Approximation-First Timeseries Query At Scale

Zeying Zhu
University of Maryland
zeyingz@umd.edu

Jonathan Chamberlain
Boston University
jdchambo@bu.edu

Kenny Wu
University of Maryland
kwu588@terpmail.umd.edu

David Starobinski
Boston University
staro@bu.edu

Zaoxing Liu
University of Maryland
zaoxing@umd.edu

ABSTRACT

Timeseries monitoring systems such as Prometheus play a crucial role in gaining observability of the underlying system infrastructure. These systems collect timeseries metrics from various system components and perform monitoring queries over periodic window-based aggregations (i.e., rule queries). However, despite wide adoption, the operational costs and query latency of rule queries remain high. In this paper, we identify major bottlenecks associated with repeated data scans and query computations concerning window overlaps in rule queries, and present PromSketch, an approximation-first query framework as intermediate caches for monitoring systems. It enables low operational costs and query latency, by combining approximate window-based query frameworks and sketch-based precomputation. PromSketch is implemented as a standalone module that can be integrated into Prometheus and VictoriaMetrics, covering 70% of Prometheus' aggregation over time queries. Our evaluation shows that PromSketch achieves up to a two-order-of-magnitude reduction in query latency over Prometheus and VictoriaMetrics, while lowering operational dollar costs of query processing by three orders of magnitude compared to Prometheus and by at least 4× compared to VictoriaMetrics with at most 5% average errors across statistics.

PVLDB Reference Format:
Zeying Zhu, Jonathan Chamberlain, Kenny Wu, David Starobinski, and Zaoxing Liu. Approximation-First Timeseries Query At Scale. PVLDB, 18(8): 2348-2361, 2025.
doi:10.14778/3742728.3742732
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Froot-NetSys/promsketch.

1 INTRODUCTION

Cloud-native timeseries monitoring systems such as Prometheus [1], VictoriaMetrics [5], and Grafana Mimir [13] are widely used as the cloud telemetry platform, where various metrics such as sensor readings [74], IP network traffic information [18, 19, 23, 49, 51], and cluster CPU and memory utilization [16, 31] are stored and monitored. Under the hood, such a monitoring system often consists of a timeseries database as the back-end and a dynamic query engine as the front-end, allowing users to perform various statistical queries over different time ranges to support downstream applications such as anomaly detection [8, 25], attack detection [81, 89, 90], and data visualization [12]. Among all queries supported for these applications, *rule* queries [21, 30, 37] are often set up to periodically compute aggregated statistics over time ranges (i.e., repeated *time range queries*) and alert users if abnormal conditions are met (e.g., quantiles, top-K, cardinality). For instance, timeseries network flow data (e.g., source/destination IPs, ports, and protocols) can be monitored in the range of seconds to aggregate distinct source IPs targeting a specific host over a recent time window, indicating a potential Distributed Denial of Service (DDoS) attack [35, 72].

While Prometheus and its variants have been a de facto standard open-source tool to handle rule queries, they struggle with non-trivial operational costs and high query latency in practice. In our evaluation, an AWS Prometheus service running 10 rule queries every minute and monitoring just on *a single rack* would approximately take $357,120 for query processing and $47,746 for data ingestion per month (§2.3). Performing a quantile query over 100K-sample windows and 10K timeseries takes 15 min on a commodity server in our testbed. Our profiling reveals two major bottlenecks in rule queries that lead to high monitoring cost and query latency: 1) *repeated data scans* from storage and 2) *repeated query computations*, based on the observation that a single rule can perform time range queries over consecutive overlapping windows and different rules may also query the same overlapping windows. For example, a rule with 10-minute windows and 1-min evaluation intervals, or queries over different time windows (e.g., 2-, 5- and 10-minute), repeatedly access the overlapping portions of data among windows but Prometheus computes them separately.

While several existing efforts aim to address the bottlenecks of Prometheus, they fall short in one or more of the dimensions in operational cost, query latency, and query accuracy. Exact monitoring systems that optimize Prometheus (e.g., VictoriaMetrics [5]) can reduce query latency through better storage engine designs and data caching for lower data retrieval time, and applying parallel query computation for lower query evaluation time. However, they do not reduce operational costs as they do not address the repeated data scanning and computational bottlenecks. While one can consider applying pre-computation approaches similarly to those optimizing SQL queries [60, 93], they tend to support a fixed time window and limited statistics such as sum and max, with small cost and performance improvements.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742732

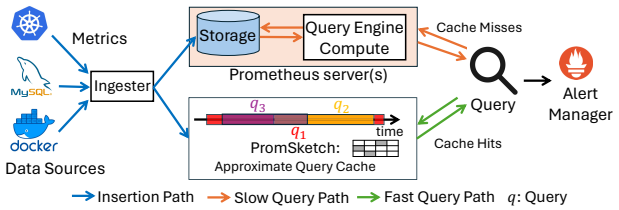


Figure 1: A Prometheus monitoring ecosystem with PromSketch.

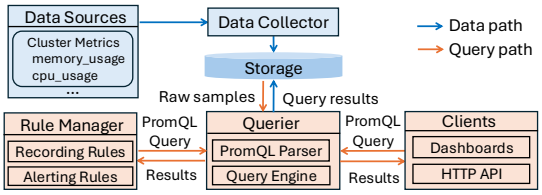


Figure 2: The architecture of a timeseries monitoring system.

Alternatively, approximate analytics (e.g., sampling- and sketch-based) offer a promising approach to trade off estimation accuracy for further lower operational costs and query latency [75] of complex queries. Applications often require near real-time analytics and tolerate approximate but highly accurate results, such as datacenter alerts [34, 70, 99], network measurements [54, 100], and more [47, 66, 101]. However, practical issues of low accuracy and low query generality remain. Sampling-based approaches (e.g., [73, 95]) can provide an estimation for any queries but suffer from worse and unpredictable accuracy for complex statistics such as quantiles and entropy. Sketch-based analytics (e.g., [32, 48, 65, 75, 79]) and sliding window sketches (e.g.,[36, 39, 40, 46]) can provide strong accuracy guarantees over querying statistics of a fixed window but are limited to answering certain queries.

In this paper, we revisit the promises of approximate analytics to improve operational efficiency and performance in timeseries monitoring systems. We present **PromSketch**, an approximate query cache that improves operational cost and query latency by up to two orders of magnitude while preserving high accuracies (e.g., >95%). In contrast to Prometheus' independent query handling, PromSketch is a framework that is able to "sketch and cache" a wide range of recent windows and statistics in the fast storage (e.g., main memory) to mitigate the bottlenecks of repeated data scans and query computations from overlapping windows (as in Fig. 1).

PromSketch is built on the combination of two key ideas. First, PromSketch caches a range of *intermediate results* instead of caching raw data or final query results integrated in today's timeseries monitoring systems [5]. This is a practical choice because (1) a raw data cache does not help reduce repeated query computations, and the memory usage can be prohibitively large; and (2) a query result cache misses the opportunities to optimize drill-down queries that are not predefined. Thus, we adopt an extended sliding window model based on the Exponential Histogram [52, 62] to maintain a list of intermediate results (called buckets) covering consecutive intervals of sizes varying exponentially in a timeseries. At query time, we linearly merge these buckets to obtain the final results on any *sub-window* of the large window. We consider this intermediate result cache as a balance between caching raw data and final results.

Second, we provably combine the extended sliding window model with popular linear sketches to support various query functions. There are a potentially large number of concurrent timeseries and query functions that need to be monitored (e.g., quantiles, entropy, and L_2 norm of CPU and memory usages from various Kubernates [3] nodes). We want to cache as many timeseries as possible given a memory budget. While exact data structures can be used to store intermediate results, they cannot scale to a large number of timeseries. To optimize memory usage, we extend the Exponential Histogram model with KLL sketch [65] and universal sketching [45, 71], and prove their memory-accuracy efficiency both theoretically and empirically, with system optimizations to reduce system runtime and operational costs.

We implement PromSketch as a Go package in 5K lines of code that is compatible with Prometheus and VictoriaMetrics, two popular open-source timeseries monitoring systems, extending PromQL and covering 70% of Prometheus' aggregation over time functions. PromSketch is also portable to other Prometheus-like systems such as [13, 24]. Our extensive experiments show that: (1) PromSketch offers robust accuracy (mean error ≤5%) while reducing operational costs of query processing by 1332× compared to Prometheus and at least 4× compared to VictoriaMetrics and (2) it reduces end-to-end query latency by up to two orders of magnitude over Prometheus and VictoriaMetrics. PromSketch's precomputation overhead is moderate as 1.3× to 3× of non-precomputed/cached Prometheus. In summary, we make the following contributions.

- We systematically analyze the rule queries in the popular timeseries monitoring systems and identify the bottlenecks and cost consequences of repeated data scans and overlapped query computations on the cloud. (§2)
- To mitigate the bottlenecks, to the best of our knowledge, PromSketch is the first work to (1) introduce an end-to-end approximate intermediate caching design for various time ranges and statistics in timeseries monitoring, (2) propose combinations of Exponential Histogram and different types of sketches (e.g., KLL, Universal Sketching) to support various windows and query statistics, and (3) analytically prove the guarantees. (§4)
- We provide ready-to-plugin PromSketch to both single-machine and distributed systems with cloud-native architecture (§5), and show its benefits on real-world and synthetic datasets over baseline systems (§6).

2 BACKGROUND AND MOTIVATION

We introduce background of timeseries monitoring systems and motivating scenarios, and discuss the limitations of existing systems and new design opportunities.

2.1 Timeseries Monitoring Systems

A monitoring system usually collects, stores, and queries timeseries data from various sources. Fig. 2 shows a typical timeseries monitor architecture. *Data collectors* scrape metrics and send them to storage. The *storage engine* appends new data to the timeseries without modifying previous data. Users can issue queries in PromQL [22] via various clients, including rule queries for periodic monitoring and alerting. The *query engine* retrieves data samples from storage and computes results based on query expressions. The rule query results can be stored for reuse.

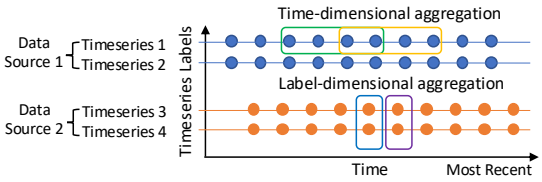


Figure 3: Timeseries data and different types of queries in Prometheus. ***Each row is a timeseries identified by its labels.***

Data Model. Timeseries data are streams of timestamped values belonging to the same data source and the same set of labeled tags. They span over two dimensions: (1) the time dimension, which consists of data samples each associated with a timestamp belonging to one timeseries; and (2) the label dimension, which consists of samples from many different data sources and label tags at a given timestamp. A data sample can be represented by $\rho = (l, t, v)$, where $l = (d_1, d_2, ..., d_m)$ contains m label dimensions, t is the timestamp, and v is the data value, either a 64-bit floating point (e.g., CPU usage) or string (e.g., IP address). An example timeseries of `cpu_usage` metric recording the CPU usage of each node and each core can be represented as `cpu_usage{node_id="node0",cpu_id="0"}`, specified with labels of node ID and CPU core ID.

Queries. Monitoring systems like Prometheus offer various queries for downstream applications, especially alerting and recording rules [20]. Users can define *rules* to automatically execute periodic monitoring queries and track alerts [91]. A rule query mainly consists of three parts as below: 1) a rule type, either recording rule (which stores results for future use), or alerting rule (which sends alerts based on conditions); 2) a rule evaluation interval T_{eval}, defining the evaluation frequency; and 3) a query expression, with an optional alert condition for triggering alerts in alerting rules.

```
rule:
- type: record|alert
- evaluation_interval: T_eval
- expr: PromQL expression, [alert condition]
```

Formally, a query expression can be defined as $Q_R = \{q(\rho), \rho : t_{cur} - T_q \leq t \leq t_{cur} \bigwedge d_{i_1} = x_{i_1} \bigwedge \cdots \bigwedge d_{i_m} = x_{i_m}\}$, with query function q on a set of timeseries data samples for T_q query window looking back from current time t_{cur}, and $d_{i_1}, \ldots, d_{i_m}$ a subset of the label dimensions to condition on. The query function q can be an aggregation over the time dimension or label dimensions at a given timestamp. Examples of timeseries data samples and rule queries are shown in Fig. 3. Aggregation queries, such as quantiles, Top-K, entropy, and cardinality, are important for understanding statistics without focusing on single values in timeseries and are more efficient than querying raw data samples. Summarizing the data over time with aggregation query functions and periodic rule queries, essentially, it forms a sliding window model. In this paper, we aim to optimize the time-dimensional aggregation queries.

2.2 Motivating Scenarios

Network Flow Monitoring. DDoS attacks can occur when attackers use TCP SYN floods to exhaust bandwidth or server resources via a botnet [35, 78]. Victims can detect ongoing attacks by monitoring the volume and entropy of SYN packets from multiple source IPs targeting a single destination [82]. For example, operators can track DDoS indicators for a virtual machine using alerting rules with a 5-second evaluation interval as follows [72]. Detection queries can identify the start and the end of a DDoS attack by monitoring flow changes and comparing metrics against alert thresholds, requiring frequent queries across various time windows (e.g., 10s, 5s) due to the uncertainty of optimal window sizes. In this case, each target server may receive millions of packets per second [83], requiring the time windows of being over 100K to 1 million data samples.

```
rules:
- evaluation_interval: 5s
- type: alert
- expr: entropy_over_time(src_ip{vm="instance1"}[10s])
        > entropy_threshold
- expr: distinct_over_time(src_ip{vm="instance1"}[10s])
        > volume_threshold
- expr: distinct_over_time(src_ip{vm="instance1"}[5s])
        > volume_threshold
```

Table 1: Comparison of operational cost between several systems. ***"PS-PM" and "PS-VM" refer to Prometheus- and VictoriaMetrics-based integrations of PromSketch.***

Costs	AWS Prometheus [6]	PS-PM	VictoriaMetrics [29]	PS-VM
Storage	$214, $0.03/GB-Month	$214	$1,065	$1,065
Data Ingestion	$47,746, ~$0.35/10M samples	$47,746	incl.	incl.
Query Processing	$357,120, $0.1/B samples	$267.8	≥ $7,443	≤ $1833
Total Costs	$405,080	$48,227.8	≥ $8,508	≤ $2,898

Cloud Resource Scaling. Cloud-native platforms autoscale resources, such as pods, to reduce costs [10] based on aggregated statistical queries (e.g., averages, quantiles) over time windows for metrics like CPU, memory, and pod counts from monitoring tools [92]. For instance, recording rules can query each container's 0.95-quantiles for memory and CPU usage, and average pod counts over the past 5 minutes, storing the results for quick retrieval by the cloud resource scheduler and downstream applications as below. Standard Google Cloud clusters can have up to 256 pods per node and up to 100 nodes per cluster [11]. Thus, cluster-level monitoring can easily result in 100K or more timeseries to query on.

```
rules:
- evaluation_interval: 1m
- type: record
- expr: quantile_over_time(0.95,
         container_memory{dimension="used"}[5m])
- expr: quantile_over_time(0.95,
         container_cpu{dimension="used"}[5m])
- expr: avg_over_time(pod_number[5m])
```

In summary, rule query use cases involve monitoring queries that repeatedly query the same metrics over time with varying window sizes, various statistical functions, and the ability to handle large data volumes, while being sensitive to query latency, providing critical observability to anomaly detection [61, 77, 90, 97], security checking [35, 94], and cloud performance monitoring [53, 97].

2.3 Operational Cost and Bottleneck Analysis

We start by comparing the operational costs of two representative systems in Table 1, monitoring a 1000-node Kubernetes cluster with each node having 1000 metrics, storing 2678 billion data samples per month. 10 concurrent rule queries run every minute and each query processes 8 billion samples. Cost estimates follow AWS Prometheus Pricing [6], which charges by storage and samples processed, and a typical cloud billing model used by VictoriaMetrics [2, 29], which charges based on resource usage such as memory, vCPUs. We defer

Table 2: CPU hotspots of evaluating a quantile rule query in Prometheus and VictoriaMetrics.

Func/Call Stack	CPU Time		Description
	Prometheus	VictoriaMetrics	
Data Scanning	**41%**	**80.2%**	Fetch data from storage
Query computation	**27.6%**	**11.7%**	Aggregation queries in rule
Go Garbage Collector	24.7%	4.3%	Golang garbage collector
mcall	4.5%	0.8%	Golang runtime scheduling

detailed analysis in §6. Query processing comprises over 85% of the total costs in Prometheus and in VictoriaMetrics.

We analyze bottlenecks in Prometheus and VictoriaMetrics to identify sources of high query costs. Using Golang pprof and the testbed in § 6, we profile recording rule queries with extended time windows. For example, we test with a 10,000-second query window, 100ms sample interval, and 1s evaluation interval, benchmarking the `quantile_over_time(0.99,metric[10000s])` query. Table 2 shows the CPU profiling results and the top two bottlenecks.

Bottleneck 1: Repeated data scans from storage. The primary bottleneck is data scanning from storage, due to repeated scans of data when query windows overlap in rule queries or concurrent queries from multiple users.

Bottleneck 2: Repeated query computations. The second major bottleneck is query computation. In both VictoriaMetrics and Prometheus, periodic rule queries are computed independently rather than as sliding window queries. They re-execute the entire query computation for overlapping portions, without leveraging intermediate results from previous overlapping windows.

2.4 Prior Work and Limitations

Exact monitoring systems. Prior work reduces timeseries query latency and costs via three categories. The first enhances storage engines with better indexing (e.g., InfluxDB [15], VictoriaMetrics [5]), optimized storage schemas (e.g., [98]), and improved compression techniques (e.g., [87]). These methods reduce storage costs and retrieval latency but don't address computational bottlenecks from repeated data scans and overlapping windows.

The second improves query performance via parallel query processing, query sharding, and precomputation. Parallel processing (e.g., VictoriaMetrics [28]) distributes query tasks by timeseries across CPU cores. Query sharding (e.g., Mimir [13], Thanos [24]) partitions queries by time or series and processes each partition sequentially to reduce memory usage and the impact of Go garbage collection. While both lower query latency, parallel processing does not lower overall query costs, and query sharding, while reducing memory costs, still maintains redundant computational overheads across queries. While precomputation (e.g., [93]) reduces costs and query latency by precomputing statistics during ingestion and thus reducing query redundancy from window overlapping, it supports only basic statistics (e.g., sum, max) and fixed intervals.

The third uses key-value caches (e.g., VictoriaMetrics [9], Mimir [14, 17]) to accelerate queries, as metadata-cache, index-cache, chunk-cache, and result-cache [14]. Metadata and index caches accelerate timeseries searches by mapping metrics to database indexes but don't remove repeated data retrieval or computational overhead. Chunk-cache stores data in memory, reducing disk retrieval time but is limited by memory capacity and doesn't address repeated computations. Result caches store query results, but frequent changes in query statistics and time ranges limit cache reuse.

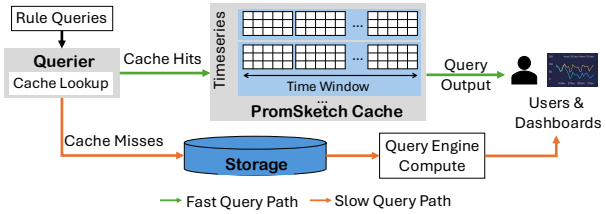


Figure 4: PromSketch Architecture.

Approximate Query Processing (AQP). In monitoring systems, approximate results are often sufficient for downstream applications [62, 71, 72, 84], offering the chance to trade off minor accuracy for lower query latency and operational costs [75], using sampling or data summarization for time-window queries.

Sampling has been widely explored in Approximate Query Processing (AQP) by pre-processing data samples for query-time use [33, 73, 86, 88]. Monitoring systems like Thanos [95] apply downsampling to reduce data retrieval and computation costs. While sampling-based frameworks offer broad applicability across various statistics and support the sliding window model [73], their accuracy guarantees weaken for complex statistics (e.g., quantiles [65]) and suffer from larger errors when zooming into small sub-windows with a low fixed sampling rate, due to limited sample availability.

Sketch-based analytics offer bounded accuracy-memory tradeoffs in sub-linear space [58, 65, 71, 79], creating compact summaries during ingestion and estimating statistics with provable error bounds. Sliding window sketches are often designed for specific query types, maintaining summaries for the entire window, such as sliding sum [40], 0-1 counting [52], heavy hitter detection [38, 39], distinct counting [46], and sliding quantiles [36]. While implementing each individually supports diverse queries, it introduces per-statistic effort and lacks sub-window query support within the recent window, leading to additional maintenance overhead. Recent approaches [62] extend fixed sliding window frameworks [44, 52] to support arbitrary sub-windows and accommodate various sketch types as subroutines, making them well-suited for periodic rule queries with varying window sizes and statistical requirements.

Summary and Opportunities. Existing solutions fall short in the tradeoffs among operational costs, query latency, and accuracy. Our analysis reveals a key optimization opportunity in removing query redundancy due to overlapping windows. Since periodic rule queries often share overlaps, caching appears as an effective approach to reduce redundant data scans and query computations. However, caching all raw data samples is not a scalable choice and does not reduce the computational costs from window overlaps. Moreover, caching some final results is an ad-hoc choice to only optimize a few predefined queries. Thus, caching intermediate results that are precomputed and flexible enough to query a wide range of windows becomes a well-informed choice.

3 PROMSKETCH: SYSTEM OVERVIEW

PromSketch Architecture. We illustrate the system components in Fig. 4. PromSketch maintains an in-memory approximate cache. Data samples are ingested into both backend storage and the cache by the data ingester. PromSketch precomputes intermediate results

for the most recent windows of timeseries selected by rule queries. When a rule query is issued, the querier first checks the cache for the required time range and statistics. If found, the query retrieves estimated results from PromSketch with reduced latency; otherwise, it falls back to the original TSDB query engine to scan raw data and compute exact statistics. The final query results, from PromSketch or the exact engine, are returned to users.

Challenges and Key Ideas. To realize the vision of PromSketch, we address several key design challenges:

Challenge 1: Caching many recent query windows and results. Query statistics and functions are various in real use cases. Caching all samples ensures generality but is memory-intensive, while caching only final results limits optimization for unforeseen drill-down queries. A raw data cache also fails to reduce redundancy from overlapping query windows.

Key Idea. We extend the window-based approximate query framework (e.g., Exponential Histogram [52, 62]) to be sub-window-capable as a flexible intermediate query cache along the time dimension for each timeseries. The cache stores intermediate results for many sub-windows within a large recent time window, allowing reuse for overlapping portions of query windows (e.g., one can query 5-, 10-, 15-min windows within a cached 30-min window without recalculating from scratch). It can support multiple and arbitrary sub-windows and different query functions through using different internal data structures to maintain intermediate results.

Challenge 2: Caching a large number of timeseries. Number of active timeseries that need to be monitored can be large [93], requiring caching as many timeseries as possible within a memory budget. While exact data structures in window-based frameworks to store the intermediate results offers high accuracy, it needs a large amount of memory as exact query processing and cannot scale to a large number of timeseries.

Key Idea. To reduce memory usage, we integrate approximate methods, such as sketches and sampling, to work as compact and low-latency intermediate data summarizations in the framework. For instance, we propose proper combinations of Exponential Histogram and KLL, and formally establish space-error bounds.

Challenge 3: Efficient caching of various query statistics. Users often query different statistics over the same timeseries, such as distinct counting and entropy of source IPs for DDoS attack detection. To support as many query statistics as existing exact monitoring systems, a strawman solution is to cache each statistic with a separate sketch instance for each timeseries. However, this approach introduces per-statistic efforts and large memory costs.

Key Idea. To avoid per-statistic effort, recent advances in universal sketching [45, 71] allows a single sketch instance to support multiple target query functions, such as L_0, L_1, L_2 norms and entropy, instead of requiring a separate sketch for each function. We combine universal sketching with EH to support multiple statistics simultaneously [62], and proposes a novel optimization that combines exact maps and universal sketching as EH buckets, reducing memory footprint while improving accuracy.

4 PROMSKETCH DETAILED DESIGN

We introduce window-based frameworks as PromSketch cache and its algorithmic building blocks, followed by detailed system designs.

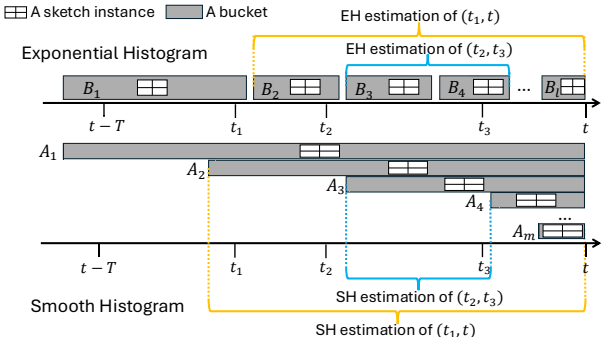


Figure 5: Exponential Histogram [52] and Smooth Histogram [44] structures and window-based queries. *(t_1, t) and (t_2, t_3) are sub-window queries within the most recent T time window and current time t.*

4.1 Window-based Frameworks as a Cache

Periodic time interval aggregation queries, such as Alerting rules and Recording rules, are essentially *sliding window* queries along the time. These queries maintain statistics of the most recent T time window $W = (t-T, t)$. Users can also query any statistics over a sub-window $(t_1, t_2) \subseteq W$ for zoom-in diagnosis of applications such as anomaly localization. To cache as many query windows as possible within limited memory budgets, approximate window-based frameworks that maintain sliding windows and sub-window structures are viable options. Currently, there are two general approximate window-based frameworks providing $o(N)$ memory with good estimations for a recent window W of N items: Exponential Histogram (EH) [52] and Smooth Histogram (SH) [44]. Intuitively, Exponential Histogram maintains non-overlapping buckets whose bucket sizes are exponentially growing when buckets are older; Smooth Histogram maintains overlapped buckets that covering time ranges with different start time points, as Fig. 5 shows.

Exponential Histograms [52] suggests to break the most recent window $W = (t-T, t)$ into a sequence of l non-overlapping intervals (buckets) $B_1, B_2, \ldots, B_l$. Window W is covered by $\bigcup_{i=1}^{l} B_i$, and contains all B_i except B_1. Then, if a target function f admits a composable sketch, maintaining such a sketch on each bucket can provide us with an estimator for f on a window $W' = \bigcup_{i=2}^{l} B_i$. $f(W)$ is sandwiched between $f(W')$ and $f(B_1 \cup W')$. Therefore, a careful choice of each bucket endpoints provides control over the difference between $f(W)$ and $f(W')$. When the window slides, new buckets are introduced, expired buckets are deleted, and buckets in between are merged. The EH approach admits non-negative, polynomially bounded functions f which in turn enable a composable sketch and are weakly additive, i.e., $\exists C_f \geq 1$, such that $\forall S_1, S_2$:

$$f(S_1) + f(S_2) \leq f(S_1 \cup S_2) \leq C_f(f(S_1) + f(S_2)). \quad (1)$$

We show the intuition of querying a sub-window by the following example: $q = (t_2, t_3)$ as depicted in Fig. 5. In the example, q is sandwiched between $B_2 \cup B_3 \cup B_4$ and B_3, where $f(\bigcup_{j=2}^{l}) = (1 \pm \varepsilon) f(t_2, t)$ and $f(\bigcup_{j=4}^{l}) = (1 \pm \varepsilon) f(t_3, t)$. Intuitively, one can expect that $f(t_2, t_3)$ can be approximated by $f(B_3 \cup B_4)$ with an additive error of $\pm \varepsilon f(t_2, t)$, related to the suffix (t_2, t).

Smooth Histograms [44] buckets $A_1, \ldots, A_m$ overlap. An example sub-window query $q = (t_2, t_3)$ is sandwiched between $f(A_2)$ and

Table 3: Example Prometheus aggregation-over-time queries supported by PromSketch.

EHKLL (Algo 1)	`quantile_over_time`	`min_over_time`	`max_over_time`
EHUniv (Algo 2)	`count_over_time` `distinct_over_time`	`entropy_over_time` `topk_over_time`	`l2_over_time`
Uniform Sampling	`count_over_time` `stddev_over_time`	`sum_over_time` `stdvar_over_time`	`avg_over_time`

Algorithm 1 EHKLL: Quantiles Based on EH

1: **Input**: EH item count error ϵ_{EH}, KLL rank error ϵ_{KLL}, confidence level δ, time window range T
2: **function** UPDATE(t, item)
3: Maintain EH with $k_{EH} = O(\frac{1}{\epsilon_{EH}})$ based on Invariant 1 and Invariant 2.
4: Each bucket B_i maintains a KLL sketch KLL_i with ϵ_{KLL}.
5: **function** QUERY(t_1, t_2, ϕ)
6: Find $B_i = (b^0, b^1)$ and $B_j = (b^2, b^3)$ s.t.: $t_1 \in B_i$ and $t_2 \in B_j$
7: Compute the merged sketch $KLL_{merge} = \bigcup_{i+1 \le r \le j} KLL_r$
8: **return** ϕ-quantile: $x_q = KLL_{merge}.query(\phi)$

$f(A_4)$ and can be approximated by $f(A_3 - A_4)$ with SH buckets, if the sketches preserve approximation upon subtraction.

We choose Exponential Histogram over Smooth Histogram in PromSketch for two main reasons. First, SH requires subtractive properties between sketches while EH requires only additive mergeability, which most sketches support, allowing us to analyze error bounds for more window/sketch combinations. Second, for potentially large-scale data ingestion, EH can offer better system performance. Specifically, when inserting an item, EH only needs inserting into the newest (and smallest) bucket, with an amortized $O(1)$ insertion cost [52], while SH inserts the item into every active bucket, resulting in an $O(\log N)$ insertion cost [44]. Additionally, EH typically has smaller buckets than those of SH because they represent non-overlapping sub-windows, and thus require smaller inner data structure allocations.

4.2 Algorithmic Building Blocks

Next, we introduce algorithmic building blocks of PromSketch, novelly combining EH window framework and configurable sketches with provable error guarantees. They cover 70% existing window aggregation queries in Prometheus and introduce capabilities for currently unsupported queries as in Table 3.

4.2.1 EH+KLL for Quantiles (EHKLL). Quantile-based rule queries such as `quantile_over_time`, `min_over_time`, `max_over_time`, consist of querying data samples over a time range and a $\varphi \in [0, 1]$, representing φ-quantile (e.g., `min` and `max` correspond to the 0-quantile and 1-quantile). We present a novel construction for arbitrary sub-window quantiles, using an EH with each bucket as a KLL sketch [65] to maintain quantiles, as in Alg. 1 (EHKLL). The user specifies the KLL rank error ϵ_{KLL}, EH error ϵ_{EH}, confidence level δ, and the most recent window size (in time range T or data count N). The new data sample is added into the latest bucket B_l's KLL sketch. If needed, buckets merge [52] to maintain EH invariants based on the number of items in each EH bucket. For a query with time range $T = [t_1, t_2]$, we identify two buckets that contain t_1 and t_2, merge the KLL sketches between the two buckets based on EH to construct a merged sketch.

Feasible Quantile Sketches with EH. We integrate KLL sketch[65] as the quantile estimator because of feasibility to aggregate the estimation errors from both EH window framework and each bucket's sketch consistently. At a high level, we can choose quantile sketches between two types – one provides rank error guarantees, and the other provides relative error guarantees. A rank error ϵ_{rank} approximate quantile sketch receives items $x_1, x_2, \ldots, x_n$, and allows one to approximate the rank of any query item up to additive error $\epsilon_{rank} n$ with probability at least $1 - \delta$. The rank of a query x is the number of items in the queried window such that $x_i \le x$. Given a φ-quantile query x_φ, a relative error ϵ_{rel} approximate quantile sketch outputs $\tilde{x}_\varphi$ such that $|\tilde{x}_\varphi - x_\varphi| \le \epsilon_{rel} x_\varphi$. Since an EH maintains buckets with rank error guarantees based on Invariant 1 and Invariant 2 as below ([52]) and KLL is a representative sketch with rank errors, we explore the novel combination of EH+KLL and analyze the aggregated rank error bounds.

Invariant 1. Define $k_{EH} = \frac{1}{\epsilon_{EH}}$ and assume $\frac{k_{EH}}{2}$ is an integer; otherwise, we replace $\frac{k_{EH}}{2}$ by $\lceil \frac{k_{EH}}{2} \rceil$. At all times, the bucket sizes $C_1, \ldots, C_l$ satisfy $\frac{C_j}{2(1+\sum_{i=j+1}^{l} C_i)} \le \frac{1}{k_{EH}}$, for all $1 \le j \le l$.

Invariant 2. At all times, the bucket sizes are nondecresing, i.e., $C_1 \le \cdots \le C_{l-1} \le C_l$. Further, the bucket sizes are constrained to the following: $\{1, 2, 4, \ldots, 2^{l'}\}$ for some $l' \le l$ and $l' \le \log \frac{2N}{k_{EH}} + 1$. For every bucket size other than the size of the last bucket, there are at most $\frac{k_{EH}}{2} + 1$ and at least $\frac{k_{EH}}{2}$ buckets of that size.

EHKLL Error Guarantee. We prove the error bound as follows. First considering queries over the entire sliding window, the oldest bucket with size C_1 that we discard for quantile estimation can contribute at most C_1 rank difference from the accurate answer. The newest bucket C_l exactly aligns with the sliding window query boundary and introduces no errors. Therefore, based on Invariant 1 [52], the rank error caused by window framework EH is at most $\frac{2}{k_{EH}}$. Assuming a KLL sketch has ϵ_{KLL} rank error of the quantile estimation after taking all N items, given its mergeability, the final estimated rank error is $\epsilon_{EHKLL} \le 2\epsilon_{EH} + \epsilon_{KLL}$. Alg. 1 can be used to query all quantiles including *min* ($\phi = 0$) and *max* ($\phi = 1$). Based on [65], a KLL sketch needs $O(\frac{1}{\epsilon_{KLL}} \log^2 \log(1/\delta\epsilon_{KLL}))$ bits of memory for estimations of all quantile. There are on the order of $O(\frac{1}{\epsilon_{EH}} \log N)$ EH buckets if we maintain EH buckets based on the number of samples inserted to each bucket. Therefore, the total memory needed by Alg. 1 is $O(\frac{1}{\epsilon_{KLL}} \log^2 \log(1/\delta\epsilon_{KLL}) \cdot \frac{1}{\epsilon_{EH}} \log N)$, which gives at most $2\epsilon_{EH} + \epsilon_{KLL}$ normalized rank error for sliding window queries with N samples in the most recent window.

Next, we extend the error guarantee for arbitrary sub-window queries with EHKLL. The sub-window query is answered by merging buckets $i + 1$ to j in Alg. 1. Similarly, the bucket B_i discarded in calculation contributes at most C_i rank difference; the bucket B_j included in the calculation contributes at most C_j rank difference. In total, the rank difference from EH sub-window query is at most $C_i - C_j$. If we provide rank error bound against (t_1, t_2), the rank error from EHKLL is $\epsilon_{EHKLL} \le \frac{C_i - C_j}{N_{t_1} - N_{t_2}} + \epsilon_{KLL} \le \frac{N_{t_1}}{N_{t_1} - N_{t_2}} \frac{C_i - C_j}{(1+\sum_{r=i+1}^{l} C_r)} + \epsilon_{KLL} \le 2\epsilon_{EH} \frac{N_{t_1}}{N_{t_1} - N_{t_2}} + \epsilon_{KLL}$, where N_{t_i} refers to the number of

Algorithm 2 EHUniv: GSum Based on EH

Input: L_2 error target ϵ, confidence level δ, time window T
function UPDATE(t, item)
    Maintain EH for L_2^2 with $k_{EH} = O(1/\epsilon^2)$ based on Invariant 3 and Invariant 4.
    On each bucket A_i maintain a universal sketch per bucket with error target ϵ.
function QUERY(t_1, t_2)
    Find $B_i = (b^0, b^1)$ and $B_j = (b^2, b^3)$ s.t.: $t_1 \in B_i$ and $t_2 \in B_j$
    Compute the merged sketch $Univ_{merge} = \bigcup_{i+1 \le r \le j} Univ_r$
    Query *gsum* from $Univ_{merge}$ based on Recursive GSum algorithm [43] (also Alg. 4 in [62])
    return *gsum*

samples between time t_i to current time t. If we provide rank error bound against (t_1, t), similarly, the rank error from EHKLL is $\epsilon_{EHKLL} \le \frac{C_i - C_j}{(1+\sum_{l=i+1}^{m} C_l)} + \epsilon_{KLL} \le 2\epsilon_{EH} + \epsilon_{KLL}$.

4.2.2 Universal sketching with EH (EHUniv). Next, we support estimations of complex statistics that are not fully supported by current systems, such as L_2 norm, distinct counting, and entropy. These are often used together and queried on the same timeseries, but naive solutions require separate caching per statistic. For example, in DDoS attack detection, one would need a distinct counting sketch for cardinality and another for entropy, doubling memory usage. Thus, we leverage a universal sketch as an EH subroutine for multiple statistics with one sketch, as Alg. 2 (EHUniv). The update process inserts samples into the newest EH bucket and maintains EH buckets according to invariants. If EH invariants are violated for a pair of buckets (B_{j-1}, B_j), they are merged into a new bucket B'_{j-1}, whose universal sketch combines those of B_{j-1} and B_j. The update process has an amortized merge time $O(1)$ and a worst-case merge time of $O(k_{EH} \log N)$, where N is the sample count in the most recent window. During queries, after finding the two buckets that contains the query window start time t_1 and end time t_2, it merges all buckets in between, including the last bucket but excluding the first one. Finally, the supported statistic can be answered by Recursive GSum [43] and the merged universal sketch.

EHUniv Benefits. The benefits of integrating universal sketch is the ability to maintain a single sketch for querying multiple statistics rather than creating separate sketches for each, and its natural mergeability. These statistical functions can be summarized as *GSum*, allowing a single *universal* sketch instance to maintain multiple statistics [43, 71]. The *GSum* function is defined as $G = \sum_{i=1}^{m} g(f_i)$, where f_i is the frequency of data sample data$_i$ and $g : \mathbf{N} \to \mathbf{N}$ is a function. The class of *GSum* functions covers many practical monitoring functions, including L_0 (distinct counting), L_1 norms (`count_over_time`), L_2 norm, entropy[1], and TopK-frequent item finding. We provide the basics of universal sketching here and refer to [43, 71] for more details. Theorem 2 in [43] states that if $g(x)$ grows slower than x^2, drops no faster than sub-polynomially, and has predictable local variability, then there is an algorithm that outputs an ϵ-approximation to G, using sub-polynomial space and only one pass over the data. For a universe with M different items in a data stream, a universal sketch maintains $\log M$ parallel copies of a "L_2-heavy hitter" (L2-HH), e.g., using a Count Sketch [48] as the L2-HH subroutine. Then, leveraging a Recursive GSum algorithm [52] (also described in Alg. 4 of [62], we omit the details here), a universal sketch estimates the statistical function g by recursively computing g on founded heavy hitters in $\log M$ layers of Count Sketches. Following [62], EH buckets can be maintained based on L_2 norm and thus can support L2-HH routines: [62] improves EH [52]'s results on L_2 that by maintaining $k_{EH} = O(\frac{1}{\epsilon^2})$ and $C_f = 2$, EH can provide ϵ-approximation for L_2 on the sliding window by maintaining Invariant 3 and 4, where $f(B_j) = L_2^2(B_j), j = 1, \ldots, l$.

***Invariant* 3.** $f(B_j) \le \frac{C_f}{k_{EH}} \sum_{i=j+1}^{l} f(B_i)$.

***Invariant* 4.** $f(B_{j-1}) + f(B_j) > \frac{1}{k_{EH}} \sum_{i=j+1}^{l} f(B_i)$.

EHUniv Error Guarantee. According to Theorem 7 in [52], the EH approach required $O(k_{EH} s(\epsilon, \delta) \log N)$ bits of memory, where $s(\epsilon, \delta)$ is the amount of memory needed for a sketch to get a $(1+\epsilon)$-approximation with a failure probability of at most δ. Theorem 3.5 in [62] shows that a Count-Sketch-based L_2 heavy hitter algorithm based on EH with $k_{EH} = O(\frac{1}{\epsilon^2})$ can solve (ϵ, L_2)-heavy hitters problem in the Sliding Window and Sub-Window query using $O(\epsilon^{-4} \log^3 N \log \delta^{-1})$ memory bits. As shown in [62], Recursive Sketch with (g, ϵ)-heavy hitter algorithm which finds all i such that $g(f_i(t_1, t_2)) \ge \epsilon G(t_1, t)$ will return $\hat{G}(t_1, t_2) = G(t_1, t_2) \pm \epsilon G(t_1, t)$ and errors with probability at most 0.3, and with $O(\log M)$ space overhead. Therefore, using EH and buckets of universal sketches with Count Sketches for L2-HH subroutines, Alg. 2 estimates GSum statistics using $O(\epsilon^{-4} \log^3 N \log M \log \delta^{-1})$ bits of memory.

EHUniv Optimizations. Straightforward EHUniv implementation can incur large memory usage, as universal sketches in EH need to be configured with the same parameters and memory for mergability among buckets, where each sketch cannot be too small to guarantee good accuracy for a bucket. However, newer EH buckets maintained in the window are usually very small-sized (e.g., sizes $1, 2, 4, \ldots$). To optimize EHUniv memory usage and runtime, we propose to use exact item frequency maps for smaller buckets (when sizes are below the sketch memory) and universal sketches for larger buckets. The hybrid sketch/map construction reduces memory footprint and per-item update time, and improves accuracy because maps provide deterministic results for small buckets. When a map size exceeds the threshold, it is converted into a universal sketch. Querying an interval among active buckets may access maps or sketches. If the time range includes only maps, we merge selected maps in the time range to calculate item frequencies and statistics. If it includes only sketches, we query the merged sketch. When both maps and sketches are present, we merge the maps into one, update a universal sketch with these frequencies, and combine all sketch buckets and the updated sketch.

4.3 Single-Machine PromSketch

PromSketch, as an intermediate result cache, can be applied to both single-node and distributed monitoring systems. We first introduce

[1]Entropy is measuring the diversity/uncertainty of the data in a timeseries, defined as $H \equiv -\sum_{i=1}^{m} \frac{f_i}{n} \log(\frac{f_i}{n})$ [67, 71], with the total item count n in a window and the number of distinct items m.

the end-to-end design for integrating it into a single-node system with the Promethues architecture.

PromSketch data ingester. Data ingester inserts collected timeseries data samples into both the backend TSDB and corresponding PromSketch cache instances in parallel.

Rule manager. Rule manager issues rule queries. When it initiates a rule query, it signals the query engine that a query is periodic and eligible for caching. The query engine then creates a PromSketch instance. If rule configurations are updated and certain rules are removed, the corresponding PromSketch instances are removed.

PromSketch query engine. The query engine registers a PromSketch cache instance when it first executes a rule query, with timeseries ID (or name), statistical function, and query window size. If multiple rule queries ***share the same timeseries and statistical function but have different window sizes***, the PromSketch cache expands its window range to the largest query window for best possible caching. When evaluating an `aggregation_over_time` query, the query engine first checks whether the timeseries has been precomputed by PromSketch. If available, it computes results using PromSketch; otherwise, it retrieves raw data samples from the cache or storage to perform exact query computation. PromSketch supports evaluating multiple timeseries sequentially (e.g., integrating with Prometheus) or in parallel across multiple cores (e.g., integrating with VictoriaMetrics).

PromSketch is designed to be compatible with PromQL-like query languages, including those used by Prometheus, VictoriaMetrics, and more [13, 24], with `aggregation_over_time` functions. To support the PromSketch cache with the query engine, we extend the query parser to include an option for utilizing the PromSketch cache at the entry point of the query's Abstract Syntax Tree, which is widely used to parse `aggregation_over_time` functions in PromQL. This preserves the original query syntax and allows outer functions to process results from the PromSketch cache. For queries that first aggregate by timeseries label and then by time (e.g., avg_over_time(max(metric)[10s])), we initiate a PromSketch instance with the inner aggregated timeseries (e.g., max(metric)) as input. Similarly, for queries that join timeseries before applying time-based aggregation, the joined timeseries samples are inserted into a PromSketch instance. In this work, we focus on optimizing aggregation over time functions, leaving optimizations for label-dimension aggregation to future work.

PromSketch cache considerations. PromSketch uses a hash table for timeseries indexing as Prometheus for prototyping. For each timeseries, insertions and queries are performed concurrently. Inserting to a PromSketch instance may require reconstruction of its EH buckets. Therefore, we add a Read-Write lock between query and insertion threads for each PromSketch instance, allowing multiple concurrent reads to the buckets while permitting only one insertion at a time. PromSketch cache is maintained dynamically: If some rules are removed by the users, PromSketch will remove cache instances that are no longer needed. Optionally, PromSketch can also integrate VictoriaMetrics' timeseries index cache for accelerated sketch instance look-up. Moreover, PromSketch has several reliability and data ordering considerations: (1) When a running PromSketch fails, the in-memory cache can be rebuilt with old data from storage, with another PromSketch instance accepting new data with current timestamps. In this case, queries are answered by merging two instances. (2) PromSketch has the same out-of-order data model as VictoriaMetrics [26] and Prometheus [4], where only the data samples with current timestamp ranges are accepted and out-of-order/duplicated samples should be rejected. In practice, PromSketch cache is placed after the deduplication and reordering component in VictoriaMetrics [27].

4.4 Extension to Distributed PromSketch

To demonstrate a distributed system design, we integrate PromSketch into a cluster-based VictoriaMetrics-like architecture [7]), where all *ingester*, *cache*, *query*, and *storage* components can be running as container nodes as microservices. This cloud-native design brings several benefits — The timeseries set is dynamically partitioned using consistent hashing [64] for possible scaling up and down, with each PromSketch node independently maintaining its own data shard without sharing data with other nodes. This is different from VictoriaMetrics' design of tightly coupling the query and caching in same nodes. Ingester and query nodes access timeseries based on the partitioning. With this design, the number of PromSketch cache and the number storage/query nodes can be adjusted accordingly based on the load. Moreover, distributed PromSketch also allows replicas of cache nodes for fault tolerance in case of node failure as well as leveraging the auto-scaling mechanism provided by Kubernates [7].

5 IMPLEMENTATION

We implement PromSketch as a plugin with 5K lines of Go code and integrate it into Prometheus (release-2.52) and VictoriaMetrics (v1.102.0). For example, users can apply a ~30-line patch to Prometheus for integration.

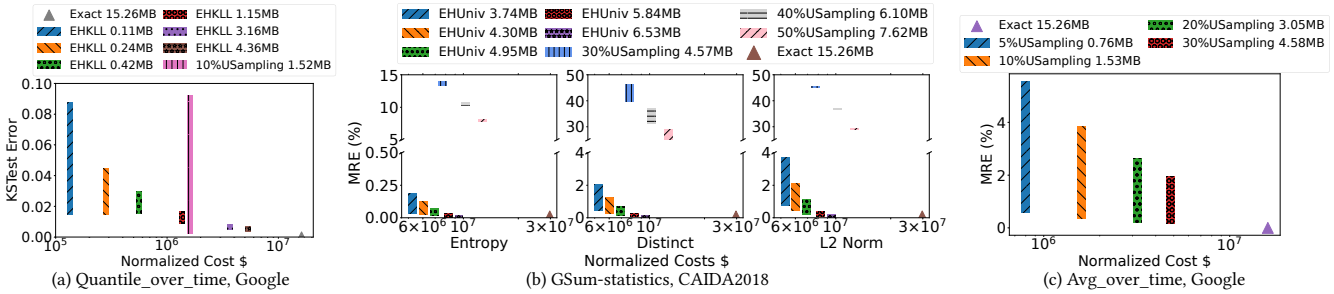
Algorithm implementation optimizations: To further improve system performance, we implement two optimizations following [101] for universal sketches in Alg. 2: (1) *One layer update:* We update only the lowest sampled layer per insertion, reducing the layers updated to one per insertion. (2) *Pyramid memory:* We use larger Count Sketches in upper layers and smaller ones in lower layers, while preserving high accuracy, as the layered-sampling in universal sketching reduces the data size reaching lower layers.

Extending to more statistics: We implement approximate caching for additional statistics using sliding window uniform sampling [73] for statistics like average, sum, standard deviation and variance. This algorithm maintains a sample set S of the most recent sliding window with time range T. New items are added with a probability $p \in (0, 1)$, and outdated samples are discarded. For a query over (t_1, t_2), it answers queries with samples within the range from S.

6 EVALUATION

We evaluate PromSketch's end-to-end performance, provide a sensitivity analysis and demonstrate that:

- PromSketch offers $\leq$ 5% mean errors across statistics at 5× to 75× less system costs (compute and memory) of exact query engines. Thus, the operational costs are reduced by 1332× and 4× compared to Prometheus and VictoriaMetrics, respectively.
- PromSketch offers up to two order of magnitude smaller query latency than Prometheus and VictoriaMetrics.



(a) Quantile_over_time, Google (b) GSum-statistics, CAIDA2018 (c) Avg_over_time, Google

Figure 6: Query cost-accuracy analysis of multiple zoomed-in sub-window queries. ***Normalized costs refer to compute (microseconds) and memory usage (Bytes) costs per set of queries (10K-, 100K, 1M-sample sub-windows) in total. Each bar shows the error region of multiple sub-window queries given a statistic.***

- PromSketch maintains up to 8× faster ingestion than alternative fixed sliding window designs when querying multiple metrics and time windows, representing a moderate 1.3× to 3× slowdown compared to non-cached Prometheus depending on the number of timeseries and PromSketch algorithm choices.

Testbed: Our single-machine experiments run on an Ubuntu 20.04 system with a 32-core Intel Xeon Gold 6142 (2.6GHz), 384GB DRAM, and a 1TB SATA HDD. The cluster experiments use three servers, each with a 24-core AMD 7402P (2.8GHz), 128GB DRAM, 1.6TB NVMe SSDs, and 100Gbps NICs.

Datasets: We use two synthetic and two real-world traces. (1) *Synthetic traces:* We generate Zipf-distributed (obeying $Pr(k) = (1+k)^{-1.01}, k \in N$, *Zipf*) 10M data samples at configurable intervals and 10M uniform distribution data (*Uniform*), with values in $[0, 10^5]$, for each timeseries. We create a dynamic dataset (*Dynamic*) that transitions between Zipf, uniform, and normal distributions ($\mu = 5 \times 10^4, \sigma = 10^4$), generating 1M data points per distribution in a continuous cycle. (2) *Real-world traces:* We use Electronic Power dataset (*Power*) [59], and Google Cluster data v3 [56] (*Google*), where we use start_time as time and average_usage.memory as memory_usage. For EHUniv evaluation, we use CAIDA datasets [96] of source IP addresses from a NYC Equinix backbone on 2018-12-20 (*CAIDA2018*) and 2019-01-17 (*CAIDA2019*). PromSketch sensitivity analysis is conducted on the first 20M points, and 2M points of *Power* due to length constraints.

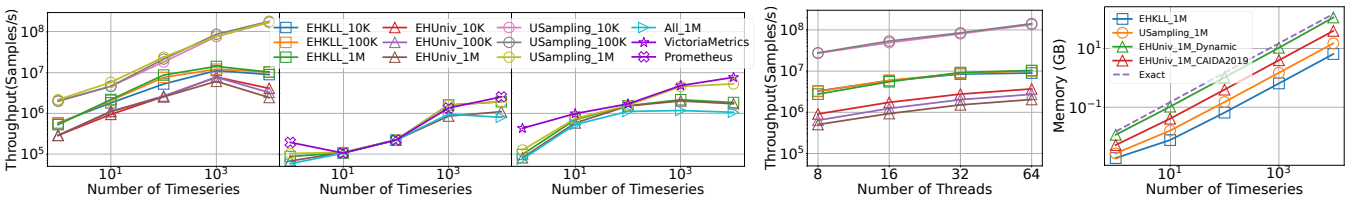
Baselines and evaluation metrics: We compare PromSketch with (1) *Prometheus* system. (2) single-machine and distributed *VictoriaMetrics*. From the space of approximate analytics engines, we compare PromSketch against (3) *Uniform Sampling*: We implement uniform sampling and insert the sampled data into an array-based caching layer for Prometheus, tested with varied sampling ratios. We evaluate rule query latency, insertion throughput, memory usage, accuracy, and operational costs. Accuracy is assessed by Mean Relative Error (MRE) against exact statistics. For quantile, min, and max estimations, we use the Kolmogorov-Smirnov test (KSTest) [41] to compare CDF differences.

6.1 End-to-End PromSketch Performance

For evaluating query latency and insertion throughput, we set uniform sampling at a 10% sampling rate; for EHKLL, we set KLL space limit parameter [65] $k_{KLL} = 256$ (where $k_{KLL} = (1/\epsilon_{KLL})\sqrt{\log(1-\delta)}$), and EH error parameter $k_{EH} = 50$, for 5% KSTest errors; for EHUniv, we set $k_{EH} = 20$ for 5% relative errors.

6.1.1 Compute and memory costs vs. accuracy. Under a cloud billing model, users pay for resources such as memory and CPU cores. Fig.6 shows the normalized operational costs for concurrent queries, including insertion, query compute, and memory usage (excluding storage and network). The costs are normalized by computation time and memory usage. Each figure shows the average errors of different sub-window sizes, with confidence levels shown as a region after five runs. Fig. 6(a) and (c) depict queries with sub-window sizes of 1M, 100K, and 10K samples, using a fixed 1M-sample sliding window. The zoom-in queries mimic anomaly detection: the user first queries a 1M window, then splits it into ten 100K sub-windows for further queries, and finally divides the last 100K sub-window into ten 10K sub-windows for finer granularity. Fig. 6(b) shows the mean relative errors and confidence levels for `entropy`, `distinct` counting, and L_2 norm, with a 1M-sample sliding window and zoomed-in sub-windows of suffix length from 100K to 1M with an interval of 10K samples. We issue queries every 100K samples. For quantiles and GSum-statistics, PromSketch offers better cost-accuracy tradeoffs than uniform sampling. PromSketch maintains less than 5% errors even for the smallest sub-windows (10K samples) while reducing costs by 75× for EHKLL, 10× for Sampling, and 5× for EHUniv compared to the exact baseline.

6.1.2 Cloud operational cost estimations. We compare operational costs of Prometheus, VictoriaMetrics, and PromSketch integration in Table 1. PromSketch respectively reduces the query processing cost by about 1332× compared to AWS Prometheus Pricing and at least 4× compared to VictoriaMetrics, while not increasing the storage and data ingestion costs. For a 1000-node Kubernetes cluster collecting 1000 metrics per node per second for a month, the total ingestion is 2,678B samples/month, requiring 1M samples/s. Assuming each metric has 20 labels with 100 unique values and averaging 30 bytes per label and 2 bytes per sample after compression [6], and 10 rule queries running 24/7, querying every minute with 8000 timeseries and 1M samples per series, the cost breakdown is as follows: (1) **AWS Prometheus pricing [6]:** Data ingestion costs $47,746, storing 7,156GB of metrics and labels costs $214/month, and query processing costs $357,120. This is an estimate for 10 alerting rules, and we envision the cost to be at least several orders of magnitude more when it scales up. (2) **VictoriaMetrics [29]:** Using the typical cloud billing model [2] and assuming each data sample has 64-bit floating point value and 64-bit timestamp associated after decompression, 10 queries concurrently require 10× 8 Billion × 16B = 1,280 GB memory, costing at least $7,443/month for compute (using x2idn.24xlarge [2] with 96 vCPUs and 1.5TB memory).



(a) PromSketch-only (b) PromSketch+Prometheus (c) PromSketch+VictoriaMetrics (d) PromSketch-only Scalability (e) #Timeseries vs. Memory

Figure 7: Single-Machine data insertion throughput with different timeseries numbers and sliding window sizes on Dynamic dataset for (a) PromSketch, (b) PromSketch integrated to Prometheus, and (c) PromSketch integrated to VictoriaMetrics; (d) insertion scalability with different thread numbers on Zipf dataset and 10K timeseries; and (e) memory usage with different timeseries numbers. ***All_1M refers to running all three algorithms together with 1M-sample window per series. In (e), we show memory consumption for the Dynamic dataset, with EHUniv also tested on CAIDA2019.***

Table 4: Total concurrent rule query latency on 10K-, 100K-, and 1M-sample windows. ***"PM" and "VM" stand for Prometheus and VictoriaMetrics, and "PS-PM" and "PS-VM" refer to corresponding PromSketch integration.***

Statistics	Datasets	PM	PS-PM (↓)	VM	PS-VM (↓)
0.9-Quantile	Zipf	4155s	32.3s (137×)	63.0s	3.1s (20×)
	Dynamic	5005s	27.7s (181×)	96.1s	3.2s (30×)
Max	Zipf	1102s	31.5s (35×)	27.0s	3.1s (9×)
	Dynamic	1421s	25.1s (57×)	29.3s	3.2s (9×)
0.5-Quantile	Zipf	12858s	64.9s (198×)	207.5s	6.5s (32×)
+0.9-Quantile	Dynamic	7537s	53.1s (142×)	493.2s	6.5s (76×)
0.5-Quantile+Max	Zipf	21494s	105.9s (203×)	402.8s	12.9s (31×)
+0.9-Quantile+Min	Dynamic	11383s	116.9s (97×)	1014.4s	12.9s (78×)
Distinct	Zipf	1779s	46.1s (39×)	29.4s	2.2s (14×)
	Dynamic	2593s	23s (113×)	100.1s	1.9s (53×)
	CAIDA2019	1688s	28.8s (59×)	31.5s	2.0s (16×)
Entropy	Zipf	2042s	53.6s (38×)	48.6 s	2.8s (17×)
	Dynamic	7871s	34.0s (231×)	179.9s	1.8s (100×)
	CAIDA2019	2105s	33.0s (64×)	44s	2.3s (19×)
L_2 Norm	Zipf	1940s	50.4s (39×)	42.26 s	2.62s (16×)
	Dynamic	3562s	26.0s (137×)	280.8s	1.77s (158×)
	CAIDA2019	1969s	30.7s (64×)	42s	2.0s (21×)
Distinct	Zipf	7517s	112s (67×)	415.48 s	5.2s (81×)
+Entropy	Dynamic	13266s	73.0s (182×)	1146s	9.9s (116×)
+L_2 Norm	CAIDA2019	6866s	77.5s (89×)	336s	6.5s (52×)
Average	Zipf	1158s	9.4s (123×)	1.84s	0.93s (2×)
	Dynamic	1164.8s	9.4s (124×)	2.44s	0.96s (2.5×)
Average	Zipf	2492s	18.4s (135×)	17.9s	1.1s (16×)
+Stddev	Dynamic	2520s	20.5s (123×)	23.3s	1.1s (21×)
0.9-Quantile+Max	Zipf	8985s	180.2s (50×)	349s	9.2s (38×)
+Average+Distinct	Dynamic	13177s	88.6s (154×)	590s	9.0s (65×)

Storage costs $1,065/month, with no data ingestion charge [29]. (3) **PromSketch-PM:** With Prometheus, PromSketch only processes each sample once, costing $267.8/month for query processing ($0.1/Billion samples). (4) **PromSketch-VM:** With VictoriaMetrics, PromSketch uses ~3MB per timeseries for a 1M-sample window and 5% error target, requiring $10 \cdot 8000 \cdot 3\text{MB} = 240$ GB, which can be handled by an m6g.16xlarge instance (64 vCPUs, 256GB) at $1,833/month. Overlapping queries on the same timeseries will further reduce costs.

6.1.3 Single-Machine Rule Query Latency. We evaluate PromSketch's end-to-end query latency in Prometheus and VictoriaMetrics across different statistics and query window sizes on 10K time series each with 100ms data insertion interval. Table 4 reports total latencies for concurrent drill-down queries over the most recent 10^5-, 10^4-, and 10^3-second time windows for each listed statistics. The total latency is averaged from 10 runs after a warm-up of 10^5 seconds. Each `aggr_over_time` query aggregates statistics across all timeseries. With Alg. 1, PromSketch improves quantile query latencies by up to 203× over Prometheus, and 78× over VictoriaMetrics. With Alg. 2, PromSketch improves distinct counting, entropy, and L_2 norm query latencies by up to 231× compared to Prometheus and up to 158× compared to VictoriaMetrics. With 10% uniform sampling, PromSketch improves `average` and `stddev` query latencies by 135× over Prometheus and 21× over VictoriaMetrics. The smaller improvement for `average` is due to VictoriaMetrics's data caching optimization for `avg`, while it does not optimize complex queries such as quantiles and distinct counting. The overall smaller improvements on VictoriaMetrics compared to Prometheus are attributed to VictoriaMetrics' more efficient storage engine.

6.1.4 Single-Machine Insertion Throughput. To evaluate the impact of PromSketch precomputation on insertion throughput with the backend TSDBs, we set the data timestamp interval as 100ms and measure the duration of inserting 60-hour data (2.16M samples) with varying timeseries numbers. Each algorithm is tested with 10K-, 100K- and 1M-sample window sizes. Fig. 7 shows the insertion throughput with timeseries evenly distributed across CPU cores. Fig. 7 (a) shows the insertion throughput of PromSketch alone without inserting into backend TSDB, can achieve 10M samples/s for EHKLL, 2M samples/s for EHUniv, and 100M samples/s for 10% uniform sampling. We show the 1M-sample window size for each algorithm in integrated systems. For Prometheus integration (Fig. 7(b)), with 10K number of timeseries, uniform sampling, EHKLL, EHuniv, and running all three algorithms together have 1.3×, 1.3×, 2.3×, and 3.1× smaller insertion throughput compared to Prometheus. For integration with VictoriaMetrics (Fig. 7(c)), PromSketch achieves over 1.7M samples/s insertion throughput with 100 to 10K timeseries. With VictoriaMetrics, uniform sampling, EHKLL, EHUniv, and running all three algorithms together has 1.4×, 4.1×, 4.5×, and 7.2× smaller insertion throughput compared to VictoriaMetrics, respectively, with 10K number of timeseries. The discrepancy is attributed to VictoriaMetrics' faster storage backend. Fig. 7 (d) shows that each algorithm scales linearly as the number of threads increases, with 10K timeseries, 10K-, 100K-, and 1M-sample time windows, and varied thread counts in a single-machine setting.

6.1.5 Distributed PromSketch Performance. We integrate PromSketch into the VictoriaMetrics cluster version [7], distributing query, sketch, and storage nodes across 3 servers.

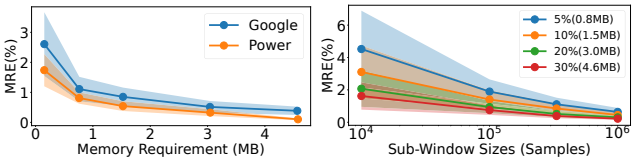
Rule Query Latency. Data is collected every second per series through a single ingestion node, with a synthetic data generator issuing Zipf-distribution data of different timeseries. Table 6 shows

Table 5: VictoriaMetrics+PromSketch cluster version insertion throughput (M/s) with different nodes and timeseries numbers.

#Timeseries	1-node	2-nodes	3-nodes
20K	1.33	2.56	3.86
40K	1.30	2.60	3.79
80K	1.30	2.57	3.79

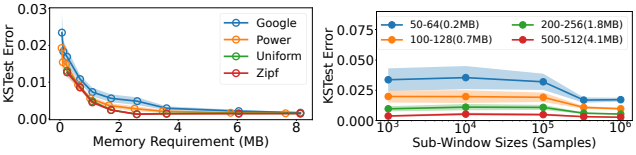
Table 6: VictoriaMetrics(w/ PromSketch) cluster version total rule query latencies (seconds) with different nodes. ***We show the speedup comparing VictoriaMetrics w/ PromSketch and without.***

#Timeseries	VictoriaMetrics		w/PromSketch	
	1-node	3-nodes	1-node (↓)	3-nodes (↓)
20K	74.49	44.43	4.53 (16×)	3.22 (14×)
40K	141.62	121.58	6.95 (20×)	4.04 (30×)



(a) 300-600K Interval (b) Multi Sub-windows, Google

Figure 8: Uniform sampling's mean average estimation error for (a) memory-error in 300-600K intervals, and (b) various sub-windows. ***A legend label `x%` refers to a configuration with `x%` sampling rate.***



(a) 300-600K Interval (b) Multi Sub-windows, Google

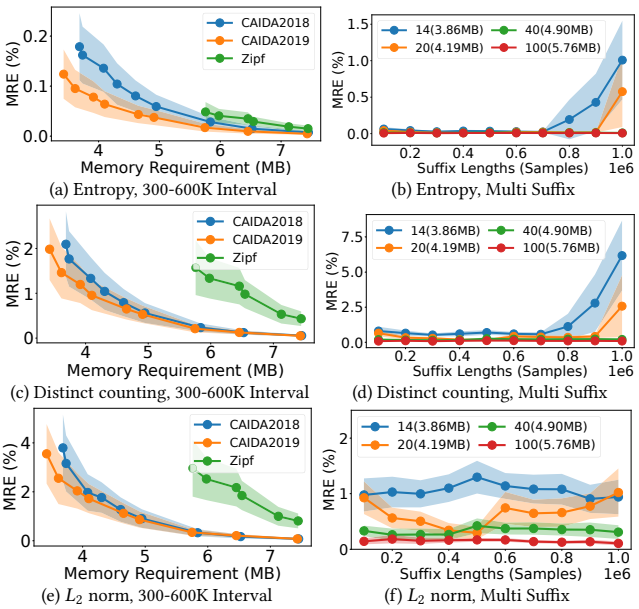
Figure 9: EHKLL's mean quantile estimation KSTest error for (a) memory-error in 300K-600K intervals, and (b) various sub-window sizes. ***A legend label `x-y(z MB)` denotes an EHKLL configuration with $k_{EH} = x$, $k_{KLL} = y$, and z MB total memory.***

the total query latencies of four statistics (0.9-quantile, max, average, and distinct) with 1K-, 10K-, and 100K-second query windows (12 concurrent queries). PromSketch cache reduces the total latency by up to 30× compared to VictoriaMetrics Cluster. Scaling from 1 to 3 nodes in PromSketch reduces latency by 1.4× for 20K timeseries and 1.7× for 40K timeseries, representing a sub-linear speedup due to not all query nodes being used by the cluster scheduler.

Insertion. Table 5 shows the throughput with different nodes and timeseries when inserting data to all three algorithm instances for each timeseries in VictoriaMetric + PromSketch. With increasing number of server nodes, PromSketch's ingestion performance scales linearly from 1.33M/s to 3.86M/s with a large number of concurrent timeseries. We observe that changing the number of monitored timeseries (from 20K to 80K) does not show a significant performance impact, indicating the feasibility of supporting larger-scale cloud infrastructure monitoring.

6.2 PromSketch Sensitivity Analysis

6.2.1 Accuracy with memory consumptions and sub-window sizes. To evaluate memory consumption and empirical errors, we set the sliding window size to 1M samples and query sub-windows



(a) Entropy, 300-600K Interval (b) Entropy, Multi Suffix

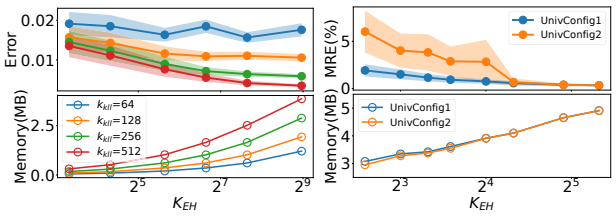
(c) Distinct counting, 300-600K Interval (d) Distinct counting, Multi Suffix

(e) L_2 norm, 300-600K Interval (f) L_2 norm, Multi Suffix

Figure 10: EHUniv's mean relative errors of entropy, distinct counting, and L_2 norm estimation for (a) memory-error in 300K-600K intervals, and (b) various suffix lengths on CAIDA2018. ***A legend label `x(z MB)` denotes an EHUniv configuration with $k_{EH} = x$, a 16-layer universal sketch (8 layers with 3-rows, 2048-column CS, and 8 layers with 3-row, 512-column CS), and z MB total memory, where CS referes to Count Sketch.***

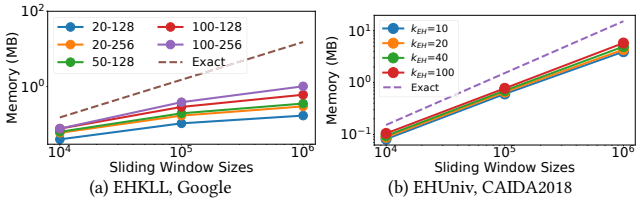
ranging from 300K to 600K samples and different suffix lengths on real-world traces. Differences in datasets primarily stem from the skewness of workload distribution. Fig. 8 and Fig. 9 present the average statistic and quantile statistic results. Both Fig. 8(a) and Fig. 9(a) show that increased memory improves estimation accuracy. Fig. 8(b) and Fig. 9(b) show that larger sub-windows also yield smaller estimation errors compared to smaller ones, and more memory enhances accuracy across all sub-window sizes. We repeat the above evaluation for entropy, distinct counting, and L_2 norm, as in Fig. 10. As memory increases, the relative errors for these estimates decrease. With 4MB memory, EHUniv can achieve 2% MRE for L_2 norm, 1% MRE for entropy, and 2% MRE for distinct counting on both CAIDA datasets for the $\frac{1}{3}$ sub-window in a 1M sliding window. Additionally, as suffix lengths increase, estimation errors for entropy and distinct counting rise due to the fixed memory space accommodating more data samples. Errors for L_2 norm may vary depending on how well suffix lengths align with EH bucket boundaries. EHUniv approaches near-zero errors when its memory approaches that of the exact algorithm. Fig. 11 shows the configuration impact on EHKLL and EHUniv. Given a KLL or Universal sketch configuration, larger k_{EH} reduces window alignment error and thus has smaller estimation errors. Given a k_{EH}, smaller KLL and Universal sketch memory configuration has larger errors. EHUniv shows smaller memory gap when k_{EH} is larger than 20 because each EH bucket is small and thus uses exact map, with errors solely due to window alignment.

6.2.2 Memory with timeseries numbers and sliding window sizes. We evaluate PromSketch memory consumption under 5% error target and 1M-sample sliding window for each timeseries. Fig. 7(e)



(a) 300-600K Interval, Google (b) 1M-Window, CAIDA2019

Figure 11: Tuning parameters for EHKLL and EHUniv. ***We show KSTest error for EHKLL. UnivConfig1 is a 16-layer universal sketch (8 layers with 3-row, 1024-column CS, and 8 layers with 3-row, 128-column CS). UnivConfig2 is a 14-layer universal sketch (8 layers with 3-row, 256-column CS, and 6 layers with 3-row, 64-column CS).***



(a) EHKLL, Google (b) EHUniv, CAIDA2018

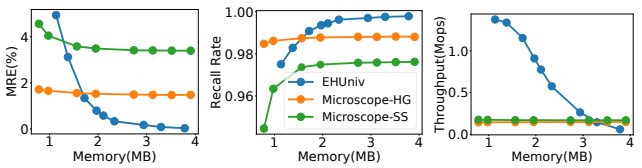
Figure 12: Memory usage with sliding window sizes. ***In (a), a legend label x-y denotes an EHKLL configuration with $k_{EH} = x$, $k_{KLL} = y$. The exact baseline stores every point with its associated timestamp.***

shows the memory increases linearly as the timeseries number increases, because we allocate one PromSketch instance per series. Since Dynamic dataset contains uniformly distributed data, EHUniv's memory usage is higher compared to the more skewed CAIDA2019 dataset. Fig. 12(a) and (b) shows the memory consumption of EHKLL and EHUniv with different sliding window sizes and parameter configurations for each timeseries. For each configuration of both EHKLL and EHUniv, the memory usage increases sublinearly as sliding window sizes grows. Larger k_{EH} and k_{KLL} in EHKLL and k_{EH} in EHUniv use more memory. Given a window size, a larger k_{EH} of EHUniv uses more memory because smaller EH buckets use hash maps for exact computation. Conversely, a smaller k_{EH} reduces memory usage since larger buckets leverage sketches for compression.

6.2.3 Comparing with sliding window algorithm. We compare PromSketch with fixed sliding window algorithm, e.g., MicroscopeSketch [100], on concurrent window queries, evaluating insertion throughput and estimation error across all query windows. We query TopK-frequent item finding over time of EHUniv, and compare MRE and average recall rate over 10 sub-windows (ranging from 100K- to 1M-sample sub-windows in a 1M-sample sliding window) against MicroscopeSketch with HeavyGuardian [102] and SpaceSaving [80] on CAIDA2019. Fig. 13 shows between 1.7MB and 3.3MB memory, PromSketch has up to 8× higher insertion throughputs, smaller errors, and higher recall rates than MicroscopeSketch, because of EH's ability to support multiple windows simultaneously.

7 RELATED WORK

Window-based summaries. While various sliding window methods exist, most do not support arbitrary sub-window queries, incurring per-window maintenance efforts. [36] addresses approximate frequency counting and quantiles. SlidingSketches [57] optimizes hash-based sketches like Bloom filter [42] and CountSketch [48] but



(a) Mean Relative Error (b) Recall Rate (c) Insertion Throughput

Figure 13: TopK-frequent item estimation comparing to MicroscopeSketch with HeavyGuardian (HG) and SpaceSaving (SS).

lacks support for quantile sketches like KLL. ECM-sketch [85] enhances CountMin Sketch [50] by replacing counters with Exponential Histogram for frequency estimation, whereas PromSketch uses sketches as EH buckets. WCSS [39] and MicroscopeSketch [100] support frequency estimation and TopK-frequent item finding, but they lack sub-window querying support and MicroscopeSketch is limited to counter-based sketches. CoopStore [55] focuses on offline precomputation of quantiles and frequencies with fixed window sizes and luxury memory during aggregation to reduce errors. SummaryStore [34] designs approximate timeseries storage with PowerLaw-based sub-window queries, while PromSketch functions as an in-memory cache and integrates to Prometheus-like monitoring systems with rule query support.

Approximate timeseries visualization. M4 [63] and MinMaxCache [76] solve timeseries data point visualization problem with approximate pixel positions in a canvas, by querying min and max values of data points of a time range. They are orthogonal to our work and don't address the window query bottlenecks.

Approximate query processing (AQP) systems. Another type of approximate query system focuses on label dimensional queries. PASS [68, 69] and AQP++ [88] combine precomputed-aggregation and sampling to support SQL queries, e.g., sum, count, min, and variance statistics. VerdictDB [86] acts as a sampling middlebox between the user interface and the backend database, enabling approximate SQL queries without requiring backend modifications. DHS [103] offers streaming frequency estimation with dynamic sketch memory layout, but does not target time window queries.

8 CONCLUSIONS

We present PromSketch, an approximation-first query caching solution that enhances scalability of queries in timeseries monitoring systems by reducing query latency and operational costs. PromSketch leverages approximate (sub)window-based query frameworks and sketches for efficient in-memory intermediate query result caching. Our evaluation shows that PromSketch reduces query latency by up to two orders of magnitude compared to Prometheus and VictoriaMetrics, and reduces query costs by up to three orders of magnitude compared to Prometheus, with 5% average errors across statistics.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Ron Parker for their valuable feedback. Zaoxing Liu and Zeying Zhu were supported in part by the U.S. NSF grants CNS-2431093 and CNS-2415758. Starobinski and Chamberlain were supported in part by grants from the Red Hat Collaboratory at Boston University (2025-01-RH05) and the U.S. NSF grant AST-2229104.

REFERENCES

[1] 2015. Prometheus: Monitoring at SoundCloud. Retrieved 2025/05/23 from https://github.com/pingcap/docs/blob/master/tidb-monitoring-framework.md

[2] 2022. Amazon EC2 On-Demand Pricing. Retrieved 2025/05/23 from https://aws.amazon.com/ec2/pricing/on-demand/

[3] 2022. Kubernetes. Retrieved 2025/05/23 from https://kubernetes.io/

[4] 2022. Prometheus handling out-of-order samples. Retrieved 2025/05/23 from https://promlabs.com/blog/2022/12/15/understanding-duplicate-samples-and-out-of-order-timestamp-errors-in-prometheus/

[5] 2022. VictoriaMetrics. Retrieved 2025/05/23 from https://victoriametrics.com

[6] 2024. Amazon Managed Service for Prometheus pricing. Retrieved 2025/05/23 from https://aws.amazon.com/prometheus/pricing/

[7] 2024. Cluster VictoriaMetrics. Retrieved 2025/05/23 from https://docs.victoriametrics.com/cluster-victoriametrics/

[8] 2024. DataDog Anomaly Detection. https://docs.datadoghq.com/monitors/types/anomaly/.

[9] 2024. Fastcache used by VictoriaMetrics. https://github.com/VictoriaMetrics/fastcache

[10] 2024. Google Cloud scales based on Monitoring metrics. https://cloud.google.com/compute/docs/autoscaler/scaling-cloud-monitoring-metrics

[11] 2024. Google Kubernetes Engine Quotas and Limits. https://cloud.google.com/kubernetes-engine/quotas

[12] 2024. Grafana Dashboards. https://grafana.com/grafana/dashboards/.

[13] 2024. Grafana Mimir. Retrieved 2025/05/23 from https://grafana.com/docs/mimir/latest/

[14] 2024. Grafana Mimir uses Redis or Memcached as chunks-cache, index-cache, results-cache and metadata-cache. https://grafana.com/docs/helm-charts/mimir-distributed/latest/configure/configure-redis-cache/

[15] 2024. InfluxDB. Retrieved 2025/05/23 from https://www.influxdata.com/

[16] 2024. Kubernetes monitoring with Prometheus. Retrieved 2025/05/23 from https://prometheus.io/docs/prometheus/latest/configuration/configuration/#kubernetes_sd_config

[17] 2024. Memcached. https://memcached.org.

[18] 2024. Monitoring Juniper Networks with Prometheus. Retrieved 2025/05/23 from https://github.com/czerwonk/junos_exporter

[19] 2024. Open5GS Metrics with Prometheus. Retrieved 2025/05/23 from https://open5gs.org/open5gs/docs/tutorial/04-metrics-prometheus/

[20] 2024. Prometheus Configurations. Retrieved 2025/05/23 from https://prometheus.io/docs/prometheus/latest/configuration/configuration/

[21] 2024. Prometheus functions. Retrieved 2025/05/23 from https://prometheus.io/docs/prometheus/latest/querying/functions/

[22] 2024. Prometheus Query Language. Retrieved 2025/05/23 from https://prometheus.io/docs/prometheus/latest/querying/basics/

[23] 2024. Prometheus SNMP exporter. Retrieved 2025/05/23 from https://github.com/prometheus/snmp_exporter

[24] 2024. Thanos. Retrieved 2025/05/23 from https://thanos.io/

[25] 2024. VictoriaMetrics Anomaly Detection. Retrieved 2025/05/23 from https://victoriametrics.com/blog/victoriametrics-anomaly-detection-handbook-chapter-2/index.html

[26] 2024. VictoriaMetrics backfilling support for out-of-order samples. Retrieved 2025/05/23 from https://docs.victoriametrics.com/#backfilling

[27] 2024. VictoriaMetrics Deduplication. Retrieved 2025/05/23 from https://docs.victoriametrics.com/#deduplication

[28] 2024. VictoriaMetrics parallel query in vm-select. https://github.com/VictoriaMetrics/VictoriaMetrics/issues/2886.

[29] 2024. VictoriaMetrics Pricing compared to Prometheus. Retrieved 2025/05/23 from https://victoriametrics.com/blog/managed-prometheus-pricing/

[30] 2024. VictoriaMetrics rollup functions. Retrieved 2025/05/23 from https://docs.victoriametrics.com/metricsql/#rollup-functions

[31] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. 2013. Scuba: Diving into data at facebook. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1057–1067.

[32] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*. 574–576.

[33] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European conference on computer systems*. 29–42.

[34] Nitin Agrawal and Ashish Vulimiri. 2017. Low-latency analytics on colossal data streams with summarystore. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 647–664.

[35] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*. 1093–1110.

[36] Arvind Arasu and Gurmeet Singh Manku. 2004. Approximate counts and quantiles over sliding windows. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 286–296.

[37] Awesome Prometheus alerts 2024. Awesome Prometheus alerts. Retrieved 2025/05/23 from https://samber.github.io/awesome-prometheus-alerts/

[38] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. 2018. Memento: Making sliding windows efficient for heavy hitters. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. 254–266.

[39] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[40] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2019. Succinct summing over sliding windows. *Algorithmica* 81 (2019), 2072–2091.

[41] Vance W Berger and YanYan Zhou. 2014. Kolmogorov–smirnov test: Overview. *Wiley statsref: Statistics reference online* (2014).

[42] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[43] Vladimir Braverman, Stephen R Chestnut, David P Woodruff, and Lin F Yang. 2016. Streaming space complexity of nearly all functions of one variable on frequency vectors. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 261–276.

[44] Vladimir Braverman and Rafail Ostrovsky. 2007. Smooth histograms for sliding windows. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*. IEEE, 283–293.

[45] Vladimir Braverman and Rafail Ostrovsky. 2010. Zero-one frequency laws. In *Proceedings of the forty-second ACM symposium on Theory of computing*. 281–290.

[46] Yousra Chabchoub and Georges Heébrail. 2010. Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window. In *2010 IEEE International Conference on Data Mining Workshops*. IEEE, 1297–1303.

[47] Shiyu Chang, Yang Zhang, Jiliang Tang, Dawei Yin, Yi Chang, Mark A Hasegawa-Johnson, and Thomas S Huang. 2017. Streaming recommender systems. In *Proceedings of the 26th international conference on world wide web*. 381–389.

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

[49] Cloudflare Blog - Monitoring our monitoring: how we validate our Prometheus alert rules 2024. Cloudflare Blog - Monitoring our monitoring: how we validate our Prometheus alert rules. Retrieved 2025/05/23 from https://blog.cloudflare.com/monitoring-our-monitoring/

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

[51] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 647–651.

[52] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining stream statistics over sliding windows. *SIAM journal on computing* 31, 6 (2002), 1794–1813.

[53] Ronen Ben David. 2021. *Kubernetes Auto-Scaling: YoYo attack vulnerability and mitigation.* Master's thesis. Reichman University (Israel).

[54] Cristian Estan and George Varghese. 2002. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 323–336.

[55] Edward Gan, Peter Bailis, and Moses Charikar. 2020. Coopstore: Optimizing precomputed summaries for aggregation. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2174–2187.

[56] Google ClusterData 2019 traces 2020. Google ClusterData 2019 traces. Retrieved 2025/05/23 from https://github.com/google/cluster-data/blob/master/ClusterData2019.md

[57] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. 2020. Sliding sketches: A framework using time zones for data stream processing in sliding windows. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1015–1025.

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

[59] Georges Hebrail and Alice Berard. 2012. Individual Household Electric Power Consumption. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58K54.

[60] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. 1997. Range queries in OLAP data cubes. *ACM SIGMOD Record* 26, 2 (1997), 73–88.

[61] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in*

Operating Systems. 150–155.

[62] Nikita Ivkin, Ran Ben Basat, Zaoxing Liu, Gil Einziger, Roy Friedman, and Vladimir Braverman. 2019. I know what you did last summer: Network monitoring using interval queries. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 3 (2019), 1–28.

[63] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. 2014. M4: a visualization-oriented time series data aggregation. *Proceedings of the VLDB Endowment* 7, 10 (2014), 797–808.

[64] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

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

[66] Yehuda Koren. 2009. Collaborative filtering with temporal dynamics. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 447–456.

[67] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. 2006. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Performance Evaluation Review* 34, 1 (2006), 145–156.

[68] Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. 2021. Combining aggregation and sampling (nearly) optimally for approximate query processing. In *Proceedings of the 2021 International Conference on Management of Data*. 1129–1141.

[69] Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. 2021. Combining Aggregation and Sampling (Nearly) Optimally for Approximate Query Processing. https://doi.org/10.48550/ARXIV.2103.15994

[70] Gangmuk Lim, Mohamed S Hassan, Ze Jin, Stavros Volos, and Myeongjae Jeon. 2020. Approximate quantiles for datacenter telemetry monitoring. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1914–1917.

[71] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 101–114.

[72] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *30th USENIX Security Symposium (USENIX Security 21)*. 3829–3846.

[73] Xuesong Lu, Wee Hyong Tok, Chedy Raissi, and Stéphane Bressan. 2010. A simple, yet effective and efficient, sliding window sampling algorithm. In *Database Systems for Advanced Applications: 15th International Conference, DASFAA 2010, Tsukuba, Japan, April 1-4, 2010, Proceedings, Part I 15*. Springer, 337–351.

[74] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2003. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 491–502.

[75] Antonis Manousis, Zhuo Cheng, Ran Ben Basat, Zaoxing Liu, and Vyas Sekar. 2022. Enabling efficient and general subpopulation analytics in multidimensional data streams. *arXiv preprint arXiv:2208.04927* (2022).

[76] Stavros Maroulis, Vassilis Stamatopoulos, George Papastefanatos, and Manolis Terrovitis. 2024. Visualization-aware Time Series Min-Max Caching with Error Bound Guarantees. *Proceedings of the VLDB Endowment* 17, 8 (2024), 2091–2103.

[77] Octavian Mart, Catalin Negru, Florin Pop, and Aniello Castiglione. 2020. Observability in kubernetes cluster: Automatic anomalies detection using prometheus. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 565–570.

[78] Artur Marzano, David Alexander, Osvaldo Fonseca, Elverton Fazzion, Cristine Hoepers, Klaus Steding-Jessen, Marcelo HPC Chaves, Ítalo Cunha, Dorgival Guedes, and Wagner Meira. 2018. The evolution of bashlite and mirai iot botnets. In *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 00813–00818.

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

[80] Michael Mitzenmacher, Thomas Steinke, and Justin Thaler. 2012. Hierarchical heavy hitters with the space saving algorithm. In *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 160–174.

[81] Muhammad Aashiq Moosa, Apurva K Vangujar, and Dnyanesh Pramod Mahajan. 2023. Detection and Analysis of DDoS Attack Using a Collaborative Network Monitoring Stack. In *2023 16th International Conference on Security of Information and Networks (SIN)*. IEEE, 1–9.

[82] Y. Ohsita, S. Ata, and M. Murata. 2004. Detecting distributed denial-of-service attacks by analyzing TCP SYN packets statistically. In *IEEE Global Telecommunications Conference, 2004. GLOBECOM '04.*, Vol. 4. 2043–2049 Vol.4. https://doi.org/10.1109/GLOCOM.2004.1378371

[83] Packets-per-second limits in EC2 2019. Packets-per-second limits in EC2. Retrieved 2025/05/23 from https://stressgrid.com/blog/pps_limits_in_ec2/

[84] Alessandro Vittorio Papadopoulos, Ahmed Ali-Eldin, Karl-Erik Årzén, Johan Tordsson, and Erik Elmroth. 2016. PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 1, 4 (2016), 1–31.

[85] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. 2012. Sketch-based querying of distributed sliding-window data streams. *arXiv preprint arXiv:1207.0139* (2012).

[86] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*. 1461–1476.

[87] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.

[88] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. 2018. Aqp++ connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proceedings of the 2018 International Conference on Management of Data*. 1477–1492.

[89] James Pope, Francesco Raimondo, Vijay Kumar, Ryan McConville, Rob Piechocki, George Oikonomou, Thomas Pasquier, Bo Luo, Dan Howarth, Ioannis Mavromatis, et al. 2021. Container escape detection for edge devices. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 532–536.

[90] Athanasios Priovolos, Dimitris Lioprasitis, Georgios Gardikis, and Socrates Costicoglou. 2021. Using anomaly detection techniques for securing 5G infrastructure and applications. In *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*. IEEE, 519–524.

[91] PromCon 2023 - Yet Another Streaming PromQL Engine 2023. PromCon 2023 - Yet Another Streaming PromQL Engine. Retrieved 2025/05/23 from https://www.youtube.com/watch?v=3kM2Asj6hcg

[92] Prometheus Metrics based autoscaling in Kubernetes 2023. Prometheus Metrics based autoscaling in Kubernetes. Retrieved 2025/05/23 from https://gcore.com/learning/prometheus-metrics-based-autoscaling-in-kubernetes/

[93] Chunhui Shen, Qianyu Ouyang, Feibo Li, Zhipeng Liu, Longcheng Zhu, Yujie Zou, Qing Su, Tianhuan Yu, Yi Yi, Jianhong Hu, et al. 2023. Lindorm TSDB: A Cloud-Native Time-Series Database for Large-Scale Monitoring Systems. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3715–3727.

[94] Mor Sides, Anat Bremler-Barr, and Elisha Rosensweig. 2015. Yo-Yo Attack: vulnerability in auto-scaling mechanism. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 103–104.

[95] Thanos Downsampling, Resolution and Retention 2024. Thanos Downsampling, Resolution and Retention. Retrieved 2025/05/23 from https://thanos.io/v0.8/components/compact/

[96] The CAIDA UCSD Anonymized Internet Traces 2024. The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/catalog/datasets/passive_dataset/. Online.

[97] James Turnbull. 2018. *Monitoring with Prometheus*. Turnbull Press.

[98] Zhiqi Wang, Jin Xue, and Zili Shao. 2021. Heracles: an efficient storage model and data flushing for performance monitoring timeseries. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1080–1092.

[99] Jamie Wilkinson. 2016. Google Promethues: A practical guide to alerting at scale. Retrieved 2025/05/23 from https://docs.google.com/presentation/d/1X1rKozAUuF2MVc1YXElFWq9wkcWv3Axdldl8LOH9Vik/edit#slide=id.g598ef96a6_0_341

[100] Yuhan Wu, Shiqi Jiang, Siyuan Dong, Zheng Zhong, Jiale Chen, Yutong Hu, Tong Yang, Steve Uhlig, and Bin Cui. 2023. MicroscopeSketch: Accurate Sliding Estimation Using Adaptive Zooming. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2660–2671.

[101] Mingran Yang, Junbo Zhang, Akshay Gadre, Zaoxing Liu, Swarun Kumar, and Vyas Sekar. 2020. Joltik: enabling energy-efficient" future-proof" analytics on low-power wide-area networks. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–14.

[102] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2584–2593.

[103] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. 2021. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of ACM SIGKDD*. 2285–2293.