vote^+$, which is equal to its frequency numerically, and the value field, which includes the Representative and Candidate similar to single-key situation. Moreover, every bucket records the negative votes $vote^-$, which is equal to the number of items that collide in the bucket.

Value Sketch Insertion Operation: To insert an item $e = (k, v)$, Value Sketch first uses the hash function $h(.)$ to map e into the bucket $B[h(k)]$ and check the cells in the bucket. Specifically, there are three sub-cases:

Case 1: The key k of the item e matches the key in some cell $B[h(k)][j]$. In this case, we just increment its positive vote $vote^+$ by 1 and insert the value v of the item e into the value field in $B[h(k)][j]$. Specifically, we generate positive (negative) infinities by probability method, and insert these infinities and the value into the Candidate. If the Candidate is full, we select two medians from the Candidate, insert them into the Representative and clear the Candidate. Moreover, if the Representative is full, we evict the largest and smallest value samples to make room for new value samples, which is just similar to single-key situation.

Case 2: The key k of the item e does not match any key in bucket $B[h(k)]$, but the bucket still has empty cells. In this case, we insert the item $e = (k, v)$ into one empty cell (suppose it is $B[h(k)][j]$). The key in $B[h(k)][j]$ is set to k, and the positive vote $vote^+$ is set to 1. Finally, we insert the value v of the item e into the value field in $B[h(k)][j]$.

Case 3: The key k of the item e does not match any key in bucket $B[h(k)]$, and the bucket does not have empty cells. In this case, we increment the negative votes $vote^-$ of bucket $B[h(k)]$ by 1, which shows that hash collision happens. Then, we find the cell in bucket $B[h(k)]$ with minimum positive vote $vote^+$ (suppose it is $B[h(k)][j]$). We now check whether $\frac{B[h(k)].vote^-}{B[h(k)][j].vote^+} \geq \lambda$ holds: If the inequality holds after we increment $B[h(k)].vote^-$, we treat this item as an infrequent item: we evict the item from Value Sketch and insert $e = (k, v)$ into the cell by clearing $B[h(k)][j]$, initializing $B[h(k)][j].k$ to k, setting $B[h(k)][j].vote^+$ to 1, inserting the value v into the value field $B[h(k)][j].value$ and resetting $B[h(k)].vote^-$ to 0; If the inequality does not hold, then nothing happens: no item will be evicted from Value Sketch, and e will not be inserted into Value Sketch either.

Value Sketch Query Operation: The query operation is simple. To query the w-quantile of value for items with key k, we again use the hash function $h(.)$ to locate bucket $B[h(k)]$. Value Sketch then traverses the bucket to find the cell which records k. Value Sketch will return the query result of the value field as the w-quantile of value for items with key k.

Running Examples: For simplicity, we choose $d = u = 2$ and $\lambda = 1$. Figure 3 shows three running examples of Value Sketch in per-key situation. In the first example, to insert item $e = (k_3, 3)$, we use a hash function $h(.)$ to map it into the bucket $B[2]$. k_3 matches the key stored in the second cell, so we increment the $vote^+$ by 1 and insert the value 3 into the value field. In the second example, to insert item $f = (k_4, 9)$,

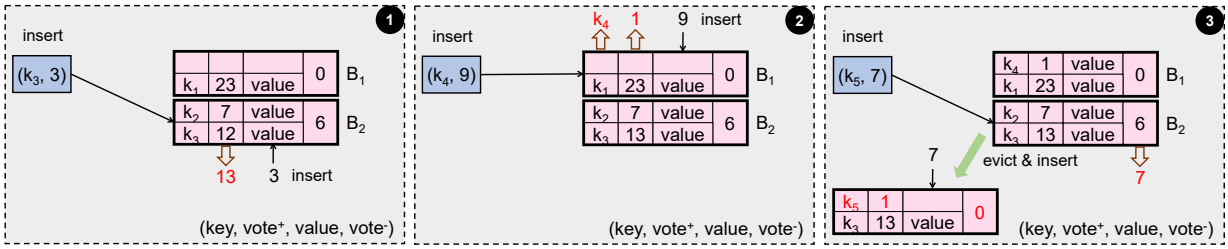


Fig. 3: Examples in Per-key Situation

we map it into the bucket $B[1]$. The key k_4 does not match any key in the bucket, but the first cell $B[1][1]$ is empty. Hence, we insert $(k_4, 9)$ into $B[1][1]$: the key field is set to k_4, the $vote^+$ is set to 1 and the value 9 is inserted into the value field. In the last example, to insert item $g = (k_5, 7)$, we map it into the bucket $B[2]$. The key k_5 does not match any key in the bucket, and the bucket does not have empty cell either. So we increment $vote^-$ by 1. Next we find the minimum $vote^+$ in the bucket $B[2]$, and check whether the inequality $\frac{vote^-}{vote^+} \geq \lambda$ holds. The cell $B[2][1]$ has the minimum $vote^+$, and $\frac{vote^-}{vote^+} = 1$. Consequently, we clear the cell $B[2][1]$ and insert $g = (k_5, 7)$ into $B[2][1]$: the key field is initialized as k_5, $vote^+$ is set to 1, the value 7 is inserted into the value field and $vote^-$ is reset to 0.

D. Optimization: Early Screening

Idea of Early Screening: To support per-key point-quantile estimation, MagnifierSketch records the value of both frequent and infrequent items. However, the quantile of infrequent items cannot be accurately estimated due to their low frequency. In addition, it is widely known that the distribution of real datasets is usually skewed, and items with low frequency make up the majority of the data stream. Inspired by the Cold Filter [38], MagnifierSketch proposes **Early Screening** to effectively filter infrequent items in advance: MagnifierSketch uses a Tower Sketch which records the frequency of every item, and only items with frequency exceeding the predefined threshold are allowed to enter Value Sketch. In this way, Tower Sketch automatically separates frequent items from infrequent items, and allows MagnifierSketch to keep the information of frequent items in Value Sketch as accurately as possible.

MagnifierSketch Insertion Operation: To insert an item $e = (k, v)$, MagnifierSketch first uses its key k to query Tower Sketch to get its frequency. If its frequency exceeds the threshold T, we insert $e = (k, v)$ into Value Sketch; Otherwise, we simply insert e into Tower Sketch and return.

MagnifierSketch Query Operation: The query operation is simple: To query the w-quantile of value for items with key k, MagnifierSketch just query Value Sketch to get the estimation: namely, use the hash function $h(.)$ to locate the bucket $B[h(k)]$ in Value Sketch, get the cell which records the information of key k, and query the value field to give an estimation of the quantile.

IV. Mathematical Analysis

In this section, we conduct mathematical analysis for MagnifierSketch. We first prove that the query result of MagnifierSketch is unbiased on the new distribution in Section IV-A. Then we derive the error bound for both the Candidate and the Representative, and give the space complexity in single-key situation in Section IV-B. We give an error bound for Value Sketch in per-key situation on Section IV-C. Finally we analyze the overall time complexity of MagnifierSketch in Section IV-D.

A. Proof of Unbiasedness

In this part, we assume that the value of all items in the data stream follows an arbitrary continuous distribution F, and the calibrated distribution is F'. All quantiles are *w.r.t.* the calibrated distribution F'. We prove that the expectation of quantile function at the query result *w.r.t.* F' is just 0.5 in single-key situation.

Theorem 1. *For every value or infinity v inserted into Value Sketch, the distribution of the quantile function at v (w.r.t. F') is uniform distribution on the interval* $[0, 1]$.

Proof. Without loss of generality, we assume $w > 0.5$. Suppose the quantile functions at v *w.r.t.* F and F' are $\hat{w}$ and $\tilde{w}$ respectively, then for $0 \leq x \leq \frac{1}{2w}$,

$$\mathbb{P}(\tilde{w} \leq x) = \mathbb{P}(v \text{ is not infinity}) \cdot \mathbb{P}(\hat{w} \leq 2wx) = \frac{1}{2w} \cdot 2wx = x.$$

Similarly, we can prove that $\mathbb{P}(\tilde{w} \leq x) = x$ for $\frac{1}{2w} \leq x \leq 1$, hence $\tilde{w}$ obeys a uniform distribution on $[0, 1]$. □

Theorem 2. *Suppose MagnifierSketch returns v as the query result, and the quantile function at v is $\tilde{w}$, then the expectation of $\tilde{w}$ is 0.5.*

Proof. The proof is based on symmetry: for every two value samples $v_1 < v_2$ sent to the Representative, suppose the quantile function at v_1 and v_2 are $\tilde{w}_1$ and $\tilde{w}_2$ respectively, then $\tilde{w}_1$ and $1 - \tilde{w}_2$ have identical distribution. As a result, $\tilde{w}$ and $1 - \tilde{w}$ have identical distribution, and $\mathbb{E}\tilde{w} = \mathbb{E}(1 - \tilde{w})$. Hence $\mathbb{E}\tilde{w} = 0.5$. □

B. Error Bound in Single-key Situation

In this part, we analyze the property of the Candidate and the Representative in single-key situation respectively. We first derive the error bound of the Candidate, then we use the error bound of Candidate to show the error bound of the Representative. Finally we prove that the space complexity of the Candidate and the Representative in single-key situation is $O\left(\frac{1}{\varepsilon}\sqrt{\log\frac{1}{\delta}}\right)$. We assume that the value of all items

in the data stream follows an arbitrary distribution F, and the calibrated distribution is F'. All quantiles are *w.r.t.* the calibrated distribution F' in this part.

Theorem 3. *For every value v which is sent from the Candidate to the Representative, suppose the real quantile function at v is $\hat{w}$. Then for any small positive number ε,*

$$\mathbb{P}(\hat{w} - 0.5 < -\varepsilon) = \mathbb{P}(\hat{w} - 0.5 > \varepsilon) \leq e^{-\varepsilon^2 r}. \quad (1)$$

To prove Theorem 3, we first cite the famous Chernoff Bound.

Lemma 4 (Chernoff Bound)**.** *Suppose $X_1, \cdots, X_n$ are i.i.d. random variables taking values in $\{0, 1\}$. Let $X = X_1 + \cdots + X_n$ denote their sum and $\mu = \mathbb{E}X$ denote its expectation. For any positive number $\eta > 0$, we have*

$$\mathbb{P}(X \geq (1+\eta)\mu) \leq e^{-\eta^2\mu/2}. \quad (2)$$

Then we apply this bound to prove Theorem 3.

Proof. Assume r value samples in the Candidate are $v_1, \cdots, v_r$, and $w_1, \cdots, w_r$ are the quantile function at $v_1, \cdots, v_r$ respectively. We define indicative variables

$$X_j = 1_{\{w_j < 0.5-\varepsilon\}}, Y_j = 1_{\{w_j > 0.5+\varepsilon\}}, j = 1, \cdots, r.$$

Let $X = X_1 + \cdots + X_r$ and $Y = Y_1 + \cdots + Y_r$, then $\mathbb{E}X = \mathbb{E}Y = (0.5 - \varepsilon)r$. Let $\eta = \frac{\varepsilon}{0.5-\varepsilon}$. Applying Lemma 4, we get

$$\mathbb{P}(X \geq (1+\eta)\mathbb{E}X) = \mathbb{P}(Y \geq (1+\eta)\mathbb{E}Y) \leq e^{-\frac{\eta^2 \mathbb{E}X}{2}} \approx e^{-\varepsilon^2 r}.$$

Note that $\hat{w} - 0.5 < -\varepsilon$ if and only if $X \geq 0.5r$, and $\hat{w} - 0.5 > \varepsilon$ if and only if $Y \geq 0.5r$, so we get the result. □

Theorem 5. *Assume that the number of items in the data stream is sufficiently large. Suppose the Representative reports v as the query result, and the real quantile function at v is $\tilde{w}$. Then for any small positive number ε,*

$$\mathbb{P}(|\tilde{w} - 0.5| > \varepsilon) \leq 2e^{-\varepsilon^2 rs}. \quad (3)$$

To prove Theorem 5, we first show a lemma in stochastic process and prove it.

Lemma 6 (Random Walk with Reflecting Barrier)**.** *Consider a random walk in $\{0, 1, \cdots, s\}$, with transition probability*

$$\begin{aligned} &\mathbb{P}(X_{l+1} = i+1 | X_l = i) = x, && 0 \leq i \leq s-1, \\ &\mathbb{P}(X_{l+1} = i-1 | X_l = i) = y, && 1 \leq i \leq s, \\ &\mathbb{P}(X_{l+1} = i | X_l = i) = 1 - x - y, && 1 \leq i \leq s-1, \end{aligned}$$

and

$$\mathbb{P}(X_{l+1} = 0 | X_l = 0) = 1 - x, \mathbb{P}(X_{l+1} = s | X_l = s) = 1 - y,$$

where X_l denotes the position at time l. Its stationary distribution is

$$\pi_i = \frac{\left(\frac{x}{y}\right)^i}{\sum_{j=0}^{s} \left(\frac{x}{y}\right)^j}. \quad (4)$$

Then we use the lemma to prove Theorem 5.

Proof. Define X_k as the number of value samples in the Representative which are smaller than the $(0.5-\varepsilon)$-quantile after k value samples are sent to the Representative, and $x = p^2, y = (1-p)^2$, where p is equal to $\mathbb{P}(\hat{w} - 0.5 < -\varepsilon)$ and $\mathbb{P}(\hat{w} - 0.5 > \varepsilon)$ in Theorem 3. It is easy to verify that X_k obeys the random walk with reflecting barrier in Lemma 6. Since the number of items in the data stream is sufficiently large, we assume that the probability distribution of X_k converges to the stationary distribution. Notice that $\tilde{w} - 0.5 < \varepsilon$ if and only if $X_k \geq 0.5s$. Hence

$$\begin{aligned} \mathbb{P}(\tilde{w} - 0.5 < -\varepsilon) = \mathbb{P}(X_k \geq 0.5s) &\approx \sum_{i=0.5s}^{s} \pi_i \\ = \frac{u^{0.5s}(1 - u^{s-0.5s})}{1 - u^s} \approx u^{0.5s} &= \left(\frac{p}{1-p}\right)^s \approx p^s, \end{aligned}$$

where $u = \frac{x}{y}$. Similarly, we can prove

$$\mathbb{P}(\tilde{w} - 0.5 > \varepsilon) \leq p^s.$$

Applying union bound, we get

$$\mathbb{P}(|\tilde{w} - 0.5| > \varepsilon) \leq 2p^s \leq 2e^{-\varepsilon^2 rs}. \quad \square$$

Remark. *According to the Dobrushin Theorem [43], the convergence rate of an irreducible aperiodic Markov chain with finite states is exponential, so the distribution of the Representative converges quickly.*

Corollary 7. *Given two small positive number ε and δ, to guarantee that $|\tilde{w} - 0.5| \leq \varepsilon$ with probability at least $1 - \delta$, the Candidate and the Representative needs $O\left(\frac{1}{\varepsilon}\sqrt{\log\frac{1}{\delta}}\right)$ memory.*

Proof. Let $r = s$ and $2e^{-\varepsilon^2 rs} \leq \delta$, we get

$$s \geq \frac{1}{\varepsilon}\sqrt{\log\frac{2}{\delta}},$$

which shows that $r + s = 2s = O\left(\frac{1}{\varepsilon}\sqrt{\log\frac{1}{\delta}}\right)$. □

C. Error Bound in Per-key Situation

In this part, we first derive the probability of hash collision in Value Sketch, then we use this conclusion to give an error bound for Value Sketch in per-key situation.

Theorem 8. *For any bucket in Value Sketch, the probability of hash collision is*

$$P_{hc} = 1 - e^{-\frac{H}{u}} \sum_{i=0}^{d} \frac{1}{i!} \left(\frac{H}{u}\right)^i, \quad (5)$$

where H is the number of distinct items entering Value Sketch.

Proof. The proof is first given in [39] when $d = 1$. We now extend the conclusion to $d \geq 2$. There are totally H distinct items which enter Value Sketch, and every distinct item is randomly mapped into a bucket by the hash function $h(.)$. Given an arbitrary bucket and an item $e = (k, v)$, the probability that e is mapped to the bucket is $\frac{1}{u}$. Therefore, for any bucket, the number of distinct items mapped to the bucket

W follows a Binomial distribution $B\left(H, \frac{1}{u}\right)$. When both H and u are large, the distribution of W can be approximated by Poisson distribution $\pi\left(\frac{H}{u}\right)$, hence

$$P(W=i)=\frac{1}{i!}\left(\frac{H}{u}\right)^{i} e^{-\frac{H}{u}}.$$

Notice that hash collision happens if and only if $W \geq d+1$, hence

$$\begin{aligned} P_{hc} &= P(W \geq d+1) = 1 - P(W \leq d) \\ &= 1 - e^{-\frac{H}{u}} \sum_{i=0}^{d} \frac{1}{i!}\left(\frac{H}{u}\right)^{i}. \end{aligned}$$

□

Corollary 9. *Let $r = s = \frac{1}{\varepsilon}\sqrt{\log\frac{2}{\delta}}$ in Corollary 7. For an arbitrary key k, suppose Value Sketch reports v as the query result, and the real quantile function at v is $\tilde{w}$. Then Value Sketch guarantees $|\tilde{w}-0.5| \leq \varepsilon$ with probability at least $(1-\delta)(1-P_{hc})$.*

D. Time Complexity

In this part, we analyze the amortized time complexity of MagnifierSketch in both single-key and per-key situations. Since MagnifierSketch has a small probability to insert a large number of infinities, we will only show the *expectation* of amortized time complexity of MagnifierSketch.

Theorem 10. *The expectation of amortized time complexity to insert an arbitrary item $e=(k,v)$ is at most $O\left(\frac{s}{r}\right)$ in single-key situation.*

Proof. To insert item e, MagnifierSketch sets Z to generate positive (negative) infinities. The expected number of positive (negative) infinities inserted into the Representative before e is inserted is $|2w-1|$, which is a constant. Every value or infinity will first be inserted into the Candidate, and if the Candidate is full, then two value samples in the Candidate will be inserted into the Representative. The insertion into the Candidate costs $O(1)$ time, and finding the median among r value samples costs $O(r)$ time. If the Representative is full, then MagnifierSketch will find the maximum and minimum value in the Representative and evict them, which costs $O(s)$ time. In conclusion, if we insert r values or infinities into the Candidate, then the time complexity is at most $O(r+s)$. Hence the expectation of amortized time complexity to insert e is at most $O\left(\frac{r+s}{r}\right) = O\left(\frac{s}{r}\right)$ in single-key situation. □

Remark. *Although MagnifierSketch occasionally inserts a large number of infinities, the expectation number of generated infinities to insert a value is $|2w-1|$, and it is smaller than 1, so the insertion throughput of MagnifierSketch will only be halved due to generating infinities in the worst case (when w is close to 0 or 1). In addition, by Kolmogorov's Strong Law of Large Numbers (SLLN), the median in the Representative will rarely be infinity, so MagnifierSketch gives a valid estimation in most cases.*

Theorem 11. *The expectation of amortized time complexity to insert an arbitrary item $e=(k,v)$ is at most $O\left(\frac{s}{r}+d\right)$ in per-key situation.*

Proof. To insert item e, MagnifierSketch first query Tower Sketch to get its frequency. If its frequency does not exceed the threshold T, e will only be inserted into Tower Sketch and return, which can be done in $O(1)$ time. Otherwise, MagnifierSketch uses a hash function $h(.)$ to locate a bucket and traverse the bucket in Value Sketch, which costs $O(d)$ time. If e is in the bucket or e is not in the bucket but the bucket still has empty cells, then e will be inserted into the cell, the expectation of amortized time complexity is $O\left(\frac{s}{r}\right)$. If e is not in the bucket and the bucket does not have empty cells, MagnifierSketch will just increment $vote^{-}$ and return. In conclusion, the expectation of amortized time complexity to insert e is at most $O\left(\frac{s}{r}+d\right)$ in per-key situation. □

As a result, if r and s are close numbers, and d is not very large, then the overall time complexity to insert an item e can be viewed as $O(1)$, which shows that MagnifierSketch can catch up with the high-speed items in the data stream and achieve high throughput.

V. Experimental Results

In this section, we provide experimental results with MagnifierSketch. First, we describe the experimental setup in Section V-A. Then, we show how parameter settings affect MagnifierSketch performance in Section V-B. We compare the performance of MagnifierSketch with state-of-the arts in both single-key and per-key situations in Section V-C and V-D respectively. Finally, we implement MagnifierSketch on RocksDB database to reduce quantile query latency in Section V-E. All related codes are released on GitHub [44].

A. Experimental Setup

Implementation: We implement MagnifierSketch and all other algorithms in C++. In all experiments, we use Bob Hash [45] with different hash seeds to implement the hash functions.
Computation Platform: We conducted all the experiments on a server with one 18-core processor (36 threads, Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz) and 128 GB DRAM memory. The processor has 64KB L1 cache, 1MB L2 cache for each core, and 24.75MB L3 cache shared by all cores.
Metrics:
1) Average Error (AE): Suppose we query w-quantile of $k_1, \cdots, k_n$, and the quantile function at the query results are $\hat{w}_1, \cdots, \hat{w}_n$, the average error is defined as $\frac{1}{n}\sum_{i=1}^{n}|\hat{w}_i - w|$.
2) Throughput: We use million of operations (insertions and queries) per second (Mops) to measure the throughput. We repeat the experiment for 10 times and calculate the average results as our throughput.
Datasets:
1) CAIDA Dataset: This Dataset is streams of anonymous IP traces collected from 2016 by CAIDA [46]. We regard the interval of two consecutive packets as its value. We use 20 million items.

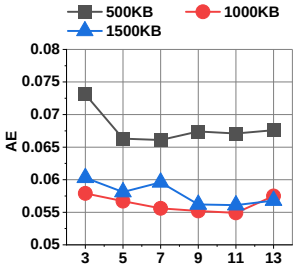


Fig. 4: Effects of d

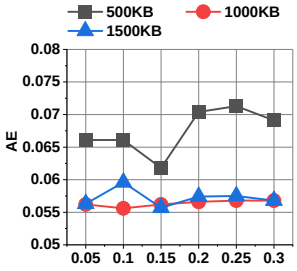


Fig. 5: Effects of q

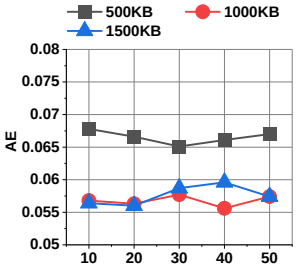


Fig. 6: Effects of T

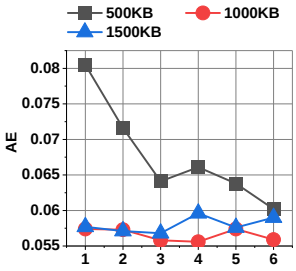


Fig. 7: Effects of λ

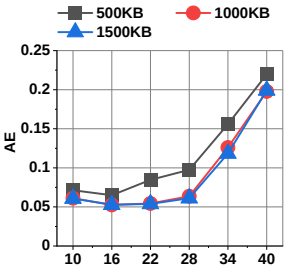


Fig. 8: Effects of r

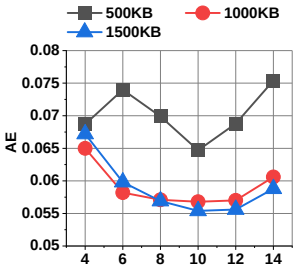


Fig. 9: Effects of s

2) Campus Dataset: The Campus Dataset is comprised of IP packets captured from the network of our campus. We regard the flow[1] ID as the key, and latency as the value. We use 14 million items

3) Seattle Dataset: The Seattle Dataset [47], [48] consists of round-trip times (RTT) between 99 nodes in the Seattle network systems. We treat RTTs from the same node as the same key, and RTTs as the value.

4) Synthetic Dataset: We generate the Synthetic Dataset following the Pareto distribution with $\alpha = 1$. We regard the interval between two consecutive items with the same key as its value. We use 20 million items.

Baseline Solution: In single-key situation, we compare the performance of MagnifierSketch with GK, KLL, DDSketch, t-digest and ReqSketch. In per-key situation, we compare the performance of MagnifierSketch with GK, KLL, DDSketch, t-digest, ReqSketch, SQUAD and SketchPolymer. Since GK, KLL, DDSketch, t-digest and ReqSketch do not focus on estimating per-key quantiles, we make a small modification to apply these algorithms for our problem, which is similar to [28]: For every algorithm, suppose it consumes m memory in single-key situation, and we allocate M memory on aggregate, then we construct $\frac{M}{m}$ buckets, and each bucket contains a copy of the data structure. For every coming item in the data stream, we use one hash function to map its key into a bucket, and record its value in the mapping bucket. Items mapped into the same bucket can be viewed to share the same key. Thus, to query the w-quantile of a key, we use the same hash function to map the key into a bucket, and calculate the quantile using these algorithms.

B. Experiments on Parameter Settings

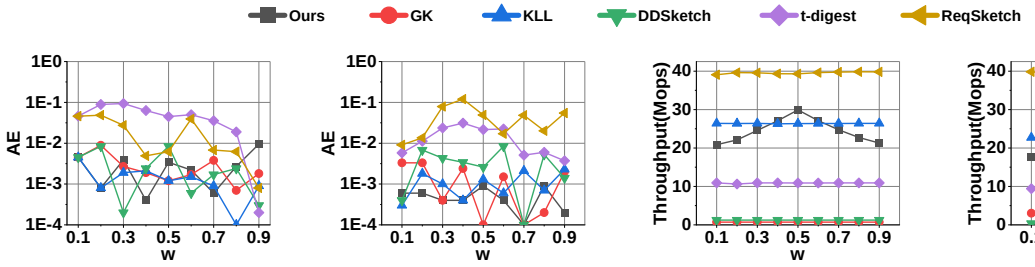
In this section, we measure the effects of some key parameters of MagnifierSketch, namely, the number of cells in each bucket d, the ratio of the memory size of Tower Sketch to the memory size of the whole MagnifierSketch q, the threshold for Tower Sketch T, the threshold for ratio in Value Sketch λ, the size of the Candidate r, and the size of the Representative s. We do not measure the effect of the number of buckets in Value Sketch u, as u can be calculated once the total memory and other parameters are fixed. We use CAIDA dataset in these experiments, and AE to measure these effects.

Effects of the number of cells in each bucket d (Figure 4): *Experimental results show that the best option of d is between 7 and 11.* We vary d from 3 to 13 in each experiment, and results show that when the memory size is 500KB, MagnifierSketch achieves lowest AE when $d = 7$. When the memory size is 1000/1500KB, MagnifierSketch performs best when $d = 11$. In fact, users can tune the parameter d to make a trade-off between accuracy and speed: a larger d will make MagnifierSketch more accurate at the sacrifice of overall throughput, as MagnifierSketch has to traverse the bucket to find a cell when inserting an item in Value Sketch. Thus, we choose $d = 7$ by default.

Effects of the ratio of the memory size of Tower Sketch to the memory size of the whole MagnifierSketch q (Figure 5): *Experimental results show that a small proportion of Tower Sketch can take up the role of filtering infrequent items.* We vary q from 0.05 to 0.3 in total memory from 500KB to 1500KB, and results show that the best choice of q is between 0.1 to 0.15. Taking into consideration that Value Sketch keeps the information of every frequent item, we allocate most memory to Value Sketch, so we set $q = 0.1$ in other experiments.

Effects of the threshold for Tower Sketch T (Figure 6): *Experimental results show that the best option of T is among 20 to 40.* In each experiment, we try different values of T, and results show that the best option of T is 30/40/20 within 500/1000/1500KB memory respectively. Since setting a larger T can filter as many infrequent items as possible in Tower

[1]A flow is generally defined as a part of the five-tuple: source IP address, destination IP address, source port, destination port, and protocol.

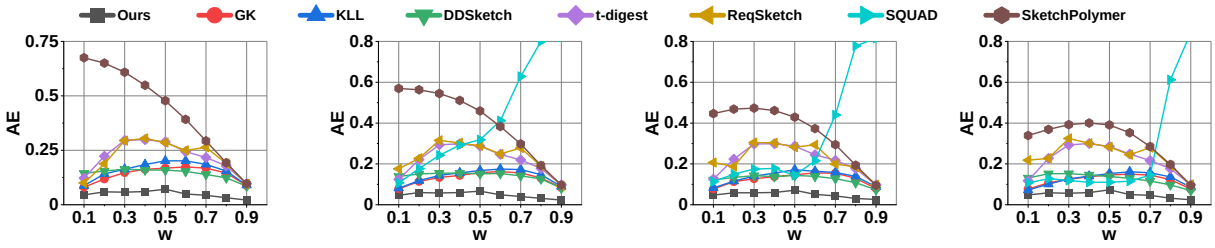


(a) CAIDA (b) Synthetic

Fig. 10: AE in Single-key Situation

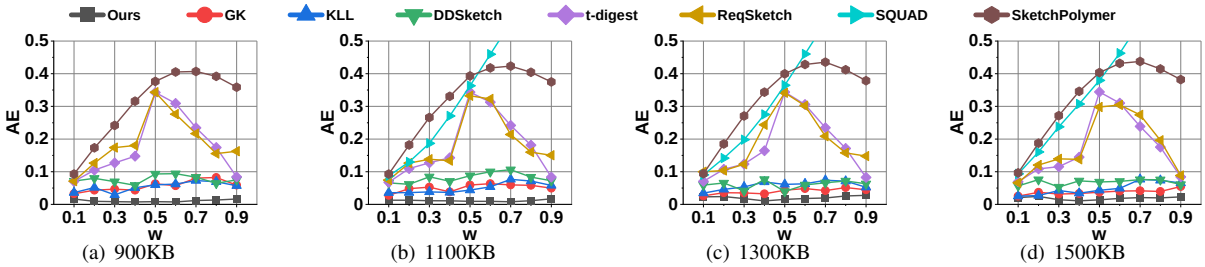
(a) CAIDA (b) Synthetic

Fig. 11: Insertion Throughput in Single-key Situation



(a) 900KB (b) 1100KB (c) 1300KB (d) 1500KB

Fig. 12: AE on CAIDA Dataset in Per-key Situation



(a) 900KB (b) 1100KB (c) 1300KB (d) 1500KB

Fig. 13: AE on Campus Dataset in Per-key Situation

Sketch, we choose $T = 40$.

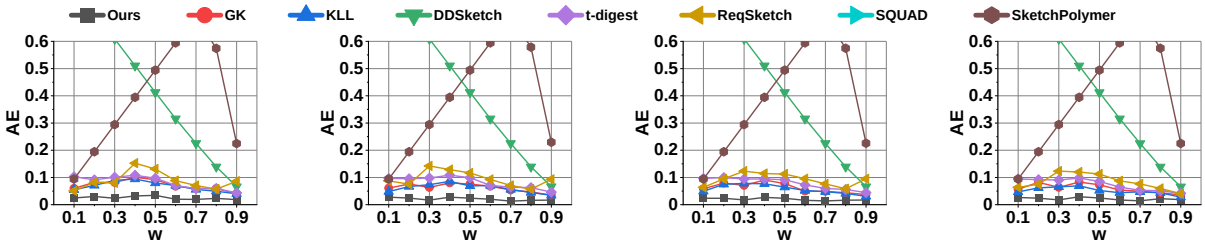
Effects of the threshold for ratio in Value Sketch λ (Figure 7): *Experimental results show that the best option of λ is between 3 to 6.* We set the memory from 500KB to 1500KB, and vary λ from 1 to 6 in the experiment. The results show that the best option of λ is 6/4/3 within 500/1000/1500KB memory respectively. If λ is too small, items might be replaced too frequently, which results in the loss of information in the data stream. If λ is too large, the mechanism of Ostracism wouldn't do much good. As a result, we choose $\lambda = 4$.

Effects of the size of the Candidate r (Figure 8): *Experimental results show that the best r is 16.* In order to choose 2 median value samples to insert to the Representative when the Candidate is full, the size of the Candidate should be an even number. We vary r from 10 to 40, and experimental results show that the best option of r is always 16 when total memory is 500/1000/1500KB. In fact, if we keep the total memory of Value Sketch unchanged, then a larger r will result in a smaller u, so hash collision will be inevitable. As a result, we set $r = 16$ by default.

Effects of the size of the Representative s (Figure 9): *Experimental results show that the best choice of s is 10.* Since we insert two numbers into the Representative every time, we should set s to be an even number, and we vary it from 4 to 14. The experimental results show that the best option of s is always 10 when the total memory is 500/1000/1500KB. In fact, the more value samples the Representative has, the fewer buckets Value Sketch contains. To ensure that there are enough buckets in Value Sketch, we set $s = 10$ by default.

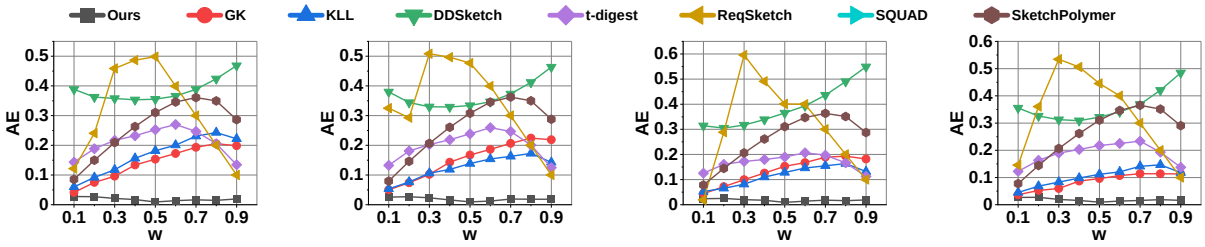
C. Experiments in Single-key Situation

In this section, we compare the performance of MagnifierSketch with prior work (GK, KLL, DDSketch, t-digest, ReqSketch) on CAIDA dataset and Synthetic dataset in single-key situation. To support single-key insertion and query, we ignore the key of all items in the data stream and we set the total memory to 10KB for every algorithm. *Experimental results* (Figure 10-11) *show that MagnifierSketch is as good as other algorithms in terms of AE, but MagnifierSketch runs much faster.* The AE of MagnifierSketch is far less than that of t-digest and ReqSketch, but it's not much different from other algorithms. The insertion throughput of MagnifierSketch is around 20M item per second, which is around 2 times higher than t-digest, 7 times higher than GK, 15 times higher than DDSketch and only lower than KLL and ReqSketch. Moreover, the insertion throughput of MagnifierSketch peaks when $w = 0.5$, as we have to insert positive (negative) infinities by probability method when $w \neq 0.5$, and the larger



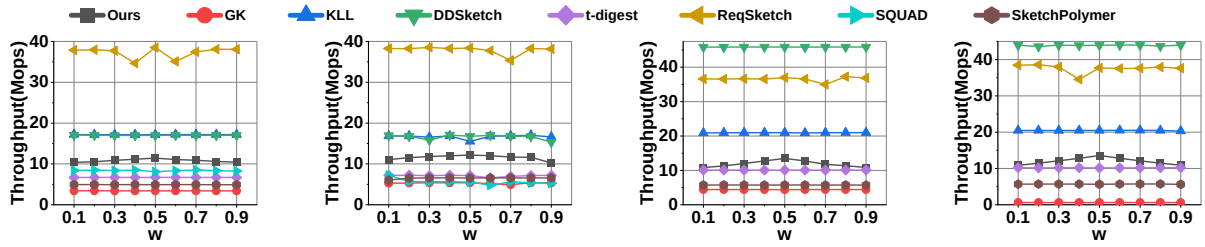
(a) 50KB (b) 60KB (c) 70KB (d) 80KB

Fig. 14: AE on Seattle Dataset in Per-key Situation



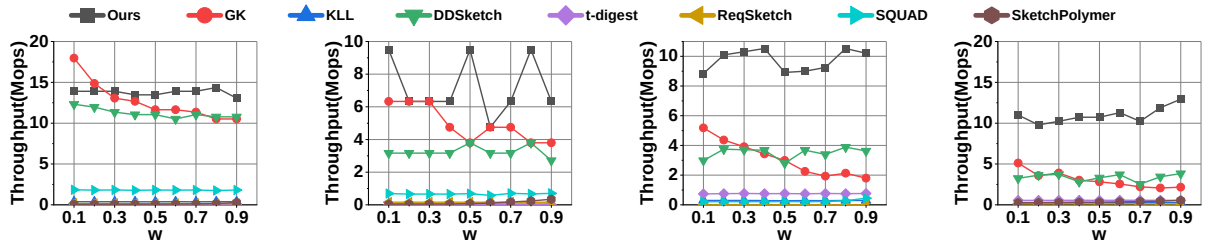
(a) 50KB (b) 60KB (c) 70KB (d) 80KB

Fig. 15: AE on Synthetic Dataset in Per-key Situation



(a) CAIDA (1500KB) (b) Campus (1500KB) (c) Seattle (80KB) (d) Synthetic (80KB)

Fig. 16: Insertion Throughput on Per-key Situation



(a) CAIDA (1500KB) (b) Campus (1500KB) (c) Seattle (80KB) (d) Synthetic (80KB)

Fig. 17: Query Throughput on Per-key Situation

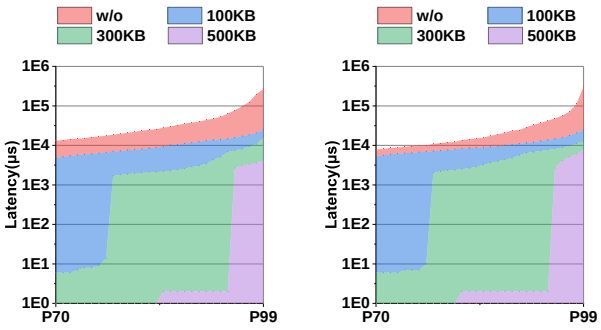
$|w - 0.5|$ is, the more infinities MagnifierSketch has to insert, which results in the lower throughput.

D. Experiments in Per-key Situation

In this section, we compare the performance of MagnifierSketch with baseline solution (GK, KLL, DDSketch, t-digest, ReqSketch), SQUAD[2] and SketchPolymer in per-key situation[3]. *Experimental results* (Figure 12-17) *show that MagnifierSketch performs much better than state-of-the-art in per-key situation.* The AE of MagnifierSketch is all lower than 0.06 on CAIDA dataset, and lower than 0.03 on Synthetic dataset, which is approximately half of other algorithms. Experimental results also show that MagnifierSketch works especially well in small memory cases, which SQUAD and SketchPolymer fails to achieve. As to throughput, MagnifierSketch achieves balance betweeen insertion throughput and query throughput. Both insertion and query throughput of MagnifierSketch are close to 10 Mops on these datasets. On CAIDA dataset, the insertion throughput of GK and SketchPolymer is smaller than 5 Mops, so they cannot catch up with high-speed items in data streams; the query throughput of KLL, t-digest, ReqSketch and SQUAD is smaller than 2 Mops, so they have to spend a lot of time to answer quantile query.

[2]Experimental results show that SQUAD has to set $\varepsilon = 1$ within 900KB on CAIDA dataset, and it cannot work within such small memory. As a result, we only provide the results of SQUAD with 1100, 1300 and 1500KB memory.

[3]Experimental results show that M4 cannot run within tight memory. With similar problem settings, M4 uses at least 6MB to run for CAIDA dataset in [29], so we do not list the results of M4 here.



(a) $w = 0.5$ (b) $w = 0.9$

Fig. 18: Experiments on RocksDB database

E. Experiments on RocksDB Database

In this section, we implement MagnifierSketch on RocksDB database [49] to accelerate quantile estimation in database scenarios. We create a synthetic RocksDB benchmark to measure the running time of quantile estimation with and without MagnifierSketch. We generate 1M items with 64-bit key and 64-bit value following Zipfian distribution with $\alpha = 1$. For items with the same key, we add distinct 16-bit suffix to the key to distinguish them, so RocksDB database will not report two records with the same key. For simplicity, we set $w = 0.5$ and $w = 0.9$ and compare the following two schemes on the running time of quantile estimation:

1) MagnifierSketch: For every item to be inserted into RocksDB database, we insert it into MagnifierSketch as well. When query the w-quantile of values with key k, we first use k to query MagnifierSketch. If the information of key k is stored in MagnifierSketch, then we directly return the query result of MagnifierSketch as its query result; otherwise we select all values with key k from the database and calculate their w-quantile.

2) Baseline Solution: No data structure in RocksDB database is specifically designed for quantile estimation. Consequently, to answer quantile query for key k, the baseline solution has to select all items in the database with key k.

We measure the tail latency of quantile query for baseline solution and different size of MagnifierSketch in Figure 18. *Our experimental results show that allocating small memory for MagnifierSketch can significantly accelerate quantile query speed in RocksDB database.* Within 100KB memory, MagnifierSketch can reduce the P90 latency by $3.23/2.21\times$ compared to baseline solution. As the allocated memory increases, MagnifierSketch can store more frequent items, and the acceleration effect is more pronounced. Within 500KB memory, the P99 latency is reduced by $67.06/35.51\times$ and is smaller than 8ms, so MagnifierSketch supports fast quantile query request in real database scenarios.

VI. Discussion

Question: To apply Distribution Calibration, MagnifierSketch assumes that all values in the data stream are i.i.d. Is this acceptable in practice?

Answer: In reality, values in datasets do not strictly follow i.i.d. distribution; this assumption serves more as an approximation. Yet, this assumption is common and reasonable in both scientific research and real datasets, with which we can give simpler and more elegant results.

In theoretical computer science, most prior work on quantile estimation applies the classical comparison-based model, where the algorithm is only allowed to compare items via the total ordering on the universe [4], [5], [24], [50]. These algorithms do not rely on prior knowledge of data distribution, and they provide optimal space complexity for this problem. However, we would like to point out that, in real database experiments, programmers shall consider more than theory. A simpler and more straightforward algorithm is preferred in implementation, maintenance, reuse, and scalability.

Our example is two popular quantile estimation algorithms, DDSketch [23] and t-digest [26]. The theory of DDSketch not only assumes that values of all items are i.i.d., but also supposes values follow the Pareto distribution. t-digest also assumes that values follow i.i.d. distribution, and t-digest does not explicitly provide its space complexity. Both DDSketch and t-digest can be problematic when the distribution is not i.i.d.: under certain input, these algorithms fail to achieve accurate quantile estimation [25]. Nonetheless, DDSketch and t-digest are still widely used in databases and actually works well [51], [52], and this is why we collect data to do empirically experiments in addition to giving theoretical guarantees.

In this paper, we run MagnifierSketch on four datasets, three of which are real-world datasets, and experimental results show that values in these datasets can be approximated as i.i.d., so MagnifierSketch still performs well in practice. Finally, to tackle non-i.i.d. data stream, we can cut the data stream into several intervals, and values in each interval can be approximated as i.i.d. The results from different intervals can be aggregated to achieve approximate quantile estimation.

VII. Conclusion

In this paper, we propose MagnifierSketch for point-quantile estimation. In single-key situation, MagnifierSketch applies **Value Focus** to keep values close to median when $w = 0.5$, and uses **Distribution Calibration** to transfer arbitrary quantile estimation problem into median estimation problem when $w \neq 0.5$. Moreover, MagnifierSketch applies **Double Filtration** to filter infrequent items in both stages to cater for per-key quantile estimation. We present rigorous mathematical analysis for MagnifierSketch, and experimental results show that MagnifierSketch outperforms existing algorithms in different scenarios. We have released the source codes of MagnifierSketch on GitHub [44].

REFERENCES

[1] R. S. Parrish, "Comparison of quantile estimators in normal sampling," *Biometrics*, pp. 247–257, 1990.

[2] J. I. Munro and M. S. Paterson, "Selection and sorting with limited storage," *Theoretical computer science*, vol. 12, no. 3, pp. 315–323, 1980.

[3] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Approximate medians and other quantiles in one pass and with limited memory," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 426–435, 1998.

[4] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 58–66, 2001.

[5] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," in *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, 2016, pp. 71–78.

[6] F. Chen, D. Lambert, and J. C. Pinheiro, "Incremental quantile estimation for massive tracking," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000, pp. 516–522.

[7] A. Timofeev and A. Sterin, "Using the quantile regression method to analyze changes in climate characteristics," *Russian Meteorology and Hydrology*, vol. 35, no. 5, pp. 310–319, 2010.

[8] D. Ferrari and S. Paterlini, "The maximum lq-likelihood method: An application to extreme quantile estimation in finance," *Methodology and Computing in Applied Probability*, vol. 11, no. 1, pp. 3–19, 2009.

[9] V. N. Padmanabhan and J. C. Mogul, "Improving http latency," *Computer Networks and ISDN Systems*, vol. 28, no. 1-2, pp. 25–35, 1995.

[10] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency." in *HotOS*, vol. 13, 2011, pp. 11–11.

[11] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, 2013, pp. 159–170.

[12] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, 2014, pp. 103–112.

[13] C. Chen, Y. Chen, K. Zhang, M. Ni, S. Wang, and R. Liang, "System redundancy enhancement of secondary frequency control under latency attacks," *IEEE Transactions on Smart Grid*, vol. 12, no. 1, pp. 647–658, 2020.

[14] M. Shahzad and A. X. Liu, "Accurate and efficient per-flow latency measurement without probing and time stamping," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3477–3492, 2016.

[15] J. Brownlee, *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python*. Machine Learning Mastery, 2020.

[16] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang, "Data cleaning: Overview and emerging challenges," in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 2201–2206.

[17] J. M. Hellerstein, "Quantitative data cleaning for large databases," *United Nations Economic Commission for Europe (UNECE)*, vol. 25, pp. 1–42, 2008.

[18] T. Dasu and T. Johnson, *Exploratory data mining and data cleaning*. John Wiley & Sons, 2003.

[19] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 325–341.

[20] F. Zhao, P. I. Khan, D. Agrawal, A. E. Abbadi, A. Gupta, and Z. Liu, "Panakos: Chasing the tails for multidimensional data streams," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1291–1304, 2023.

[21] P. Harrison, *Median wages and productivity growth in Canada and the United States*. Centre for the Study of Living Standards, 2009.

[22] L. Harris, "Transaction data tests of the mixture of distributions hypothesis," *Journal of Financial and Quantitative Analysis*, vol. 22, no. 2, p. 127–141, 1987.

[23] C. Masson, J. E. Rim, and H. K. Lee, "Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *arXiv preprint arXiv:1908.10693*, 2019.

[24] G. Cormode, Z. Karnin, E. Liberty, J. Thaler, and P. Veselỳ, "Relative error streaming quantiles," in *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2021, pp. 96–108.

[25] G. Cormode, A. Mishra, J. Ross, and P. Veselỳ, "Theory meets practice at the median: A worst case comparison of relative error quantile algorithms," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 2722–2731.

[26] T. Dunning and O. Ertl, "Computing extremely accurate quantiles using t-digests," *arXiv preprint arXiv:1902.04023*, 2019.

[27] R. Shahout, R. Friedman, and R. B. Basat, "Squad: Combining sketching and sampling is better than either for per-item quantile estimation," *arXiv preprint arXiv:2201.01958*, 2022.

[28] J. Guo, Y. Hong, Y. Wu, Y. Liu, T. Yang, and B. Cui, "Sketchpolymer: Estimate per-item tail quantile using one sketch," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 590–601.

[29] S. Dong, Z. Fan, T. Bai, T. Yang, H. Xue, P. Chen, and Y. Wu, "M4: A framework for per-flow quantile estimation."

[30] K. Yang, Y. Li, Z. Liu, T. Yang, Y. Zhou, J. He, T. Zhao, Z. Jia, Y. Yang *et al.*, "Sketchint: Empowering int with towersketch for per-flow per-switch measurement," in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 2021, pp. 1–12.

[31] A. Zhou, F. Cao, Y. Yan, C. Sha, and X. He, "Distributed data stream clustering: A fast em-based approach," in *2007 IEEE 23rd international conference on data engineering*. IEEE, 2007, pp. 736–745.

[32] H.-L. Nguyen, Y.-K. Woon, and W.-K. Ng, "A survey on data stream clustering and classification," *Knowledge and information systems*, vol. 45, pp. 535–569, 2015.

[33] P. B. Gibbons, Y. Matias, and V. Poosala, "Fast incremental maintenance of approximate histograms," in *VLDB*, vol. 97, 1997, pp. 466–475.

[34] V. Paxson, "End-to-end internet packet dynamics," in *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, 1997, pp. 139–152.

[35] J. Correa, P. Foncea, R. Hoeksma, T. Oosterwijk, and T. Vredeveld, "Posted price mechanisms for a random stream of customers," in *Proceedings of the 2017 ACM Conference on Economics and Computation*, 2017, pp. 169–186.

[36] C. Faloutsos, Y. Matias, and A. Silberschatz, "Modeling skewed distribution using multifractals and the80-20'law," in *Proceedings of the International Conference on Very Large Data Bases*. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1996, pp. 307–317.

[37] J. Guo, B. Chen, K. Yang, T. Yang, Z. Liu, Q. Yin, S. Wang, Y. Wu, X. Wang, B. Cui, T. Li, X. Peng, R. Chen, and G. Zhang, "Hourglasssketch: An efficient and scalable framework for graph stream summarization," in *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 2025.

[38] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 741–756.

[39] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[40] S. Guha and A. McGregor, "Stream order and order statistics: Quantile estimation in random-order streams," *SIAM Journal on Computing*, vol. 38, no. 5, pp. 2044–2059, 2009.

[41] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[42] Z. Fan, J. Guo, X. Li, T. Yang, Y. Zhao, Y. Wu, B. Cui, Y. Xu, S. Uhlig, and G. Zhang, "Finding simplex items in data streams," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 1953–1966.

[43] R. L. Dobrushin, "Central limit theorem for nonstationary markov chains. i," *Theory of Probability & Its Applications*, vol. 1, no. 1, pp. 65–80, 1956.

[44] "The source codes related to MagnifierSketch." https://github.com/MagnifierSketch/MagnifierSketch-code.

[45] "The source code of Bob Hash." http://burtleburtle.net/bob/hash/evahash.html.

[46] "The CAIDA Anonymized Internet Traces." http://www.caida.org/data/overview/.

[47] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Seattle: a platform for educational cloud computing," in *Proceedings of the 40th ACM technical symposium on Computer science education*, 2009, pp. 111–115.
[48] R. Zhu, B. Liu, D. Niu, Z. Li, and H. V. Zhao, "Network latency estimation for personal devices: A matrix completion approach," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 724–737, 2016.
[49] "Facebook RocksDB." http://rocksdb.org/.
[50] E. Gribelyuk, P. Sawettamalya, H. Wu, and H. Yu, "Simple & optimal quantile sketch: Combining greenwald-khanna with khanna-greenwald," *Proceedings of the ACM on Management of Data*, vol. 2, no. 2, pp. 1–25, 2024.
[51] Y. Li, G. Yu, P. Chen, C. Zhang, and Z. Zheng, "Microsketch: Lightweight and adaptive sketch based performance issue detection and localization in microservice systems," in *International Conference on Service-Oriented Computing*. Springer, 2022, pp. 219–236.
[52] T. Dunning, "The t-digest: Efficient estimates of distributions," *Software Impacts*, vol. 7, p. 100049, 2021.

APPENDIX A
PSEUDO-CODE OF MAGNIFIERSKETCH

A. Single-key Situation when $w = 0.5$

The pseudo-code of insertion and query operation in single-key situation when $w = 0.5$ is shown in Algorithm 1.

Algorithm 1: Single-key Situation when $w = 0.5$

// initialize the Candidate and Representative as empty
$C, R \leftarrow \varnothing$;
// Insertion Procedure
for *every item* $e = (k, v)$ *in the data stream* **do**
  $C \leftarrow C \cup \{v\}$;
  if $|C| = r$ **then**
    // the Candidate is full
    $v', v'' \leftarrow$ median of C;
    $C \leftarrow \varnothing$; $R \leftarrow R \cup \{v', v''\}$;
    if $|R| > s$ **then**
      // the Representative is full
      $v_M \leftarrow \max_{v \in R} v$; $v_m \leftarrow \min_{v \in R} v$;
      // keep the size of R unchanged
      $R \leftarrow R \backslash \{v_M, v_m\}$;
// Query Procedure
return *0.5-quantile of* R

B. Value Sketch Insertion Procedure

The pseudo-code of insertion operation of Value Sketch is shown in Algorithm 2.

Algorithm 2: Value Sketch Insertion Procedure

Input: an item $e = (k, v)$
if $\exists 1 \le j \le d$, *s.t.* $B[h(k)][j].key = k$ **then**
  $B[h(k)][j].vote^+ \leftarrow B[h(k)][j].vote^+ + 1$;
  $B[h(k)][j].value$.`insert(`v`)`;
else if $\exists 1 \le j \le d$, *s.t.* $B[h(k)][j]$ *is empty* **then**
  $B[h(k)][j].key \leftarrow k$;
  $B[h(k)][j].vote^+ \leftarrow 1$;
  $B[h(k)][j].value$.`insert(`v`)`;
else
  $B[h(k)].vote^- \leftarrow B[h(k)].vote^- + 1$;
  // find the cell with minimum $vote^+$ in $B[h(k)]$
  $j \leftarrow \arg\min_{1 \le i \le d} B[h(k)][i].vote^+$;
  if $B[h(k)].vote^- \ge \lambda \times B[h(k)][j].vote^+$ **then**
    // clear the cell and insert e
    $B[h(k)][j]$.`clear()`;
    $B[h(k)][j].key \leftarrow k$;
    $B[h(k)][j].vote^+ \leftarrow 1$;
    $B[h(k)][j].value$.`insert(`v`)`;
    $B[h(k)].vote^- \leftarrow 0$;

C. MagnifierSketch Insertion Procedure

The pseudo-code of insertion operation of MagnifierSketch is shown in Algorithm 3.

Algorithm 3: MagnifierSketch Insertion Procedure

Input: an item $e = (k, v)$
if `TowerSketch.query(`k`)` $< T$ **then**
  `TowerSketch.insert(`k`)`;
  return;
else
  `ValueSketch.insert(`k, v`)`;