Experiments & Analysis Paper

An Experimental Analysis of Quantile Sketches over Data Streams

Lasantha Fernando, Harsh Bindra, Khuzaima Daudjee
Cheriton School of Computer Science
University of Waterloo
firstname.lastname@uwaterloo.ca

ABSTRACT

Streaming systems process large data sets in a single pass while applying operations on the data. Quantiles are one such operation used in streaming systems. Quantiles can outline the behaviour and the cumulative distribution of a data set. We study five recent quantile sketching algorithms designed for streaming settings: KLL Sketch, Moments Sketch, DDSketch, UDDSketch, and ReqSketch. Key aspects of the sketching algorithms in terms of speed, accuracy, and mergeability are examined. The accuracy of these algorithms is evaluated in Apache Flink, a popular open source streaming system, while the speed and mergeability is evaluated in a separate Java implementation. Results show that UDDSketch has the best relative-error accuracy guarantees, while DDSketch and ReqSketch also achieve consistently high accuracy, particularly with long-tailed data distributions. DDSketch has the fastest query and insertion times, while Moments Sketch has the fastest merge times. Our evaluations show that there is no single algorithm that dominates overall performance and different algorithms excel under the different accuracy and run-time performance criteria considered in our study.

1 INTRODUCTION

Stream Processing Engines (SPEs) are expected to process large numbers of events at high speeds and deliver results rapidly. However, high volumes of events and complex queries can overwhelm a streaming system leading to performance degradation. Consequently, there exist a number of algorithms that strive to answer queries quickly using efficient data structures [19, 34]. In particular, given a query type, there is a class of algorithms known as *sketches* that are able to generate approximate results for that query type based on a summary of the data [4, 36]. The term *sketch* can generally be used to describe an algorithm that creates the sketch or the data structure holding the relevant summary of the observed data. In a streaming system environment, a sketch is created based on a single pass of the data, and allows the SPE to answer a query with significantly less space compared to the size of the ingested data stream.

Quantile computation is an important stream processing operation that has many applications in domains such as network monitoring [15], web server response time monitoring [32], financial stock market monitoring [28], database query optimization [30], search engine log analysis, and mobile network health monitoring [13]. Quantiles can be used more generally to describe the cumulative distribution function and subsequently the probability distribution function of a data set. This is particularly useful when the parametric equations of the data are not known or are difficult to find.

© 2023 Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-093-6 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

It is well known that it is impossible to compute exact quantiles in a single pass without storing all of the data [35]. Early research showed that there is a lower bound of $1.5N$ comparisons required to compute an exact quantile of a data set of size N [8]. Within SPEs, finding quantiles is a challenging problem since these systems typically have limited memory, see the data only once, and do not have precise knowledge of the size of the data. In contrast, a storage system like a relational database can store data, re-create summaries and make multiple passes of the data [6] as and when desired.

Fortunately, many applications can work with inexact quantiles and do not need to incur the high cost of computing exact quantiles. There exists a class of sketching algorithms known as *quantile sketches*[1] that is ideal for computation of approximate quantiles in stream processing since they are able to process a data set and hold a sketch summary for quantile estimation, all in one pass. There are a number of quantile sketching algorithms developed for finding inexact quantiles in high velocity data streams [21, 23, 29, 30, 32, 40]. The most recent of these include Moments Sketch [21], DDSketch [32], UDDSketch [9, 18], KLL Sketch [26, 27, 40], and ReqSketch [14]. The respective papers highlight theoretical and empirical space improvements, mergeability, insertion speed, and quantile computation speed of these algorithms. Although these algorithms are suited for use in streaming settings, they have not been evaluated on a real streaming system. In this paper, we fill this void by conducting a detailed, practical, implementation-based performance evaluation and analysis of key quantile sketching algorithms.

The design of quantile sketching algorithms generally assumes that all data within the period of interest are consumed and the quantile is computed afterwards. However, this is not the case with windowed operations within streaming systems due to the network delay it takes for an event to come from the source to the streaming system. Another contribution of this paper is to analyze how the best performing quantile sketches fare in a setting with and without late-arriving data as found in windowed computations.

A key metric of our study is accuracy, which we evaluate uniformly by using the same criteria, data sets and implementation in Apache Flink, an industrial-strength streaming system. We also use uniform criteria and data sets to evaluate speed, space requirements, and adaptability through a standalone implementation for performance isolation.

The key contributions of our paper can be summarized as follows:

(1) We conduct a detailed performance evaluation of key quantile sketching algorithms.
(2) We evaluate accuracy of these algorithms under uniform experimental settings in Apache Flink, an industrial-strength streaming system, with and without late-arriving data.

[1] We use the terms *quantile sketches*, *sketching algorithms*, and *quantile sketching algorithms* interchangeably in the paper.

Table 1: Example data set with corresponding rank and quantile for each element

x	3	7	8	8	11	14	16	18	30	51
Rank(x)	1	2	3	4	5	6	7	8	9	10
Quantile$^{-1}(x)$	.1	.2	.3	.4	.5	.6	.7	.8	.9	1

(3) We provide a comparison of adaptability between algorithms.
(4) We implement UDDSketch in Java to enable a uniform experimental comparison.

The rest of this paper is organized as follows. In Sec. 2, we provide background on relevant stream processing concepts, quantile computations, error measurements and statistical metrics. Sec. 3 describes the sketching algorithms evaluated in our study. Our experimental results and analyses are presented in Sec. 4. Sec. 5 surveys work that includes past related studies as well as other sketching algorithms to provide additional context on the selection of algorithms studied in this paper. Sec. 6 concludes the paper.

2 BACKGROUND

This section provides an overview of the basics of quantiles and the types of commonly considered errors for quantile queries. Additionally, a method of measuring the weight of the tail of a distribution, kurtosis, is presented which can aid analysis. We also discuss *mergeability*, a property of sketching algorithms that is highly desirable in a distributed streaming setting. We also discuss some key stream processing system concepts such as window operations and late-arriving data.

2.1 Quantiles

Given a data set D of size N, the rank of an element, Rank(x), is the index of that value in the sorted version of D. A q-quantile, x_q, is the item whose rank in sorted D is $\lceil qN \rceil$ for $0 < q \leq 1$. The rank of x can be interpreted as the number of elements less than or equal to x. Similarly, the value of the q-quantile, x_q, says that approximately $100q$ percent of the data is smaller or equal to x_q. Quantile$^{-1}(x)$ returns the value of q such that the q-quantile query is x. That is, Quantile$^{-1}(x) \rightarrow q$ and q-quantile $\rightarrow x$. Examples of Rank and an inverse q-quantile function are given in Table 1.

2.2 Relative Error vs Rank Error

Rank and relative error are best understood with an example. Consider a user interested in the 0.9 quantile[2] for the data set in Table 1. The true 0.9-quantile of this data set is 30 but assume some quantile sketch estimates the value of 18 for the 0.9-quantile query. Formally, Rank error for a q-quantile which outputs an estimate $\hat{x}_q$ is defined as:

$$\frac{|(\text{Rank}(x_q) - \text{Rank}(\hat{x}_q)|}{N} = \left| q - \frac{Rank(\hat{x}_q)}{N} \right|$$

Essentially, the rank error of a q-quantile query is how far Quantile$^{-1}(\hat{x}_q)$ is from q. In this case, $\hat{x}_q = 18$ has a rank of 8 (i.e. 18 is the 8th index) but the true 0.9-quantile would have a rank of 9, so the rank error is $0.9 - 8/10 = 0.1$ or 10%. In contrast, the relative error of the query is how far the estimated value $\hat{x}_q$ is from the true value x_q of the quantile, divided by the true value. Formally, the relative error can be written as:

$$\frac{|x_q - \hat{x}_q|}{x_q}$$

The relative error for this instance becomes $|30 - 18|/30 = 0.4$ or 40%. This exercise shows that rank error and relative error are not equal particularly when it comes to the tail end of the distribution. Many applications such as those processing web request latencies [32] handle data with long tails. Moreover, relative error is related to the absolute error of an estimate, and hence represents a more intuitive error measure. Therefore, relative error is used in our evaluations.

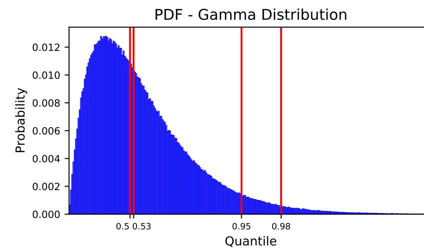


Figure 1: Probability distribution function for a data set sampled from gamma distribution with a tail (kurtosis of 3)

Fig. 1 provides a probability density function (PDF) representation of a data set sampled from a gamma distribution with a tail on the right hand side. The x-ticks represent where to find the values of the 0.5^{th}, 0.53^{th}, 0.95^{th}, 0.98^{th} quantiles. Consider if one wanted to use the value at the 0.5^{th} quantile to approximate the 0.53^{th} quantile, and the 0.95^{th} to approximate the 0.98^{th}. Visually, one can see that the values for the 0.5^{th} and 0.53^{th} quantiles are close together and so they may provide a reasonable approximation for one another. In contrast, the values for the 0.95^{th} and 0.98^{th} quantile are noticeably farther apart and may not be a good approximation for one another. In both cases, there would be a 3% rank error but the relative error would be higher when the 0.95^{th} quantile approximates the 0.98^{th} quantile. As seen in this example, the relative error provides a better representation of the actual error of the estimate, regardless of the distribution.

2.3 Kurtosis

Kurtosis can be used as a measure of how heavy or light the tail of a distribution is compared to a normal distribution – the longer the tail of a distribution, the larger the kurtosis. Our evaluation uses the definition which removes excess kurtosis to make the normal distribution have a kurtosis of 0 [1]. The kurtosis of the distribution in Fig. 1 is 3, which implies it has a long tail relative to a normal distribution.

2.4 Mergeability

Mergeability is the ability to merge 2 sketches into 1 sketch that represents all of the data from both sketches. This should be accomplished without any change to the error guarantees of the merged sketch. Mergeability is essential in a distributed setting where the partitioned data can be summarized locally so that

[2] q quantiles can also be referred to as the $q \times 100^{th}$ percentile. The 0.9^{th} quantile and the 90th percentile refer to the same value.

only the sketch summaries need to be merged across different machines. Otherwise, moving all data to a central sketch can be a significant bottleneck.

2.5 Window Operations

A key operation in streaming systems is to use windows to group together events over some attribute and provide results periodically. SPEs will use windows to group events for processing different queries like joins, aggregations, filters and quantiles. There are time-based windows and sequence-based windows. A 10-second time-based window would group events generated in the next 10 seconds, whereas a sequence-based window of length 10 would group the next 10 events.

Time-based windows can be further classified as fixed windows, sliding windows or session windows [7]. A fixed window with a length of 10 seconds would group events generated or ingested from time t to $t+10$ s and emit the results before starting the next group from $t + 10$ to $t + 20$ s. A sliding window of the same length and a period of 1 s would create a group from time t to $t + 10$ s, create another group from $t + 1$ s to $t + 11$ s, and so on. A session window with a timeout of 10 s would start grouping events at time t and keep collecting events until a period of inactivity for 10 s, i.e., if the last event arrived at $t + 23$ s, events would be grouped from t to $t + 33$ s.

Windows can group events by event generated time or ingestion time. Generated time alludes to the time an event was created at the source, and ingestion time is the time that event was received by the SPE. The delay between the time the event is created at the source and the time it is received by the streaming system is called network delay. This is a general term to encompass delays, including the time required to travel from source to destination, e.g., due to network latency. For example, if you are querying for the median fare price of taxi rides between 1:00AM and 1:02AM, you would like to use generated time since the time the taxi ride event was ingested by the streaming system is irrelevant. Due to the popularity of such use cases, this paper groups events using generated time and uses time-based fixed windows in its streaming window operations.

2.6 Late-arriving Data

The design of quantile sketching algorithms generally assumes that all data within the period of interest are consumed and the quantile is computed afterwards. However, this is not the case with windowed operations within streaming systems due to the network delay it takes for an event to come from the source to the streaming system. Events that fall within the scope of a window, but come after the SPE has already processed the window, are identified as late-arriving data. Late-arriving data can be handled using different techniques depending on the context. For the purpose of our study, we assume late-arriving events will be dropped. These dropped events, if not included in the computation of the window result, can consequently deteriorate the accuracy of the result. This paper provides an analysis on how quantile sketches fare with late-arriving data.

3 ALGORITHMS

We evaluate the following quantile sketches: KLL Sketch, Moments Sketch, DDSketch, UDDSketch, and ReqSketch. As discussed in Sec. 5.2, these five algorithms are amongst the strongest in one or more aspects of quantile computations for streaming data. KLL Sketch [26] extends Random to outperform many of the algorithms from Sec. 5.2 and has theoretical guarantees of optimality in terms of additive rank error [29]. However, no direct performance comparisons of KLL Sketch have been carried out against the other four algorithms evaluated in our study. In prior work, Moments Sketch has demonstrated superior merge and insertion speeds, and stronger accuracy guarantees than most other algorithms [21]. DDSketch [32] is a histogram-based algorithm and the first relative-error sketch with strong accuracy guarantees. Based on DDSketch, UDDSketch improves the algorithm to maintain the relative error guarantee over a fixed number of bucket collapses [18]. ReqSketch [14] is a recently proposed randomized algorithm that provides multiplicative rank error guarantees. To the best of our knowledge, the only direct performance-based comparison of ReqSketch is against t-digest [16]. All five sketches are mergeable, have low memory usage, and most of them represent different methods of maintaining data and computing quantiles.

3.1 KLL Sketch

Karnin et al. introduced KLL Sketch [27] that builds upon the revised version of the Random algorithm [30] [29]. Ivkin et al. [26] made a number of improvements to accuracy and space complexity of KLL Sketch and experimentally evaluated its performance. Finally, Zhao et al. introduced a mechanism to allow deletions [40].

The core structures behind KLL Sketch are *compactors* that have incrementally increasing height, h, starting from 0. KLL Sketch places elements into compactors and runs a *compaction* algorithm when it becomes full. Compaction sorts the data, randomly discards all even or odd elements, and moves the remaining elements to the compactor at the next height. The weight w of each element in the compactor, determined by the formula $w = 2^h$, indicates the number of entries represented by that element and also determines the size of the compactor. KLL sketch uses a sampler to represent the bottom compactors of size 2 to provide logically equivalent functionality with reduced space complexity.

Consider a simplified example of a sketch with 2 compactors. Assume the first compactor of height 0 has a maximum size of 10. If the compactor initially contains the first 9 values from Table 1 and then receives the last value of 51, this would result in a compaction. The compaction would discard all elements in odd indices and move the remaining elements to the compactor at the next height, and the elements have their weight implicitly increased from 2^0 to 2^1.

To compute the q-quantile, values in each compactor at height h are copied $w = 2^h$ times, sorted, and the value at $\lceil qN \rceil^{th}$ index is taken. The exact state of the KLL Sketch at this point is illustrated in Table 2 (top 2 rows under Compactors). When merging two sketches, it combines two compactors at the same height into one and compacting any level containing more than k_h elements, where k_h is based on the maximal height of the combined sketch.

KLL Sketch is one of the two algorithms in our study whose estimates of the query are actual values from the data set, with ReqSketch being the other. Due to its randomized nature, KLL Sketch will sometimes get the exact quantile value instead of an estimate and have zero error. KLL Sketch achieves ϵ rank error with high probability $1 - \delta$ and space complexity $O((1/\epsilon)\sqrt{\log(1/\epsilon)})$.

Table 2: State of two compactors of KLL Sketch after consuming the example data from Table 1

	Compactors									
h=0	< no elements >									
h=1	3		8		11		16		30	
	Query Calculation									
x	3	3	8	8	11	11	16	16	30	30
Rank(x)	1	2	3	4	5	6	7	8	9	10
Quantile^{-1}(x)	.1	.2	.3	.4	.5	.6	.7	.8	.9	1

3.2 Moments Sketch

Moments Sketch was introduced by Gan et al. [21] to improve the merge time of sketches representing data sets with a high number of unique elements. A minimum cardinality of 5 is required for this sketch or its underlying algorithm will fail.

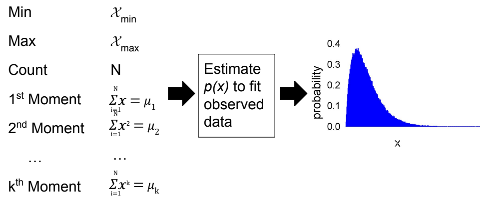


Figure 2: Visual representation of Moments Sketch steps (log moments are omitted for simplicity)

Moments Sketch holds the *min*, *max*, k *moments* and k *log moments* of the data set where k is some small number usually between 10 to 22. The nth moment of a data point x is simply x^n and the log moment is $\log^n(x)$. An `arcsin` transformation is recommended for large magnitude data since substantial exponents of a number can cause overflow errors. Moments Sketch selects the distribution $p(x)$ that fits the data and also maximizes its Shannon Entropy using the Method of Moments. The Method of Moments constructs a distribution $p(x)$ whose moments are the same as the data set [24]. The resulting distribution $p(x)$ can then be used to estimate the q-quantile by solving for x in the equation $q = \int_{-\infty}^{x} p(x)\,dx$. These steps are illustrated in Fig. 2. The merge operation involves simply adding together only the stored moments from the two sketches and recomputing the minimum and maximum as needed. Moments Sketch does not provide any rigorous error bound on a quantile query, and provides a bound on only the *average* quantile error. The average quantile error decreases as the number of moments held, k, increases.

3.3 DDSketch

DDSketch [31, 32] is a recently introduced histogram-based deterministic algorithm that provides relative error guarantees. The motivation for DDSketch according to its authors is the dissatisfaction with the rank error guarantees provided by the other algorithms.

In DDSketch, a bucket representing the histogram counts the number of elements observed in the stream between $(\gamma^{i-1}, \gamma^i]$, where $\gamma = (1+\alpha)/(1-\alpha)$ and α is the maximum relative error. A data element, x, is indexed to a bucket B_i by $\log_\phi(x) = i$. This indexing method allows the histogram to handle a wide range of values with a small number of buckets.

DDSketch computes a quantile based on the number of elements in each bucket. Starting from bucket B_0, the number of values in each bucket is summed up in order until it reaches the bth bucket such that $\sum_{i=0}^{b} B_i/N >= q$, where B_i is the number of values in bucket i. When $i = b$, this means the q-quantile lies in the ith bucket and the estimated quantile is $2\phi^i/(\phi+1)$. Since each bucket contains elements between $(\gamma^{i-1}, \gamma^i]$ the maximum error for the q-quantile estimate $\hat{x}_q = 2\phi^i/(\phi+1)$ can be $\frac{\hat{x}_q - \gamma^{i-1}}{\gamma^{i-1}}$ or $\frac{\gamma^i - \hat{x}_q}{\gamma^i}$. In both cases, you can substitute the value for γ and find that the error is less than α. These concepts are illustrated in Fig. 3.

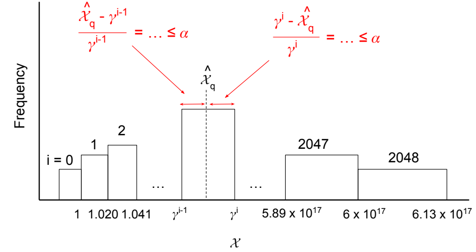


Figure 3: Visual representation of DDSketch with $\gamma = 1.0202$ and $\alpha = 0.01$

The number of buckets is usually not restricted since a small number of buckets can represent a large range of numbers with minimal space requirements. There is a variant of the algorithm that limits the number of buckets that can be used. In that case, if the sketch runs out of space due to a large data range, the buckets holding lower values will be merged, which would violate the accuracy guarantees of the lower quantiles. The merge operation combines buckets from the two sketches with a common γ by adding the counts of buckets in the same range. It can also trigger a range extension to align the bucket ranges of the two sketches being merged. The DDSketch implementation [31] evaluated in our study uses arrays to store bucket information.

3.4 UDDSketch

Uniform DDSketch or simply UDDSketch [18] was inspired by the design of DDSketch and retains the histogram-based approach to estimating quantiles. The main difference between UDDSketch and DDSketch lies in how bucket collapsing is done when the current range of the sketch cannot support an element being inserted into the sketch. As described in Sec. 3.3, the DDSketch variant that restricts the number of buckets would collapse the two lowest-indexed buckets in order to extend the range of the sketch. UDDSketch collapses all of the adjacent pairs of buckets instead of collapsing only the bucket pair with the lowest indices. More formally, UDDSketch replaces each of the buckets with indices i and $i+1$, where i is odd and at least one bucket is non-empty, with a new bucket of index $\lceil \frac{i}{2} \rceil$ having the sum of the count of the two buckets as its new count.

UDDSketch's uniform bucket collapsing algorithm leads to a uniform deterioration of the relative error guarantee in comparison to DDSketch (with a fixed number of buckets). UDDSketch's relative error guarantee α' is given by $\alpha' = 2\alpha/(1+\alpha^2)$, allowing the calculation of a deterministic error guarantee at any time based on the initial relative error threshold α_0 and the subsequent number of bucket collapsing operations k. More importantly, this allows us to reverse the deterministic error guarantee calculation to derive the initial relative error threshold for an estimated number of bucket collapses k using $\alpha_0 = \tanh(\operatorname{arctanh} \alpha_k / 2^{k-1})$,

where α_k is the required final relative error guarantee. Mirroring the original C implementation, our implementation of UDDSketch uses a map data structure as its bucket store.

UDDSketch's merging algorithm [9] iterates through the buckets of two sketches having a common γ, combines those within the same interval, and potentially performs a costly bucket collapsing operation at the end. Bucket ranges of the two sketches being merged align if they have the same γ value since UDDSketch collapses buckets uniformly.

3.5 ReqSketch

Relative Error Quantile Sketch or ReqSketch [14] is the most recent of the algorithms considered in our study. Similarly to KLL Sketch (Sec. 3.1), it retains a sample of the observed data in a sequence of compactors known as *relative-compactors.* ReqSketch inserts elements into a sketch starting from the relative-compactor at height 0. A relative-compactor at height h maintains a buffer of capacity B and executes a compaction operation whenever B is full, similar to KLL Sketch. ReqSketch considers only the L largest sorted items ($L \leq B/2$) in the buffer and randomly promotes the odd or even items to the compactor at height $h+1$ and discards the others from the considered L items, while retaining all of the smallest $B-L$ items in the buffer. Additionally, the number of items that are considered for compaction for a given relative-compactor changes based on the state of its compaction schedule. A compaction schedule is maintained so that larger items of a buffer are compacted more frequently and smaller items are compacted less frequently. ReqSketch's merge operation concatenates compactors of the same level and any exceeding its capacity are compacted similarly to KLL Sketch. The merged compaction schedule state is obtained by taking the bitwise OR of the schedule states of the two merging compactors.

ReqSketch provides a space complexity of only $O(\log^{1.5}(\epsilon n)/\epsilon)$ under practical assumptions while maintaining a multiplicative error guarantee of $|\hat{Rank}(x) - Rank(x)| \leq \epsilon\hat{Rank}(x)$ with high probability. Note that the multiplicative error, defined by the relation $|\hat{Rank}(x) - Rank(x)| \leq \epsilon\hat{Rank}(x)$, is also referred to as the relative error [14]. However, it is distinct from the relative error definition (Sec. 2.2) used in this study.

4 EVALUATION

The primary focus of this evaluation section will be on speed and accuracy. In line with the practical and empirical focus of this study, we also provide an analysis of data structure space requirements in Sec. 4.3 to focus on the implementation aspect of the sketches.

4.1 Data Sets

Our survey uses both synthetic and real-world data sets to evaluate performance. For the synthetic data sets, we generate data based on the Pareto and uniform distributions. The Pareto distribution helps to understand how well the algorithms perform on a distribution with a very long tail, as used in prior evaluations [18, 32]. The Uniform distribution helps to understand how well the algorithms perform on evenly spread out data.

Synthetic data is generally created using a function from a library that samples from a set distribution. Such a data set follows a nearly perfect distribution and does not have a strong resemblance to real-world data. Many real-world data sets would lie between the evenly spread uniform distribution and the heavy-tailed Pareto distribution in terms of skew. Using these two synthetic data sets allows us to obtain a general understanding of the behaviour of the algorithms across a range of data distributions that otherwise would not be possible. Additionally, we periodically sample the synthetic data generation parameters from normal distributions to make the synthetic data set resemble real-world data. The distribution parameters are updated every millisecond, resulting in samples drawn from a slightly modified distribution. The experiments were run with a Pareto distribution whose shape parameter α and scale parameter X_m were determined using a normal distribution $\mathcal{N}(1, 0.05)$. For the Uniform data set, the minimum is generated by an $\mathcal{N}(100, 25)$ distribution, and the maximum is generated by an $\mathcal{N}(1000, 100)$ distribution.

There are two real world data sets used for experimentation in our survey. The first is NYT data set which contains the trip fare information from the 2013 New York Taxi (NYT) data [12]. Second, the Power data set which contains global active power measurements from the UCI Individual Household Electric Power Consumption data [25]. The Power data set has been commonly used in empirical evaluations of quantile sketches in previous studies [18, 21, 32]. The PDFs of all the data sets can be seen in Fig. 4. We measure insertion and query times after consuming values sampled from a Pareto distribution ($\alpha = 1$ and $X_m = 1$). The sketches were populated from a uniform ($U(30, 100)$), binomial ($p = 0.2$ and $n = 100$) and Zipf (20 elements and an exponent of 0.6) distribution when evaluating merge times. Lastly, we use a synthetically generated data set from a uniform ($U(30, 100)$) and a discrete binomial ($p = 0.4$ and $n = 40$) distribution to execute the test of adaptability described in Sec. 4.5.7.

4.2 Methodology

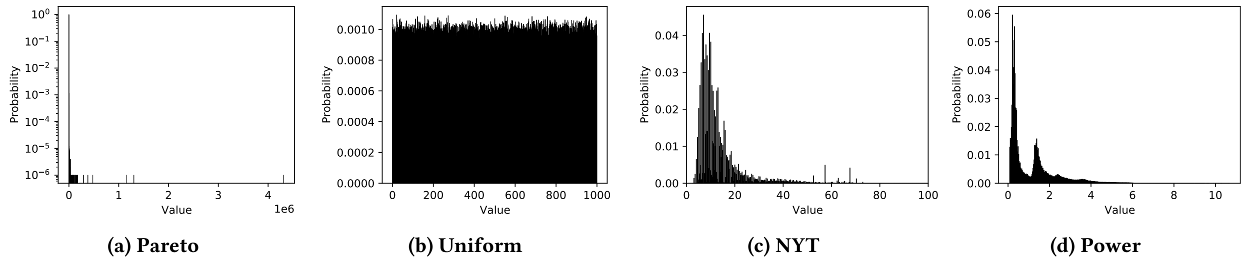
We evaluate the KLL Sketch and ReqSketch implementations from the Apache Data Sketches library [2]. The authors' implementation was for DDSketch [31] and Moments Sketch [22]. We provide an implementation of UDDSketch in Java since the authors' implementation is in C [18]. The parameters for DDSketch, UDDSketch, KLL Sketch, and ReqSketch in the experiments are chosen so they have a similar memory footprint and a rank or relative accuracy closer to 1% (0.01). In the case of Moments Sketch, the memory requirement is significantly less than that of the other algorithms, and its parameter *num_moments*[3] does not allow direct tuning of accuracy thresholds. We choose 0.01 or 1% as the target accuracy since this was the considered error threshold in [32].

In our experiments, KLL Sketch's *max_compactor_size*[4] was set to 350 and consequently the expected rank error is 0.97%. ReqSketch's *num_sections*[5] was set to 30, which is used to derive the number of elements L considered for compaction and the buffer size of a relative-compactor. We enable the high rank accuracy (HRA) parameter when using ReqSketch, which will compact from the smallest L elements of the buffer and not the largest L elements as described in Sec. 3.5. This parameter is enabled because it significantly reduces the relative error when estimating the more interesting upper quantiles. We use DDSketch with an

[3]For clarity, Moments Sketch's parameter k representing the number of moments is called *num_moments*.

[4]KLL Sketch's parameter k representing the maximum size of the highest level compactor is called *max_compactor_size* for clarity.

[5]ReqSketch's parameter k representing the number of sections considered for compaction is called *num_sections* for clarity.



(a) Pareto **(b) Uniform** **(c) NYT** **(d) Power**

Figure 4: Histogram representations of data sets used

unbounded dense store and the relative error parameter is set to $\alpha = 0.01$, resulting in $\gamma = 1.0202$. For UDDSketch, the maximum number of buckets was set to 1024 and the bucket collapsing threshold *num_collapses*[6] was set to 12. This initially results in $\alpha = 4.88x10^{-7}$ and reaches the threshold of $\alpha = 0.01$ after 11 bucket collapses. As noted in [21], we experienced numerical stability issues with anything more than 15 moments. Therefore, we keep 12 moments to ensure it is able to provide fairly accurate estimates without leading to instability. Per [22], we apply a log transformation to Pareto and Power data sets since these data sets span over many orders of magnitude of data.

The speed of the algorithms was measured for merge, insertion and query times. Insertion time is an important metric since a data set of size N will generate N insert operations for which speedy insertions into the sketch is required to avoid system bottlenecks in a streaming setting. Merge time was evaluated to ensure that the algorithms can support distributed queries at fast speeds. Lastly, query times indicate how quickly the result can be obtained from the sketch once it has collected all the data. The significance of this operation is application dependent – an application that is querying for a quantile every few seconds will demand faster query times than one that queries at a much slower rate, e.g., every minute. All of the experiments measuring the speed of the algorithms were run as single-threaded standalone Java applications to achieve performance isolation.

Algorithm accuracy was tested through several different data sets, as well as with and without late-arriving data. These were run on the Apache Flink streaming system with tumbling window operations that consume data at 50,000 events per second. The windows were of size 20 seconds, thereby containing about 1 million elements each, which is the data size we use for the experiments on kurtosis and adaptability. In a single run, the experiment was run for 220 seconds for a given algorithm and data set, and the results for the first tumbling window are discarded and the average accuracy of the remaining 10 windows is then computed to obtain the accuracy of a single run. All of the experiment results provided in our study are averaged over 10 independent runs. Error bars on graphs represent 95% confidence intervals around the means. All experiments were run on a single server machine with 2× Intel E5-2620v2 CPUs (with each processor containing 6 CPU cores/12 threads) and 32 GB of RAM on Ubuntu 20.04.

The following quantiles were queried: 0.05, 0.25, 0.5 (median), 0.75, 0.95, 0.98, and 0.99. A range of quantile computations is ideal since one quantile computation could produce vastly different results from another [30]. There is a bias towards quantiles 0.9 and higher since those quantiles are typically of greater interest.

[6] UDDSketch's parameter k representing the number of bucket collapses is called *num_collapses* for clarity.

Table 3: Final memory usage of each sketch (in KB) after consuming 1 million data points of data set

	REQ	KLL	UDDS	DDS	Moments
Pareto	16.99	4.24	27.96	5.42	0.14
Uniform	16.99	4.24	20.9	1.84	0.14
NYT	17	4.24	22.53	2.15	0.14
Power	17	4.24	22.61	2.04	0.14

For example, a typical web request monitoring application can indicate an increase in web response time from 2 to 20 seconds for a 0.01 quantile difference at around the 0.99^{th} quantile [32]. This can indicate a serious service disruption affecting a limited number of users, and such insights can be derived only from accurate estimation of upper quantiles. We group the results into *mid* and *upper* quantiles, where mid consists of the quantiles 0.05, 0.25, 0.5, 0.75, 0.9 and upper are the 0.95^{th} and 0.98^{th} quantiles. The tail end of a distribution, where the 0.98^{th} and sometimes the 0.95^{th} quantiles reside, experience a larger spread and variation for skewed data, which is why they are shown as a separate category of their own. The accuracy for 0.99^{th} quantile is reported separately since it is of interest in many practical use cases.

4.3 Data Structure Analysis

Quantile sketching algorithms do not need much memory because they aggregate data to create very small summaries of them. This is evident by the observed memory usage in Table 3 for each of the algorithms with parameter values as described in Sec. 4.2 and consuming 1 million entries from a given dataset. All of the algorithms consume less than 0.03 MB in our experiments. Additionally, we provide an analysis of the space requirements by the numerical size of each of the sketches. This is possible since implementations, e.g., using data structures and pointers, of the five sketches do not add much data complexity.

DDsketch with an unbounded dense store would initially create a count array of 64 buckets, and expand the array based on the range of the values observed. Thus, the number of buckets maintained by DDSketch is independent of the data size, and depends on the distribution and the range of the data observed. In most of our experiments, the final number of buckets created remained below 1024 for a relative error threshold of 0.01. DDSketch needed a maximum of about only 120 buckets for the Power data set (Fig. 4d) whose range is $[0, 11]$ and about 670 buckets for the Pareto data set (Fig. 4a) with a range of approximately $[0, 6 \times 10^6]$. A sketch of 1024 buckets needs an array of only 1024 elements to maintain the bucket count and variables such

as index offset, minimum index, and maximum index to maintain state while still being able to accept values in the range of $[1, 7.69 \times 10^8]$ when $\alpha = 0.01$. We also evaluated DDSketch with a collapsing dense store of 1024 buckets for the same α value, but did not observe any significant difference in space requirements largely due to the fact that 1024 buckets can already support a large range of values in practice.

In the case of UDDSketch, the actual space requirements are higher than that of DDSketch because we used a map-based implementation that maintained a map index, bucket index, and a bucket count for each bucket. However, this implementation of UDDSketch would still maintain less than 3100 numbers for a bucket size of 1024. Note that similar to DDSketch, the actual number of buckets needed depends on the data distribution and range for UDDSketch. For example, we observed that UDDSketch utilized 981 buckets after inserting 1 million data points sampled from the Pareto distribution when $\alpha_k = 0.01$ and $num_collapses = 12$. KLL Sketch's implementation has a total sample size of 1048 across all compactors when $max_compactor_size = 350$ for a stream size of 1 million data points. Similarly, ReqSketch retained 4,177 items for $num_sections = 30$ after inserting 1 million data points sampled from the Pareto distribution. The Moments Sketch implementation [22] used in our experiments keeps only standard moments and avoids maintaining log moments, resulting in a sketch summary that stores less than 20 numbers[7] when $num_moments = 12$.

4.4 Speed

This section presents the experimental results obtained for the evaluated run times of insertion, query, and merge operations that affect a quantile sketch's overall performance.

4.4.1 Insertion Speed. Fig. 5a shows the average run time of adding a single element from a Pareto distribution to each sketch. All of the considered algorithms lie within the same order of magnitude and have insertion times of less than 0.2 μs. The insertion time is independent of the current data size of the sketch. Moments Sketch updates each of the *num_moments* moments and potentially the minimum and maximum values when an element is observed. DDSketch and UDDSketch both simply derive the index for a bucket that a particular element falls into and updates the bucket counter. However, we use DDSketch with an unbounded store that does not collapse buckets, while UDDSketch's insert operation has the potential to trigger a uniform bucket collapse. A uniform bucket collapse is a costly operation which has a worst-case time complexity of $O(m)$ where m is the maximum number of buckets. DDSketch uses an optimized array-based implementation to keep track of the buckets, whereas UDDSketch's unoptimized map-based implementation can have higher access and update times. These factors result in UDDSketch having the worst insertion time of all the considered algorithms. Both KLL Sketch and ReqSketch insertions consist of appending the current element to the compactor at height 0, which can potentially trigger one or more compaction operations. However, ReqSketch compaction involves promoting specific items of a compactor based on a compaction schedule, and uses a list-based implementation as opposed to the optimized array-based implementation used in KLL Sketch. This results in higher insertion times for ReqSketch.

[7] A double-precision floating point number would consume 8 bytes.

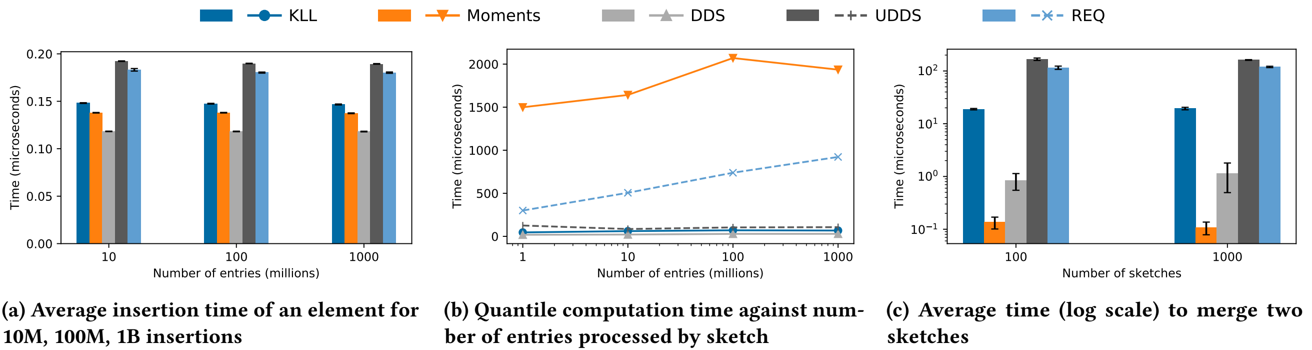
4.4.2 Quantile Query Times. Fig. 5b shows the average run time measurements for computing a quantile using each of the different sketching algorithms as a function of the data size. We conduct this experiment by filling each of the sketches up to the specified data size from a pre-sampled Pareto distribution and measuring the query time for estimating the quantiles as specified in Sec. 4.2.

In Fig. 5b, we observe that Moments Sketch has the worst query time in contrast to the merge time results, even though it is able to maintain an average query time of less than 2 ms for most data sizes. Quantile estimation in Moments Sketch involves solving the unconstrained convex optimization problem [21] using the maintained *num_moments* moments, which is independent of the data size and is computation-intensive compared to other sketches. This is reflected by the inconsistently varying and large average query times of Moments Sketch for different data sizes. The query time for both DDSketch and UDDSketch depends on the number of non-empty buckets. The maximum number of buckets that the universe of data can fill in DDSketch for our pre-sampled Pareto distribution is realized at around 100 million, and therefore the query time stabilizes thereafter. In the case of UDDSketch, the number of buckets depends additionally on the number of bucket collapses. We observed the number of buckets to be 990 with 10 collapses for a sketch of size 1 million. This changed to 641 buckets with 11 collapses for data sizes of 10 million or higher. Therefore, the higher number of buckets for a data size of 1 million leads to a comparatively larger query time for the smaller data size in the case of UDDSketch.

KLL Sketch populates and sorts the data so that it can perform a binary search to find the approximate or actual value of the relevant q-quantile. The maintenance of a fully packed shared array that is sorted except at the zeroth level compactor leads to fast data population times, and in turn faster query results. ReqSketch iterates through the compactors to populate a sorted auxiliary data structure and uses binary search to estimate a given quantile. In contrast to KLL Sketch, the higher level compactors are sorted at the time of populating the sorted sketch data. Therefore, as larger data sizes result in a larger number of samples being retained, which in turn results in a larger number of compactors that need to be iterated through and sorted, the query time also increases sub-linearly for ReqSketch.

Our analysis shows that of the considered sketches, the ones that utilize a summary to estimate quantiles have query times independent of data size and dependent on the distribution of data, while sketches that retain a sample for estimation are generally dependent on the data size.

4.4.3 Merge Times. Fig. 5c shows the average time to merge two sketches when merging 100 and 1000 sketches for each of the algorithms. Each individual sketch consumed 1 million events sampled from either a uniform, binomial or Zipf distribution before being merged. Overall, the merge times are relatively low when considering that each of the sketches represent 1 million data points. One of the contributing factors for merge time is the actual size of the sketch, and the low memory footprint of all the algorithms leads to a comparatively small time of less than 1 ms ($1000\mu s$) for merging two sketches. This is reflected in how the merge times roughly correlate with the space requirements of each of the algorithms. The other contributing factor that differentiates merge times is the particular algorithm used for merging, and how effectively it uses the structural characteristics of the sketch to perform a merge operation.



(a) Average insertion time of an element for 10M, 100M, 1B insertions

(b) Quantile computation time against number of entries processed by sketch

(c) Average time (log scale) to merge two sketches

Figure 5: Algorithm Operation Speed Evaluation

Moments Sketch's merge operation only needs to add together the 12 moments from the two sketches and recompute the minimum and maximum as needed, which allows it to have the fastest merge time by at least an order of magnitude. DDSketch's merging algorithm combines buckets from the two sketches by adding the counts of buckets in the same range. It can also trigger a range extension to align the bucket ranges of the two sketches being merged. The combining of a few hundred buckets and potentially adjusting bucket ranges is the reason for its slower performance compared to Moments Sketch. The average merge times of Moments Sketch and DDSketch are numerically smaller leading to higher variance. The difference in average merge times between 100 and 1000 sketches, even though noticeable, is not significant as indicated by the overlapping error bars. UDDSketch's merging algorithm [9] has to iterate through the buckets of each sketch to combine counts, and potentially perform a costly bucket collapsing operation at the end. Its map based implementation can have higher overheads when iterating over the data compared to an array-based bucket store such as that of DDSketch, leading to significantly worse merge times.

KLL Sketch has to combine compactors at the same height, and perform compaction if the size exceeds the new capacity limit of the combined sketch. The process of combining and potentially executing multiple compaction operations results in significant processing time and explains why KLL Sketch has a relatively high merge time. ReqSketch has a more involved compaction procedure compared to KLL Sketch, where the index of the elements being promoted on compaction changes each time based on the compaction schedule. This results in higher compaction times, and in turn higher merge times than KLL Sketch. Overall, sketches such as KLL Sketch and ReqSketch that retain a sample of the observed data have higher merge times in general compared to the sketches that maintain a summary of the data, such as Moments Sketch and DDSketch, with UDDSketch being the only algorithm that bucks this trend.

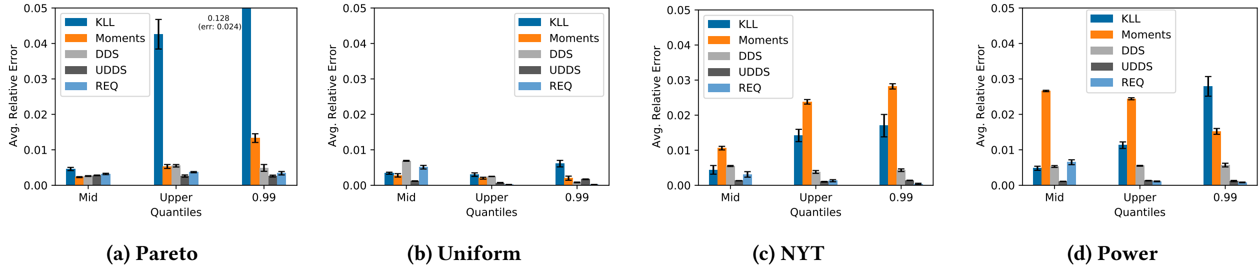
4.5 Accuracy

We present results for the important metric of estimation accuracy of the sketching algorithms in this section.

4.5.1 Pareto Data Set. Fig. 6a shows the accuracy of the five sketches processing the extremely long-tailed Pareto distribution from Fig. 4a. KLL Sketch's accuracy suffers in the upper quantiles due to highly scattered data at the tail, particularly for the 0.99^{th} quantile. Recall that KLL Sketch's compaction algorithm sorts and randomly discards either odd or even elements, promoting the remaining items to the next level. Therefore, it outputs the next closest available element as an approximation if the exact value was discarded during compaction. For example, if the actual value of the 0.99^{th} quantile was discarded during compaction, depending on the actual rank of the retained samples, KLL Sketch may output the value of the 0.991^{th} quantile. A data set derived from a heavy-tailed distribution such as Pareto would have a larger difference between values of neighboring ranks if sampled from the tail, which would result in a large relative error even though the rank error is minimal. Conversely, the 0.5^{th} and 0.501^{th} quantiles may actually be good approximations for one another even when the data is from a heavy-tailed distribution, since the difference in value between neighboring ranks is much smaller in the non-tail region (Fig. 1 visually illustrates this concept).

ReqSketch discards elements from its buffers when its capacity is exceeded. However, the elements that are discarded are selected from the $B/2$ smallest items in the buffer (of size B) when high rank accuracy is enabled. Therefore, ReqSketch performs well for the upper quantiles because there is higher probability for the larger values to have been retained by the sketch. It performs reasonably well even for the lower quantiles since similar lower values would be sampled with higher probability for the Pareto distribution, leading to accurate approximations even when the actual value has not been retained by the sketch. DDSketch and UDDSketch achieve high accuracy since their bucket ranges are derived based on the relative accuracy thresholds specified. Moments Sketch also performs well since the data is sampled from a probability distribution, which it can accurately approximate using the method of moments.

4.5.2 Uniform Data Set. All algorithms were able to achieve an average relative error below the threshold of 0.01 for the Uniform data set as shown in Fig. 6b. Even with the added parameter randomization, the uniform distribution is predictable and has a kurtosis close to 0. Therefore, all of the considered algorithms provide accurate estimates most of the time. ReqSketch's and DDSketch's mid-quantile relative error for the Uniform data set is higher than that of the Pareto data set. This is because the density of the Pareto distribution is much higher for the lower and mid quantile ranges, resulting in many similar values being sampled in that quantile range compared to the uniform distribution. The extremely high accuracy of ReqSketch in estimating the upper quantiles of the uniform distribution is due to a combination of higher retention of larger values by the sketch and better approximations for upper quantiles (when the actual value



(a) Pareto **(b) Uniform** **(c) NYT** **(d) Power**

Figure 6: Accuracy evaluations of each algorithm against data sets

is not available) due to the uniformity of the variance. The main takeaway from Fig. 6b is that all five algorithms perform very well against uniformly varying data in general.

4.5.3 NYT Data Set. Fig. 6c shows the accuracy for this real-world data set used in our evaluation. Moments Sketch computes quantiles from an estimated probability distribution, which introduces a quantile estimation error whenever the real world dataset deviates from the estimated distribution given that a limited number of moments and a limited grid size is used in its estimate. Even for the NYT data set, UDDSketch has the best accuracy overall since there is a theoretical guarantee on the relative error [18] for a given number of bucket collapses. DDSketch also provides the same theoretical guarantee when the unbounded version of the algorithm is used [32], which is reflected in its consistent average error for both mid and upper quantiles.

The NYT data set has a high number of frequently occurring data points in comparison to other data sets. The top 10 most frequently occurring data points in the Power data account for about approximately 4.5% of the total 2 million data points. The Uniform and Pareto data sets would have even fewer frequently occurring data points. In contrast, the top 10 most frequently occurring data points in NYT data set account for approximately 31.2% of the total 14.7 million data points. For algorithms that retain a sample of the observed data such as KLL Sketch and ReqSketch, this would mean that its compactors contain representations of a small number of elements which occur many times. On all of the runs, the estimates for the 0.25 quantiles were precise, consisting of 6.5, 7.5, 8.0, and 9.0, with each value repeated over $200,000$ times in the data set. As a result, KLL Sketch would have a high probability of keeping the exact element for the 0.25 quantile even after multiple compactions. KLL Sketch achieves a similarly high accuracy for 0.05 and 0.5 quantiles, but its mid quantile accuracy gets affected by the errors in other quantiles that do not come from an area of the distribution where there are many repeating values close together. ReqSketch also benefits from the repeated values similar to KLL Sketch. However, since ReqSketch is biased towards discarding smaller values and retaining larger values when high rank accuracy is enabled, a higher accuracy is achieved by ReqSketch for upper quantiles and the 0.99^{th} quantile, while the difference in mid quantile accuracy is not statistically significant when compared to KLL Sketch.

4.5.4 Power Data Set. The real-world Power data set has a bimodal shape as seen in Fig. 4, for which Moments Sketch is not able to accurately estimate a probability distribution. The mid quantiles are between the humps, which is where Moments Sketch sees an increase in error. The distribution becomes more predictable towards the tail end, which is why the error drops down slightly in the upper quantiles and more profoundly in the 0.99^{th} quantile for Moments Sketch. UDDSketch and DDSketch both exceed the relative error guarantee thresholds and perform extremely well in terms of accuracy for all evaluated quantiles. KLL Sketch performs relatively well for mid quantiles where the number of repeated data elements would be high for the power data set, but performs poorly for quantiles that lie on the long tail of the distribution, as was the case for Pareto and NYT data sets. ReqSketch has the best accuracy out of all the algorithms for estimating the upper and 0.99 quantiles and a comparatively worse accuracy for the mid quantiles for the power data set due to its high rank accuracy setting (as discussed in Sec. 4.2).

4.5.5 Overall Accuracy. UDDSketch consistently gives the best relative error across all data sets. However, it sets an exponentially smaller initial relative error threshold in anticipation of future bucket collapses. If the specified number of bucket collapses are not met, the actual relative error threshold ends up being much lower than the desired relative error threshold. UDDSketch maintained sketches at 0.005, 0.0006, 0.0025, and 0.0025 relative error thresholds for the Pareto, Uniform, NYT, and Power data sets respectively. This is the reason for UDDSketch's accuracy exceeding the specified threshold when compared to DDSketch. Even though DDSketch's results are reported for an unbounded bucket store, the error difference was 0.14% on average for mid quantiles and 0.07% for the upper quantiles when compared to DDSketch with a collapsing dense store of 1024 buckets.

Moments Sketch is able to provide estimates below the 0.01 relative error threshold for both mid and upper quantiles when the data set is synthetic but not for the two real-world data sets. Since Moments Sketch is estimating a probability distribution, any data points that deviate from the expected distribution in a real-world data set introduce quantile estimation errors. Additionally, the accuracy can be increased at the cost of increased query time by increasing the grid size parameter for the moments solver. We used the default parameters (per [22]) across all our experiments.

KLL Sketch performs comparatively well in terms of accuracy when the data distribution is close to uniform or when the quantile estimates are from a region where the probability density is higher in the data distribution. This is due to the sampling approach that randomly discards either the odd or even sorted elements from a level when compaction occurs in KLL Sketch. In ReqSketch, the sampling approach can be biased towards lower or higher valued elements based on the HRA parameter, which allows us to obtain very accurate upper-quantile estimates by sacrificing lower-quantile accuracy and vice versa.

4.5.6 Kurtosis. Fig. 7 shows how the accuracy at the 0.98^{th} quantile varies as the kurtosis of the data increases. It shows a

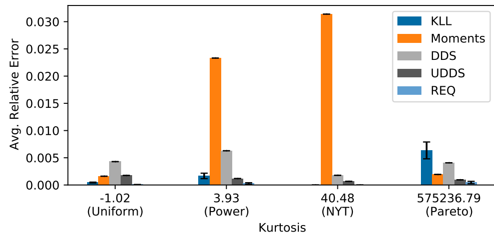
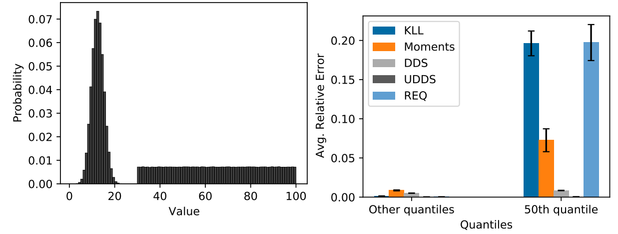


Figure 7: Accuracy of 0.98^{th} quantile query as a function of kurtosis



(a) Histogram of the evaluated data set

(b) Accuracy evaluations

Figure 8: Histogram representation of data set and accuracy results of adaptability evaluation experiment

general decrease in accuracy for algorithms that depend on the data distribution as the 98th quantile becomes more difficult to estimate accurately for such algorithms when there is increased positive skew. The first data point along the x-axis in Fig. 7 is the uniform distribution which does not have a tail, meaning the probability of occurrence of events around the 0.98^{th} quantile is as good as around any other point such as the median. Hence, the low error across all algorithms for the uniform data set. As previously mentioned, DDSketch's relative error is not affected by the data distribution and therefore the algorithm is able to maintain a stable error under 0.01×10^{-2} for all data sets. UDDSketch is able to similarly maintain a stable error as long as the number of bucket collapses do not exceed the threshold *num_collapses*. Conforming to the trend that was observed in Sec. 4.5, it becomes difficult for Moments Sketch to estimate the value for real-world datasets that deviate from a probability distribution that can be easily approximated. However, it provides reasonably accurate estimates for synthetic data sets, even when the kurtosis is high. ReqSketch and KLL Sketch are able to estimate the 0.98^{th}-quantile precisely since the actual value of 57.3 for the quantile is repeated more than 4,000 times in a sample of 1 million data points from the NYT dataset, significantly increasing the probability of that value being retained. ReqSketch performs better than KLL Sketch for the Pareto data set due to the biased sampling of ReqSketch that allows it to retain more values from its long tail.

4.5.7 Adaptability. This experiment is based on a similar experiment [40] and measures the accuracy of the algorithms on a data set whose distribution changes halfway through. The first half is one million data points of a discrete binomial distribution with parameters $n = 30$ and $p = 0.4$, and subsequently 1 million data points from a uniform distribution with minimum of 30 and maximum of 100. Fig. 8a shows the distribution of data which is essentially fragmented into 2 sections for the 2 distributions.

The results of the adaptability experiment are shown in Fig. 8b. The results are insignificant for most of the algorithms except at the 0.5^{th} quantile where there is a jump in error for most of the algorithms. This is because the 0.5^{th} quantile lies at the exact end of the binomial section as seen in Fig. 8a. It is difficult to definitively point to the cause of the estimation error in Moments Sketch by back-tracing its multiple approximation techniques from the limited information retained by the algorithm. However, we surmise that its lack of accuracy in estimating data sets with small number of discrete values [21] combined with the fact the estimated PDF using moments would not be able to account for the sudden change in the distribution, would lead to higher estimation errors around the 0.5^{th} quantile. Due to compaction and the large weights assigned to the element representing the 0.5^{th} quantile, both KLL Sketch and ReqSketch discard the value at 0.5^{th} quantile with high probability. Instead, they output a retained element from the beginning of the uniform section or a preceding value from the binomial section as an approximation. The significant jump from the largest binomial value to the smallest uniform value therefore results in a large relative error for KLL Sketch and ReqSketch. DDSketch and UDDSketch are not affected by the data distribution, hence their accuracy at the 0.5^{th} quantile remains stable.

4.6 Late-Arriving Data

Late-arriving data is common in streaming windowed operations because network delays can cause events to arrive after the window has already run its query [19, 20, 37]. Hence, we ran the same experiments as in Sec. 4.5 but with late-arriving data dropped by the stream processing engine. We emulate late arrivals by applying an offset from an exponential distribution with 150 ms as the mean network delay to the timestamp at event generation. With this mean being from an exponential distribution, the tail is long and a large majority of events have a considerably high latency, allowing us to understand the accuracy behaviour of the algorithms with a high number of dropped late-arrivers. We observed about a 2% loss of events from a single window on average when events were generated with these tail-latency inducing settings. However, a sketch with an accurate summary of the data is not affected significantly by missing a small percentage of data. Overall, the error was slightly higher for the late-arriving data experiments when compared against the results of Fig. 6. However, the core analysis remains the same as in Sec. 4.5.

4.7 Sensitivity Analysis of Accuracy on Window Size

We ran the accuracy experiments on Flink for different window sizes of 5 s, 10 s, and 20 s to understand if the window size has an impact on accuracy. Overall, we observed that the average relative error was consistent for synthetic data sets, but had some variation within real-world data sets. For the NYT and Power data sets, Moments Sketch's average relative error decreased by 0.0018 overall when the window size increased from 5 s to 20 s, while it increased by 0.0007 and 0.0014 on average for KLL Sketch and ReqSketch respectively. Moments Sketch can approximate the distribution better for larger window sizes since the shape of the observed data is more likely to become smoother and match the correct probability distribution for a given grid of the sketch. For KLL Sketch and ReqSketch, larger window sizes result in a higher number of compactions, increasing the probability of the

Table 4: Each algorithm is evaluated on different characteristics relative to the other algorithms

Characteristic	KLL Sketch	Moments	DDSketch	UDDSketch	ReqSketch (HRA)
Sketching approach	Sampling	Summary	Summary	Summary	Sampling
High Tail Accuracy	Non-Skewed	Synthetic	All	All	All
High Non-Tail Accuracy	All	Synthetic	All	All	All
Insertion Speed	Medium	Medium	High	Low	Low
Query Speed	High	Low	High	High	Medium
Merge Speed	Medium	High	Medium	Low	Low
Adaptability	Inconsistent	Low	High	High	Inconsistent

actual and nearby values of a given quantile to be discarded. The average relative error for DDSketch and UDDSketch were fairly consistent across window sizes and did not show any overall trend. We report the accuracy for 20 s windows with 1 million data points since this represents the distribution of all of the considered data sets robustly.

4.8 Summary of Results

DDSketch and UDDSketch are deterministic algorithms that provide relative error guarantees irrespective of the data distribution of the stream. Their accuracy can be affected by the data range instead. However, both can handle data values up to 6.13 x 10^{17} using just 2048 contiguous positive-indexed buckets while guaranteeing a relative error of 0.01. DDSketch's superior insert, query, and merge times would make it an ideal choice if overall runtime performance is a concern, while UDDSketch can provide better accuracy. Both the algorithms have high adaptability. Moment's sketch, while having weak accuracy in the face of real-world data, has superior merge times. KLL Sketch provides reasonable runtime performance, and is suitable when the data is not skewed. However, its accuracy falls when processing data distributions with large spread in values. This is because KLL Sketch is a randomized algorithm that retains a sample of the observed data, and the element from the sample that is returned as an estimate will have a larger error when the data spread is higher. ReqSketch's selective compaction procedure allows it to have extremely accurate upper or lower quantile estimates, making it an ideal candidate when a relevant subset of the quantiles needs to be estimated, given that its comparatively weaker runtime speeds are not a concern.

Table 4 represents a summary of our evaluations. Each algorithm is evaluated on a range of characteristics, based on how well it performed relative to the other algorithms. We classify the *Sketching Approach* of each algorithm as *Summary* or *Sampling* depending on whether it utilizes a statistical summary or retains samples to estimate quantiles. *High Tail Accuracy* indicates whether the algorithm was able achieve good tail accuracy on all data sets or only on data sets with specific characteristics. *High Non-tail Accuracy* has a similar definition. Tail accuracy considers the upper end of the distribution that includes the popularly queried 95th and 98th percentiles. Non-tail accuracy considers the other end of the distribution. *Insertion*, *Query* and *Merge Speed* were evaluated as having a *Low*, *Medium*, or *High* value by making an overall comparison. *Space* requirements are not included in our comparison since the space requirement is extremely small compared to the actual data size for all of the algorithms. *Adaptability* is evaluated to be *Low* or *High* based on the results from Sec. 4.5.7, and marked as *Inconsistent* when having good adaptability overall except for the 50th percentile.

5 RELATED WORK

This section examines prior work under two criteria. Sec. 5.1 examines prior studies of quantile streaming sketch evaluations and Sec. 5.2 examines quantile sketches that are related to the ones we evaluate.

5.1 Related Studies

An experimental study published by Luo et al. [29] is the work most related to this paper. Their study extends their earlier work by introducing GKArray [29], an improved implementation over GKAdaptive [39], which is a variant of the earlier GK algorithm [23]. This study predates all of the five algorithms we consider, and hence does not have a comparison of the more recent quantile sketches that provide better performance and accuracy covered in this paper. Additionally, we evaluate the accuracy of the algorithms on Apache Flink, an industrial-strength streaming system, using real-world data. Our study demonstrates the feasibility of using these sketches without significant re-engineering in a modern, multi-threaded streaming environment. It also evaluates the impact on accuracy in the face of late-arriving data, a common scenario in many stream processing deployments.

Luo et al.'s study [29] classifies sketching algorithms as cash register algorithms and turnstile algorithms. Cash register algorithms allow elements to only be inserted into the sketch, whereas turnstile algorithms allow both insertions and deletions. Turnstile algorithms require more space to achieve accuracy comparable to cash register algorithms, and have worse performance [29]. We evaluate only cash register algorithms and do not consider turnstile algorithms since the main criteria for selecting an algorithm to evaluate were accuracy and performance.

Chen and Zhan [11] survey some quantile sketches and analyze their behaviour with a focus on space complexity, update time, and accuracy. Their study does not include an experimental analysis and most of the algorithms considered predate the ones evaluated in our study. Another recent study by Mitchell et al. [33] conducts in-depth experiments and provides an empirical analysis of how different lightweight moment estimators affect accuracy and sketch time of quantile approximations using Moments Sketch [21]. However, their evaluations compare only against KLL Sketch [27] and includes two-pass estimators, which makes it difficult to compare against results obtained in a streaming setting.

Studies introducing or improving a sketching algorithm generally provide performance comparisons against other quantile sketches, but with each study relying on a platform or an implementation different to that of the other. This leads to discrepancies when comparing performance. For example, Moments Sketch was implemented in Java while the implementations of KLL Sketch [26] [40] used Python. Similarly, the comparison

between UDDSketch and DDSketch was implemented in C [18] while the comparison of DDSketch and Moments Sketch was done through Java implementations [32]. Other gaps include differences in testing methods such as how KLL Sketch was tested for adaptability by changing the distribution of the data [40]; this experiment was not performed in other papers. Our study uses uniform experimental settings to evaluate the key metrics of accuracy, speed, space requirements, and adaptability.

5.2 Related Sketching Algorithms

This section examines prior work on quantile streaming [5, 10, 29, 30] algorithms aside from the five algorithms that we have described and evaluated. Thus, this section provides context for why KLL Sketch, ReqSketch, Moments Sketch, DDSketch, and UDDSketch are the algorithms chosen in our evaluation.

5.2.1 Random. This algorithm's roots trace back to a sketching algorithm by Manku et al. [30]. It maintains a reservoir sample of all the events seen in the stream. A buffer maintains the elements that are sampled. The buffer is collapsed by simply discarding half of the elements when the buffer is full. The collapse function increases the weight of the remaining elements by a factor of 2. Conceptually, a query is answered by copying each element w times and then sorting to compute the quantile. Associating the weight w to the element is sufficient without keeping multiple copies. This method provides an ϵ-approximate q-quantile with high probability $1 - \delta$ where ϵ is the rank error. This algorithm, with subsequent upgrades, was one of the best performing algorithms (per [29]). Random's space and accuracy guarantees were further improved in KLL Sketch [27] [26] that we evaluate.

5.2.2 Histograms. Histograms are a useful and intuitive way to summarize and approximate a data set's distribution function [5]. The HDR histogram is a modern histogram with fast insertion speeds, mergeability property and strong relative accuracy claims [38]. The HDR histogram performed comparably to DDSketch on accuracy and insertion speed but performed worse on merge speed and total sketch size [32]. Masson et al. [32] show that DDSketch is comparable or superior to the HDR histogram across different performance categories; thus, we do not evaluate HDR histogram in our study.

5.2.3 Dyadic Count Sketch. Dyadic Count Sketch (DCS) was evaluated to be the best turnstile algorithm [29]. DCS maintains $\log(u)$ dyadic levels in increasing order where the ith level has $u/2^i$ intervals of size 2^i, where u is the universe of possible elements in data set D. Count-Sketches [10] on each level track the number of elements in each interval. Knowing how many elements fall into each interval allows the algorithm to estimate a q-quantile query. A comparison of KLL Sketch against DCS showed KLL outperforms DCS in terms of memory usage, speed and accuracy [40]. Due to its larger memory footprint requiring prior knowledge of size, and being outperformed by KLL Sketch, DCS is not included in our evaluation.

5.2.4 t-digest. t-digest [17] clusters elements based on their value to form a set of weighted centroids. Each cluster keeps track of the number of elements consumed and their mean. t-digest's scale function k determines the size of each cluster, and allows a particular range of quantiles to have smaller clusters with higher accuracy while sacrificing accuracy of the larger clusters. t-digest does not provide a theoretical bound on its estimation error and its merging algorithm can weaken the accuracy of the original sketches. Even though t-digest had comparable accuracy for practical data sets when evaluated against ReqSketch [16] and KLL Sketch [3], its accuracy was shown to be worse against Moments Sketch [21]. Its update time is comparatively slower than KLL Sketch [3] and ReqSketch [16]. Additionally, given that t-digest cannot provide consistent accuracy guarantees across all quantiles, it did not warrant inclusion in our experiments.

6 CONCLUSION

Five recent quantile sketching algorithms, KLL Sketch, Moments Sketch, DDSketch, UDDSketch, and ReqSketch were evaluated and analyzed experimentally for speed and accuracy through implementations in Java and Apache Flink. The evaluations found consistent accuracy with DDSketch and UDDSketch due to their relative error gurantees. Moments Sketch has relatively weak accuracy against real-world data since it always relies on estimated probably distributions to compute a quantile. Since KLL Sketch relies on retaining a sample of the data and discarding the rest, it has high relative error when the data set is skewed or when it does not contain repeated values. ReqSketch mitigates this issue by sampling more selectively to favour either higher or lower ranked elements through its compaction algorithm, providing extremely accurate estimates for the upper or lower quantiles. All of the considered algorithms show consistent accuracy in the face of missing data due to late-arrivals. However, UDDSketch and DDSketch are the only algorithms that have consistent accuracy when the data distribution is drastically varying as shown in Sec. 4.5.7.

In terms of performance, DDSketch, UDDSketch, and KLL Sketch have the fastest query times while Moments Sketch's merge times are at least an order of magnitude faster than the other algorithms. In terms of insertion speed, DDSketch is the fastest algorithm while UDDSketch is the slowest. However, all of the algorithms are comparably fast with an average insertion time that is well below a microsecond.

Our evaluations suggest that DDSketch is an algorithm suitable for almost any application with very good runtime performance. UDDSketch provides better accuracy, but with worse runtime performance. However, KLL Sketch and ReqSketch can provide the exact value for a quantile query due its sampling-based nature, and their sample size can be increased to increase accuracy. If highly accurate estimates are required for upper or lower quantiles, ReqSketch is ideal, while KLL Sketch can provide better performance at the cost of accuracy. Moments Sketch would be a suitable candidate if the dominant factor in selecting an algorithm is its merge time.

Acknowledgements This project was supported with funding from the Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation (CFI) and Ontario Research Fund (ORF). We thank the anonymous reviewers for their constructive feedback.

REFERENCES

[1] [n.d.]. 1.3.5.11. Measures of Skewness and Kurtosis. https://www.itl.nist.gov/div898/handbook/eda/section3/eda35b.htm
[2] 2020. Apache Data Sketches. https://datasketches.apache.org/. Accessed: 2021-11-25.
[3] 2020. DataSketches | KLL sketch vs t-digest. https://datasketches.apache.org/docs/QuantilesStudies/KllSketchVsTDigest.html
[4] 2020. DataSketches | Sketch Origins. https://datasketches.apache.org/docs/Background/SketchOrigins.html

[5] Jayadev Acharya, Ilias Diakonikolas, Chinmay Hegde, Jerry Zheng Li, and Ludwig Schmidt. 2015. Fast and Near-Optimal Algorithms for Approximating Distributions by Histograms. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Melbourne, Victoria, Australia) *(PODS '15)*. Association for Computing Machinery, New York, NY, USA, 249–263. https://doi.org/10.1145/2745754.2745772

[6] 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* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/2465351.2465355

[7] Tyler Akidau. 2015. Streaming 101: The world beyond batch. https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/

[8] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, Robert Endre Tarjan, et al. 1973. Time bounds for selection. *J. Comput. Syst. Sci.* 7, 4 (1973), 448–461.

[9] Massimo Cafaro, Catiuscia Melle, Italo Epicoco, and Marco Pulimeno. 2021. Data stream fusion for accurate quantile tracking and analysis. *arXiv preprint arXiv:2101.06758* (2021).

[10] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Automata, Languages and Programming*, Peter Widmayer, Stephan Eidenbenz, Francisco Triguero, Rafael Morales, Ricardo Conejo, and Matthew Hennessy (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 693–703.

[11] Zhiwei Chen and Aoqian Zhang. 2020. A Survey of Approximate Quantile Computation on Large-Scale Data. *IEEE Access* 8 (2020), 34585–34597. https://doi.org/10.1109/ACCESS.2020.2974919

[12] NYC Taxi and Limousine Commission. 2014. NYC Taxi Trip Data 2013 (FOIA/FOIL). https://archive.org/details/nycTaxiTripData2013

[13] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2011. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends® in Databases* 4, 1–3 (2011), 1–294. https://doi.org/10.1561/1900000004

[14] Graham Cormode, Zohar Karnin, Edo Liberty, Justin Thaler, and Pavel Veselý. 2021. Relative Error Streaming Quantiles. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, Virtual Event China, 96–108. https://doi.org/10.1145/3452021.3458323

[15] G. Cormode, Korn, and Muthukrishnan. 2004. Holistic UDAFs at streaming speeds. In *SIGMOD*. ACM, 35–46.

[16] Graham Cormode, Abhinav Mishra, Joseph Ross, and Pavel Veselý. 2021. Theory meets Practice at the Median: A Worst Case Comparison of Relative Error Quantile Algorithms. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. ACM, Virtual Event Singapore, 2722–2731. https://doi.org/10.1145/3447548.3467152

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

[18] Italo Epicoco, Catiuscia Melle, Massimo Cafaro, Marco Pulimeno, and Giuseppe Morleo. 2020. UDDSketch: Accurate Tracking of Quantiles in Data Streams. *IEEE Access* 8 (2020), 147604–147617. https://doi.org/10.1109/ACCESS.2020.3015599

[19] O. Farhat, H. Bindra, and K. Daudjee. 2020. Leaving Stragglers at the Window: Low-Latency Stream Sampling with Accuracy Guarantees. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/3401025.3401732

[20] Omar Farhat, Khuzaima Daudjee, and Leonardo Querzoni. 2021. *Klink: Progress-Aware Scheduling for Streaming Data Systems.* Association for Computing Machinery, New York, NY, USA, 485–498. https://doi.org/10.1145/3448016.3452794

[21] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-Based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *Proc. VLDB Endow.* 11, 11, 1647–1660. https://doi.org/10.14778/3236187.3236212

[22] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2019. Moments Sketch. https://github.com/stanford-futuredata/momentsketch original-date: 2018-09-04T16:50:51Z.

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

[24] Alastair R Hall et al. 2005. *Generalized method of moments.* Oxford university press.

[25] Georges Hebrail and Alice Berard. 2012. UCI Machine Learning Repository: Individual household electric power consumption Data Set. https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption

[26] Nikita Ivkin, Edo Liberty, Kevin Lang, Zohar Karnin, and Vladimir Braverman. 2019. Streaming quantiles algorithms with small space and update time. *arXiv preprint arXiv:1907.00236* (2019).

[27] 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, New Brunswick, NJ, USA, 71–78. https://doi.org/10.1109/FOCS.2016.17

[28] X. Lin, J. Xu, Q. Zhang, Hongjun Lu, Jeffrey Xu Yu, X. Zhou, and Y. Yuan. 2006. Approximate processing of massive continuous quantile queries over high-speed data streams. *IEEE Transactions on Knowledge and Data Engineering* 18, 5 (2006), 683–698. https://doi.org/10.1109/TKDE.2006.73

[29] G. Luo, L. Wang, K. Yi, and G. Cormode. 2016. Quantiles over data streams: experimental comparisons, new analyses, and further improvements. *The VLDB Journal* 25, 4 (2016), 449–472.

[30] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. 1999. Random sampling techniques for space efficient online computation of order statistics of large datasets. *ACM SIGMOD Record* 28, 2 (1999), 251–262.

[31] Charles Masson. 2021. DDSketch. https://github.com/DataDog/sketches-java original-date: 2019-05-28T13:04:50Z.

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

[33] Rory Mitchell, Eibe Frank, and Geoffrey Holmes. 2021. An Empirical Study of Moment Estimators for Quantile Approximation. *ACM Transactions on Database Systems* 46, 1 (April 2021), 1–21. https://doi.org/10.1145/3442337

[34] Barzan Mozafari and Carlo Zaniolo. 2010. Optimal load shedding with aggregates and mining queries. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 76–88.

[35] J Ian Munro and Mike S Paterson. 1980. Selection and sorting with limited storage. *Theoretical computer science* 12, 3 (1980), 315–323.

[36] Lee Rhodes. 2015. Data Sketches. *https://yahooeng.tumblr.com/post/135390948446/data-sketches* (2015).

[37] Nicolo Rivetti, Nikos Zacheilas, Avigdor Gal, and Vana Kalogeraki. 2018. Probabilistic Management of Late Arrival of Events. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems* (Hamilton, New Zealand) *(DEBS '18)*. Association for Computing Machinery, New York, NY, USA, 52–63. https://doi.org/10.1145/3210284.3210293

[38] Gil Tene. [n.d.]. *HDR Histogram.* https://github.com/HdrHistogram/HdrHistogram

[39] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. 2013. Quantiles over data streams: an experimental study. In *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*. ACM Press, New York, New York, USA, 737. https://doi.org/10.1145/2463676.2465312

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