Estimating Quantiles from the Union of Historical and Streaming Data

Sneha Aman Singh
Iowa State University
sneha@iastate.edu

Divesh Srivastava
AT&T Labs – Research
divesh@research.att.com

Srikanta Tirthapura
Iowa State University
snt@iastate.edu

ABSTRACT

Modern enterprises generate huge amounts of streaming data, for example, micro-blog feeds, financial data, network monitoring and industrial application monitoring. While Data Stream Management Systems have proven successful in providing support for real-time alerting, many applications, such as network monitoring for intrusion detection and real-time bidding, require complex analytics over historical and real-time data over the data streams. We present a new method to process one of the most fundamental analytical primitives, quantile queries, on the union of historical and streaming data. Our method combines an index on historical data with a memory-efficient sketch on streaming data to answer quantile queries with accuracy-resource tradeoffs that are significantly better than current solutions that are based solely on disk-resident indexes or solely on streaming algorithms.

1 Introduction

A quantile is a fundamental analytical primitive, defined as follows. Let D denote a dataset of n elements chosen from a totally ordered universe. For an element $e \in D$, the rank of the element, denoted by $\mathtt{rank}(e, D)$, is defined as the number of elements in D that are less than or equal to e.

DEFINITION 1. *For $0 < \phi \leq 1$, a ϕ-quantile of D is defined as the smallest element e such that $\mathtt{rank}(e, D) \geq \phi n$.*

Quantiles are widely used to describe and understand the distribution of data. For instance, the median is the 0.5-quantile. The median is widely used as a measure of the "average" of data, and is less sensitive to outliers than the mean. The set consisting of the 0.25-quantile, the median, and the 0.75-quantile is known as the quartiles of data.

Quantile computation on large dynamic data is important in many applications, for instance, in monitoring of web server latency [10]. Latency, defined as the time elapsed between a request issued at the client and the receipt of the response from the server, is an important measure of the performance of a web service. The median latency is a measure of the "typical" performance experienced by users, and the 0.95-quantile and 0.99-quantile are used to get a detailed insight on the performance that most users experience. Similarly, quantiles find application in network performance measurement, e.g, .to determine the skewness in the TCP round trip time (RTT) [7]. Such quantile computations are a key functionality provided by many Data Stream Management Systems (DSMS), such as GS Tool [7], that provide support for real-time alerting over high-velocity streaming data generated by modern enterprises.

While DSMSes have proven immensely successful in supporting real-time analytics over streaming data, they lack the ability to perform sophisticated analysis of streaming data *in the context of historical data*, for example, comparing current trends in the streaming data with those observed over different time periods in the last few years. Such an integrated analysis of historical and streaming data is required by many emerging applications including network monitoring for intrusion detection [23, 4], financial trading, real-time bidding [28], and traffic monitoring [25]. To address the demands of such applications, *data stream warehousing systems*, such as TidalRace [16], have recently emerged. In such systems data streams, in addition to being analyzed in real-time, are also archived in a data warehouse for further analysis. At the time the streams are observed, it is also necessary to take an integrated view of streaming and archived historical data, to enable comparisons with historical trends, and to utilize past data to bring more context to the current data [4, 12]. Such an integrated processing has been considered significant in complex event processing (CEP) [9] and is used for predictive analysis and correlation of streaming and historical data.

While there has been quite a bit of work on query processing methods for the union of historical and streaming data (see the related work section for more details), there has been no prior work on methods for fundamental analytical primitives, such as quantiles and heavy hitters, in this setting. The use of pure streaming methods for analytical primitives, while feasible, results in unacceptably high error rates or memory usage for data stream warehouses. Our work takes a first step in the direction of designing *integrated* query processing methods for historical and streaming data for the estimation of quantiles, resulting in significantly lower error and memory usage than pure streaming methods for this problem.

1.1 Problem

We consider a setup where a data stream is captured and processed in real time. The data stream is collected for the duration of a "time step" into a "batch", and then loaded into a data warehouse. For example, a time step may be an hour or a day. Data that is not yet loaded into the warehouse is referred as "streaming data" or "data stream". Data that has been archived in the warehouse is called "historical data". Historical data is typically larger than the data stream by a factor of thousands. See Figure 1 for an illustration of the setup for data processing. Let U denote the universe with a total
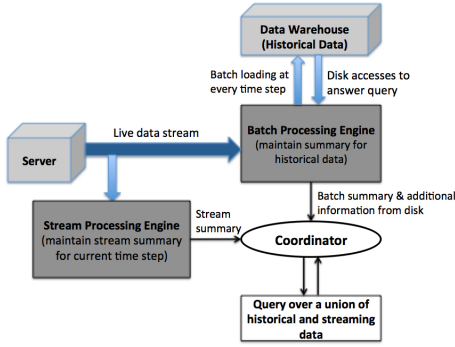
Figure 1: Setup for Integrated Processing of Historical and Streaming data

order among all elements. Let $\mathcal{H}$ denote the historical data and $\mathcal{R}$ the streaming data. Let n denote the size of $\mathcal{H}$ and m denote the size of $\mathcal{R}$. Let $\mathcal{H}[1] \leq \mathcal{H}[2] \leq \ldots, \leq \mathcal{H}[n]$ be the elements of $\mathcal{H}$ according to their total order in U and $\mathcal{R}[1] \leq \mathcal{R}[2] \leq \ldots \leq \mathcal{R}[m]$ be the elements of $\mathcal{R}$ according to their total order in U. Note that the elements of $\mathcal{H}$ and $\mathcal{R}$ are not necessarily stored in a sorted order. Let $\mathcal{T} = \mathcal{H} \cup \mathcal{R}$, and let $N = n + m$ denote the size of $\mathcal{T}$.

The task is to answer queries on $\mathcal{T}$, which is changing constantly due to arrival of new data. In general, it is expensive to answer quantile queries exactly on evolving data [20], in that it requires either very large main memory, or a large number of disk accesses. Hence, we focus on approximate computation of quantiles, where there is uncertainty in the rank (within $\mathcal{T}$) of the element returned versus the desired rank, and the approximation error is defined to be the worst case difference between the rank of the element that is returned and the derived rank.

Given an approximation parameter $\epsilon \in (0, 1]$, and a constant $\phi \in (0, 1]$, our goal is to design a method that identifies an approximate ϕ-quantile, e, from $\mathcal{T}$ such that $|rank(e, \mathcal{T}) - \phi N| < \epsilon m$. Our desired approximation error is expressed as a fraction of the size of streaming data. We can equivalently express the error as a fraction of the total data size, N, but since the goal is to have this error be significantly smaller than the size of streaming data, it is convenient to express it as an appropriate fraction of m. The amount of main memory available is much smaller than either $\mathcal{R}$ or $\mathcal{H}$, but secondary storage is abundant.

1.2 Contributions

We present a method for processing streaming and historical data that enables fast and accurate quantile queries on the union of historical and streaming data. Our method provides the following guarantees.

- A query for a ϕ-quantile on $\mathcal{T}$ is answered with approximation error ϵm where m is the size of the streaming data. While m is much smaller than the size of $\mathcal{T}$, it could still be much larger than the size of main memory. For instance, m could be of the order of 1TB. The relative error in the answer, when compared with the size of the entire dataset, becomes increasingly smaller as the size of historical data increases.

- We provide an upper bound on the memory requirement of the algorithm. We show (both theoretically as well as in practice) that the resulting accuracy-memory tradeoff is much better than what can be achieved using state-of-the-art streaming algorithms for quantile computation. We also provide

theoretical upper bounds on the number of disk accesses required to add a batch of streaming data to the warehouse, and the number of disk accesses required to answer a query for a quantile.

- We present detailed experimental results that show the performance that can be expected in practice. A quantile query on $\mathcal{T}$ is answered with accuracy about 100 times better than the best streaming algorithms while using the same amount of main memory, with the additional cost of a few hundred disk accesses for datasets of size 50 to 100 Gigabytes, which are of the order of a million disk pages. The number of disk accesses required to load a batch of streaming data into the warehouse is typically not much more than what is required to simply write the batch to disk.

1.3 Related Work

Quantile computation on large data is a well-studied problem [20, 3, 18, 6, 15, 11, 24], both in the context of stored data [20] and streaming data [3, 18, 6, 15, 11, 24]. To compute quantiles from data stored in a data warehouse or a database, data is processed using multiple passes through the disk, and hence it is possible to compute exact quantiles in a deterministic manner. In contrast, in the case of a data stream, only a single pass over the data is possible and the quantile is computed using in-memory structures that are not able to store the entire data seen so far. Hence, quantile estimation in a data stream is generally approximate, with a provable guarantee on the quality of approximation.

Munro and Paterson [20] proposed a p-pass algorithm to compute exact quantiles and showed a lower bound that the memory required to exactly compute quantiles in p passes is at least $\Omega(N^{1/p})$, where N is the number of elements in the dataset. Manku et al. in [18] proposed a single pass deterministic algorithm to estimate ϵ-approximate ϕ-quantiles using space $O(\frac{1}{\epsilon}\log^2(\epsilon N))$. They also proposed randomized algorithms, MRL98 and MRL99 [18, 19] that identify ϵ-approximate ϕ-quantiles with probability at least $(1-\delta)$, $0 < \delta < 1$, using $O\left(\frac{1}{\epsilon} \log^2\left(\frac{1}{\epsilon} \log^2\left(\frac{1}{\delta}\right)\right)\right)$ memory.

Greenwald and Khanna [15] present a deterministic single pass streaming algorithm for ϵ-approximate quantiles with worst case space requirement $O\left(\frac{1}{\epsilon} \log(\epsilon N)\right)$. Shrivastava et al. [24] present a streaming algorithm for ϵ-approximate quantiles called the "QDigest" that has a space complexity of $O(\frac{1}{\epsilon} \log U)$, where U is the size of the input domain. Wang et al. [26] performed an experimental evaluation of different streaming algorithms [15, 24, 19]. They concluded that MRL99 [19] and Greenwald-Khanna [15] are two very competitive algorithms with MRL99 performing slightly better than Greenwald-Khanna in terms of space requirement and time for a given accuracy. Since Greenwald-Khanna is a deterministic algorithm and its performance is close to MRL99, Wang et al. suggest that Greenwald-Khanna be used when it is desired to have a worst-case guarantee on the error. They also propose a simplified version of [19] called RANDOM, which performs slightly better than [19] in terms of the processing time.

Current literature on integrated processing of historical and streaming data has focused on developing efficient query processing models for data integration [22, 8, 9, 25, 4, 1, 5, 27]. Frameworks [13, 4, 22, 8] have been proposed to address complex event processing (CEP) over the integration of historical and streaming data from systems perspective. In the direction of query processing, Moirae [4], DataDepot [13], HYPE [22] and Dejavu [8, 9], and also [23] propose query processing models and techniques for enabling and optimizing CEP over the union of historical and streaming data. These works focus on declarative pattern matching, semantic level query specification and complex event processing. [23] proposes the use

of bitmap indices to store and query historical data, for enabling fast and real time query processing of streaming and historical data. Our work is different and complementary to these in that we investigate query processing strategies for a specific analytical primitive in detail and we provide guarantees on the quality of the answer as well as the performance. Our approach to indexing historical data through merging partitions is similar to that of the LSM-Tree [21]. However, the goals of the two data structures are very different – the LSM-Tree is designed for search in the presence of a high insert rate, while our structure is designed for quantile queries, with occasional batched inserts.

2 Approach

A memory-efficient approach to computing quantiles from the union of historical and streaming data is to apply a streaming algorithm, say the Greenwald-Khanna algorithm [15] or the QDigest algorithm [24] to $\mathcal{T}$. The streaming algorithm runs continuously and processes data within a single time step. At the end of the time step, the entire batch is loaded into the warehouse. The streaming algorithm maintains an in-memory summary that can be used at any time to answer quantile queries on the entire dataset seen so far. We call this the "pure-streaming" approach. The pure-streaming approach can estimate quantiles with an approximation error of ϵN using main memory of $O\left(\frac{\log{(\epsilon N)}}{\epsilon}\right)$ words (if the Greenwald-Khanna algorithm is used). Note that the approximation error is proportional to the size of the entire dataset, which keeps increasing as more data is loaded into the warehouse.

Another strawman approach is to process $\mathcal{H}$ and $\mathcal{R}$ separately, by different methods. $\mathcal{H}$ is kept on disk, sorted at all times, and an existing streaming algorithm is used to process $\mathcal{R}$ and maintain an in-memory summary of the streaming data at all times. A quantile query is answered by combining the stream summary with $\mathcal{H}$. The approximation error in the result is only due to the streaming algorithm. Hence, it is possible to achieve error proportional to m, the size of the streaming data only. Since $m \ll N$, the accuracy given by this approach is significantly better than the pure-streaming approach. However, this approach is expensive in terms of number of disk operations, because at each time step, a new dataset has to be merged into the existing sorted structure. This can lead to a large number of disk I/O operations for each time step.

Our goal is to improve upon the *accuracy of the pure streaming approach* and the *performance of the strawman approach*. We aim for accuracy comparable to the strawman approach (whose error is significantly smaller than ϵN) using a similar amount of main memory as the pure streaming algorithm and limited number of disk I/Os.

Intuition. Keeping the data fully sorted on disk at all times is not feasible, due to the large number of disk accesses needed for doing so. The other extreme, of not sorting data at all, is not feasible either, since computing quantiles will require multiple scans of the disk (at query time). We try to find a good middle ground. First, we note that sorting all data that arrives within a time step is easy to do. We repeatedly merge older partitions to create larger partitions, where each partition has data within it sorted. We perform this recursively in such a manner that (1) the number of partitions on disk is small, logarithmic in the number of time steps and (2) each data element is involved in only a few merges, so that the total amortized cost of merging partitions remains small. As a result, we maintain the historical data $\mathcal{H}$ on the disk in a structure HD that allows for fast updates, but still has only a small number of sorted partitions.

Symbol	Meaning
U	Universe from which elements are chosen
$\mathcal{H}, \mathcal{R}$	Historical data and streaming data, respectively
$\mathcal{T}$	$\mathcal{H} \cup \mathcal{R}$
n, m, N	Sizes of $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{T}$ respectively
T	Total number of time steps
ϵ	Error parameter
HD	On-disk data structure for $\mathcal{H}$
HS	In-memory summary for $\mathcal{H}$
κ	A small integer parameter, the "merge threshold"
$\mathcal{P}_\ell^i$	A partition within level ℓ of HD
SS, TS	An in-memory summary of $\mathcal{R}$ and $\mathcal{T}$, respectively.

Table 1: Summary of Notation Used

In addition to the on-disk structure HD, we maintain an in-memory summary HS that provides us quick access to elements at different ranks within each sorted partition. This summary of $\mathcal{H}$ is updated at each time step with the addition of a new dataset to the warehouse and also when partitions are merged together. We also maintain an in-memory summary SS for the streaming data $\mathcal{R}$. This summary is updated with every new incoming element. At the end of each time step, when the data stream is loaded into the warehouse, SS is reset. Quantile queries are answered using a combination of HS and SS to generate a quick estimate, followed by making few queries to the disk resident data, to get a more accurate estimate. We show that our approach is more accurate than the pure-streaming approach and makes significantly fewer disk I/Os compared to the strawman approach.

Table 1 has a summary of the notation used in this section. We present the method for processing historical data and maintaining HD and HS in Section 2.1, the method for processing streaming data and maintaining SS in Section 2.2, and the method for answering queries in Section 2.3. The data structures are initialized as in Algorithm 1.

Algorithm 1: Initialize Data Structures

1 Input parameters: error parameter ϵ
2 $\epsilon_1 \leftarrow \epsilon/2, \epsilon_2 \leftarrow \epsilon/4$
3 $\beta_1 \leftarrow \lceil \frac{1}{\epsilon_1} + 1 \rceil$, and $\beta_2 = \lceil \frac{1}{\epsilon_2} + 1 \rceil$
4 Call HistInit(ϵ_1, β_1) and StreamInit(ϵ_2, β_2)

2.1 Processing Historical Data

When a new dataset $\mathcal{D}$ is added to HD at the end of a time step, $\mathcal{D}$ is sorted and stored as a separate data partition. Doing this has two advantages – (1) if we tried to merge in $\mathcal{D}$ into the existing sorted partition, the cost of merging the partitions will be very high; by keeping it as a separate partition, this cost is avoided, and (2) it also enables us to maintain data from different time steps separately, hence allowing for queries restricted to a specific time window of a certain number of time steps. However, keeping a separate partition for each time step can lead to a very high query cost, hence some merging of existing partitions is necessary. Our approach to this problem is to let a few unmerged partitions accumulate, and when the number of such partitions crosses a threshold, merge them into a larger partition. This process is recursively repeated by allowing a few larger partitions accumulate, followed by merging them into an even larger partition, and so on. Doing this systematically allows for the merging cost to be controlled, and also leads to a low cost during query processing, since much of the data is spread around a
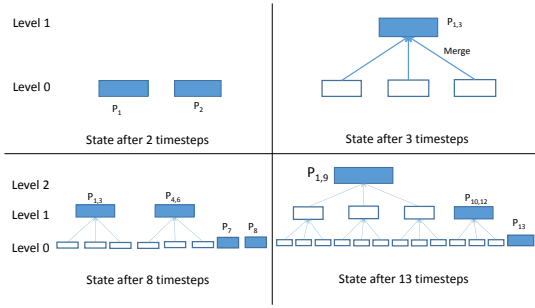
Figure 2: An example of how the data partitions within HD evolve, over 13 time steps. $P_{i,j}$ **denotes a partition formed by the union of all data that came in time steps** i **till** j **(both endpoints inclusive). We assume** $\kappa = 2$. **The empty nodes in the picture represent partitions that are no longer stored separately i.e. have been merged into larger partitions.**

small number of partitions. Figure 2 shows an example of how the data partitions evolve over 13 time steps.

More precisely, let $\kappa > 1$ be a small integer parameter chosen before the algorithm begins execution. Each data partition of HD is associated with a "level", a small positive integer. We maintain the following invariant: *each level can have a maximum of κ data partitions at any point of time.* Let the partitions at level ℓ be denoted $\mathcal{P}_\ell^0, \mathcal{P}_\ell^1, \ldots, \mathcal{P}_\ell^{j-1}, j \leq \kappa$.

Suppose a newly arrived dataset $\mathcal{D}$, of size η needs to be added to HD. Then, $\mathcal{D}$ is first sorted and stored at level 0 of HD; the sorting can be performed in-memory, or using an external sort [14], depending on the size of $\mathcal{D}$. κ is also called the "merge threshold". The maximum number of levels in HD is $\log_\kappa T$, where T is the total number of time steps. If there are more than κ partitions in level 0, then all partitions within level 0 are merged to form a single partition in level 1, so that our invariant is maintained. For instance, in Figure 2, at the end of two time steps, there are two separate partitions at level 0. At the end of the next time step, when there are three partitions at level 0, they are merged into a single partition at level 1 (denoted $P_{1,3}$ in the figure), to maintain the invariant. Similarly, if there are more than κ partitions in level 1, they are recursively merged to form larger partitions at level 2, and so on, until we reach a level that has κ or fewer partitions. For instance, in Figure 2, at the end of 9 time steps, there will be 3 partitions at level 1, which will be merged together into a single partition at level 2 (denoted $P_{1,9}$ in the figure). Partitions at higher levels contain data aggregated from a number of time steps, while partitions at lower levels are smaller and contain data from fewer time steps.

When a quantile query is executed over HD, a common operation is to determine the number of elements in HD that are lesser than a given value. To answer this query, our structure needs to consult a small (logarithmic in the number of time steps) number of data partitions. At the same time, to add a new dataset to the warehouse, our structure will not need to manipulate many partitions; the larger data partitions are rarely touched.

Along with the batch data structure HD, an in-memory summary HS is also maintained for historical data. We next describe how this summary HS is constructed. Algorithm 3 describes the steps required to maintain the data structures for the historical data, including the on-disk data structure HD, and its in-memory summary HS.

Summary of Historical Data HS HS is an in-memory data structure that summarizes historical data stored in different data parti-

---

Algorithm 2: `PartitionSummary`$(\mathcal{P}, \beta_1)$

Input: Partition $\mathcal{P}$ on disk, sorted in increasing order
Output: Return a summary of $\mathcal{P}$ of size β_1

1   $S \leftarrow \emptyset$
2   Let η denote the size of the partition
3   Set $S[0]$ to be the smallest element in the partition
4   **for** i *from* 1 *to* $\beta_1 - 1$ **do**
5   Set $S[i]$ to the element at rank $(i\epsilon_1\eta)$ in $\mathcal{P}$
6   **return** S

Algorithm 3: Maintenance of the on-disk data structure HD, and its in-memory summary HS for historical data.

1   **def** $HistInit(\epsilon_1, \beta_1)$
      /* Initialize data structures for historical data   */
2   Remember ϵ_1, β_1
3   HD $\leftarrow \emptyset$
4   HS $\leftarrow \emptyset$
5   **def** $HistUpdate(\mathcal{D})$
      /* Update HD and HS with a new dataset $\mathcal{D}$.   */
6   Sort $\mathcal{D}$ and add as a new partition to level 0 of HD
7   Add `PartitionSummary`$(\mathcal{D}, \beta_1)$ to level 0 of HS
8      $\ell \leftarrow 0$
9   **while** *(more than κ partitions at level ℓ)* **do**
10   Multi-way merge the sorted partitions at level ℓ, $\mathcal{P}_\ell^1, \mathcal{P}_\ell^2, \ldots, \mathcal{P}_\ell^\kappa$ into a single sorted partition $\mathcal{P}'$ using a single pass through the partitions
11   Add $\mathcal{P}'$ to level $(\ell + 1)$ in HD and `PartitionSummary`$(\mathcal{P}', \beta_1)$ to $HS_{\ell+1}$
12         $HS_\ell \leftarrow \emptyset$
13         $\ell \leftarrow (\ell + 1)$
14   **def** $HistSummary()$
15   **return** HS

---

tions of HD. Naturally following a parallel structure to the on-disk organization, HS consists of data structures at $\lambda = \lceil \log_\kappa T \rceil$ different levels, one corresponding to each level of HD, HS_0, HS_1, HS_2, $\ldots$, $HS_{\lambda-1}$. Figure 3 shows an example computation of the summaries for historical data. In this example, there are three partitions in HD, P1, P2, and P3, with 100, 100, and 200 elements respectively. Each partition is stored on disk, and has a corresponding in-memory summary, also shown in the figure.

Let $\epsilon_1 = \frac{\epsilon}{2}$ and $\beta_1 = \lceil \frac{1}{\epsilon_1} + 1 \rceil$. For $0 \leq \ell \leq \lambda - 1$, each data structure HS_ℓ, is a set of no more than κ summaries, HS_ℓ^0, HS_ℓ^1, $\ldots$, HS_ℓ^{j-1}, $j \leq \kappa$. The data partition in HD corresponding to HS_ℓ^i is denoted $\mathcal{P}_\ell^i$. Each summary HS_ℓ^i is an array of length β_1.

When a new data partition $\mathcal{D}$ is created, either due to adding a new dataset to HD at level 0, or due to merging smaller partitions, a new summary is generated for this partition as described in Algorithm 2. After $\mathcal{D}$ is sorted, it is divided into β_1 equal subsequences, and the first element of each subsequence is chosen into the summary. In the example shown in Figure 3, there is a summary for each of P1, P2, and P3, each with five elements, since $\epsilon_1 = 1/4$. Each item of the summary, in addition to having the value of the element, also has a pointer to the on-disk address, for fast lookup in the data warehouse. Note that the generation of a new data partition and the corresponding summary occur simultaneously so no additional disk access is required for computing the summary, beyond those taken for generating the new data partition.

Partition P1 (on disk)	1	2	3	.....	100
Partition P2 (on disk)	101	102	103	.....	200
Partition P3 (on disk)	2	3	4	.....	201
Stream S (not stored)	401	402	403	.....	600

Summary for P1	1	25	50	75	100
Summary for P2	101	125	150	175	200
Summary for P3	2	51	101	151	201

Stream Summary (SS)

401	438	452	480	520	530	565	595	600

Below is TS = The Summary of the Entire Data = Union of the Historical and Batch Summaries, followed by sorting them

1	2	25	50	51	75	100	101	101	125	150	151	175	200	201	401	438	452	480	520	530	565	595	600

Below is shown for each element in TS, its actual rank in the union of historical and streaming data

1	3	49	99	101	149	199	201	201	249	299	301	349	399	400	401	438	452	480	520	530	565	595	600

Below is shown for each element in TS, the lower bound L on its rank

0	0	25	50	100	125	150	200	200	225	250	300	325	350	400	400	425	450	475	500	525	550	575	600

Below is shown for each element in TS, the upper bound U on its rank

25	75	100	125	175	200	225	300	300	325	350	400	425	450	500	525	550	575	600	625	650	675	700	725

Figure 3: An example of how the summaries and L_i and U_i are computed. We assume $\epsilon = 1/2$, so that $\epsilon_1 = 1/4$ and $\epsilon_2 = 1/8$. There are three partitions on disk, P1, P2, and P3. The length of the summary for each partition is 5 and the length of the stream summary is 9.

2.2 Processing Streaming Data

Streaming data is processed using an ϵ_2-approximate streaming quantiles algorithm where $\epsilon_2 = \epsilon/4$. Given a desired rank r, this algorithm returns an element whose rank $\hat{r}$ in a stream $\mathcal{R}$ of size m lies between $[r - \epsilon_2 m, r + \epsilon_2 m]$. For our work, we seek worst-case (not randomized) guarantees on the error, and hence we use the Greenwald-Khanna algorithm [15], though other algorithms such as the Q-digest [24] can also be used.

Let $\beta_2 = \lceil \frac{1}{\epsilon_2} + 1 \rceil$. When a query is received, the streaming algorithm is used to generate a summary SS, an array of length β_2, using steps shown in Algorithm 4. StreamSummary uses the streaming algorithm to find elements of approximate rank $i\epsilon_2 m$ from $\mathcal{R}$, for i ranging from 0 to $1/\epsilon_2$, and add these elements to SS. In the example shown in Figure 3, there is a single summary of size 9 for the stream, since $\epsilon_2 = 1/8$. Note that unlike the summary for historical data, the β_2 elements in the streaming summary are not spaced uniformly among the elements of $\mathcal{R}$, but are spaced approximately uniformly. Due to the guarantee provided by the streaming algorithm, each of these β_2 elements are identified with a maximum error of $\epsilon_2 m$ in their ranks. We use the following guarantees from the Greenwald-Khanna [15] algorithm. We call the summary used by the Greenwald-Khanna as the "GK summary".

THEOREM 1 (GREENWALD-KHANNA [15]). *There is an algorithm that processes a stream $\mathcal{R}$ of numbers and maintains a summary of size $\left(\frac{1}{\epsilon}\log(\epsilon m)\right)$ where m is the size of the stream so far and ϵ is an error parameter. For a given rank r, $0 \le r \le m$, the summary returns a value whose rank r' is guaranteed to be in the interval $[r, r + \epsilon m]$.*

The guarantee provided by the stream summary follows directly from Theorem 1 and is described in the following observation. Let m denote the size of the stream.

LEMMA 1. *For $1 \le i \le \beta_2 - 1$, SS$[i]$ is at rank r_i in $\mathcal{R}$ where $i\epsilon_2 m \le r_i \le (i+1)\epsilon_2 m$.*

2.3 Answering a Quantile Query over the Union of Historical and Streaming data

Our algorithm gives two kinds of responses to a quantile query (1) a quick response with a rough estimate, using only the in-memory structures, as described in Algorithm 5 and (2) a slower, but more accurate response using the in-memory summaries as well as disk accesses, as described in Algorithm 6.

Algorithm 4: Maintenance of the data structure for the stream $\mathcal{R}$

1 **def** *StreamInit*(ϵ_2, β_2)
2   Remember ϵ_2, β_2
3   GK is an empty GK summary with error parameter ϵ_2

4 **def** *StreamUpdate*(e)
    /* When an element e arrives in $\mathcal{R}$ */
5   Insert e into GK

6 **def** *StreamSummary*()
    /* Return a stream summary, to be used in answering a query   */
7   SS is a vector of length β_2
8   Set SS[0] to be the smallest element in the stream so far
9   **for** i *from* 1 *to* $\beta_2 - 1$ **do**
10   Set SS[i] to the element in GK at approximate rank $i\epsilon_2 m$
11   **return** SS

12 **def** *StreamReset*()
    /* Called at the end of each time step. The stream summary is reset to empty.   */
13   GK is an empty GK summary with error parameter ϵ_2

2.3.1 Quick Response

On receiving a quantile query, Algorithm 5 provides a quick answer that has an absolute error in rank proportional to the size of $\mathcal{T}$. When a query is received, the stream summary SS is computed as described in Section 2.2. The query algorithm then sorts the union of SS and HS yielding TS, a summary of the entire data $\mathcal{T}$ observed so far. Let the size of TS be δ. Let TS$[i]$ denote the ith element in the sorted order of TS, $0 \le i < \delta$. In the example shown in Figure 3, TS consists of $3 \cdot \beta_1 + \beta_2$ elements, for a total of 24 elements, arranged in sorted order. There are a total of 600 elements within the data, and it can be seen that the actual rank of elements in TS among the 600 elements are approximately uniformly spaced.

For $0 \le i \le \delta - 1$, let L_i denote a lower bound on the rank of TS$[i]$ in $\mathcal{T}$, and U_i denote an upper bound on the rank of TS$[i]$ in $\mathcal{T}$. L_i and U_i are defined as follows. Consider an element TS$[i]$, for $0 \le i \le (\delta - 1)$. Let α_S denote the number of elements from SS that are less than or equal to TS$[i]$. For each partition $P \in$ HS,

let α_P denote the number of elements from partition P that are in HS and are less than or equal to TS[i]. And, let m_P denote the size of partition P in HD. Further, suppose that TS[i] belonged to partition I. L_i and U_i are defined as follows. Let b be defined to be 1 if $\alpha_S > 0$ and 0 otherwise.

$$L_i = m\epsilon_2 b(\alpha_S - 1) + \sum_{P \in \text{HS}, \alpha_P > 0} m_P \epsilon_1 (\alpha_P - 1)$$

$$U_i = m\epsilon_2 b(\alpha_S + 1) + \sum_{P \in \text{HS}, \alpha_P > 0} m_P \epsilon_1 (\alpha_P)$$

See Figure 3 for an example of the computation of L_i and U_i for each $0 \le i \le \delta - 1$.

Algorithm 5: QuantilesQuickResponse(r)

/* *Quickly return an element of approximate rank r from $\mathcal{T}$ with error $O(\epsilon N)$* */
1 Let SS $\leftarrow$ StreamSummary(), from Algorithm 4
2 Let HS $\leftarrow$ HistSummary(), from Algorithm 3
3 TS $\leftarrow (\cup_{s \in \text{HS}} s) \cup$ SS, sorted in an increasing order
4 Compute L_i for each $0 \le i \le (\delta - 1)$
5 Let j be the smallest integer $0 \le j \le (\delta - 1)$ such that $L_j \ge r$. If no such j exists, then $j = (\delta - 1)$.
6 **return** TS[j]

LEMMA 2. *The following guarantees hold for L_i and U_i, for $0 \le i < \delta$. Let $r_i = \text{rank}(TS[i], \mathcal{T})$. Then,*

1. $L_i \le r_i \le U_i$.

2. $U_i - L_i \le \epsilon N$.

PROOF. Let β_S denote the number of elements from $\mathcal{R}$ that are less than or equal to TS[i]. For each partition $P \in$ HD, let β_P denote the number of elements from P that are less than or equal to TS[i]. It follows that $\text{rank}(TS[i], \mathcal{T}) = \beta_S + \sum_{P \in \text{HD}} \beta_P$.

There are α_S elements from SS that are less than or equal to TS[i]. If $\alpha_S > 0$, then the largest of these is SS[$\alpha_S - 1$], and is at rank between $(\alpha_S - 1)\epsilon_2 m$ and $\alpha_S \epsilon_2 m$ in $\mathcal{R}$, from Lemma 1. Hence, $\beta_S \ge (\alpha_S - 1)\epsilon_2 m$. Similarly, for each partition $P \in$ HD such that $\alpha_P > 0$, $\beta_P \ge m_P \epsilon_1(\alpha_P - 1)$. Note that if $\alpha_P = 0$, then no element from the partition P is less than TS[i], since the smallest element from P is also a part of the summary. In this case, $\beta_P = 0$. Adding these up, we get that $r_i \ge (\alpha_S - 1)\epsilon_2 mb + \sum_{P \in \text{HS}, \alpha_P > 0} m_P \epsilon_1(\alpha_P - 1)$, showing that $L_i \le r_i$.

For the upper bound U_i, consider the smallest element from SS that is larger than TS[i]. This must be the α_Sth element in SS. Using Lemma 1, the rank of this element in $\mathcal{R}$ is no more than $(\alpha_S + 1)\epsilon_2 m$. Hence, we have $\beta_S \le (\alpha_S + 1)\epsilon_2 m$. For each partition P, we can similarly show $\beta_P \le (\alpha_P)\epsilon_2 m_P$, leading to the proof that $U_i \ge r_i$.

By the definition of U_i and L_i, we have $U_i - L_i \le (2\epsilon_2 m + \sum_{P \in \text{HS}} m_P \epsilon_1) = (2\epsilon_2 m + \epsilon_1 n) \le \epsilon N$. $\square$

LEMMA 3. *In response to a query for a r-quantile, QuantilesQuickResponse (Algorithm 5) returns an element of rank $\hat{r}$ such that $|\hat{r} - r| \le 1.5\epsilon N$.*

PROOF. Case I: Suppose there exists an $i, 0 \le i \le (\delta - 1)$ such that $L_i \ge r$. Then the algorithm returns TS[j] for the smallest j such that $L_j \ge r$. In this case, it must be true that $L_{j-1} < r$. Also note that $L_j \le L_{j-1} + \epsilon_1 N$. Hence we have $r \le L_j \le$

$(r + \epsilon_1 N)$. The rank of the element returned, $\hat{r}$, is $\text{rank}(TS[j], \mathcal{T})$. From Lemma 2, we have $L_j \le \hat{r} \le L_j + \epsilon N$. Overall, we get $r \le \hat{r} \le r + (\epsilon + \epsilon_1)N$, proving this case.

Case II: Suppose that $L_{\delta-1} < r$. In this case, $\hat{r} = \text{rank}(TS[\delta - 1], \mathcal{T})$. From Lemma 2, we have that $\hat{r} < r + \epsilon N$. We can also show that $\hat{r} \ge N - \epsilon N$, since this is the last element in TS. Since $r \le N$, we have $\hat{r} \ge r - \epsilon N$. Thus, we have $r - \epsilon N \le \hat{r} \le r + \epsilon N$, proving this case. $\square$

Algorithm 6: QuantilesAccurateResponse(r)

/* *Return an element of approximate rank r from $\mathcal{T}$ with error $O(\epsilon m)$* */
1 Let SS $\leftarrow$ StreamSummary(), from Algorithm 4
2 Let HS $\leftarrow$ HistSummary(), from Algorithm 3
3 TS $\leftarrow (\cup_{s \in \text{HS}} s) \cup$ SS, sorted in an increasing order
4 $(u, v) \leftarrow$ GenerateFilters(TS, r)
5 **return** Query(SS, HS, u, v))

2.3.2 Accurate Response

We now present Algorithm 6 which returns a response to a quantile query that is more accurate than Algorithm 5. As a first step, the algorithm finds a pair of elements u and v from TS such that the element of desired rank r is guaranteed to lie between these elements, i.e $\text{rank}(u, \mathcal{T}) \le r \le \text{rank}(v, \mathcal{T})$. We refer to this pair as Filters. Filters are generated by calling GenerateFilters(X, r), described in Algorithm 7.

Algorithm 7: GenerateFilters(TS, r)

/* *Find elements u, v from TS such that the element of rank r in $\mathcal{T}$ is guaranteed to lie between u and v* */
1 Let x be the largest $i, 0 \le i < \delta$ such that $U_i \le r$
2 Let y be the smallest $i, 0 \le i < \delta$ such that $L_i \ge r$
3 **return** (TS[x], TS[y])

LEMMA 4. *Given rank r and summary TS, Algorithm 7 returns elements $u, v \in$ TS such that $\text{rank}(u, \mathcal{T}) \le r \le \text{rank}(v, \mathcal{T})$ and $(\text{rank}(v, \mathcal{T}) - \text{rank}(u, \mathcal{T})) < 4\epsilon N$, where $N = |\mathcal{T}|$.*

PROOF. $u = $ TS[x] where $U_x \le r$. From Lemma 2, $\text{rank}(u, \mathcal{T}) \le U_x$, and hence we have $\text{rank}(u, \mathcal{T}) \le r$. Similarly, we have $\text{rank}(v, \mathcal{T}) \ge r$.

It must be true that $U_{x+1} > r$. Let α denote the rank of TS[$x+1$] in $\mathcal{T}$. Using Lemma 2, we have $\alpha > r - \epsilon N$. Since successive elements in TS cannot differ by more than ϵN in rank, we have $\text{rank}(u, \mathcal{T}) > r - 2\epsilon N$.

Similarly, we note that $L_{y-1} < r$. Let β denote the rank of TS[$y - 1$] in $\mathcal{T}$. From Lemma 2, we have $\beta < r + \epsilon N$. Since successive elements in TS cannot differ by more than ϵN in rank, we have $\text{rank}(v, \mathcal{T}) < r + 2\epsilon N$. $\square$

After computing u and v, the algorithm for an accurate response makes a series of recursive calls to the function Query (described in Algorithm 8) to narrow down the range of elements between u and v. This algorithm uses a binary search on the range of possible values to find a new pair of filters with smaller interval size. The idea is to narrow down the range of elements between the pair of filters to a point where all elements between the filters in HD can be loaded into memory. These consecutive elements from HD are used in combination with SS to accurately answer the quantile query.

Algorithm 8: Query(SS, HS, u, v, r)

/* Return an element at approximate rank r within $\mathcal{T}$. SS
   is a summary of the stream, and HS is a summary of
   HD. The desired element is guaranteed to lie between
   u and v */

1   $z \leftarrow (u + v)/2$
   /* Lines 2-7 compute ρ_1, the rank of z in HD   */

2   $\rho_1 \leftarrow 0$

3   **foreach** *summary ψ in* HS **do**
     /* It is necessary to iterate through different partitions
      (corresponding to the summaries) since the rank of
      z within $\mathcal{T}$ has contributions from many different
      partitions within HD.   */

4   Let $\mathcal{P}$ be the data partition corresponding to ψ
     /* l and v help in narrowing the range of elements we
      need to search in $\mathcal{P}$ to compute the rank of z in $\mathcal{P}$
      */

5   Let l be the rank in $\mathcal{P}$ of the largest element $x \in \psi$ such
     that $x \leq u$ and p the rank in $\mathcal{P}$ of the smallest element
     $y \in \psi$ such that $y \geq v$

6   Compute ρ', the rank of z in $\mathcal{P}$ using a binary search on
     $\mathcal{P}$, starting with left endpoint l and right endpoint p

7      $\rho_1 \leftarrow \rho_1 + \rho'$
   /* Compute ρ_2, the approximate rank of z in $\mathcal{R}$ */

8   **for** i *from* 0 *to* $|SS - 1|$ **do**

9   **if** $z \geq SS[i]$ **then**

10        $\rho_2 \leftarrow \rho_2 + \epsilon_2 m$

11   $\rho \leftarrow \rho_1 + \rho_2$ // The approximate rank of z in $\mathcal{T}$

12   **if** $(r < (\rho - \epsilon m))$ **then**

13   **return** Query(SS, HS, u, z, r)

14   **else if** $(r > (\rho + \epsilon m))$ **then**

15   **return** Query(SS, HS, z, v, r)

16   **else**

17   return z

LEMMA 5. *Given a query for an element of rank r from $\mathcal{T}$, Algorithm 6 returns an element whose rank in $\mathcal{T}$ is $\hat{r}$ such that $|r - \hat{r}| = O(\epsilon m)$, where $0 < \epsilon < 1$ is an error parameter .*

PROOF. Algorithm 6 starts off computing a pair of filters u and v from TS such that $\text{rank}(u, \mathcal{T}) \leq r \leq \text{rank}(v, \mathcal{T})$ and $\text{rank}(v, \mathcal{T}) - \text{rank}(u, \mathcal{T}) \leq 4\epsilon N$. Algorithm 8, uses these filters, and visits the disk to find a new pair of filters u' and v' such that $\text{rank}(u, \mathcal{T}) \leq r \leq \text{rank}(v, \mathcal{T})$ still holds true, while u' and v' are closer in rank when compared to the previous pair of filters. This recursion continues till the number of elements between v' and u' is no more than $\frac{1}{\epsilon}$. At this point, the algorithm retrieves elements from HD between u and v into the query data structure Q. Had there not been the streaming data, Q would have returned the element of rank r with error 0. However, there is an error of $\epsilon_2 m$ because the ranks of elements of SS in $\mathcal{R}$ are only approximate with an error of $O(\epsilon_2 m)$. Hence the error of our algorithm is $O(\epsilon m)$. $\square$

2.4 Performance

We first consider the number of **disk accesses** made by the algorithm, for update, and for answering a query. We note that the disk accesses required for updating HD in Algorithm 3 are sequential I/Os, while the disk accesses required for answering queries are mostly random I/Os. While random I/Os are more expensive than

sequential, the number of random I/Os made during a query is relatively small.

LEMMA 6. *The amortized number of disk accesses required per time step to update* HD *and* HS *when a new dataset $\mathcal{D}$ is added is $O(\frac{n}{BT} \log_\kappa T)$, where n is the size of $\mathcal{H}$, B is the block size and T is the number of time steps.*

PROOF. When a new dataset $\mathcal{D}$ arrives, disk I/Os are made for two reasons: (1) sort and add $\mathcal{D}$ into a new partition in HD, and (2) if needed, merge data partitions.

When a dataset of size η is added to HD, it needs to be sorted, using external sorting [14]. Prior work [2] has shown that in practice, external sorting can be done using a constant number of passes over data, each pass involving a multi-way merge of data, even for very large datasets. Thus, the total number of disk accesses for sorting is linear in the number of blocks in the partition, $O(\eta/B)$. To find the amortized number of accesses per time step, we sum $O(\eta/B)$ over all datasets inserted into HD, and divide by the number of time steps, leading to $O(\frac{n}{BT})$.

The second kind of disk access is due to the merging of older partitions into a single partition. Consider an element e that was inserted into the warehouse. It will first belong to one sorted partition, which will further get merged into another partition, and so on. The number of levels of merging is $\log_\kappa T$, since each level of merging reduces the number of partitions by a factor of κ. In each level of merge, the total amount of data that is merged is n items. Since merging takes sequential I/Os of the order of the number of disk blocks being merged, the total I/Os for merging at a single level is $O(n/B)$. Summing this across all the levels, we get a total disk accesses of $O(\frac{n}{B} \log_\kappa T)$. The total number of disk I/Os is the sum of the above two terms, averaged across T time steps, which is $O(\frac{n}{BT} \log_\kappa T)$. $\square$

LEMMA 7. *The number of disk accesses required by Algorithm 6 to identify an element of rank r from $\mathcal{T}$, is $O(\log_\kappa T \log \frac{\epsilon n}{B} \log |U|)$*

PROOF. The query has to execute multiple recursive calls. Within each recursive call, the algorithm, for each partition $\mathcal{P}$ in HD, computes l and p. This computation does not require any disk accesses, since for each element in ψ, its rank with the corresponding partition is explicitly computed and stored, while the summary was being computed. Computing the rank ρ' of z however takes disk accesses. Since the difference in rank of l and p within $\mathcal{P}$ starts off at no more than ϵn, and the binary search can stop when we have reached the state where all relevant data fits within one disk block of size B, the binary search takes $O(\log \frac{\epsilon n}{B})$ disk accesses. Summed over all partitions, the total is $O(\log_\kappa T \log \frac{\epsilon n}{B})$ disk accesses per recursive call. Since the depth of recursion is $O(\log |U|)$, the total number of disk accesses follows. $\square$

LEMMA 8. *Total main memory required by the summary for historical data,* HS *is $O\left(\frac{\kappa \log_\kappa (T)}{\epsilon}\right)$.*

PROOF. The number of levels in HS is $\log_\kappa T$. There are no more than κ partitions in each level, and the size of the summary for each partition is $\beta_1 = O(1/\epsilon)$. Therefore, the total size of HS is $O\left(\frac{\kappa \log_\kappa (T)}{\epsilon}\right)$. $\square$

LEMMA 9. *Total main memory required by the summary of streaming data* SS *is $O\left(\frac{\log(\epsilon m)}{\epsilon}\right)$.*

PROOF. Given a rank r, an ϵ_2-approximate streaming algorithm requires a total memory of $O\left(\frac{\log(\epsilon m)}{\epsilon}\right)$ (Theorem 1) to return an

element with an approximate rank r from $\mathcal{R}$. The structure SS requires $\lceil \frac{1}{\epsilon_2} + 1 \rceil = O\left(\frac{1}{\epsilon}\right)$ words of memory. Hence total memory required to construct and maintain SS is $O\left(\frac{\log(\epsilon m)}{\epsilon}\right)$. $\quad\square$

The total main memory required by our algorithm follows from Lemmas 9 and 8 and is given by the following observation.

OBSERVATION 1. *The total main memory required by our algorithm to maintain the summaries is*
$O\left(\frac{1}{\epsilon}(\log(\epsilon m) + \kappa \log_\kappa(T))\right).$

We present the overall guarantees in Theorem 2.

THEOREM 2. *Our algorithm, when given an integer $r \in [0, N)$, returns an element $e \in \mathcal{T}$ such that $(r - \epsilon m) \leq \mathtt{rank}(e, \mathcal{T}) \leq (r + \epsilon m)$. The total main memory requirement of our algorithm is $O\left(\frac{1}{\epsilon}(\log(\epsilon m) + \kappa \log_\kappa(T))\right)$. The amortized number of disk accesses required to update HD at each time step is $O(\frac{n}{BT} \log_\kappa T)$ and the number of disk accesses to answer a query is $O(\log_\kappa T \log \frac{\epsilon n}{B} \log |U|)$ where B is the block size, T is the number of time steps, m the size of the streaming data, n the size of historical data, and $N = (n + m)$.*

Let us consider an example for illustration. Suppose that a time step is a day. Also, suppose that 10TB of data is loaded into the data warehouse at each time step, for 3 years, and that the block size is 100KB. The average number of disk operations required each day to add data to the warehouse is about $\frac{10^8}{3 \times 365} \times \log(10^8)$, which is of the order of 10^6. This includes the disk accesses needed to add new data as well as merge older partitions. Assuming that a fast hard disk can access 1 block per millisecond, this will take approximately 1000 seconds. The processing time can be reduced further by parallelizing the merge operations [17]. Assuming that approximation parameter ϵ is 10^{-6}, total number of disk accesses required to answer a query is in the order of 350, using order of 300000 words of memory.

Optimization. When compared with Algorithm 6, we made an optimization to reduce the number of disk accesses for query processing. As described, when a quantile query is received Algorithm 7 finds a pair of elements within which the quantile is guaranteed to lie. Following this, Algorithm 8 is recursively called to narrow down the range $[u, v]$, always making sure that the quantile lies within the range. The optimization is that the recursive search needs to proceed only as long as the pair of elements u and v are in different disk blocks. Once u and v are within the same disk block, we do not use any further disk operations, and store the block in memory for further iterations. This yielded a reduction in the number of disk accesses.

Queries Over Windows. It may be necessary to answer queries over "time windows", a subset of data that has arrived during recent time steps. The algorithms that we presented can be directly used to answer windowed queries if the window sizes are aligned with the partition boundaries in the warehouse. For such queries, we can restrict our attention to data structures corresponding to the partitions that fall within the window.

3 Experiments

We report the results of our simulation-based experiments, where we evaluate the accuracy, memory usage, and runtime of the algorithm.

3.1 Experimental Setup

We used a 64-bit Intel Core i5 (4 cores) Macbook - OS X Yosemite, with a processor speed of 2.6GHz and 8GB RAM. We implemented all algorithms using Java 7. We assumed a block size B of 100 KB.

Datasets We used two synthetic datasets "Normal" and "Uniform Random", and two real world datasets, one derived from Wikipedia page view statistics and the other a network traffic trace.

(1) The **Normal** dataset was generated using normal distribution with a mean of 100 million and a standard deviation of 10 million. The size of the streaming data (total data collected at each time step, not yet loaded into the warehouse) is 1GB. The total data volume at each time step is 1GB, and there are 100 time steps. Thus, the total size of historical data is 100GB.

(2) The **Uniform** dataset was generated by choosing elements uniformly at random from a universe of integers ranging from 10^8 to 10^9. The maximum size of the streaming data is 1GB. With 100 time steps, the total size of historical data is 100GB, with 1GB per time step.

(3) The **Wikipedia** dataset was generated using page view stats from a Wikipedia dump [1]. Each tuple of this dataset is the size of the page returned by a request to Wikipedia. The maximum size of the streaming part of the dataset is 500MB. There were 116 time steps, and the total size of the historical data is 58 GB.

(4) The **Network Trace** was generated from anonymized traffic traces taken at a west coast OC48 peering link for an ISP. This data was generated over a period of about 15 hours. Each tuple is a source-destination pair. The maximum size of the streaming part of the dataset is 600MB. There were 100 time steps, and the total size of the historical data is 60GB.

Performance Metrics: The three main performance measures are accuracy, memory usage, and runtime of the algorithm. The runtime of the algorithm is measured in terms of the processing and query time, as well as the number of disk accesses. The accuracy is measured using *relative error*, defined as $\frac{|r - \hat{r}|}{\phi N}$ where r is the rank desired by the quantile query, and $\hat{r}$ is the actual rank of the element returned by the algorithm. The different components of runtime are: (1) update/processing time per time step, (2) the number of disk accesses required per time step to add a new dataset to the historical data, (3) time taken to answer a query for the ϕ-quantile using the data structures and the historical data, and (4) the number of disk accesses required to answer a query for the ϕ-quantile.

Implementation Details: We used the Greenwald-Khanna (GK) algorithm to process streaming data, in conjunction with on-disk and in-memory historical data structures. For comparison, we implemented a "pure-streaming" approach using two prominent deterministic streaming quantile algorithms - GK [15] and QDigest [24]. Given a memory budget, we allocate 50 percent of the memory to the stream summary and 50 percent of the memory to the historical summary. Note that giving half the memory to the streaming summary and half to the historical summary leads to an error no worse than twice of what can be obtained using the optimal split, since each summary receives at least half of the maximum possible memory that it could have received. A detailed investigation of the optimal memory split between the historical and streaming summary is an interesting question for future investigation.

3.2 Results

Accuracy: We measured the relative error of different approaches as a function of the memory, by varying the memory from 100 MB to 500 MB, for $\kappa = 10$. We report the median of 7 different runs in Figures 4a, 4b, 4c and 4d. Our algorithm with the accurate response is labeled as "Our Algorithm", and quick response is labeled as "Quick Response". We observe that the accuracy of "quick response" is close to the QDigest algorithm, and the accuracy of the accurate response algorithm is significantly better than the rest.

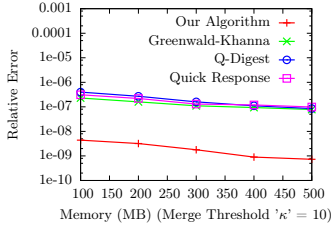---

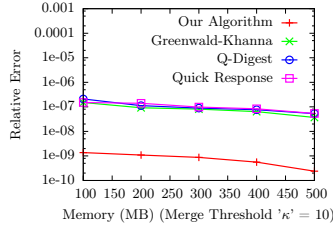[1]http://dumps.wikimedia.org/other/pagecounts-raw/
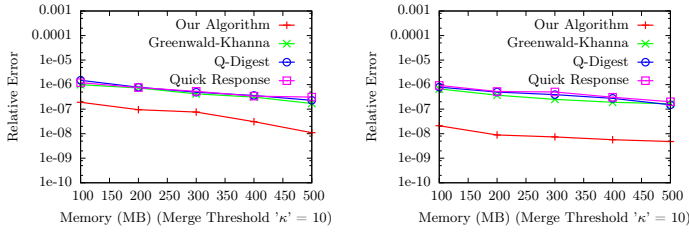
(a) Uniform Random, size 101GB
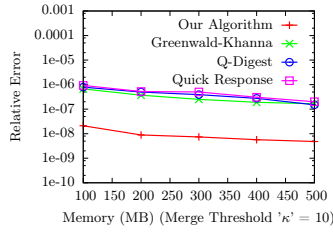
(b) Normal, size 101GB

(c) Wikipedia, size 58.5GB

(d) Network Trace, size 60GB

Figure 4: Accuracy (Relative Error) vs Memory
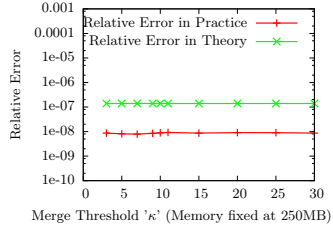

(a) Uniform Random

(b) Normal

(c) Wikipedia

(d) Network Trace

Figure 6: Update time vs memory, $\kappa = 10$

Overall, we found that the accuracy of our algorithm is substantially better (usually by a factor of more than 100) than that of a pure streaming algorithm, given the same amount of main memory.

In Figures 5a, 5b, 5c and 5d, we show the dependence of the accuracy on κ, labeled as "Relative Error in Practice". Keeping memory fixed at 250 MB, we vary κ from 2 to 30. We observe that the accuracy of the algorithm does not depend on the merge threshold κ, which is consistent with Theorem 2 which says that the accuracy depends only on ϵ and the size of the stream. We also compare with an upper bound on relative error derived from theory, labeled as "Relative Error in Theory", and observe that the observed error is significantly smaller than the theoretical bound.
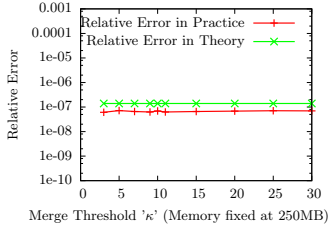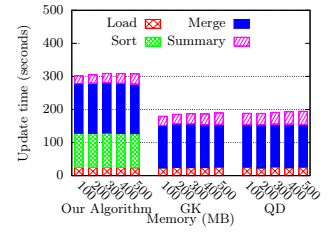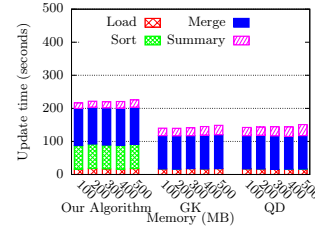

(a) Uniform Random

(b) Normal
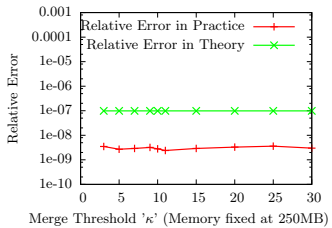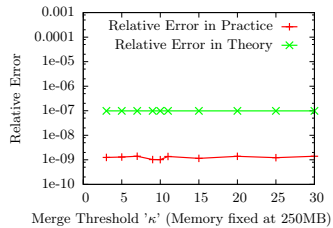
(c) Wikipedia

(d) Network Trace

Figure 5: Accuracy (relative error) vs merge threshold κ, memory fixed at 250MB

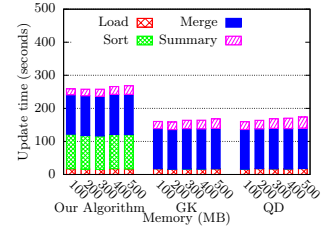*Update Time per Time Step.* Figures 6a, 6b, 6c, and 6d show the update time (for a single time step) as a function of the memory when memory is varied from 100MB to 500MB, while κ is fixed at 10. To understand where the time is being spent, we have broken down the update time into four components - (1) time to load new data to the data warehouse (2) time to sort data, (3) (amortized) time to merge historical data partitions according to Algorithm 3, and (4) time to update the in-memory summary. We observe that the time taken to sort and merge the datasets are the two most expensive components.

We also compare the update time of our algorithm with that of pure streaming algorithms GK and Q-Digest, shown in Figures 6a, 6b, 6c, and 6d. In the pure streaming approach, we use the same loading paradigm, i.e. loading new data to the data warehouse at every time step, and same partitioning scheme as used in our algorithm. This partitioning scheme ensures that the new data is loaded using minimal resource that is proportional to the size of the new data. It also ensures that older data are merged together so that they are handled in a similar fashion without having to access large number of partitions. However a pure streaming algorithm does not require the data set to be sorted and hence do not have to pay for sorting. We observe that the update time of our algorithm is about 1.5 times as much as the pure streaming algorithms. Time taken for loading and merging the data sets are similar for all algorithms. However, the time taken to generate the summary is slighly smaller in the case of our algorithm. This is so because the time taken for generating summary of our algorithm is mostly due to the stream summary. Summary for historical data is generated during loading and index generation.

Figures 7a, 7b, 7c, and 7d show the dependence of the update time as a function of κ, keeping memory fixed at 250MB. In each figure, the y-axis on the left side shows the update time in seconds, while the y-axis on the right side shows the number of disk accesses to update the historical data. Since these two numbers are closely correlated, we show them in the same figure. Indeed, we observe that the update time and the number of disk accesses show similar trends. We observe that in general, the update time decreases slightly as κ increases. The reason is that as κ decreases, merges of data partitions are more frequent. For instance, if $\kappa = 2$ and $T = 100$, the maximum number of levels is $\lceil \log_\kappa(T) \rceil =$
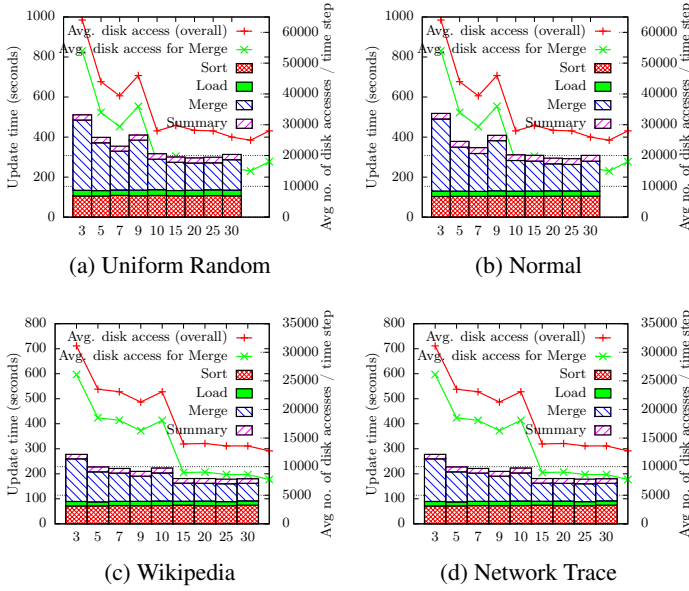
(a) Uniform Random (b) Normal

(c) Wikipedia (d) Network Trace

Figure 7: Update time and Number of disk accesses (per time step) vs merge threshold κ, memory fixed at 250MB

(a) Uniform Random (b) Normal

(c) Wikipedia (d) Network Trace

Figure 9: Query Runtime and Disk Accesses vs memory, $\kappa = 10$

$\lceil \log_2 100 \rceil = 7$ and the maximum number of merge operations that a single data item is involved in is 7. If $\kappa = 10$ and $T = 100$, the corresponding number is $\log_{10} 100 = 2$.

In the plots for update time as a function of κ, we see an anomaly at $\kappa = 9, 10$, where the number of disk accesses increases from $\kappa = 9$ to 10, and then decreases thereafter as κ increases further. To understand this, see Figure 8 which shows the cumulative frequency distribution of the number of disk accesses.

The x-axis of the figure shows the number of disk accesses per time step and the y-axis shows a percentage. Point (x, y) in this graph says that y percent of disk accesses took less than or equal to x steps. For the case $\kappa = 9$, the figure shows that in 89 percent of the time steps, disk accesses are made only to add the new batch to level 0 without any merging of partitions, and this takes 10K disk accesses. In 10 percent of time steps, level 0 partitions are additionally merged to a level 1 partition – this takes 190K disk accesses. In another 1 percent of the time steps, there is an additional merge from level 1 to level 2, requiring 1810K disk accesses. The peak at $\kappa = 9$ shown in Figure 7a, is explained by this one time step requiring 1810K disk accesses. Note that similar to the case for $\kappa = 9$, there is one time step out of 100 time steps for the case of $\kappa = 7$ (Figure 8), where level 0 partitions are merged to level 1 and level 1 partitions are merged to level 2, requiring 1130K disk accesses.
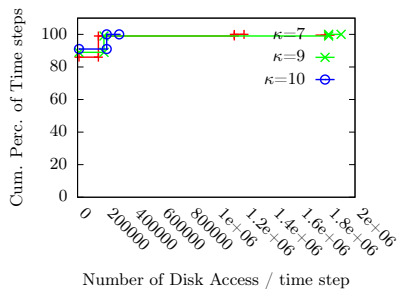

Figure 8: Cumulative percentage of the number of time steps where update less than or equal to the number of disk accesses / time step shown in X-axis, 100 total time steps, for Normal
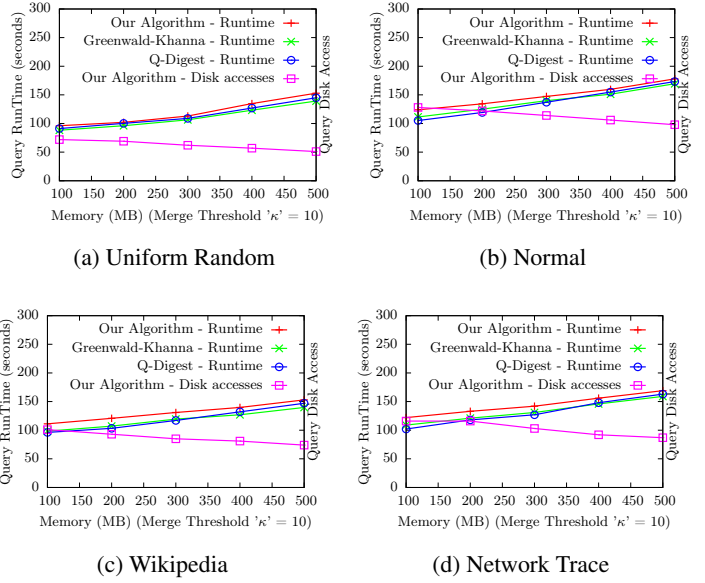
But this value is small compared to the number of disk accesses required by $\kappa = 9$. The single merge from level 1 to level 2 for $\kappa = 9$ will payoff for future time steps, but our experiments used only 100 time steps, which is why the cost for $\kappa = 9$ is anomalously high. On the other hand, for $\kappa = 10$, in 91 percent of time steps, a new dataset is added to the warehouse without any merging of data partitions, and in 9 percent of time steps level 0 partitions merge to level 1. The data partitions from level 1 do not merge to higher level and hence, the number of update disk accesses for $\kappa = 10$ is noticeably low compared to $\kappa = 9$. If we ran the experiments for a large number of time steps, then we expect to see a smooth decrease in the number of disk accesses with increasing κ.

Query Time. Figures 9a, 9b, 9c and 9d show the time taken by the algorithms to answer a query as a function of the amount of memory allocated, while keeping κ fixed at 10. The y-axis on the left side shows query time in seconds, and the right side shows the number of disk accesses for a query. We observe that the query time increases in general with the increase in the amount of memory allocated to the algorithms. We also observe that the query time of our algorithm is only slightly more than the query time of pure streaming algorithms. We observe that the total number of disk accesses decreases slightly with an increase in memory, but however, the overall query time still increases due to an increase in the time taken to process the stream summary during a query.

Figures 10a, 10b, 10c and 10d show the query time as a function of κ, while keeping the memory fixed at 250 MB. The y-axis on the left and right side show the time (in seconds) and the disk accesses, respectively. The increase in the number of disk accesses for increasing value of κ shows a similar trend as the query time. The explanation of the result is as follows. For a fixed amount of memory, the query time due to the stream summary does not change with κ. As κ increases, the number of partitions per level increases. Since the total memory is fixed, the size of the summary per data partition decreases, which leads to a larger number of disk accesses (for each partition) to answer a query accurately, and hence a larger runtime. We report the median of the disk accesses over 7 different runs of the algorithm.
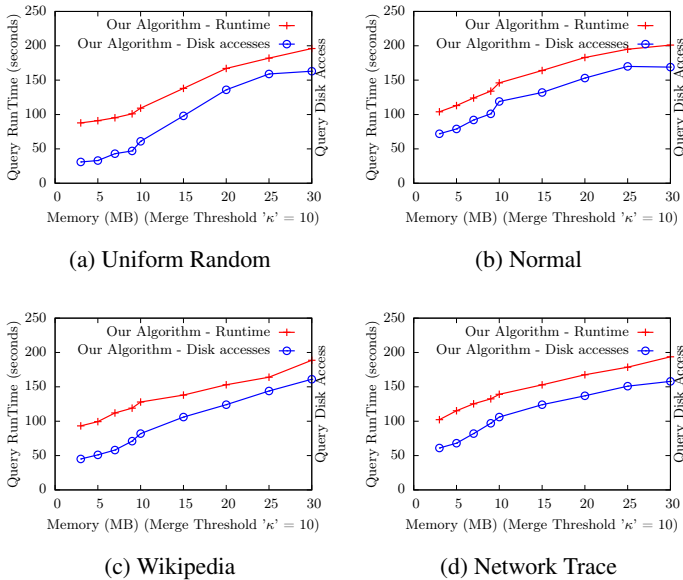
(a) Uniform Random

(b) Normal

(c) Wikipedia

(d) Network Trace

Figure 10: Query Runtime and Disk Accesses vs κ, memory fixed at 250MB

Queries over a Window. Figure 11 shows the possible window sizes, in terms of time steps, over which a query can be answered, for Normal dataset with 100 time steps. We show the graphs for $\kappa = 3$ and $\kappa = 10$. We observe that for $\kappa = 3$, a query can be made over windows of sizes 1, 4, 7, 10, 19, 45, 72, 100 time steps, whereas for $\kappa = 5$, the window sizes for querying are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 time steps. We observe that for larger values of κ, we have more window size selections because the number of merges are fewer. The number of disk accesses for answering queries increases with the window size, due to an increase in the size of the data within the window.

Scalability Test. In this set of experiments, we measured the variation of the accuracy and running time with the size of historical data and the size of data stream.

In the first experiment, we varied the size of the historical data while keeping the size of streaming data fixed. Figure 12a shows the relative error as the size of historical data is varied from 10GB to 100GB for the normal dataset, by keeping the number of time steps fixed at 100, and varying the size of the stream data added at each time step. For this experiment, we fix the stream size at 1GB, the main memory at 250MB, and set $\kappa = 10$. We observe that the relative error decreases with an increase in the size of the historical data. This is consistent with the theoretical guarantees given by Lemma 5, which shows that the absolute error in the rank of the algorithm does not vary with the size of the historical data,
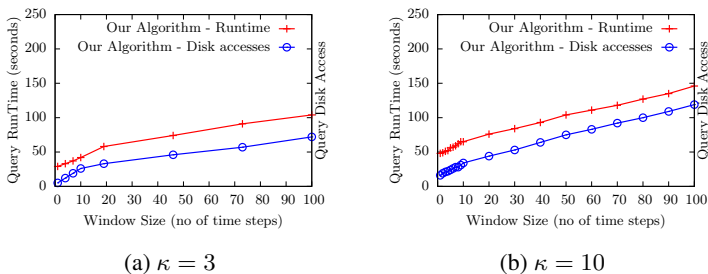

(a) $\kappa = 3$

(b) $\kappa = 10$

Figure 11: Runtime and number of disk accesses for a query vs the window size, Normal data, memory fixed at 250MB

and thus the relative error decreases as the size of the historical data increases. We observe from Figure 12b that the time taken as well as the number of disk accesses per time step for an update increases with the size of the historical data. This is also consistent with the theoretical guarantee in Lemma 6. We observe from Figure 12c that the time taken and the number of disk accesses for answering a query increases as we increase the size of the historical data set, which is consistent with the theoretical guarantee in Lemma 7.

In the final experiment, we varied the size of the data stream while keeping the size of the historical data fixed. Figures 13a, 13b and 13c show the relative error, update time, and the query time, respectively, as a function of the size of the stream, keeping the size of the historical data set fixed at 100 GB, main memory size fixed at 250 MB and $\kappa = 10$. We observe in figure 13a that the relative error of our algorithm increases linearly with the stream size. This is consistent with our theory in Lemma 5. We also observe that the time taken for an update and to answer a query do not vary with stream size. We also observe that the number of disk accesses, both for update and query (Figures 13b and 13c respectively), do not depend on the size of the stream. This is again consistent with Lemmas 6 and 7.

4 Conclusion

Many "real-time big data" scenarios require an integrated analysis of live streaming data and historical data. While there have been multiple attempts at designing data processing architectures and query processing strategies for such an analysis, query processing strategies for fundamental analytical primitives are lacking. We present a new method to process one of the most fundamental analytical primitives, quantile queries, on the union of historical and streaming data. Our method combines an index on historical data with a memory-efficient sketch on streaming data to answer quantile queries with accuracy-resource tradeoffs that are significantly better and more flexible than current solutions that are based solely on disk-resident indexes or solely on streaming algorithms. Our theory and experiments indicate that ours is a practical algorithm, potentially scalable to very large historical data.

Our work explores a natural three-way tradeoff between accuracy, memory, and the number of disk accesses. It is possible to improve accuracy by increasing memory usage while keeping the number of disk accesses fixed, through giving more memory to the stream summary. It is possible to reduce the number of disk accesses by increasing memory while keeping the accuracy fixed, through giving more memory to the index for enabling disk lookups and thus decreasing the range of disk-resident data to be processed at query time. It maybe possible (within limits) to reduce the number of disk accesses by reducing the accuracy while keeping the memory usage fixed, through stopping the search of the on-disk structure early, thus increasing the error in rank incurred due to disk-resident data. In this work, we have explored a portion of this space of tradeoffs. An interesting direction for future work is to explore this space of tradeoffs further, and also consider how to improve these tradeoffs through the use of improved data structures.

Another direction for future research is on parallel methods for processing data, maintaining the index, and answering queries. During query processing on historical data, different disk partitions can be processed in parallel, leading to a lower latency by overlapping different disk reads (assuming that the storage itself can support parallel reads). The on-disk index HD is updated using a set of sort and merge operations, which can potentially be parallelized using parallel sorting. Yet another direction for future work is to consider other classes of aggregates in this model of integrated processing of historical and streaming data.
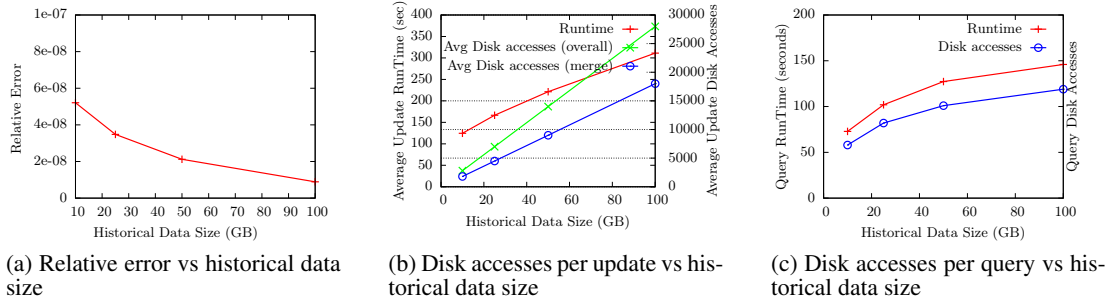
(a) Relative error vs historical data size

(b) Disk accesses per update vs historical data size

(c) Disk accesses per query vs historical data size

Figure 12: Accuracy and cost vs. historical data size, stream size fixed at 1GB, memory fixed at 250MB, $\kappa = 10$, Normal data

(a) Relative error vs stream size

(b) Disk accesses per update vs. stream size

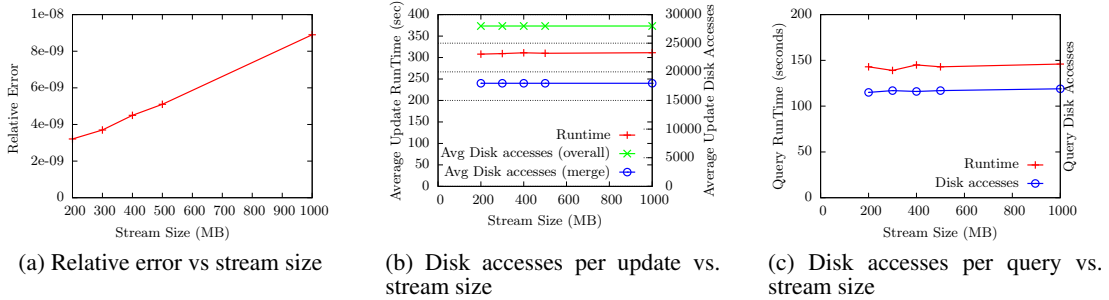(c) Disk accesses per query vs. stream size

Figure 13: Accuracy and cost vs. stream size, historical data size fixed at 100GB, memory fixed at 250MB, $\kappa = 10$, Normal data

5 Acknowledgments

6 References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, Sept. 1988.

[3] R. Agrawal and A. N. Swami. A one-pass space-efficient algorithm for finding quantiles. In *COMAD*, 1995.

[4] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee. Moirae: History-enhanced monitoring. In *CIDR*, pages 375–386, 2007.

[5] S. Chandrasekaran. *Query Processing over Live and Archived Data Streams*. PhD thesis, Berkeley, CA, USA, 2005.

[6] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD*, pages 436–447, 1998.

[7] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *SIGMOD*, pages 35–46, 2004.

[8] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *DEBS*, pages 243–254, 2011.

[9] N. Dindar, P. M. Fischer, and N. Tatbul. Dejavu: A complex event processing system for pattern matching over live and historical data streams. In *DEBS*, pages 399–400, 2011.

[10] U. Fiedler and B. Plattner. Using latency quantiles to engineer qos guarantees for web services. In *IWQoS*, pages 345–362, 2003.

[11] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Trans. Database Syst.*, 27(3):261–298, Sept. 2002.

[12] L. Golab and T. Johnson. Consistency in a stream warehouse. In *CIDR*, pages 114–122, 2011.

[13] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In *SIGMOD*, pages 847–854, 2009.

[14] G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), Sept. 2006.

[15] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, pages 58–66, 2001.

[16] T. Johnson and V. Shkapenyuk. Data stream warehousing in tidalrace. In *CIDR*, 2015.

[17] C. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. Comput.*, 32(10):942–946, Oct. 1983.

[18] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD*, pages 426–435, 1998.

[19] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *SIGMOD*, pages 251–262, 1999.

[20] J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.

[21] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, 1996.

[22] S. Peng, Z. Li, Q. Li, Q. Chen, W. Pan, H. Liu, and Y. Nie. Event detection over live and archived streams. In *Proc. WAIM*, pages 566–577, 2011.

[23] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling real-time querying of live and historical stream data. In *SSDBM*, pages 28–, 2007.

[24] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *SenSys*, pages 239–249, 2004.

[25] K. Tufte, J. Li, D. Maier, V. Papadimos, R. L. Bertini, and J. Rucker. Travel time estimation using NiagaraST and Latte. In *SIGMOD*, pages 1091–1093, 2007.

[26] L. Wang, G. Luo, K. Yi, and G. Cormode. Quantiles over data streams: An experimental study. In *SIGMOD*, pages 737–748, 2013.

[27] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.

[28] Q. Zhou, Y. Simmhan, and V. Prasanna. Towards hybrid online on-demand querying of realtime data with stateful complex event processing. In *IEEE BigData*, 2013.