IOWA STATE UNIVERSITY
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2015

Techniques for online analysis of large distributed data

Sneha Aman Singh
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

 Part of the Computer Engineering Commons

Recommended Citation
Singh, Sneha Aman, "Techniques for online analysis of large distributed data" (2015). *Graduate Theses and Dissertations*. 14907.
https://lib.dr.iastate.edu/etd/14907

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

www.manaraa.com

Techniques for online analysis of large distributed data

by

Sneha Aman Singh

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Srikanta Tirthapura, Major Professor

Daji Qiao

Doug W Jacobson

Suraj C Kothari

Tien Nguyen

Iowa State University

Ames, Iowa

2015

Copyright © Sneha Aman Singh, 2015. All rights reserved.

www.manaraa.com

DEDICATION

I would like to dedicate my thesis to my parents, Dr. Anil Kumar Aman and Chanda Aman Singh, my brother Kanishka Aman Singh, my sisters Pallavi Aman Singh and Rohini Singh Chatterjee, and my love Arko Provo Mukherjee

- for their unconditional love, support and encouragement.

www.manaraa.com

TABLE OF CONTENTS

www.manaraa.com

www.manaraa.com

www.manaraa.com

LIST OF TABLES

www.manaraa.com

LIST OF FIGURES

www.manaraa.com

www.manaraa.com

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof Srikanta Tirthapura for his continuous support, patient guidance and motivation. It was a privilege working under his mentorship. I have learned a great deal working with him. I appreciate him for guiding me through my academic and non-academic problems and showing me the right directions.

I also wish to acknowledge my POS Committee members, Dr. Daji Qiao, Dr. Suraj Kothari, Dr. Doug Jacobson and Dr. Tien Nguyen for their insightful comments that helped me understand my research problem from various perspectives. I really appreciate their ever-willingness to answer all my questions.

I would like to thank IBM for supporting my PhD for two consecutive years through IBM PhD Fellowship and for giving me a chance to pursue an internship with them. I am particularly grateful to Sam Ellis from IBM for being my mentor and giving me the opportunity to work with his team during my internship, and also for helping me improve my inter personal skills.

I would like to thank my fellow lab mates for all our academic and non-academic discussions. Working in the lab would not have been as much fun without them. I would also like to thank my friends in Ames for sharing all the happy and sad moments with me and for making my PhD journey so memorable - a special thanks to two of the most special people in my life, Arko and Kanishka.

I would like to acknowledge my entire family and particularly like to mention the Chatterjee family and Sinha family, because of whom I have a home away from home. I am thankful to them for always being there for me.

Finally, speaking of family, I wish to convey my sincere and heartiest gratitude to my parents, my brother Kanishka Aman Singh, my sisters Pallavi Aman Singh and Rohini Singh Chatterjee and my fianc Arko Provo Mukherjee, for their unconditional love and constant moral support. I am grateful to my parents for making me the person I am today. My father inspired

www.manaraa.com

me to be inquisitive and self dependent. Since childhood, I have seen my father try to figure out and fix his several small and big broken/dysfunctional things, that he deemed worth fixing, using minimal materials at his disposal. By involving me in his mini projects, he encouraged me to think out of the box and be creative. My mother motivated me to be ambitious, courageous and independent and instilled in me the true value of education. I am grateful to my parents for teaching me the importance and value of knowledge in their own unique ways.

www.manaraa.com

ABSTRACT

With the advancement of technology, there has been an exponential growth in the volume of data that is continuously being generated by several applications in domains such as finance, networking, security. Examples of such continuously streaming data include internet traffic data, sensor readings, tweets, stock market data, telecommunication records. As a result, processing and analyzing data to derive useful insights from them in real time is becoming increasingly important.

The goal of my research is to propose techniques to effectively find aggregates and patterns from massive distributed data stream in real time. In many real world applications, there may be specific user requirements for analyzing data. We consider three different user requirements for our work - Sliding window, Distributed data stream, and a Union of historical and streaming data.

We aim to address the following problems in our research : First, we present a detailed experimental evaluation of streaming algorithms over sliding window for distinct counting, which is a fundamental aggregation problem widely applied in database query optimization and network monitoring. Next, we present the first communication-efficient distributed algorithm for tracking persistent items in a distributed data stream over both infinite and sliding window. We present theoretical analysis on communication cost and accuracy, and provide experimental results to validate the guarantees. Finally, we present the design and evaluation of a low cost algorithm that identifies quantiles from a union of historical and streaming data with improved accuracy.

www.manaraa.com

CHAPTER 1. INTRODUCTION

With the advancement of technology, several domains, such as finance, health, energy, security, have become extremely dependent on data (or information). As a result, in recent years, there has been an explosion in the volume of data that is continuously being generated by applications in these domains. These large, continuous, rapid and generally unbounded flow of data are called data stream [64]. Examples of data stream include internet traffic data, sensor readings, tweets, stock market data, telecommunication records.

Real time analysis of data stream has been recognized as extremely insightful and important for many applications. A few examples of such applications are cyber security (for instance, identifying network attacks such as worm propagation and DDoS attack [82]), stock market (stock price prediction), Internet of Things (decision making in sensors), telecom (fraudulent detection). Consequently, there has been a lot of study on solving fundamental as well as complex problems over a data stream. Some of the example problems are counting unique elements [30, 3, 6, 38], identifying quantiles [63, 46, 73], identifying heavy hitters (or frequent items) [62, 57, 17], identifying persistent items [41, 55].

However, even the most fundamental database problem, such as count distinct [30] or percentile [63], gets harder as the size of data grows. These problems especially become challenging in a streaming context due to the dynamic nature of data stream and its concept drifting, i.e properties of data continuously evolving with time. Given that the memory size of a stream processing system is very limited compared to the size of a data stream that it processes, it is extremely hard to solve most of the streaming problems in real time. Hence, the state-of-the-art research on solving streaming problems provide a trade-off between the memory requirement of their proposed algorithms and accuracy of the results given by these algorithms.

www.manaraa.com

In our work, we have proposed algorithms to identify interesting events or compute aggregates from a large scale data stream in real time and performed experimental evaluation of these algorithms to demonstrate their performance in practice. In many real world applications, these streaming problems are required to be solved for specific user requirements. We have considered three such requirements for our work: Sliding Window [22], Distributed Data Stream [38] and Union of historical data with live data stream. They have been discussed below in details:

1.1 User Requirements for Data Analytics

In this section we describe the three user requirements that we have considered to solve important problems on large data.

1.1.1 Sliding Window

Window is a concept used in data stream analytics to limit the scope of data on which analysis is performed. Sliding window model [22, 38, 42, 12, 67, 8, 33] is used to restrict the scope to the most recently observed data elements. There are two commonly considered types of sliding windows, "count-based" and "time-based" window. A count-based window of size W is the set of W most recent elements in the stream. A time-based window of size T is the set of all stream elements that have arrived within T most recent time units. Depending on the application, the expired elements, i.e. elements which are no more a part of the current window, are either discarded or archived.

Aggregation over a sliding window arises naturally in real-time monitoring situations such as network traffic engineering, telecom analytics, and cyber security (e.g. [22, 32, 38, 42, 9, 78, 7]). For instance, in network traffic engineering ([31]), current network performance is monitored over a sliding window to adjust the bandwidth of the network dynamically. The abstraction of a sliding window is well accepted today and has found its way into the query processing interface of major stream processing systems, including IBM Infosphere Streams ([65]) and Apache Spark Streaming ([90]). For instance, in IBM Infosphere Streams, it is possible to apply each stream aggregation operator (including distinct counting) over a sliding window.

www.manaraa.com

The major challenge involved with analyzing data stream in the context of a sliding window is that the deletion of the expired elements from current window is implicit [8], i.e it is not possible to identify the deleted elements of a data stream without storing the entire window.

We use the term "infinite window" for a scenario where the scope of analysis is the entire data stream observed so far.

1.1.2 Distributed Data Stream

Motivated by distributed network monitoring, we consider monitoring of data stream that is distributed across multiple sites, so that no single processor observes all the data. A single node can only observe the local stream, and the target of monitoring is the logical stream that is formed through the union of all local streams. Monitoring each individual stream in isolation may not yield the desired insights, and the interaction between the different streams needs to be considered carefully. This setup is called the distributed streams model [37, 38, 20, 76, 56, 21, 79]. Web log analytics [27, 52, 69] is an example application that can be modeled as a distributed stream. In this system, there are multiple web servers each of which have their own log of web accesses, which is a (large) locally observable stream, but patterns such as typical user behavior, and anomalies that could lead to malicious user behavior, have to be detected on the union of the distributed web logs.

We consider the following distributed streaming model, that has also been adopted in prior work [37, 38, 20]. The distributed system has k sites, numbered from 1 to k; each site i receives a local stream $\mathcal{R}_i$. There is a special coordinator site that communicates with the individual sites and is required to perform all aggregation and mining tasks in the (logical) stream $\bigcup_{i=1}^{k} \mathcal{R}_i$ formed by the union of all streams. The challenge here is to minimize the communication cost between the coordinator and the local nodes, while analyzing the distributed stream.

As opposed to the above defined model, when a single node observes and processes the entire logical data stream, it is referred to as *Centralized Model*.

www.manaraa.com

1.1.3 Union of Historical and Streaming Data

In many applications, after the real time analysis of a live data stream, it is archived as historical data in a data warehouse for later and deeper analysis by batch processing system. Recently, researchers have recognized the importance of integrating the data stream processing with historical data processing for enabling longer term analysis of data, and utilizing past data to bring more context to the current data [5, 43]. This integration has been considered significant in complex event processing (CEP) [24] and can also be used for predictive analysis and correlation of streaming and historical data.

We consider a setup where a data stream is captured and processed in real time. The data stream is then loaded into a data warehouse. The loading and indexing of data is done periodically in batch, in intervals of a "time step". For instance, a time step maybe one day or a week. Data that is not loaded into the warehouse yet is referred to as the "streaming data" or "data stream".

Some of the example applications, where analysis of the union of historical and streaming data, is useful, are network monitoring for intrusion detection [70, 5], financial trading and real-time bidding [94], traffic monitoring [81].

Current literature mostly focus on developing efficient architectural model [91, 44] for data integration [68, 23, 24, 81, 5, 1, 11, 91]. The paper [44] proposes an architectural framework of streaming warehouse called DataDepot, designed to store streaming data, thus allowing analysis of massive amount of historical data over a time frame of many years. The paper [91] proposes a model which enables data analysis over the union of historical and streaming data.

1.2 Hypothesis

It is possible to design, implement and evaluate low cost streaming algorithms to find patterns such as persistence from a distributed data stream over both infinite and sliding window and to compute aggregates such as quantiles from an integration of historical and streaming data, with proven guarantees on performance and accuracy, in real time. It is also possible to perform an experimental evaluation of streaming algorithms such as that for distinct counting, to find the algorithm that performs the best in terms of accuracy and runtime.

www.manaraa.com

1.3 Contributions

We addressed the following problems as part of my research goal:

1.3.1 Evaluation of Distinct Count Streaming Algorithms over a Sliding Window [75]

Distinct counting is the problem of computing the number of distinct or unique elements in a data stream. It is a fundamental problem in databases with a wide variety of applications in database query processing and optimization ([72, 85, 89, 84, 34]) and network monitoring ([80, 54, 74]). It is one of the earliest problems studied in the area of streaming algorithms.

An example application of distinct counting in network monitoring is to track the number of distinct network connections established by a source IP address. Tracking sources that establish a large number of distinct connections can help identify network anomalies such as worm propagation and DDoS attacks ([82]). Since a network monitor has to simultaneously monitor a number of sources, it cannot afford to use much memory for each source, and needs a small-space data structure for counting the number of distinct identifiers per source. Further, it is necessary to count the number of distinct identifiers within a subsequence of the stream consisting of the most recently observed elements, commonly modeled using a "sliding window" in the stream.

From a theoretical perspective, distinct counting is widely studied e.g. [30, 3, 37, 6, 35, 13, 47, 50, 25, 32, 40, 84, 86, 29, 16]. However, there has not been much attention to engineering a good implementation. Most current algorithms for distinct counting over a stream (e.g [30, 3, 37, 6, 13, 47, 50, 25, 32, 40, 84, 86]) have been designed for the case of "infinite window", where the scope of aggregation is all the elements seen so far. There have been some algorithms designed for a sliding window (e.g [38, 22, 93]), but so far, there has not been a comprehensive evaluation and comparison of different approaches.

We present the first detailed experimental evaluation of algorithms for distinct counting over a sliding window. We consider prominent algorithms and evaluate them with respect to their memory consumption, processing time, query time, and accuracy. In some cases that we

www.manaraa.com

considered, it was known previously how to extend the algorithm to a sliding window, while in other cases, we design an extension to a sliding window of a distinct counting algorithm originally designed for the infinite window.

1.3.2 Identification of Persistent Items from Distributed Streams [75]

We address the identification of a feature called a "persistent item" from a massive distributed data stream. A persistent item is one that occurs regularly in the stream, but does not necessarily contribute significantly to the volume of the stream. If a stream is divided into n equal timeslots, then persistence of an item is the number of distinct timeslots out of the n timeslots, where the item appeared in the stream. A persistent item in a distributed set up is defined as the number of distinct time slots where the item appeared in the union of all the streams. Note that multiple occurrences of the same item in the same timeslot, whether at the same site or at different sites, do not contribute repeatedly to the persistence of the item.

Persistence is typical of many stealthy types of traffic on the Internet. Identifying persistent items can help in identifying anomalous and potentially malicious behavior in a network. For instance, Giroire et al. [41] showed that tracking all persistent destinations arising in traffic from end hosts in a domain led one to identify botnet Command and Control (C&C) destinations. The C&C destinations take control over compromised end hosts to create a botnet and carry out malicious activities in the network. Giroire et al. observed that the C&C centers had to be in regular contact with the compromised end hosts to carry out their activities, and hence the persistence of the C&C destinations were high in the streams emanating from the end hosts. However, in order to evade detection by traditional volume-based anomaly detectors, the C&C traffic was designed to be low-volume and hence the C&C centers did not show up as heavy-hitters within the streams. Another instance is in Pay-Per-Click Online Advertising [1], where identifying persistent items can be used to detect click fraud [92]. In this instance, rival companies generate false clicks on advertisements at regular but infrequent intervals. In order to evade detection by volume-based detectors, the volume of such false clicks is kept low, and hence these do not appear as heavy hitters in the click stream.

[1] http://en.wikipedia.org/wiki/Pay_per_click

www.manaraa.com

In general, persistence captures behavior when a set of entities (perhaps controlled by a malicious user) together have regular communication with a remote entity, but try to hide the communication by keeping its volume small and having it originate from different entities at different times. Such behaviors are not caught by tracking frequent items within a stream.

The goal of this work is to devise an algorithm for identifying persistent items, which minimizes (1) the communication between the processors and (2) the memory footprint of the algorithm, both per node, and overall.

1.3.3 Quantile Computation from a Union of Historical and Streaming data

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 $\texttt{rank}(e, D)$, is defined as the number of elements in D that are less than or equal to e.

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

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 the monitoring of web server latency [28]. 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) [18]. Such quantile computations are a key functionality provided by many Data Stream Management Systems (DSMS), such as GS Tool [18], that provide support for real-time alerting over high-velocity streaming data generated by modern enterprises.

www.manaraa.com

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 [70, 5], financial trading, real-time bidding [94], and traffic monitoring [81]. To address the demands of such applications, *data stream warehousing systems*, such as TidalRace [49], 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 [5, 43]. Such an integrated processing has been considered significant in complex event processing (CEP) [24] and is used for predictive analysis and correlation of streaming and historical data.

However, there has been little work on query processing methods for the union of historical and streaming data. While there is a vast literature on query processing on purely streaming data, and for query processing on stored data based on indexes, these methods do not work directly for integrated processing of streaming and historical data. Our work takes a first step in this direction of designing integrated query processing methods for historical and streaming data by considering the estimation of quantiles, one of the most fundamental analytical primitives.

www.manaraa.com

CHAPTER 2. AN EVALUATION OF STREAMING ALGORITHMS FOR DISTINCT COUNTING OVER A SLIDING WINDOW

Let S be a stream of identifiers, each chosen from a universe U. We consider the problem of maintaining the number of distinct identifiers in S in a single pass through S using limited memory, a problem we henceforth refer to as "distinct counting". In this work, we consider the efficient implementation of distinct counting over a sliding window of a stream.

2.0.3.1 Goal

We present the first detailed experimental evaluation of algorithms for distinct counting over a sliding window. We consider prominent algorithms and evaluate them with respect to their memory consumption, processing time, query time, and accuracy. In some cases that we considered, it was known previously how to extend the algorithm to a sliding window, while in other cases, we design an extension to a sliding window of a distinct counting algorithm originally designed for the infinite window. We set out to answer the following questions.

- How do different algorithms compare in terms of accuracy and processing time, given the same amount of main memory?
- Most algorithms for distinct counting work as follows. They first design a "rough" estimator whose output is a random variable, but whose error can be large. Then, many such estimators are aggregated in order to improve the accuracy. A few different methods are used for boosting accuracy, including"median of many", "split and add", and "stochastic averaging"; these methods are described in Section 2.3. Which aggregation method is suitable for each algorithm?

www.manaraa.com

- Every algorithm known for distinct counting uses a hash function that maps input identifiers, which maybe non-uniformly distributed within the input universe, to another universe, where they are uniformly distributed. The hash function has a significant impact on the accuracy and the runtime. Which hash function gives the best performance?

- How is the performance of an algorithm affected by the relative frequencies of queries (for the number of distinct elements) versus element arrivals?

2.1 Contributions

We present the first experimental evaluation of algorithms for distinct counting over a sliding window. **Algorithms:** We consider the following prominent algorithms for distinct counting: Randomized Wave (RW) [38], Probabilistic Counting with Stochastic Averaging (PCSA) [30, 22], Linear Counting (LC) [84], Loglog (DF) [25], and the first algorithm due to Bar-Yossef et al. [6], which we call BJKST1. Among these, RW is the only algorithm which was designed for distinct counting over a sliding window. Though PCSA was originally designed for an infinite window, an extension to a sliding window was described in [22]. For the rest of the algorithms, LC, DF, and BJKST1, we present an extension for the case of a sliding window.

Hash Functions: Given a set of distinct inputs, an ideal hash function generates mutually independent random numbers as its output, but we do not know of such an ideal hash function , and hence use functions that have been empirically observed to work well. We compared five popular hash functions, MurmurHash [1], Jenkins Hash [2], modulo congruential hash, Fowler-Noll-Vo (FNV) [3] hash and the Secure Hash Algorithm 1 (SHA-1) [4]., to identify the one that performs best for our purpose.

Accuracy Boosting Method: We compared the performance of each algorithm combined with different methods for boosting accuracy. In the popular "Median-of-k" approach, proposed for use in [37, 38, 50] many independent instances of the algorithm are run, and the final estimate is the median of the estimates returned by all instances. In "Split-and-Add", the

[1] https://sites.google.com/site/murmurhash/
[2] http://www.burtleburtle.net/bob/hash/doobs.html
[3] http://www.isthe.com/chongo/tech/comp/fnv/
[4] https://tools.ietf.org/html/rfc3174

www.manaraa.com

universe is partitioned into k non-overlapping sets of approximately equal size using a hash function, which induces k substreams of the original stream. These substreams are processed using different estimators. The final estimate is obtained by adding the different estimates from individual instances. In "Stochastic Averaging", used in [30, 25], the universe is partitioned into k non-overlapping intervals using a hash function. The final estimate is obtained by computing a certain function over a substream corresponding to each partition, averaging these over all partitions, and then finally computing a different function over the average. Stochastic averaging is further described in Section 2.3.1.1.

Relative Frequency of Queries versus Updates: Some algorithms, such as LC, perform well when queries are infrequent, and their performance degrades when queries become more frequent. We investigated the impact of the frequency of queries by comparing the performance under different mixes of query/update.

2.1.1 Summary of Results

- **Accuracy:** Given equal memory on the same dataset, we observed that the Randomized Wave (RW) consistently produces the most accurate estimate, followed by PCSA and then BJKST1. We also observe that Linear Counting (LC) performs with good accuracy when the memory allotted is large relative to the number of distinct elements within a sliding window; when the memory allotted is smaller, LC is unable to produce a reasonable estimate.

- **Runtime:** When the frequency of updates (element arrivals) is much larger than the frequency of queries, PCSA and DF are the fastest algorithms, followed by RW and BJKST1. However, if the frequency of queries increases, then the runtimes of PCSA, DF, and LC increase significantly, and RW and BJKST1 are the fastest algorithms.

- **Hash Function:** We found that all algorithms consistently give the most accurate estimates when *MurmurHash* is used as the hash function. Fortunately, it is also the fastest of all five hash functions that we considered, so that MurmurHash is unambiguously the best hash function among those we considered. While the accuracy of Jenkins hash is

www.manaraa.com

close to MurmurHash, it is slower than MurmurHash. The popular modulo congruential hash function performs much worse than MurmurHash and Jenkins hash, in terms of accuracy.

- **Accuracy Boosting Method:** We observed that PCSA and DF work best with stochastic averaging. This is to be expected, since PCSA and DF were designed with Stochastic Averaging in mind. Surprisingly, we found that the remaining algorithms (LC, RW, and BJKST1) performed with the smallest average error when the entire space is allotted to a single instance of the algorithm, with no further boosting of accuracy.

  Note that our comparison keeps the total space fixed for different accuracy boosting methods. For instance, if we used the median of five estimators as our accuracy boosting method, then the space allocated to each instance of the algorithm is only a fifth of the total space. Thus, our results do not contradict earlier results due to [6] and [38], who advocate using the median of many estimators. Their observation is that the probability of being inaccurate can be reduced by taking the median of many estimators, at the expense of greater space. Our experiments show that if space is held fixed, then the smallest average error is achieved when the entire space (memory) is given to a single estimator.

Overall, if accuracy is the most important criterion, then RW performs best. RW is also the fastest algorithm when the rate of updates is low relative to the rate of queries (approximately, less than 100 updates per query). PCSA is the best choice if processing time is the most important criterion and the rate of querying is not very frequent.

2.2 Related Work

Prior work on experimental evaluations of distinct counting include [4, 26, 60, 71], who compare the performance of different algorithms for distinct counting such as PCSA and Linear Counting over an infinite window. The most detailed comparison for distinct counting algorithms over infinite window seems to be due to [60], who grouped the algorithms into different categories: Logarithmic Hashing e.g PCSA ([30]), Interval-based e.g BJKST1 ([6]),

www.manaraa.com

Pure Bucket-based e.g. Linear Counting ([84]), Hybrid Bucket Sampling e.g. Distinct Sampling ([37]), Hybrid Bucket Logarithmic e.g Multiresolution Bitmap ([26]), and concluded that Linear Counting is overall the best algorithm, both in terms of accuracy and runtime.

Our work differs from that of [60] in the following ways. Mainly, we consider aggregation over a sliding window while they consider aggregation over an infinite window. The algorithms involved are different, and the results that we obtain are also different. In particular, we observe that Linear Counting (LC) does not perform very well over a sliding window. The accuracy of LC over a sliding window is very inconsistent for our datasets when the memory used is less than 1000-2000KB; in some cases it does not even produce an estimate. In contrast, the accuracy of RW and PCSA are consistently within 1 percent, even when the total memory is less than 1000KB. The difference in results between the sliding window case and the infinite window case is because the sliding window data structure needs to maintain a timestamp (of expiry) for each bit in the data structure maintained by LC. This overhead significantly increases the space required by LC to maintain an estimate of the distinct count, and consequently decreases its accuracy for a given space budget. In addition, our evaluation considers important decisions such as the choice of hash function, and the accuracy boosting method. All implementations in [60] used the modulo congruential hash function; our experiments show that other hash functions perform much better. Further, different accuracy boosting methods are not explored. The size of datasets that we consider (up to 100 million distinct elements) are much larger than in the experiments of [60] (approximately 2 million distinct elements).

2.3 Materials and Methods

There are two types of sliding windows commonly considered, count-based window and time-based window. A count-based window of size W is the set of the W most recent elements in the stream. A time-based window of size T is the set of all stream elements that have arrived within the T most recent time units. We consider a time-based window, since a count-based window is a special case of a time-based window. An algorithm for a time-based window can also be used for a count-based window by setting the timestamp to be equal to the stream position.

www.manaraa.com

2.3.1 Algorithms

We present an overview of the algorithms that we consider. For the following discussion, we assume that the domain of elements is $[N] = \{1, 2, \ldots, N\}$, and that N is a power of 2. Each element of the stream is a tuple (e, t), where $e \in [N]$ and $t \geq 0$ is an integer timestamp. We assume that timestamps are in a non-decreasing order, but not necessarily consecutive. When a query is posed at time t, the requirement is to estimate the number of distinct elements within a timestamp based sliding window of size T, i.e. those elements with timestamps r such that $(t - T + 1) \leq r \leq t$.

2.3.1.1 Probabilistic Counting with Stochastic Averaging (PCSA)

We recall the PCSA algorithm for an infinite window [30]. The algorithm maintains a bit vector B of size $\log_2 N$. It uses a hash function $h : [N] \rightarrow \{1, 2, \ldots, \log_2 N\}$, such that for each $e \in [N]$, and $b \in \{1, 2, \ldots, \log_2 N\}$, $\Pr[h(e) = b] = 2^{-b}$. Initially, all bits of B are set to 0. When an element e arrives, $B[h(e)]$ is set to 1. The intuition is that approximately 2^i distinct elements must be seen before $B[i]$ is set to 1. When there is a query for the number of distinct elements, the bits of B are scanned from position 1 onwards, to find the index of the lowest bit x that is not set. The estimate returned is $1.29281 \times 2^{x+1}$.

To adapt this to a sliding window, we use ideas from [22] and [93]. Instead of a bit vector B, we use a vector M of length $\log_2 N$, indexed from 1 till $\log_2 N$, to store timestamps. Initially, all entries of M are set to 0. When an element (e, t) arrives, $M[h(e)]$ is set to t. Note that $M[i]$ tracks the latest timestamp at which an element hashes to index i. When there is a query for the number of distinct elements within a time-based sliding window of size T, the algorithm scans M to find the smallest index x such that either $M[x]$ is 0, or the timestamp of x has expired, i.e. $M[x] < (t - T + 1)$, where t is the current time. The estimate returned is $1.29281 \times 2^{x+1}$, as before.

We implement an enhancement of the above basic scheme, based on stochastic averaging (PCSA), also proposed in [30]. In PCSA, k copies of the above data structure are used. Input elements are first partitioned into k non-overlapping groups, using a hash function g; an

www.manaraa.com

element (e, t) is forwarded to one of the k data structures, according to $g(e)$. The final estimate is $1.29281 \times k \times 2^{\hat{x}+1}$, where $\hat{x}$ is the average of the individual xs obtained from the k different data structures. Similar to PCSA for an infinite window, the processing time per element of PCSA for a sliding window is $O(1)$ and the query time is $O(k \log N)$. Suppose that a timestamp can be stored in $\mathcal{T}$ bits. The space taken by the sliding window version is $O(\mathcal{T} k \log N)$ bits, which is a factor $\Theta(\mathcal{T})$ larger than the infinite window version which takes $O(k \log N)$ bits of space.

AMS, due to [3] is another algorithm for distinct counting for an infinite window, with the same intuition as PCSA. Though AMS provides a cleaner theoretical guarantee than PCSA, PCSA has been found to be more accurate in practice than AMS, for example, as in the evaluation by [35].

2.3.1.2 Linear Counting (LC)

Linear Counting, due to [84], uses a bit vector B of size $n = D_{max}/\rho$, where D_{max} is an upper bound on the maximum number of distinct elements in the data stream, and ρ is a constant called the "load factor". The algorithm uses a hash function $h : [N] \rightarrow \{1, 2, \ldots, n\}$ such that for each $e \in [N]$, and $b \in \{1, 2, \ldots n\}$, $\Pr[h(e) = b] = 1/n$. Initially, all bits in B are set to 0. Each element e of the data stream is uniformly and independently hashed to an index in the bit vector, and the corresponding bit is set to 1. When a query is made, the number of distinct elements is estimated as $m \ln(n/m)$ where m is the number of bits in B that are still 0. [84] show that accurate estimates can be obtained when $\rho \leq 12$. However, when ρ is significantly larger than 12, the estimates are poor due to a large density of 1s in the bit array.

We extend the above to a sliding window as follows. Instead of a bit vector, we use a vector of timestamps, M, of size n, indexed from 1 till n. When element (e, t) arrives, $M[h(e)]$ is set to t. When a query is made for the number of distinct elements within the window, the entire vector M is scanned and the number of indices that either have value 0 or whose timestamps have expired, is used instead of m in the above formula. Note that the processing time per element is $O(1)$ and the time to answer a query is $O(n)$, which is expensive since n is linear in the number of distinct elements. The total time is still reasonable if the frequency of queries is

www.manaraa.com

small when compared with the frequency of updates (infrequent queries), but poor if queries are more frequent.

For the case of frequent queries, we modified LC by introducing a data structures in addition to M – a list L that comprises of tuples of the form (t, a) and is ordered according to t, the time stamp of observation, and a is the value to which the element hashes to. In the vector M, in addition to a timestamp t, there is also a pointer to the occurrence of t in L, so that if an element with a new timestamp hashes to an index in M, the corresponding entry with older timestamp is deleted from the list, and the newer entry with current timestamp is made at the head.

The modified version of LC, which we call "LC2", not only requires $2\mathcal{T}$ bytes to maintain two copies of timestamp per index of M, but also an overhead for maintaining an list, which can be twice the pointer size in a typical implementation such as the C++ Standard Template Library. The expired timestamp is determined from the tail of L in constant time. A single variable can keep track of the number of indexes with expired timestamps or with an initial value of zero. When a query is posed, the number of relevant bits can be determined in $O(1)$ time. Overall, we get $O(1)$ time for update as well as a query, but at the cost of a significant space overhead. A significant drawback of LC is that ρ cannot exceed 12 ([84]), so that the space used by the algorithm is at least $D_{max}/12$. The accuracy of the estimate falls drastically as ρ increases.

2.3.1.3 BJKST1

BJKST1 is the first algorithm in [6]. We first describe the infinite window version and then present an adaptation to a sliding window. Each stream element is hashed uniformly using a function $h : [N] \rightarrow [N^3]$. At each instant the algorithm maintains the τ smallest hash outputs, for some τ that depends on the desired accuracy. When a query is posed, an estimate of distinct count is returned as $\frac{\tau N^3}{v_\tau}$, where v_τ is the τ-th smallest hash output.

We propose the following adaptation to the sliding window. We associate with each value among the τ smallest hash outputs, a timestamp equal to the most recent time when this value was observed. It is not possible to maintain τ-th minimum of hash outputs exactly in a

www.manaraa.com

sliding window using a bounded space for a fixed value of τ (as discussed in [22], maintaining the minimum over a sliding window requires linear space in the worst case). So we vary the value of τ so that the algorithm uses a bounded space to estimate distinct count. As the hash outputs of data elements are generated randomly, the algorithm uses an expected space cost of $O(\frac{\log N}{\epsilon^2})$ to estimate distinct count, where we set the maximum value of τ to a constant θ which depends on the space allocated to the algorithm. Our idea is influenced by the algorithm for computing minimum element over a sliding window in [22, 32]

We maintain a list L of (h(e), t) tuples, where e is the element ID observed at timestamp t. When a new element (e, t) is observed, all the elements e' with $h(e')$ greater than $h(e)$ are deleted, and $(h(e), t)$ is inserted at the head of the list. Note that in some cases, as a consequence of this deletion, the size of the list may even reduce to 1 (consider the case when element (e, t) has the smallest value of hash output $h(e)$ in current window). If the size of the list is at least θ, we set the value of τ as θ to compute the number of distinct elements, else we set it to the current size of the list.

Thus, at any point of time, the list is ordered by both timestamp and hash value of the element ID, i.e for a sequence of elements $(e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n)$, $h(e_1) < h(e_2) < \ldots < h(e_n)$ and $t_1 < t_2 \ldots < t_n$, and this allows us to retrieve the τ-th smallest hash values within the window.

We did not implement the second and third algorithms in [6] for the following reasons. These algorithms theoretically use slightly smaller space than BJKST1, but there are additional factors hidden in the $\tilde{O}$ notation, as well as large constant factors, so practically their space requirement is much larger, as also analyzed in [60]. Both algorithms suppress factors involving $\log(1/\epsilon)$ and $\log\log(N)$ factors from the space cost. The algorithm by [32] is similar to the one by BJKST1, but does not give a smooth trade-off between space and relative error, like in BJKST1. The algorithm due to [50] is theoretically space-optimal and can potentially be extended to sliding windows. But we are not aware of an implementation of this algorithm, even in the infinite window case.

www.manaraa.com

2.3.1.4 Durand-Flajolet (DF)

The Loglog algorithm by [25] for infinite window derives its name from the space cost of the algorithm which is $O(\log \log N)$. However, the sliding window version of the Loglog algorithm does not have a space complexity of $O(\log \log N)$, due to the need to maintain timestamps, and is more expensive. Hence, the name "Loglog" is not applicable here, and we simply call it the "DF algorithm".

The algorithm hashes each stream element to a binary string y of length $O(\log N)$, and finds the rank of first 1-bit from the left in y, $r(y)$. It finds the maximum $r(y)$, say r, over all stream elements. This requires only $O(\log \log(N))$ space, since a single variable needs to be maintained to keep a track of the maximum. Similar to PCSA, DF uses I different bit vectors, and does a stochastic averaging to find the average of maximum r from all bit vectors. When a query is posed, the estimate of the distinct count is returned as $0.39701 \times I \times 2^{(avg(\max(r))+1)}$.

However, in the sliding window case, there is no easy way to maintain r over all bits set by active elements, since this value is not a non-decreasing number, like in the case of infinite window. Instead, similar to the PCSA algorithm, we use a vector, M, of length $\mathcal{T}$ to store timestamps. In particular, each index i of the vector M maintains the most recent timestamp during which an element was hashed to y, such that $r(y) = i$. The space taken by this data structure is no longer $O(\log \log N)$. In answering a query, $\max(r)$ is determined as the rank of the highest index in M which contains a non-expired timestamp.

Super Loglog ([25]) and HyperLogLog ([48]) are modifications of Loglog that use smaller bit vectors to reduce the space cost of the algorithm. See the study on engineering a distinct count algorithm by [48] for further details. However, these modifications do not payoff in the sliding window scenario due to the additional cost of maintaining the timestamps, which dominate the memory cost and negate the advantage due to smaller bit vectors.

2.3.1.5 Randomized Wave (RW)

The RW algorithm by [38, 39] is based on sampling via a hash function. A hash function $h : [N] \rightarrow [0, \ldots \log_2 N]$ is used that maps elements to levels as follows: the probability of an

www.manaraa.com

element being assigned to level j is $2^{-(j+1)}$. At each level i, the algorithm maintains a (doubly linked) list of elements (e, t) L_i ordered by the timestamp t of observation. Element e when observed at time t is inserted into lists $L_0, L_1, L_2, \ldots L_{h(e)}$. If the element has already appeared in L_i, then its timestamp is updated to equal the current time. To determine if an element has been observed in the current window, an additional data structure, a hash map, is used, with the element identifier as the key, and the pointer to its occurrence in L_i as the value. If a level becomes full (i.e. its size exceeds the budget allotted to it), then the oldest elements in the level are deleted from the hash map as well as the list. Further, when an element expires from the sliding window, it is also discarded from the data structure. Discarding oldest elements is a constant time operation because the oldest elements are stored at the tail of the list.

When a query is made, the lowest numbered level which contains the entire current sliding window is determined, say ℓ. The estimate of the number of distinct elements is computed as $2^{\ell}|S_{\ell}|$, where S_{ℓ} is the set of all elements in level ℓ. We have optimized the above algorithm by inserting element e into only level $h(e)$, rather than all levels from 0 to $h(e)$. This improves the processing time for an element by roughly a factor of two, while somewhat increasing the query time, since in order to process a query the algorithm needs to consider elements in all levels $0 \ldots \ell$, rather than only at level ℓ.

The adaptive sampling algorithm by Wegman(see [29] for a description) is another algorithm originally designed to compute distinct count over an infinite window. We note that if this algorithm is adapted to a sliding window setting, the result is an algorithm similar to RW.

2.3.2 Accuracy Boosting Methods

In the "median of many" approach, used in RW and BJKST1, k independent copies of the algorithm are run in parallel on the input stream, and the final estimate is the median of the estimates returned by the k different copies. In "split and add", the universe of input identifiers is partitioned into k non-overlapping sets of approximately equal size using a hash function. This induces k non-overlapping substreams of the original stream, each of which is processed separately by individual copies of the algorithm. The final estimate is obtained by adding the estimates produced from the k copies. In "Stochastic Averaging", used in PCSA and DF, the

www.manaraa.com

universe is partitioned into k non-overlapping intervals using a hash function, inducing k non-overlapping substreams of the original stream. The final estimate is obtained by computing a function f_i over the i-th substream, and applying a different function g over the average of the outputs of the functions over the k partitions.

2.4 Experiments

2.4.1 Experimental Setup

We performed all experiments on a 64-bit Red Hat Linux machine with 4 cores and a processor speed of 3.50GHz, with 16GB RAM. We used C++ with the Standard Template Library (STL), and the gcc compiler. We implemented the algorithms PCSA, LC, BJKST1, DF, and RW, as described in Section 2.3.1. We also implemented an exact algorithm for the number of distinct elements over a sliding window having a high space complexity.

2.4.1.1 Datasets

We used eight datasets for our experiments - five synthetically generated datasets following a Uniform random or Zipfian distribution, a network traffic trace, bigrams of a text file, and a dataset derived from a real-world graph.

The **Uniform Random** dataset was synthetically generated by choosing elements uniformly at random from the set of unsigned integers ranging 1 to 100 million. This has a total of 500 million elements, with approximately 100 million distinct elements and an average of about 97 million distinct elements in a sliding window of size 45 minutes. We added timestamps to the dataset so that the total time of observation of the data is about an hour. Since the dataset has a uniform distribution, each element occurs with approximately the same frequency.

We generated four **Zipfian** datasets by choosing elements through a Zipfian distribution with α-parameter 1.3, 1.35, 1.4 and 1.5 from the set of integers ranging 1 to 5 million. Each dataset has a total of 500 million elements. In this paper, we have chosen to display the graphs from the Zipfian dataset of α-parameter 1.3 with approximately 2.8 million distinct elements and an average of about 2.3 million distinct elements in a 45 minute sliding window. The

www.manaraa.com

results from the remaining three datasets are similar to the one we have shown in the paper. The total time of observation of the data set is set to 1 hour.

The **Network Trace** data is generated from anonymized traffic traces taken at a west coast OC48 peering link for a large ISP [5]. We consider each source-destination pair as a single element. This has about 400 million elements, with approximately 26 million distinct elements and an average of about 19 million distinct elements in a 45 minute sliding window. Data was generated over a time period of 1 hour.

The **Bigrams in a Text File** is generated by compiling all text versions of ebooks provided by Project Gutenberg [6], and then generating bigrams from the compiled text. This dataset has about 181 million elements with approximately 4.31 million distinct elements and an average of about 35 million distinct elements in a 45 minute sliding window. We added timestamps so that the total time of observation of the dataset is one hour.

The **Friendster Social Network graph** is obtained from the Stanford Network Analysis Project [7]. The graph has about 66 million vertices and about 1.8 billion edges. We use this network to construct a dataset as follows: we select each edge in the graph with probability 0.6 and include endpoints of selected edge as two elements in the stream. The dataset has approximately 2 billion elements with about 55 million distinct elements and an average of about 51 million distinct elements in a 45 minute sliding window.

2.4.1.2 Evaluation Metrics

We compare different algorithms by running them on the same datasets and allotting to each of them the same memory budget. The main measures of the performance of these algorithms are the accuracy and the running time.

The accuracy of an algorithm is expected to improve as the allotted memory increases. We use two measures of accuracy, the average relative error and the worst case relative error. The relative error for a single query is defined as $\frac{|d-\hat{d}|}{n}$, where d is the exact number of distinct elements within the window, and $\hat{d}$ is the estimate of the distinct number of elements within

[5] https://data.caida.org/datasets/oc48/oc48-original/
[6] https://www.gutenberg.org/
[7] http://snap.stanford.edu/data/index.html

www.manaraa.com

the window returned by the algorithm. The average relative error is the mean of the relative errors taken over all queries for a dataset, and the worst case relative error is the maximum of the relative errors across all queries. To get stable results, every data point in the plot is the median of 10 runs of the algorithm.

The running time of the algorithm is the total time taken by the algorithm to process all the elements observed in the datastream and answer the distinct count query.

2.4.2 Results

The performance of an algorithm is influenced by the hash function used and the accuracy boosting method. In the following experiments, we first determine the best hash function and accuracy boosting method for each algorithm, and then use these in further comparing different algorithms.

We further run experiments to see the trend of the accuracy and the runtime variation with the change in size of the window for a fixed memory budget.

2.4.3 Evaluation of Hash Function

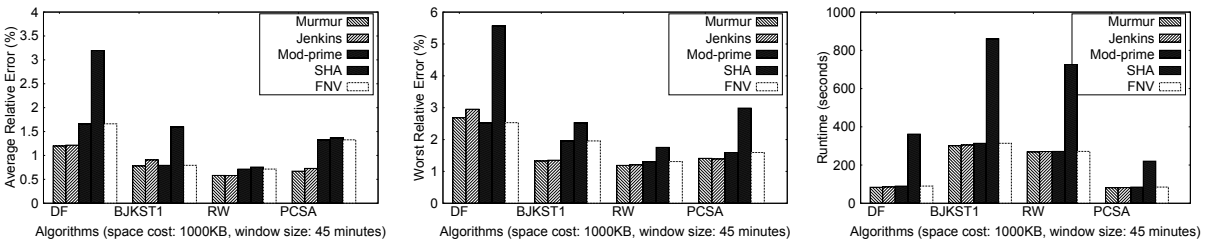
The goal of our first set of experiments is to find the best hash function to use with these algorithms. We implemented the distinct counting algorithms in an identical manner, except for the hash function. We tried five different hash functions - MurmurHash, Jenkins, Modulo congruential hash, SHA-1 and Fowler–Noll–Vo hash or FNV. We used the most recent version of MurmurHash, called "MurmurHash3", and of Jenkins, called "Spooky hash". Modulo congruential hash is the function $h(x) = (a \cdot x + b) \mod p$, where p is a large prime number and a, b are randomly chosen integers modulo p. While simple, this function has interesting theoretical properties ([10]). We use SHA-1 rather than SHA-2 since SHA-1 performs as fast as SHA-2 and requires smaller memory. Though SHA-1 is less secure than SHA-2, this is not an issue for distinct counting. We used the most recent version of FNV, FNV-1a.

The space budget for these experiments was fixed at 1000KB, and the window size was set at 45 minutes. The results of the performance of algorithms for different hash functions have been shown for the real network trace in Figure 2.1. Similar results were obtained for the

www.manaraa.com

other datasets, but they are not shown here due to space constraint. For the LC algorithm, no reasonable results were obtained for any dataset with 1000KB space, and hence we have not shown results for LC.

Observation. MurmurHash has the most consistent accuracy, and its accuracy is better than all other hash functions that we considered. It also runs faster than the others. Jenkins and FNV are close to MurmurHash in terms of both accuracy and runtime. The total runtime of Modulo congruential hash is close to MurmurHash but its accuracy is poor and inconsistent. SHA-1 performs the worst in terms of runtime, and the total runtime using SHA-1 is almost 2-3x slower than using MurmurHash, Modulo congruential hash and Jenkins. The accuracy of SHA-1 is consistent and better than Modulo congruential hash, but not as good as MurmurHash, Jenkins or FNV. Hence, we have chosen MurmurHash for the rest of our experiments.



(a) Average Relative Error of hash functions for different algorithms

(b) Worst Relative Error of hash functions for different algorithms

(c) Runtime of hash functions for different algorithms

Figure 2.1: Comparison of Hash Functions for different algorithms using Network Trace

2.4.4 Evaluation of Accuracy Boosting Method

The idea here is that the estimation accuracy can be improved by running multiple instances of an estimator, and combining the results in some manner. The use of accuracy boosting methods have been advocated in the past for distinct counting, including in [30, 37, 6, 50]. The goal of this set of experiments is to determine which accuracy boosting method serves best for an algorithm. We considered three methods, "median-of-many", "split-and-add", and "stochastic averaging", which have been explained earlier in Section 2.3.2.

Any method such as the above certainly improves accuracy when compared with each individual estimator. However, when total memory of each algorithm is held fixed while increasing

www.manaraa.com

the number of instances, each instance gets proportionately smaller memory, resulting in lower accuracy for each individual instance. So it is not clear that accuracy boosting is useful to improve the overall accuracy of an algorithm given a fixed memory budget. Note that the past literature proves that accuracy boosting method such as the "median of many" method improves the overall accuracy of the algorithm provided the space allocated for the algorithm is also linearly increased.

From our experiments, we observed that, for the "median of many" method, the runtime increases linearly with the number of parallel instances, but it stays almost the same for split and add and stochastic averaging. The reason is that in the "median of many" method, the entire data stream is passed as input to, say, k different instances of algorithm resulting in linear increase in processing time for "median of many" method, contrary to the other two methods where data stream is divided into non-overlapping subsets which are in turn passed as input, each to an individual estimator of algorithm, hence resulting in no increase in runtime.

We implemented different accuracy boosting methods for each algorithm on every dataset. The findings for each dataset were similar. We found that RW performs best without accuracy boosting, i.e. when a single instance is used and the entire memory is given to that instance. Figure 2.2 shows the results of experiments performed using a space budget of 1000KB and window size 45 minutes on the network trace. Overall, the average relative error of RW without accuracy boosting method was better than with any accuracy boosting method. As the number of instances increase, the accuracy of the algorithm decreases due to the aforementioned reason. The total memory required by RW is $O(\log(1/\delta)/\epsilon^2)$ words, where $O(\log(1/\delta))$ is the number of instances run for the algorithm. If the space budget is kept fixed, then the value of ϵ increases with the increase in the number of instances, resulting in an lower accuracy.

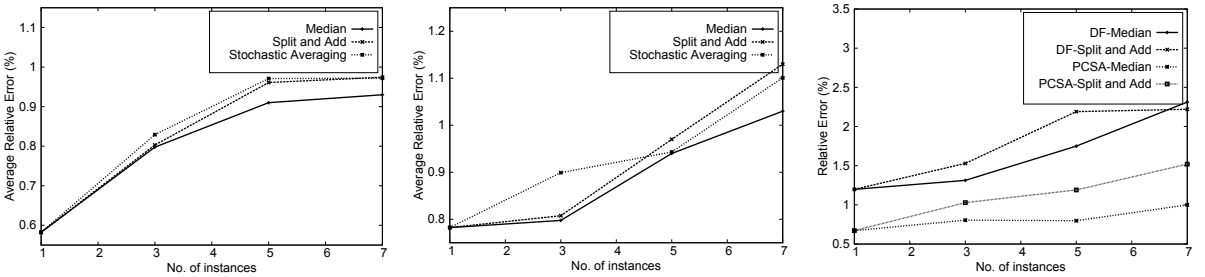
A similar result is observed with BJKST1 (Figure 2.2). The algorithm maintains the τ smallest hash values for each instance, and when multiple instances are used, the value of τ decreases proportionally to keep the overall memory constant. We found that the average relative error of BJKST1 was the smallest without accuracy boosting. While [6] suggested using the median-of-k, the study by [40] and [93] suggested stochastic averaging as an improvement. Note that these did not focus on keeping the memory budget fixed while applying the accuracy

www.manaraa.com

boosting method to the algorithms. Per our observation, the average relative error is minimized by giving the entire memory to a single instance.

PCSA and DF combines many instances of a basic algorithm, which uses a bit vector of a fixed size, using stochastic averaging. We tried combining multiple instances of PCSA and DF using median-of-k method as well as split-and-add, but the error was worse when compared with using a single instance for both PCSA and DF (giving the entire memory to stochastic averaging). In Figure 2.2, using a single instance implies that only stochastic averaging is used over a fixed number of bit vectors determined from the space budget.

LC performs best when the entire space is given to a single instance. We tried using independent smaller bit vectors, followed by accuracy boosting, but this mostly led to invalid results. In case of median-of-k, the load factor of each smaller bit vector increased drastically, and the instances often failed to produce any reasonable estimates.



(a) Average Relative Error vs No. of Instances for RW

(b) Average Relative Error vs No. of Instances for BJKST1

(c) Average Relative Error vs No. of Instances for PCSA and DF

Figure 2.2: Comparison of Accuracy Boosting methods using Network Trace (space cost: 1000KB, window size: 45 minutes)

2.4.5 Evaluation of Algorithms

We implemented each algorithm using the best hash function (MurmurHash), and also the best accuracy boosting method specific to the algorithm. We ran experiments over all the 8 datasets for different space budget keeping the time-based sliding window fixed at 45 minutes. We also ran experiments over the 8 datasets for varying window size, keeping the space budget fixed at 1000KB. We have shown results for only 5 of the 8 datasets due to space constraint. We study the performance of algorithms for 1) different space budgets given a fixed window size

www.manaraa.com

(shown in Figures 2.3, 2.5, 2.4, 2.6, and 2.7).and 2) different window sizes given a fixed space budget (shown in Figures 2.9 and 2.10). We find the median of 10 runs of each experiment to get the corresponding data point to obtain a consistent graph plot.

2.4.5.1 Accuracy

We observe that for a fixed window size, the accuracy of RW and PCSA are the best for small space budgets. As the allocated space is increased, the accuracy of RW becomes better than that of the other algorithms. We observe that as space increases, improvement in accuracy is the most significant in RW.

LC does not produce any result below a space threshold which depends on the number of distinct elements within the sliding window. Given a window size of 45 minutes, LC yields result for a minimum space of 1700KB for the Zipfian dataset. It does not produce a valid result for other datasets for a space budget as large as 2MB. Amongst the other algorithms, we observe that PCSA and RW algorithm produces the most accurate result. As we increase the space budget, RW outperforms PCSA in terms of accuracy. According to the figures, DF algorithm is the least accurate algorithm.

In the study by [60], LC emerged as the most accurate algorithm for distinct counting for a given space budget, beating out PCSA and other alternatives. Our conclusions are different from those of [60], for the following reasons. First, note that their evaluation was for a different problem, that of distinct counting over an infinite window. The algorithms in [60] used a bit vector as a data structure to implement LC, which can accommodate a large number of distinct elements before the load factor gets too large, while we need to have a vector of timestamps to implement LC, which takes much more space (we used 32 bit timestamps). Further, the number of distinct elements in the dataset used in [60] is approximately 2 million, while the number of distinct elements in our datasets is much larger, leading to a higher load factor for the same memory allocated for LC. Since the accuracy of LC is very dependent on the number of distinct elements in the dataset, it is no longer the most accurate algorithm in our study, except for the case when the allocated memory is relatively large.

www.manaraa.com

RW and PCSA are the two most accurate algorithms so we use Figure 2.8 to show the variation in the results obtained for RW and PCSA respectively over 10 different runs. These figures show the minimum, maximum, median and first and third quartile value of average relative error obtained for RW and PCSA. From the figures, it is evident that these algorithms consistently perform well and that other than a few outliers, the variation in the average relative error is small (less than 1%). Due to a constraint in space, we have not added the graph for BJKST1 but the variation in BJKST1 is similar to RW and PCSA. We have also shown the variation in the results for DF algorithm over 10 runs in Figure 2.8. We observe that there is a large variation in the results for DF for small space budget which gets better as the space allocated to the algorithm is increased.

We also performed an evaluation of the effect of the window size on the accuracy of each algorithms for a fixed space budget of 1000KB(shown in Figures 2.9 and 2.10). We conclude from the figures that there is no clear correlation between the window size and the accuracy of algorithms. Also, the runtime of the algorithms do not seem to be affected when the window size is varied. The runtime of the algorithms do not vary with the window size because the total size of the dataset remains the same even when we vary the size of the window over which aggregation is performed. Though we have shown results for only the Bigrams dataset and the Friendster graph, we obtained the same result for the rest of the dataset.

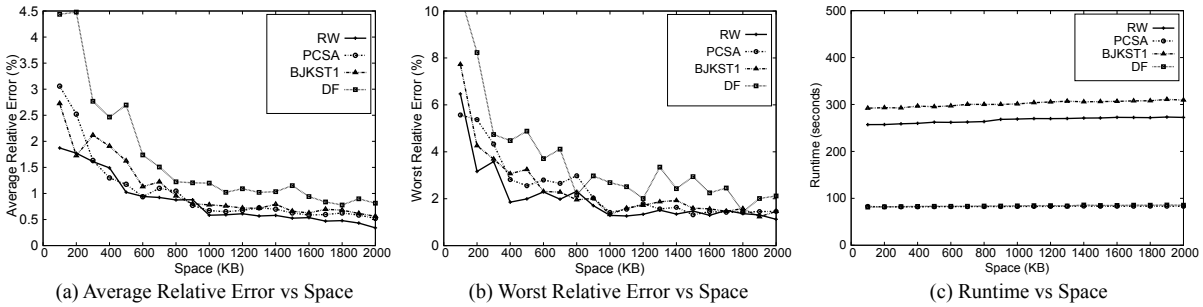
2.4.5.2 Runtime

The running time of LC is the smallest followed by PCSA and DF. These algorithms are faster than RW (approximately 2-4 times) and BJKST1 (approximately 4-5 times), as shown in Figures 2.3, 2.4, 2.5, 2.6, 2.7. We observed that on increasing the space allotted to each algorithm, the runtime for each algorithm increases only slightly. While all the algorithms that we considered have $O(1)$ (amortized) processing time per item, the processing times of these algorithms are different since PCSA, DF, and LC use very simple data structures (arrays) while RW and BJKST1 use a hash table as well as a list, which are relatively more expensive than an array.

www.manaraa.com

We ran an experiment to compare the runtime performance of an array, an STL list, an STL unordered map (a hashmap with $O(1)$ lookup time), and an STL map (an ordered map with $O(\log(n))$ lookup time, where n is the size of a dataset). We observed that the total time taken to insert 100000 elements (elements inserted were valued 1 to 100000) into an array is 1.3 milliseconds whereas a simple insertion of elements in STL unordered map, STL map, and STL list are 14 milliseconds, 41 milliseconds and 6.7 milliseconds respectively. Further, we performed an additional experiment to measure the total time taken by STL unordered and ordered map to simultaneously insert and delete each input so that at no point in time, the map size is greater than 1. The runtime for the unordered and ordered map for this experiment were 20 milliseconds and 25 milliseconds respectively.

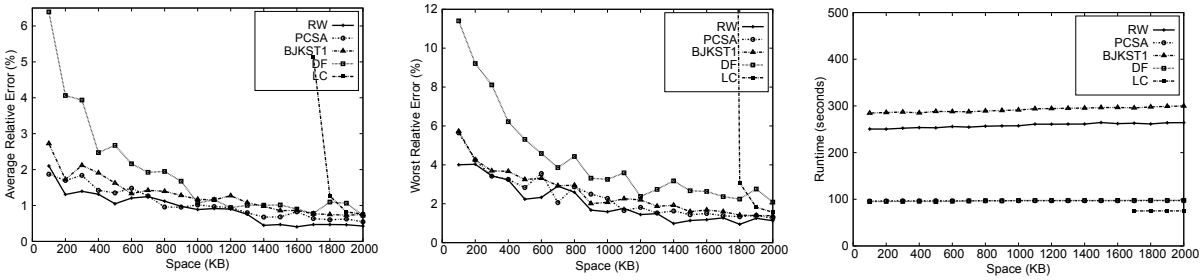
For RW and BJKST1 algorithms, there is an insertion in both map and list for each incoming element, whereas the deletion of elements from these datastructures occur at the end of every time unit. PCSA and DF algorithms require a simple insertion in an array for each incoming element. Considering that the number of insertions in array for PCSA and DF are the same as the number of insertions in both map and list for RW and BJKST1, in addition to the deletion of expired elements from RW and BJKST1 (note that PCSA and DF does not require to update the arrays for expired elements which leads to the significantly high query time for these algorithms), the difference in efficiency of list, map and array is responsible for the difference in algorithm runtimes.



(a) Average Relative Error vs Space (b) Worst Relative Error vs Space (c) Runtime vs Space

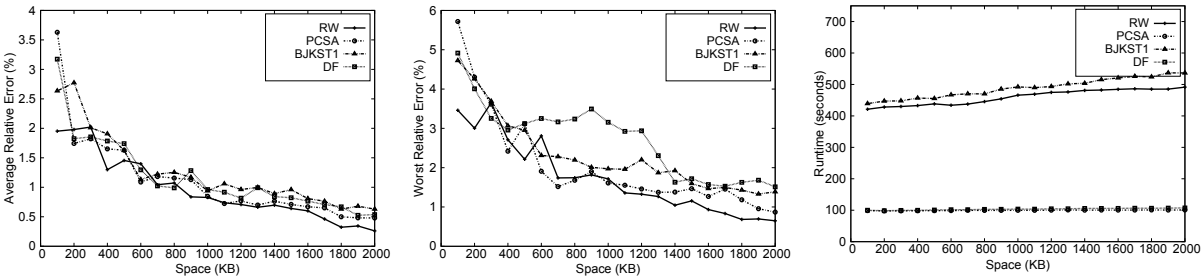
Figure 2.3: Dependence on Space for Network Trace with a window size of 45 min

www.manaraa.com



(a) Average Relative Error vs Space (b) Worst Relative Error vs Space (c) Runtime vs Space

Figure 2.4: Dependence on Space for Zipfian data with a window size of 45 min



(a) Average Relative Error vs Space (b) Worst Relative Error vs Space (c) Runtime vs Space

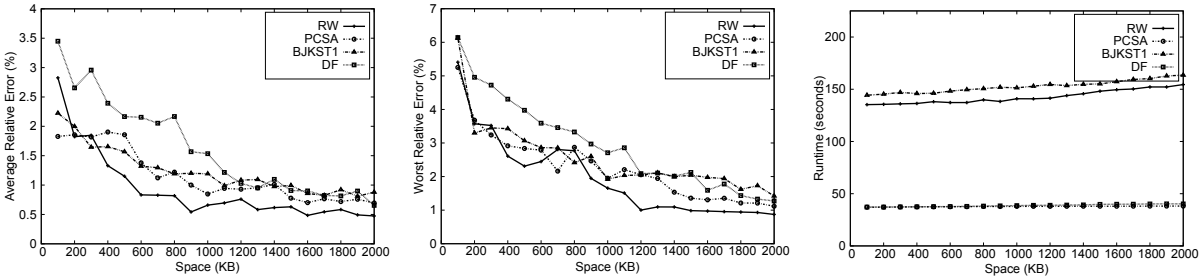
Figure 2.5: Dependence on Space for Uniform Random dataset with a window size of 45 min

2.4.5.3 Query Frequency

The rate of queries (relative to the rate of updates) has an important bearing on the performance of an algorithm. We can think of two extremes here: in one extreme is continuous monitoring, where there is a query after each update, and in the other extreme there is a query only at the end of observation. We use the term "query ratio" to mean the ratio between the number of updates and the number of queries for determining the number of distinct elements in the window.

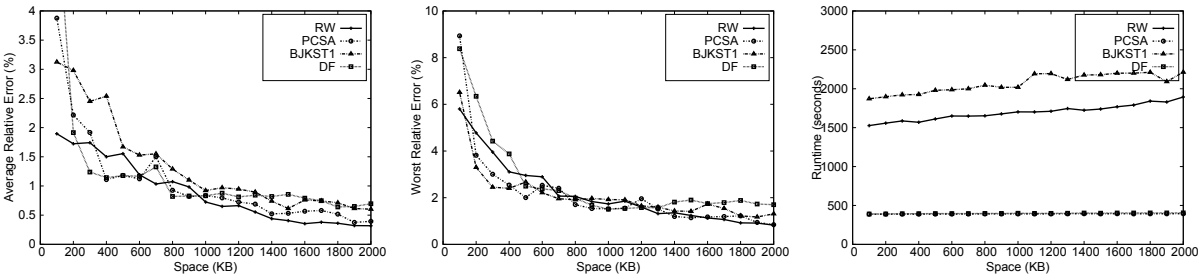
While the performance of RW and BJKST1 are not affected much by the query ratio, LC, PCSA and DF algorithm is significantly affected. In particular, answering a query using these algorithms requires a scan of the entire vector, which is very expensive. As described in Section 2.3.1, we also implemented a version of LC optimized for frequent queries, which we call LC2. We call the version that is not optimized for frequent queries as LC1. We could show the result pertaining to LC only for the zipfian dataset. Other datasets need space much

www.manaraa.com



(a) Average Relative Error vs Space (b) Worst Relative Error vs Space (c) Runtime vs Space

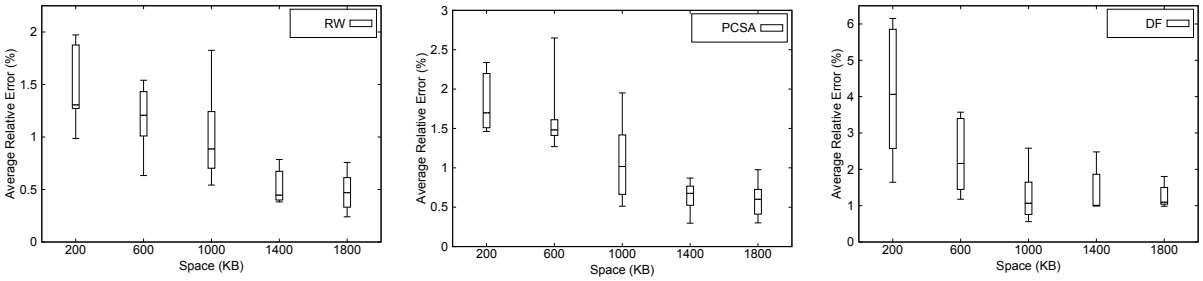
Figure 2.6: Dependence on Space for Bigram dataset with a window size of 45 min



(a) Average Relative Error vs Space (b) Worst Relative Error vs Space (c) Runtime vs Space

Figure 2.7: Dependence on Space for Friendster Graph data with a window size of 45 min

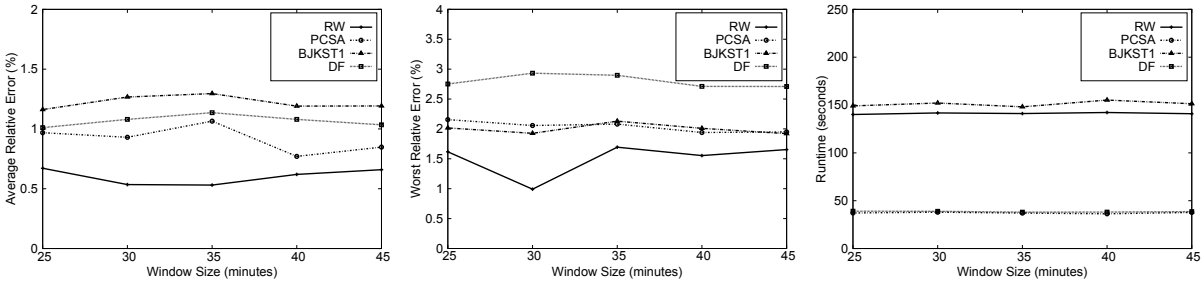
larger than 3MB for producing a valid result for LC1 and LC2. We ran experiments over all the datasets using a space budget of 3MB and a window size of 45 minutes. Smaller space allocation ($<$ 3MB) did not work for LC2 even for the zipfian dataset as the data structure used by LC2 requires large space. The X-axis represents the rate at which a query is posed. For instance, the x-value, 10000, implies that a query is made every 10000 updates. Figure 2.11 shows a



(a) Average Relative Error vs Space for RW (b) Average Relative Error vs Space for PCSA (c) Average Relative Error vs Space for DF

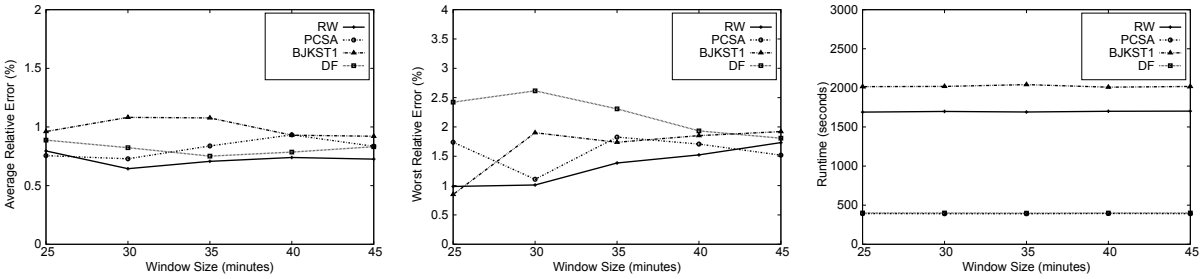
Figure 2.8: Distribution of Average Relative Error of RW, PCSA and DF for Zipfian data (bottom and top of each box are 1st and 3rd quartile resp, band within the box is the median, uppermost and lowermost end of whiskers of each box are max and min resp)

www.manaraa.com



(a) Average Relative Error vs Window Size (b) Worst Relative Error vs Window Size (c) Runtime vs Window Size

Figure 2.9: Dependence on Window Size for Bigram dataset for a fixed space budget of 1000KB



(a) Average Relative Error vs Window Size (b) Worst Relative Error vs Window Size (c) Runtime vs Window Size

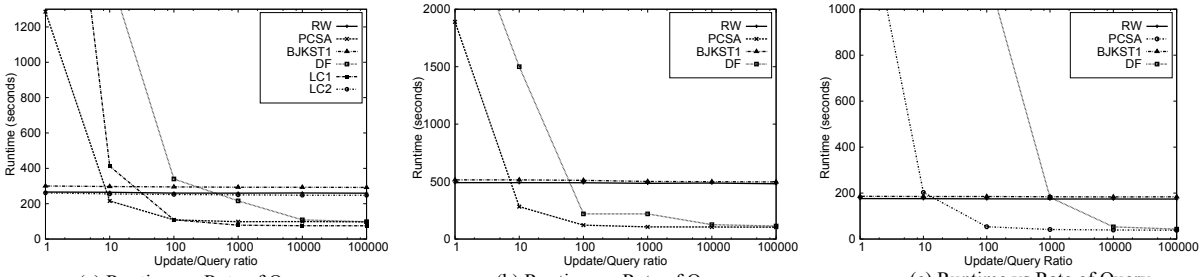
Figure 2.10: Dependence on Window Size for Friendster graph data for a fixed space budget of 1000KB

significant increase in the runtime of LC1 as the rate of querying increases. LC2 performs consistently, without being affected by the rate of querying, similar to the other algorithms. However this runtime of LC2 comes at the expense of accuracy, since additional space is taken up by the data structures for improving the query time. This also implies that LC2 would require much larger space for producing a valid result for a dataset.

The Figure 2.11 also show that the runtime of PCSA and DF is significantly large when query ratio is close to 1. However, the runtime of PCSA and DF, similar to LC1, reduces quickly with a decrease in query rate. The results from the Friendster graph and the network trace imply the same but we have excluded it from the paper due to space constraint.

The runtime of LC1, PCSA and DF algorithms increase drastically with query rate. This is because these algorithms require to perform a linear scan on the vector maintained by them so as to find the information that is required to compute the distinct count.

www.manaraa.com



(a) Runtime vs Rate of Query for Zipfian data

(b) Runtime vs Rate of Query for Uniform Random data

(c) Runtime vs Rate of Query for Bigrams

Figure 2.11: Dependence on rate of Query for a space cost 3000KB and a window size of 45 min

2.5 Conclusion

We presented a detailed experimental evaluation of algorithms for distinct counting over a sliding window. We considered alternatives for different aspects of an implementation, including the basic algorithm, the hash function, the method for boosting accuracy, and the impact of query/update ratio. While there is no clear "best" method that works better than the rest under all situations, our experiments bring out a few combinations that work close to the best.

For a given space budget, if the average relative error is the most important criterion, then using a single instance of Randomized Wave algorithm with the Murmurhash function is close to the best in most situations. If execution time is the most important criterion, then for the scenario where the ratio of number of updates to the number of queries is low, PCSA using Murmurhash performs close to the fastest under most situations. However, if the ratio of updates to queries decreases, then the runtimes of PCSA and DF increase, and when this ratio is small (less than 100, in many cases), RW and BJKST1 perform better in terms of runtime. In such cases, RW is clearly the best option, both in terms of accuracy as well as runtime.

Overall, we observe that for a given space budget, random sampling-based schemes such as RW perform better than bitmap based schemes such as LC. This is because bitmap-based schemes such as PCSA and LC become more expensive space-wise when a timestamp is added to each bit in the vector, while a random sampling-based algorithm such as RW is not affected as much since it already stores the actual element identifiers in the sample, and adding a timestamp to the identifier does not increase the overhead by very much.

www.manaraa.com

CHAPTER 3. MONITORING PERSISTENT ITEMS IN THE UNION OF DISTRIBUTED STREAMS

We address the identification of a feature called a "persistent item" from a massive distributed data stream. A persistent item is one that occurs regularly in the stream, but does not necessarily contribute significantly to the volume of the stream. Let n denote the total number of timeslots in a stream $\mathcal{R}$. The *persistence* of an item d, denoted $p(d)$, is defined as the number of distinct timeslots where d appeared in a stream. Clearly, $0 \leq p(d) \leq n$. Note that multiple occurrences of the same item in the same timeslot do not contribute repeatedly to the persistence of the item. For a parameter $0 \leq \alpha < 1$, an *α-persistent item* is defined as an item whose persistence is at least αn. The above metric was used in [41] in the context of botnet traffic detection.

A persistent item in a distributed set up is defined as follows. Suppose time is divided into "timeslots"[1] Each local site observes a stream of tuples (d, t), where d is an item identifier, and t the timeslot at which d appeared. The *persistence* of an item d, denoted $p(d)$, in a distributed set up is defined as the number of distinct time slots where d appeared in $\bigcup_{i=1}^{k} \mathcal{R}_i$. Clearly, $0 \leq p(d) \leq n$. Note that multiple occurrences of the same item in the same timeslot, whether at the same site or at different sites, do not contribute repeatedly to the persistence of the item.

An item can be highly persistent in the distributed stream without being persistent in any single local stream. Consider the following situation where a particular destination IP address was present in every timeslot from 1 till n, but kept moving from one local site to another in different timeslots, in order to evade detection. The persistence of this destination IP address in any local stream $\mathcal{R}_i$ is only $1/k$, but its overall persistence in the distributed stream is

[1] We assume that time is loosely synchronized between the different sites, so that the different sites agree on which "timeslot" is currently in play. Since timeslots are typically of the order of minutes or more [41], the clocks only need to be synchronized to within a few seconds or more.

www.manaraa.com

100%. Identifying persistent items in a distributed stream can help detect such coordinated and distributed malicious behavior.

A persistent item is different from a "frequent item" in the stream (often known as a "heavy-hitter"). A frequent item is one that appears with a high frequency in the stream, and hence contribute significantly to the volume of the stream. A persistent item need not be a frequent item. For instance, consider an item that occurs exactly once in every timeslot, so that its persistence is 100 percent. The frequency of this item is very low, so that it will not be considered a frequent item in the stream. Similarly, a frequent item may not be persistent either; consider for example an item that occurs in a bursty manner within a timeslot, but never reoccurs within other timeslots. While this item contributes significantly to the volume of the stream, its persistence is very low.

3.1 Goal

The goal of this work is to devise an algorithm for identifying persistent items, which minimizes (1) the communication between the processors and (2) the memory footprint of the algorithm, both per node, and overall. While memory has always been a primary concern in data stream algorithm design in a centralized setting, in a distributed stream, the communication cost is even more important [37, 38, 20, 76, 66, 88], hence communication will be our primary metric.

We first note that any algorithm for exactly identifying persistent items and none other than the persistent items must necessarily incur a large communication cost. In the worst case, this would need communication of the order of the total stream size. Hence, we consider approximate identification of persistent items, with a provable guarantee on the quality of approximation.

Problem Definition: Given persistence threshold $\alpha, 0 < \alpha \leq 1$, approximation parameter $\epsilon, 0 < \epsilon < \alpha$, error probability $\delta \in [0, 1]$, the task is to design a low communication cost and space efficient algorithm that identifies α-persistent items from $\bigcup_{i=1}^{k} \mathcal{R}_i$, with the following properties:

www.manaraa.com

- Low False Negative: If an item d has a persistence $p(d) \geq \alpha n$, then d is identified as α-persistent, with probability at least $(1 - \delta)$.
- Low False Positive: If an item d has a persistence $p(d) < (\alpha - \epsilon)n$, d is not reported as α-persistent, with probability at least $(1 - \delta)$.

We assume a synchronous communication model, where the system progresses in rounds. In each round, each site can observe one element (or none), send a message to the coordinator, and receive a response from the coordinator. The coordinator may receive up to k messages in a round, and respond to each of them in the same round. This model is essentially identical to the model assumed in previous work on distributed stream monitoring [20]. Our results do not change if the sites communicated at the end of each timeslot, rather than at the end of observing each element. The sizes of the different local streams at the sites, their order of arrival, and the interleaving of the arrivals at different sites, can all be arbitrary. The algorithm cannot make any assumption about these.

We consider both the infinite and sliding window models for the identification of persistent items.

3.2 Contribution

We present the first communication-efficient algorithms for tracking persistent items over the union of multiple distributed streams, with approximation parameter ϵ and error probability δ.

3.2.1 Infinite Window Algorithm

We first present an algorithm for the setting where the data of interest is the union of all items over all the k streams observed so far. Let n denote the total number of timeslots so far. The expected space complexity over all sites is $O\left(\frac{k\alpha^2 \log\left(\frac{\log(1/\delta)}{2\delta}\right)P}{\epsilon^3 n}\right)$ and the expected number of bytes transmitted across all sites is $O\left(\frac{k\alpha^2 \log\left(\frac{\log(1/\delta)}{2\delta}\right)P}{\epsilon^3 n}\right)$ bytes, where $P = \sum_{d \in M} p(d)$,

www.manaraa.com

where M is the set of all distinct items observed in the stream, i.e. P is the sum of persistence values of all distinct items observed in the union of all streams.

3.2.2 Sliding Window Algorithm

Next we consider the setting where the data of interest is the union of data observed by all the k streams during the n most recent timeslots, and we present an algorithm for identifying persistent items within this data. The expected space complexity of our distributed algorithm over all sites is $O\left(\frac{k\alpha^2 \log\left(\frac{\log(1/\delta)}{2\delta}\right)P}{\epsilon^3 n}\right)$ and the expected number of communication bytes over all sites is $O\left(\frac{k\alpha^2 \log\left(\frac{\log(1/\delta)}{2\delta}\right)P}{\epsilon^3 n}\right)$ bytes, where P is the sum of persistence values within the last n time slots, of all the distinct items seen in all the streams.

3.2.3 Simulations

We simulated our algorithm using real-world network trace data as well as synthetic data. These simulations show that our algorithm tracks α-persistent items with the observed guarantees, and that the communication and space overhead are much smaller than distributed implementation of existing algorithms.

3.3 Related Work

Prior work on identifying persistent items in a stream has considered the centralized case. This includes work by Giroire et al. [41], who track persistent items in a centralized stream by exactly computing the persistence of each distinct item in the stream, and an improved small space approximation algorithm by Lahiri et al. [54].

A frequent item or a "heavy hitter" in a stream is one whose frequency in the stream is significant when compared with the volume of the stream. There is much prior work on identifying frequent items or heavy hitters from a data stream, including [17, 62, 57, 61, 73, 77, 55]. As discussed earlier, a frequent item may not be persistent, and a persistent item in not necessarily frequent either. Frequent item identification algorithms that are based on "sketches", such as the Count Sketch [14] and the Count-Min Sketch [19], can be implemented

www.manaraa.com

in a distributed manner. These algorithms maintain multiple counters, each of which is the sum of many random variables. The sketch for the union of several streams is simply the sum of the sketches over all the streams. However, adapting these algorithms for the case of persistent items does not seem to be easy since these sketches count the number of occurrences (frequency) as opposed to the number of occurrences in distinct timeslots (persistence).

3.4 Solution Overview

A straightforward approach to tracking persistent items is as follows. Every site i maintains a data structure S_i, which contains the set of all distinct timeslots that each item has appeared in stream $\mathcal{R}_i$. Note that this requires storing a set of up to n elements for each distinct item that has appeared in $\mathcal{R}_i$. Upon a query, each site i sends S_i to the coordinator, who can exactly compute the persistence of the item in the distributed stream, and return only those items that has persistence of at least αn. While this requires no coordination among the sites prior to the query, the total communication required at the time of the query is prohibitive, since it communicates up to $\Theta(n)$ bytes per item, per site, which can be very large when the number of distinct items is large. Clearly, this approach is expensive in terms of space required per site as well.

A centralized small-space streaming algorithm for persistent items such as the one in [54] can be used to track persistent items in each individual stream, but cannot be directly used in a distributed context. The reason is that the algorithm in [54] depends on using a simple counter at each site to track the number of slots an item has appeared in. In the distributed case, overlapping occurrences of the same item in the same timeslot across different sites should still be discounted. Hence, a simple extension of the centralized algorithm will not work here.

In our approach, we first reduce the number of items that are tracked using a hash-based random sampler, similar to the one used in the centralized streaming algorithm in [54]. This sampler is used to (with high probability) selectively maintain state for only those items whose persistence crosses a given threshold. The threshold is chosen such that an α-persistent item is very likely to be tracked. Once tracking state has been established for an item, we still need to maintain its persistence as more copies of this item arrive. We could potentially do this through

www.manaraa.com

maintaining for each such item, a list of timeslots where the item has appeared. When a query for persistent items is posed, the lists for different tracked items is sent to the coordinator, who computes the union of different lists across all the sites, to compute the persistence of the tracked item. However, this naive approach leads to a high memory requirement and communication cost.

We reduce the memory and communication cost through using a distributed distinct counting algorithm [30, 47, 3, 13, 37, 6, 87, 51] to maintain, in a coordinated manner, a count of distinct timeslots of occurrence for each tracked item across all the sites. A distributed distinct counting algorithm estimates the number of distinct elements in the union of multiple distributed streams. More precisely, if $dist(\mathcal{R})$ is the number of distinct elements in a distributed stream $\mathcal{R}$, then given a relative error $0 < \gamma < 1$ and an error probability $0 < \upsilon < 1$, a (γ, υ)-approximate distinct counting algorithm returns an estimate X such that $\Pr[|dist(\mathcal{R}) - X| > \gamma.dist(\mathcal{R})] \leq \upsilon$. The algorithm that we use from [37, 6], is practical, and has an overall space requirement of $O\left(\frac{\log 1/\upsilon}{\gamma^2}\right)$ words.

With our approach, there are two sources of error in the estimated persistence of an item. (1) We do not track the persistence of each item, but only those which pass through the sampler. While this results in reduced communication when compared with tracking the persistence of each item, it also results in an error in the measured persistence of each item, even for those items that are tracked. (2) After tracking state has been established for an item, the overall persistence in the distributed stream in forthcoming timeslots is only computed approximately. Our analysis ensures that the combined error from these two sources does not exceed the desired threshold. To achieve this, the total error budget is divided among the two sources of error such that the communication cost is minimized and the final approximation guarantees are achieved. Our analysis for the infinite window case the sliding window case are described in Sections 3.5 and 3.6 respectively.

www.manaraa.com

3.5 Infinite Window

We now present an algorithm for the case of an infinite window, i.e. when the data of interest is the union of all items from the beginning of time, that arrived across all streams.

Intuition: To reduce space and communication, the first step is to avoid tracking every item, especially items with a low value of persistence. While tracking items with a low persistence cannot be completely avoided, it can be reduced through sampling. Sites 1 through k share a common hash function $h : ([1, m] \times [1, n]) \rightarrow (0, 1)$. For two tuples (d_1, t_1) and (d_2, t_2) that are unequal either in one attribute or both, $h(d_1, t_1)$ and $h(d_2, t_2)$ are independent random values chosen uniformly at random from the interval of real numbers $(0, 1)$.

Each site i maintains state for a subset of items that have arrived so far. When an item (d, t) arrives in $\mathcal{R}_i$, if d is already being tracked by i, then the state corresponding to d is updated by adding t to the set of time slots that d has appeared in. If d is not being tracked by i, then tracking state is established for d if $h(d, t) < \tau$ (for a value τ to be decided), and a message is sent to all sites to start tracking d. Clearly, if an item d appears in time slot t, tracking state is established for d with probability τ. We note the following.

- Multiple occurrences of d within the same time slot t do not increase the probability of d being tracked.
- A low-persistence item d which appears only in a few distinct time slots is not likely to be tracked. On the other hand, a high-persistence item d' which appeared in many distinct slots will be tracked with a high probability.
- Since the same hash function h is shared by all sites, the result after distributed occurrences of d is the same as if d was being observed by the same site.

Once tracking state has been established for an item d, future occurrences of d in subsequent time slots are treated without needing further communication among the sites. A challenge here is that even with state maintained at different nodes for an item d, it is still non-trivial to track the number of occurrences of d in distinct time slots. For this purpose, we use a distributed distinct counter, from [37]. Equivalently, we could use other algorithms for distinct counting

www.manaraa.com

that can be implemented in a distributed setting, such as the one by Bar-Yossef et al. [6]. We use the one in [37] because it is simple and gives very good practical performance. The accuracy and error probability of this distinct counter influences the overall space complexity of our algorithm. Before we present the formal algorithm description, we present the guarantees expected from the distinct counter.

When a query is posed for the set of persistent items, the coordinator combines the estimates of all the distributed distinct counters to compute an estimate of the persistence of each item being tracked. This estimate is used to decide whether or not an item is persistent. There are two sources of error in this estimator: (1) the error due to sampling, before the item starts being tracked, and (2) the error due to the approximate distinct counter for the item which is already being tracked. We first present the guarantees provided by a distributed distinct counting algorithm. For a relative error parameter $0 < \gamma < 1$ and an error probability parameter $0 < \upsilon < 1$, a distinct counter $\mathcal{D}_\gamma^\upsilon$ takes as input a stream of updates S and maintains an estimate of $dist(S)$, the number of distinct items in S.

Theorem 1 ([37]). *There is a distinct counter $\mathcal{D}_\gamma^\upsilon$ that takes space $O\left(\frac{\log 1/\upsilon}{\gamma^2}\right)$ words of space, and whenever a query is asked for $dist(S)$, returns an estimate X such that $\Pr[|X - dist(S)| > \gamma \cdot dist(S)] \leq \upsilon$, for $0 < \gamma < 1$ and $0 < \upsilon < 1$. This distinct counter can handle distributed updates, and the distributed state can be combined together at the end of observation.*

Note that we express the space complexity above in terms of the number of words, assuming that each item identifier and timestamp can be stored in a constant number of words.

Algorithm Description: The inputs to our algorithm are: 1) m - domain size of the identifiers, 2) n - total number of timeslots in the distributed stream, 3) α - persistence threshold, 4) ϵ - approximation parameter, and 5) δ - error probability. The distributed algorithm has three parts: Algorithm 1 defines the input parameter, and describes the initialization of the datastructures and global variables used, Algorithm 2 describes the algorithm at each local site, and Algorithm 3 describes the algorithm at the coordinator node C.

Each site i maintains a sketch S_i for stream $\mathcal{R}_i$ seen so far, comprising of tuples of the form $(d, \mathcal{D}_{\delta_2}^{\epsilon_2}[i](d))$. Here, d is an item ID, and $\mathcal{D}_{\delta_2}^{\epsilon_2}[i](d)$ is the distinct counting datastructure

www.manaraa.com

maintained for estimating the number of distinct time slots when the item d appeared. The distinct counting data structure is maintained using the distributed distinct counting algorithm with approximation parameter ϵ_2 and error probability δ_2. Here, ϵ_2 and δ_2 are parameters whose values are determined based on optimizing communication cost while obeying the correctness constraints

The coordinator C maintains a sketch S which has tuples of the form (d, t_d) where d is the item ID and t_d is the time when the item d was first tracked in the distributed stream. When an item (d, t) arrives at site i in timeslot t, then the algorithm first looks into S_i to check if d is being tracked (Algo 2: line 2). If not, then (d, t) is passed through the hash function h. The algorithm starts tracking d if $h(d, t) < \tau$, where $\tau = 2/(\epsilon_1 n)$ and $\epsilon_1 = c_\epsilon \epsilon$ (Algo 2: line 3), where $0 < c_\epsilon < 1$ is a constant; note this happens with probability τ. Site i communicates with the other sites (Algo 2: lines 4-5) and the coordinator C (Algo 2: line 6) to inform them about the newly tracked item. Once site i starts tracking the item d, it makes an entry of the form $(d, \mathcal{D}^{\epsilon_2}_{\delta_2})$ in S_i and starts maintaining a distinct count datastructure $\mathcal{D}^{\epsilon_2}_{\delta_2}[i](d)$ for item d (Algo 2: line 8). Once the item is tracked, with every appearance of the item in a new timeslot, $\mathcal{D}^{\epsilon_2}_{\delta_2}[i](d)$ is updated using the distinct counting algorithm (Algo 2: line 10).

If site i receives information about a newly tracked item d from some other site, it starts tracking d and creates a new local entry $(d, \mathcal{D}^{\epsilon_2}_{\delta_2})$ in S_i (Algo 2: lines 12-15). It then updates $(d, \mathcal{D}^{\epsilon_2}_{\delta_2})$ in S_i for d with every appearance of item d in a new timeslot in site i (Algo 2: line 8). When the coordinator receives information about a newly tracked item d from any of the sites, it makes a new entry (d, t_d) in S (Algo 3: line 1).

When a query is made to the coordinator C, for each tracked item $d \in S$, it takes a union of the corresponding distinct count data structure $(d, \mathcal{D}^{\epsilon_2}_{\delta_2})[j](d)$ across all sites as per the distinct counting algorithm to estimate the number of distinct slots $\hat{n}_d$, where item d appeared in the distributed stream since it was tracked (Algo 3: lines 3-6). While we do not know how many distinct slots d may have appeared in before it was first tracked, we can see this number is a geometric random variable X with parameter τ; we estimate the value of X using its expectation. We estimate the persistence of d, equal to the total number of distinct slots where d appeared in the entire distributed stream, as $\hat{p}_d$ as $\hat{n}_d + E(X) = \hat{n}_d + 1/\tau$ (Algo 3: line 10).

www.manaraa.com

Also, in order to optimize our results, we compute $\hat{p}_d$ as $\hat{n}_d + t_d$ for the condition $(1/\tau) > t_d$, (Algo 3: lines 7-8).

Algorithm 1: Infinite Window : Initialization

Input: m - Domain Size of identifiers; n - Total no. of time slots; α - persistence threshold; ϵ - error parameter; δ - error probability

1 Hash function $h : ([1, m] \times [1, n]) \rightarrow (0, 1)$
2 Approximation Parameters: $\epsilon_1 \leftarrow c_\epsilon \epsilon$, $\epsilon_2 \leftarrow (1 - c_\epsilon)\epsilon/4\alpha$ `// 0 < c_ε < 1 is a constant`
3 Error Probability: $\delta_2 = c_\delta \delta$ `//` $0 < c_\delta \leq min\left(1, \frac{2}{\log(1/\delta)}\right)$ `is a constant`
4 Filter parameter $\tau \leftarrow \frac{2}{\epsilon_1 n}$
5 Threshold $T \leftarrow (1 - \epsilon_2)(\alpha n - \frac{1}{\tau} + 1)$
6 $S \leftarrow \emptyset$
7 **for** $i = 1, 2, .., k$ **do**
8 $\quad S_i \leftarrow \emptyset$

Algorithm 2: Infinite Window: Algorithm at node i

1 **On receiving item** (d, t) **at node** i
2 **if** $(d \notin S_i)$ **then**
3 $\quad$ **if** $h(d, t) < \tau$) **then**
4 $\quad\quad$ **for** *every node* $j = 1 \ldots k$, $j \neq i$ **do**
5 $\quad\quad\quad$ Send "Start Tracking (d, t)" to j
6 $\quad\quad$ Send (d, t) to the coordinator
7 $\quad\quad$ $S_i \leftarrow S_i \cup \{(d, \mathcal{D}_{\delta_2}^{\epsilon_2}[i](d))\}$
8 $\quad\quad$ Insert t into $\mathcal{D}_{\delta_2}^{\epsilon_2}[i](d)$
9 **else** `//` `if` $d \in S_i$
10 $\quad$ Insert t into $\mathcal{D}_{\delta_2}^{\epsilon_2}[i](d)$
11 **On receiving message "Start Tracking** (d, t)**"**
12 `// Create a new data structure tracking` d
13 $S_i \leftarrow S_i \cup \{(d, \mathcal{D}_{\delta_2}^{\epsilon_2}[i](d))\}$
14 Insert t into $\mathcal{D}_{\delta_2}^{\epsilon_2}[i](d)$

3.5.1 Infinite Window : Correctness

Let $G(\tau)$ be the geometric random variable with parameter τ. Let $p(d)$ denote the persistence of item d, and n_d denote the number of distinct slots where d appeared in $\mathcal{R}$ after (and including) the time slot when the algorithm started tracking d.

www.manaraa.com

Algorithm 3: Infinite Window: Algorithm at the coordinator C.

Upon receiving (d, t_d)**:** Insert (d, t_d) into S
Upon receiving a query for the set of Persistent Items:
for *each* $(d, t_d) \in S$ **do**
    for $i = 0, 1, ..., k-1$ **do**
        Compute the union of $D^{\delta_2}_{\epsilon_2}[i](d)$ data structures over all sites i.
        Let $\hat{n}_d$ be the estimate of the distinct count over this union.
    if $t_d < (1/\tau)$ **then**
        $\hat{p}(d) \leftarrow \hat{n}_d + t_d$
    else
        $\hat{p}(d) \leftarrow \hat{n}_d + \frac{1}{\tau}$
    if $\hat{p}(d) \geq T$ **then**
        Report d as α-persistent

Lemma 1. *If* $G(\tau) \leq p(d)$*, then* $n_d = p(d) - G(\tau) + 1$*, else* $n_d = 0$*.*

Proof. Let the distinct time slots that d appears in distributed stream be $t_1, t_2, \ldots$, in increasing order. d is not tracked until we reach a time slot t_i such that $h(d, t_i) < \tau$. The number of time slots required for this to occur is $G(\tau)$. Note this is true even though the different sites are observing the tuples in a distributed manner, since their decisions are based on the output of a hash function on (d, t). The expression for n_d follows. □

Lemma 2. ***False Negative:*** *If an item d is α-persistent, then the probability that it is not reported by the coordinator in Algorithm 3 is at most* $\left(e^{-2} + \frac{2\delta}{\log(1/\delta)}\right)$.

Proof. Consider an α-persistent item d. Let A denote the event that d is not reported. Let $\hat{p}(d)$ be the estimate of its persistence at the end of observation. Also, let $\hat{n}_d$ be an estimate of n_d, the number of distinct time slots where d appeared in $\mathcal{R}$ after being tracked. $\hat{n}_d$ is obtained from the union of the distinct count datastructures over the k sites. Per the above algorithm, $\hat{p}(d) = \hat{n}_d + 1/\tau$, and d is not reported if $\hat{p}(d) < T$. Consider that $T = (1 - \epsilon_2)(\alpha + 1 - 1/\tau)$ and $\delta_2 = c_\delta \delta$ where $c_\delta \leq \frac{2}{\log(1/\delta)}$.

$$\Pr[A] = \Pr[\hat{p}(d) < T] = \Pr\left[\hat{n}_d < \left(T - \frac{1}{\tau}\right)\right]$$

www.manaraa.com

Let B denote the event $(1-\epsilon_2)n_d \leq \hat{n}_d$. We have the following:

$$\Pr[A] = \Pr[A|B]\Pr[B] + \Pr[A|\bar{B}]\Pr[\bar{B}] \leq \Pr[A|B] + \Pr[\bar{B}] \tag{3.1}$$

$$\begin{aligned}
\Pr[A|B] &= \Pr\left[\hat{n}_d < \left(T - \frac{1}{\tau}\right) \middle| (1-\epsilon_2)n_d \leq \hat{n}_d\right] \\
&\leq \Pr\left[(1-\epsilon_2)n_d < \left(T - \frac{1}{\tau}\right)\right] \\
&= \Pr\left[p(d) - G(\tau) + 1 < \frac{(T - 1/\tau)}{(1-\epsilon_2)}\right] && \text{using Lemma 1} \\
&= \Pr\left[G(\tau) > p(d) + 1 - \frac{T}{(1-\epsilon_2)} + \frac{1}{(1-\epsilon_2)\tau}\right] \\
&\leq \Pr\left[G(\tau) > \alpha n + 1 - \left(\alpha n + 1 - \frac{1}{\tau}\right) + \frac{1}{(1-\epsilon_2)\tau}\right] && \text{substituting } T \text{ and given } p(d) \geq \alpha n \\
&\leq \Pr\left[G(\tau) > \frac{2}{\tau}\right] \qquad = (1-\tau)^{\frac{2}{\tau}} \\
&\leq e^{-2} \qquad \text{since } (1-\tau)^{\theta} \leq e^{-\tau\theta}
\end{aligned}$$

The probability of $\bar{B}$ depends on the guarantee given by the distinct counter $\mathcal{D}_{\epsilon_2}^{\delta_2}$. Note that the number of insertions into the distinct counter is n_d, and the estimate returned by the distinct counter is $\hat{n}_d$. Using Theorem 1, we have: $\Pr[(1-\epsilon_2)n_d \leq \hat{n}_d \leq (1+\epsilon_2)n_d] \geq (1-\delta_2)$. Hence,

$$\begin{aligned}
\Pr[\bar{B}] &= \Pr[\hat{n}_d < (1-\epsilon_2)n_d] \\
&\leq \Pr[\hat{n}_d < (1-\epsilon_2)n_d] + \Pr[\hat{n}_d > (1+\epsilon_2)n_d] \\
&\leq \delta_2 \qquad \leq \frac{2\delta}{\log(1/\delta)} \qquad \text{given } \delta_2 = c_\delta \delta \leq \frac{2\delta}{\log(1/\delta)}
\end{aligned}$$

Using these back in Equation 3.1, we get the desired result. □

Lemma 3. False Positives: *An item d with persistence $p(d) < (\alpha - \epsilon)n$ is reported by the coordinator in Algorithm 3 with probability at most $\frac{2\delta}{\log(1/\delta)}$.*

Proof. Consider an item d with persistence $p(d) < (\alpha - \epsilon)n$. Let A denote the event that d is reported as being α-persistent. If $\hat{p}(d)$ is the estimate of persistence of d at the end of observation, then $\hat{p}(d) > T$. Also, per the algorithm, $\hat{p}(d) = \hat{n}_d + 1/\tau$.

www.manaraa.com

$$\Pr[A] = \Pr\left[\hat{p}(d) > T\right] = \Pr\left[\hat{n}_d > \left(T - \frac{1}{\tau}\right)\right]$$

Let B denote the event $\hat{n}_d \leq (1+\epsilon_2)n_d$. If $T = (1-\epsilon_2)\left(\alpha n + 1 - \frac{1}{\tau}\right)$, $\epsilon_1 = c_\epsilon \epsilon$, and $\epsilon_2 = \frac{\epsilon(1-c_\epsilon)}{4\alpha}$, where, c_ϵ is a constant s.t. $0 < c_\epsilon < 1$ then,

$$\begin{aligned}
\Pr[A|B] &= \Pr\left[\hat{n_d} > T - \frac{1}{\tau}\middle|\hat{n}_d \leq (1+\epsilon_2)n_d\right] \\
&\leq \Pr\left[(1+\epsilon_2)n_d \geq T - \frac{1}{\tau}\right] \\
&= \Pr\left[p(d) - G(\tau) + 1 \geq \left(\frac{1-\epsilon_2}{1+\epsilon_2}\right)\left(\alpha n + 1 - \frac{1}{\tau}\right) - \frac{1}{(1+\epsilon_2)\tau}\right] \qquad \text{substituting } T \text{ and using Lemma} \\
&= \Pr\left[G(\tau) \leq p(d) + 1 - \left(\frac{1-\epsilon_2}{1+\epsilon_2}\right)\left(\alpha n + 1 - \frac{1}{\tau}\right) + \frac{1}{(1+\epsilon_2)\tau}\right] \\
&\leq \Pr\left[G(\tau) \leq (\alpha n - \epsilon n) + 1 - \left(\frac{1-\epsilon_2}{1+\epsilon_2}\right)(\alpha n + 1) + \frac{2-\epsilon_2}{(1+\epsilon_2)\tau}\right] \qquad \text{given } p(d) < (\alpha - \epsilon)n \\
&\leq \Pr\left[G(\tau) \leq (\alpha n - \epsilon n) + 1 - (1-2\epsilon_2)(\alpha n + 1) + \frac{2}{\tau}\right] \qquad \text{as, } (1-2\epsilon_2) < (1-\epsilon_2)/(1+\epsilon_2) \\
&= \Pr\left[G(\tau) \leq 2\epsilon_2 \alpha n + 2\epsilon_2 + \frac{2}{\tau} - \epsilon n\right] \\
&\leq \Pr\left[G(\tau) \leq 4\epsilon_2 \alpha n + \epsilon_1 n - \epsilon n\right] = 0 \qquad \text{using } \tau = 2/\epsilon_1 n, \text{ and, } (2\epsilon_2\alpha n + 2\epsilon_2 \leq 4\epsilon_2 \alpha n)
\end{aligned}$$

Using Theorem 1, we have: $\Pr[(1-\epsilon_2)n_d \leq \hat{n}_d \leq (1+\epsilon_2)n_d] \geq (1-\delta_2)$ Hence,

$$\begin{aligned}
\Pr[\bar{B}] &= \Pr[\hat{n}_d > (1+\epsilon_2)n_d] \\
&\leq \Pr[\hat{n}_d < (1-\epsilon_2)n_d] + \Pr[\hat{n}_d > (1+\epsilon_2)n_d] \\
&\leq \delta_2 \qquad \leq \frac{2\delta}{\log(1/\delta)}
\end{aligned}$$

Using the relation $\Pr[A] \leq \Pr[A|B] + \Pr[\bar{B}]$, we get the desired result. We obtain that for $c_\epsilon = 1/3$, the space cost and the communication cost of this algorithm is optimized. □

Theorem 2. *By running at least* $\frac{\log \delta}{\log(e^{-2} + c_\delta \delta)}$ *parallel instances of the algorithm, we get the following guarantee:*

1. *An item* d *with persistence* $p(d) \geq (\alpha n)$ *is reported as* α*-persistent with probability at least* $1 - \delta$.

www.manaraa.com

2. *An item d with persistence $p(d) < (\alpha - \epsilon)n$ is not reported as persistent with probability at least $1 - \delta$.*

Proof. Let $\theta = \frac{\log \delta}{\log (e^{-2} + c_\delta \delta)}$. We return the union of all persistent items returned by all the parallel instances. For an α-persistent item d, the probability that d is not reported is equal to the probability that it is not reported by any of the θ instances. This probability is no more than $(e^{-2} + c_\delta \delta)^\theta$, which is bounded by δ (where $0 < c_\delta \leq \frac{2}{\log (1/\delta)}$).

Consider an item d with persistence less than $(\alpha - \epsilon)n$. The probability that d is reported is the probability that d is reported by at least one of the θ parallel instances. Using the union bound, this probability is no more than $\frac{2\delta\theta}{\log (1/\delta)}$. Upon substituting θ, we get the desired result. □

3.5.2 Infinite Window: Complexity

We present an analysis of the communication and space complexity of the algorithm for an infinite window. Let P be the sum of the persistence of all the distinct items in the distributed stream, n be the total number of time slots in the stream. Recall that k is the number of sites.

Theorem 3. *The expected space complexity of the distributed algorithm per site is* $O\left(\frac{\alpha^2 \log\left(\frac{\log (1/\delta)}{2\delta}\right) P}{\epsilon^3 n}\right)$, *and the expected space complexity over all sites is* $O\left(\frac{k\alpha^2 \log\left(\frac{\log (1/\delta)}{2\delta}\right) P}{\epsilon^3 n}\right)$

Proof. The space complexity of tracking a single item is equal to the cost of an approximate distinct count data structure $\mathcal{D}_{\epsilon_2}^{\delta_2}$, for maintaining the number of distinct time slots for the item. Let $Z(d)$ be a random variable for item d such that $Z(d) = 1$ if item d is tracked, else $Z(d) = 0$.

$$\begin{aligned}
\Pr[Z(d) = 1] = 1 - \Pr[Z(d) = 0] &= 1 - (1 - \tau)^{p(d)} \\
&\leq 1 - e^{-2\tau p(d)} \quad \text{using Taylor's expansion} \\
&\leq 2\tau p(d) \leq 1 - (1 - 2\tau p(d)) \leq 2\tau p(d)
\end{aligned}$$

www.manaraa.com

We know that space taken by distinct count operator for each item d at each site is $O\left(\frac{\log(1/\delta_2)}{{\epsilon_2}^2}\right)$ (Theorem 1). The expected space taken by an item d per site is:

$$\Pr[Z(d)=1]\left(\frac{\log(1/\delta_2)}{{\epsilon_2}^2}\right) \leq \frac{2\tau p(d)\log(1/\delta_2)}{{\epsilon_2}^2} = \frac{4p(d)\log(1/\delta_2)}{{\epsilon_2}^2\epsilon_1 n}$$
$$= O\left(\frac{p(d)\log(1/\delta_2)\alpha^2}{\epsilon^3 n}\right) \quad \text{since } \epsilon_2 = O\left(\frac{\epsilon}{\alpha}\right); \epsilon_1 = O(\epsilon); \delta_2 = O(\delta)$$
$$\leq O\left(\frac{p(d)\log\left(\frac{\log(1/\delta)}{2\delta}\right)\alpha^2}{\epsilon^3 n}\right) \qquad \text{given } \delta_2 = c_\delta\delta \leq \frac{2\delta}{\log(1/\delta)}$$

Hence, total expected space taken by the algorithm per site is:

$$= O\left(\sum_d \frac{p(d)\log\left(\frac{\log(1/\delta)}{2\delta}\right)\alpha^2}{\epsilon^3 n}\right) = O\left(\frac{\alpha^2\log\left(\frac{\log(1/\delta)}{2\delta}\right)P}{\epsilon^3 n}\right)$$

The total space over the entire distributed algorithm is k times the space cost of each site. □

Theorem 4. ***Communication:*** *The expected communication complexity of the distributed algorithm taken over all sites is* $O\left(\frac{k\alpha^2\log\left(\frac{\log(1/\delta)}{2\delta}\right)P}{\epsilon^3 n}\right)$ *bytes.*

Proof. For each item that is tracked, the algorithm incurs $O(k)$ messages to begin tracking the item. Finally, in order to identify persistent items, it is necessary to have another round of communication among all the sites and the coordinator. The total number of messages exchanged is thus $O(kN)$ where N is the number of items that are tracked. Since we know that $\mathbb{E}[E] = O\left(\frac{P}{\epsilon n}\right)$ (see the proof in Theorem 3), the expected number of messages communicated over all sites is $O(\frac{kP}{\epsilon n})$.

If we consider the number of bytes communicated, we find that each item leads to a communication of $O(\frac{\alpha^2\log(1/\delta_2)}{\epsilon^2})$ bytes, due to the distinct count data structure. Hence, the expected number of message bytes communicated between the sites and the coordinator is $O\left(\frac{k\alpha^2\log\left(\frac{\log(1/\delta)}{2\delta}\right)P}{\epsilon^3 n}\right)$ □

3.6 Sliding Window

At time slot c, the current window of size n is defined as the set of all events within the n most recent time slots, i.e. slots $(c-n+1)$ to c, both endpoints inclusive. An item d is defined

www.manaraa.com

to be α-persistent in a sliding window of size n if it occurred in at least αn distinct time slots within the current window. We now present a distributed algorithm for approximately tracking the set of all α-persistent items within the sliding window of size n.

Intuition: The sliding window algorithm uses the same sampling technique as for the infinite window case, and if a site decides to track an item, it communicates with the other sites, following which each site sets up local state for this item, and future occurrences of this item are handled locally without requiring further communication. The main challenge with the sliding window case is that as future time slots arrive, old occurrences go out of scope and have to be removed from consideration from the data structures. At each site i, S_i is continuously updated to discard expired time slots for each item.

Unlike the algorithm for infinite window, for each item d that is tracked, the error due to sampling is a concern only as long as the starting time slot for tracking d, i.e. t_d, does not expire from the sliding window. After slot $t_d + n$, a summary of all subsequent occurrences of d are tracked approximately by the data structure. Thus the query processing will distinguish between the cases when the query is made after $t_d + n$ (Algo 6: lines 11-13), and when the query is made before $t_d + n$ (Algo 6: lines 6-10). Another change is that the distinct elements algorithm should work over sliding windows, rather than for the entire stream.

Algorithm Description: Similar to the infinite window version of persistence algorithm (Section 3.5), the sliding window algorithm has three parts: Algorithm 4 initializes data structures, Algorithm 5 is the algorithm run at each site, and Algorithm 6 gives the algorithm run at the coordinator node C. The input to our algorithm is the same as that of infinite window version, except that n, in this case, is the maximum number of timeslots in a window (Algorithm 4).

The algorithm (Algo 5) used at each site i is similar to the one used for infinite window. However, the distinct count data structure is now maintained by the distinct counting algorithm for a sliding window [22, 38, 93].

When a query is made, the coordinator C takes a union of the distinct count datastructures for the k sites (Algo 6: lines 2-5) and computes $\hat{n}_d$, number of distinct slots when d appeared in the current window, per the distinct counting algorithm for sliding windows. As discussed

www.manaraa.com

above, the query processing in our algorithm distinguishes between the cases when the query is made after $t_d + n$ (Algo 6: lines 11-13), and when the query is made before $t_d + n$ (Algo 6: lines 6-10). If a query is made before $t_d + n$, then the persistent items are tracked in the same way as done by the infinite window version of the algorithm, Algo 3 . However, if a query is made after $t_d + n$, then persistence of an item d is estimated as $\hat{n}_d$.

For relative error parameter $0 < \gamma < 1$ and an error probability parameter $0 < \upsilon < 1$, a distinct counter $\mathcal{D}_\gamma^\upsilon$ takes as input a stream of updates S and at any given time t maintains an estimate of $dist(\mathcal{R})$, the number of distinct elements in $\mathcal{R}$, for the elements that occurred in most recent n slots.

Theorem 5 ([38, 22, 93]). *There is a distinct counter $\mathcal{D}_\gamma^\upsilon$ that takes space $O(\frac{\log 1/\upsilon}{\gamma^2})$ words of space, and whenever a query is asked for $dist(\mathcal{R})$ in the most recent n time slots, returns an estimate X such that $\Pr[|X - dist(\mathcal{R})| > \gamma \cdot dist(\mathcal{R})] \leq \upsilon$, where $0 < \gamma < 1$ and $0 < \upsilon < 1$.*

A detailed description of the algorithm is presented in Algorithms 4, 5, and 6.

Algorithm 4: Sliding Window: Initialization

Input: m - Domain Size of identifiers; n- maximum no. of time slots in a window; α - persistence threshold; ϵ - error parameter; δ - error probability;

1 Hash function $h : ([1, m] \times [1, n]) \rightarrow (0, 1)$
2 Approx. parameters $\epsilon_1 \leftarrow c_\epsilon \epsilon$; $\epsilon_2 \leftarrow (1 - c_\epsilon)\epsilon/4\alpha$ `// $0 < c_\epsilon < 1$ is a constant`
3 Error Probability: $\delta_2 = c_\delta \delta$ `// $0 < c_\delta \leq min\left(1, \frac{2}{\log(1/\delta)}\right)$ is a constant`
4 Filter parameter $\tau \leftarrow \frac{2}{\epsilon_1 n}$
5 Threshold $T \leftarrow (1 - \epsilon_2)(\alpha n + 1 - \frac{1}{\tau})$
6 Sketch at coordinator $S \leftarrow \emptyset$
7 **for** *each site* $i = 1 \ldots k$ **do**
8 $\quad S_i \leftarrow \emptyset$

3.6.1 Sliding Window : Correctness

Let t_d be the slot when item d started to be tracked. Also, for a query q made on the distributed streams, let t_q be the last slot of the most recent window $[t_q - n + 1, t_q]$ on which query q has been posed.

www.manaraa.com

Algorithm 5: Sliding Window: Algorithm at node i

```
On receiving item (d, t) at node i
if (d, t) ∉ S_i then
    if h(d, t) < τ) then
        for every node j = 1 ... k, j ≠ i do
            Send "Start Tracking (d, t)" to j
        Send (d, t) to the coordinator
        // Create a new data structure for d
        S_i ← S_i ∪ {(d, D^{ε2}_{δ2}[i](d))}
        Insert t in D^{ε2}_{δ2}[i](d))
else // if d ∈ S_i
    Insert t in D^{ε2}_{δ2}[i](d)
On receiving message "Start Tracking (d, t)"
// Create a new data structure for d
S_i ← S_i ∪ {(d, D^{ε2}_{δ2}[i](d))}
Insert t in D^{ε2}_{δ2}[i](d)
```

Lemma 4. *Let $G(\tau)$ be the geometric random variable with parameter τ. Also, let n_d denote the number of distinct slots in the distributed streams where d appears in current window after being tracked by the algorithm. For each item d, n_d can be expressed differently depending on t_q and $G(\tau)$ in the following manner.*

1. **If** $t_q \leq t_d + n$*: if* $G(\tau) \leq p(d)$, $n_d = p(d) - G(\tau) + 1$, *else* $n_d = 0$.

2. **If** $t_q > t_d + n$*:* $n_d = p(d)$.

Proof. The proof of the above Lemma is divided into two parts, for the two cases described above. Proof of part 1) is the same as that of proof of Lemma 1.

For part 2), at $(t_d + n)$-th slot, the first slot when d was tracked expires, i.e. t_d expires. From the slot t_d onwards, every occurrence of d is tracked. Hence, if the current window of the most recent n slots does not include t_d, then persistence of d, $p(d)$, over the current window is the number of occurrences of d in the current window, i.e. $p(d) = n_d$. □

Lemma 5. Low False Negative: *An α-persistent item d having a persistence $p(d) \geq \alpha n$ during the most recent n slots is not reported as α-persistent by the coordinator in Algorithm 6 with a probability at most*

www.manaraa.com

Algorithm 6: Sliding Window:Algorithm at the coordinator C:

On receiving tuple (d, t_d) **:** $S \leftarrow S \cup \{(d,t)\}$.
On receiving a query for the set of Persistent Items: for *each* $(d, t_d) \in S$ **do**
  for $i = 1 \dots k$ **do**
    Compute the union of $D^{\delta_2}_{\epsilon_2}[i](d)$ data structures over all sites i.
    Let $\hat{n}_d$ be the estimate of the distinct count over this union.
  if $t \leq t_d + n$ **then** `// t - current slot`
    if $t_d < (1/\tau)$ **then**
      $\hat{p}(d) \leftarrow \hat{n}_d + t_d$;
    else
      $\hat{p}(d) \leftarrow \hat{n}_d + (1/\tau)$;
  else
    $\hat{p}(d) \leftarrow \hat{n}_d$;
    $T \leftarrow (1 - \epsilon_2)\alpha n$;
  if $\hat{p}(d) \geq T$ **then**
    Report d as α-persistent item;

1. $e^{-2} + \frac{2\delta}{\log(1/\delta)}$ *if* $t_q \leq t_d + n$
2. $\frac{2\delta}{\log(1/\delta)}$ *if* $t_q > t_d + n$

Proof. Consider an α-persistent item d, with persistence $p(d) \geq \alpha n$.

1. **If** $t_q \leq t_d + n$**:** Proof is same as that of Lemma 2, where n is the maximum number of slots in current window instead of the entire distributed stream. Note that $\delta_2 = c_\delta \delta$. The error probability can be reduced to δ by running $\frac{\log(\delta)}{\log(e^{-2}+c_\delta\delta)}$ parallel instances.
2. **If** $t_q > t_d + n$ Let A denote the event that d is not reported, i.e. the event that false negative occurs. Using the proof in Lemma 4, we can also conclude that the estimate of persistence of d, $\hat{p}(d)$, in the most recent n slots is the estimate returned by the distinct counter for item d, $\hat{n}_d$, to approximate the count of distinct number of slots where d occurred over the most recent n slots, i.e. $\hat{p}(d) = \hat{n}_d$. Per the above algorithm, item d is not reported as α-persistent if $\hat{p}(d) < T$. $\Pr[A] = \Pr[\hat{p}(d) < T] = \Pr[\hat{n}_d < T]$.

www.manaraa.com

Let B denote the event $(1-\epsilon_2)n_d < \hat{n}_d$. From the above algorithm, $T = (1-\epsilon_2)\alpha n$.

$$\begin{aligned}
\Pr[A|B] &= \Pr[\hat{n}_d < T | (1-\epsilon_2)n_d < \hat{n}_d] \\
&\leq \Pr[(1-\epsilon_2)n_d < T] \\
&= \Pr[(1-\epsilon_2)p(d) < T] \qquad \text{using Lemma 4} \\
&\leq \Pr[(1-\epsilon_2)\alpha n < T] \qquad \text{given } p(d) \geq \alpha n \\
&= 0 \qquad \text{substituting } T
\end{aligned}$$

The probability of $\bar{B}$ depends on the guarantee given by the distinct counter $\mathcal{D}_{\epsilon_2}^{\delta_2}$. Using Theorem 5 we have: $\Pr[(1-\epsilon_2)n_d \leq \hat{n}_d \leq (1+\epsilon_2)n_d] \geq (1-\delta_2)$ Hence,

$$\begin{aligned}
\Pr[\bar{B}] &= \Pr[\hat{n}_d < (1-\epsilon_2)n_d] \\
&\leq \Pr[\hat{n}_d < (1-\epsilon_2)n_d] + \Pr[\hat{n}_d > (1+\epsilon_2)n_d] \\
&\leq \delta_2 \qquad \leq \frac{2\delta}{\log(1/\delta)}
\end{aligned}$$

Using $\Pr[A] \leq \Pr[A|B] + \Pr[\bar{B}]$, we get the desired result.

□

Lemma 6. Low False Positive: *An item d which is far from persistent in the most recent n slots, i.e, whose persistence $p(d) < (\alpha-\epsilon)n$ in the current window of size n, is reported as α-persistent with probability at most $\frac{2\delta}{\log(1/\delta)}$.*

Proof. Consider an item d with persistence $p(d) < (\alpha-\epsilon)n$. d is far from persistent. The proof has two cases based on when query is posed with respect to t_d, the slot when d is first tracked by algorithm.

1. **If $t_q \leq t_d + n$:**, proof of above lemma is the same as that of Lemma 3, n denoting the maximum number of slots in a sliding window, and not the total number of slots in the entire distributed stream.
2. **If $t_q > t_d + n$:** Persistence of an item d is the number of occurrences of d within the current window which equals n_d, i.e. $p(d) = n_d$ from Lemma 4. Also, $\hat{p}(d) = \hat{n}_d$. Let

www.manaraa.com

A denote the event that d is reported, i.e. the false positive occurs. Also, per the above algorithm, d is reported if $\hat{p}(d) \geq T$.

$$\Pr[A] = \Pr[\hat{p}(d) \geq T] = \Pr[\hat{n}_d \geq T]$$

Let B denote the event $(1+\epsilon_2)n_d \geq \hat{n}_d$. We have $T = (1-\epsilon_2)\alpha n$ and $\epsilon_2 = (1-c_\epsilon)\epsilon/4\alpha$.

$$\begin{aligned}
\Pr[A|B] &= \Pr[\hat{n}_d \geq T | (1+\epsilon_2)n_d \geq \hat{n}_d] \\
&\leq \Pr[(1+\epsilon_2)n_d \geq T] \\
&= \Pr[(1+\epsilon_2)p(d) \geq T] \\
&\leq \Pr\left[(\alpha-\epsilon)n \geq \frac{T}{(1+\epsilon_2)}\right] \qquad \text{given } p(d) < (\alpha-\epsilon)n \\
&\leq \Pr\left[0 \leq \alpha n - \epsilon n - (1-2\epsilon_2)\alpha n\right] \qquad \text{substituting } T, \text{ and } \tfrac{1-\epsilon_2}{1+\epsilon_2} > (1-2\epsilon_2) \\
&= \Pr\left[0 \leq 2\epsilon_2\alpha - \epsilon\right] = 0 \qquad \text{since } \epsilon_2 < \epsilon/2\alpha
\end{aligned}$$

Using Theorem 5,

$$\begin{aligned}
\Pr[\bar{B}] &= \Pr[\hat{n}_d > (1+\epsilon_2)n_d] \\
&\leq \Pr[\hat{n}_d < (1-\epsilon_2)n_d] + \Pr[\hat{n}_d > (1+\epsilon_2)n_d] \\
&\leq \delta_2 \qquad \leq \frac{2\delta}{\log(1/\delta)}
\end{aligned}$$

□

Theorem 6. *By running at least* $\frac{\log\delta}{\log(e^{-2}+c_\delta\delta)}$ *parallel instances of the above algorithm, where* $c_\delta \leq \frac{2}{\log(1/\delta)}$, *$\alpha$-persistent items can be tracked with the following properties:*

1. *An item d with persistence $p(d) \geq (\alpha n)$ is reported as α-persistent with probability at least $1-\delta$.*
2. *An item d with persistence $p(d) < (\alpha-\epsilon)n$ is not reported as persistent with probability at least $1-\delta$.*

Proof. Suppose we run θ parallel instances of the above algorithm, and take the union of the items returned by all the instances. For the first part, consider an α-persistent item d. If d is not returned, it must not be returned by any of the instances. With respect to the time of

www.manaraa.com

arrival of a persistent item d, if a query q is posed on a window $[t_q - n + 1, t_q]$, then we have two cases: 1) If $t_q \leq t_d + n$, then d is reported with probability $(e^{-2} + c_\delta \delta)^\theta$, where $0 < c_\delta \leq \frac{2}{\log(1/\delta)}$. So, if we run $\frac{\log \delta}{\log(e^{-2} + c_\delta \delta)}$ parallel instances, the probability that false negative occurs is at most δ. 2) If $t_q > t_d + n$, then d is reported with probability $\left(\frac{2\delta}{\log(1/\delta)}\right)^\theta$, and the proof follows.

For an item d with persistence less than $(\alpha - \epsilon)n$, the proof is similar to the case of infinite window. □

3.6.2 Sliding Window: Complexity Analysis

Theorem 7. ***Expected Space:*** *Total expected space required by the sliding window algorithm per site is* $O\left(\frac{\alpha^2 \log\left(\frac{\log(1/\delta)}{2\delta}\right) P}{\epsilon^3 n}\right)$ *and the expected space complexity over all sites is* $O\left(\frac{k\alpha^2 \log\left(\frac{\log(1/\delta)}{2\delta}\right) P}{\epsilon^3 n}\right)$

Proof. Here, P is the sum of the persistence (total persistence till current time slot, c) of all the distinct items in the distributed stream during the period $[c - n + 1, c]$. The proof is similar to that of Theorem 3. □

Theorem 8. ***Low Communication Overhead:*** *The expected communication complexity of the sliding window algorithm over all sites is* $O\left(\frac{k\alpha^2 \log\left(\frac{\log(1/\delta)}{2\delta}\right) P}{\epsilon^3 n}\right)$

Proof. Proof is similar to that of Theorem 4, except that the communication occurs between sites only when a new item is tracked and for the same item no further communication is done until the query is posed. □

3.7 Experiments

We report on the observed performance of our implementation of the infinite window and the sliding window algorithms.

Data We have used synthetic as well as real-world data sets for our experiments. The real-world data is a *network traffic trace* from CAIDA taken at a US west coast OC48 peering link for a large ISP in 2002 and 2003, where we consider each source-destination pair to be an item. The network trace has approximately 400 million tuples with about 26 million distinct

www.manaraa.com

items. The trace has been captured over a duration of 1 hour. We also generated synthetic data using a zipfian distribution, with $\alpha = 1.5$. This dataset has 500 million tuples, consisting of approximately 40 million distinct items.

Algorithms We compared our algorithm with two other algorithms. The first one which we call algorithm A, is an exact distributed algorithm for tracking persistence, which identifies α-persistent items by keeping track of the exact persistence number of each item, through maintaining the distinct slots it occurred in. Algorithm A is the most expensive in terms of space and communication.

The second algorithm, which we call Algorithm B, is a small space algorithm which selectively tracks items using a hash-based sampler in the same way as our algorithm does. But for each item d that Algorithm B tracks, it computes the exact number of distinct time slots where the item reappears by maintaining a list of distinct time slots of appearance. Since Algorithm B does not incur an error cost in counting the number of re-occurrences once it starts tracking an item, it can actually track fewer items than our algorithm for the same value of ϵ. However, for each item tracked, B has to maintain significant amount of state for the item (a list of all slots where the item appeared, at each site).

Our experiments evaluate the performance of our algorithms in terms of communication cost and accuracy, where accuracy is measured through the false positive and false negative rates. We also performed experiments to show the effect of the width of time slot on the communication cost of the dataset. Unless specified otherwise, we set the error probability δ to e^{-2}. For the infinite window case, we divided the real world trace into 34 million non-overlapping time slots (width of each time slot being 0.1 millisecond) and the zipfian dataset into 36 million non-overlapping slots. To evaluate the sliding window version of the algorithm, we considered a window size of 30 million distinct time slots (width of each time slot being 0.1 millisecond) for real world trace and 25 million distinct time slots for zipfian dataset.

Communication Cost vs Accuracy In the first set of experiments, we kept the number of sites constant, at 10, and varied the approximation parameter ϵ. The results from the exper-

www.manaraa.com

iments on zipfian data is shown in Figure 3.1a for the infinite window case and in Figure 3.1b for the sliding window case. From this data we make the following observations.

There is a clear trade-off between accuracy and communication cost. The communication cost decreases as we increase the value of ϵ for our algorithm as well as for Algorithm B. The communication cost incurred by our algorithm for both infinite and sliding window is an order of magnitude smaller than that of Algorithms B and A. In fact, we observe that the communication cost of algorithm B is only slightly smaller than the naive algorithm A.

The results of experiments on the network trace are presented in Figures 3.1c and 3.1d. These are similar to the results for zipfian data, and our algorithm has significantly lower communication cost when compared with Algorithms A and B. However, since the size of the dataset is smaller and it has a relatively small number of distinct time slots, the cost incurred by algorithm B is low. Hence, in this case the communication cost of algorithm B, though higher than our algorithm, is not as high as in the case of zipfian data file. This shows that the benefits of our algorithm are even greater on large datasets with a large number of time slots.

Communication Cost vs Number of Sites In order to evaluate the scalability with the size of the distributed system, we varied the number of sites in the system, while keeping the approximation error ϵ fixed at 0.025. The results for zipfian data are shown in Figures 3.2a and 3.2b, and for the network trace in Figures 3.2c and 3.2d. We observe that the communication cost of the algorithm increases linearly with the number of sites in the system, in accordance with the theoretical results. The results also show that our algorithm consistently performs better than the other two algorithms. Algorithms A and B also show a similar linear increase in communication cost with the number of sites.

The increase in communication cost of our algorithms and that of algorithm B is due to the fact that every site has a copy of each item tracked in the distributed system. Hence, increase in the number of sites would lead to an almost linear increase of the space requirement and communication cost. In the case of the algorithm A, the reason for the increase in the communication cost is as follows: multiple appearance of an item in the same time slot does not affect the size of the datastructure maintained by the site, but if an item appears multiple

www.manaraa.com

times in the same slot across different sites, then multiple copies (same as the number of sites where they appear) of the items need to be maintained, increasing the communication cost.

Communication Cost vs Width of Timeslot We performed experiments to study the effect of the width of time slots for a given dataset on communication cost. We keep the value of ϵ fixed at .025 and the number of sites fixed at 10. We use real network trace for our experiments. Using the same dataset traces, we vary the width of each time slot from 0.1 millisecond to 2 milliseconds and measure the communication cost of our algorithm.

The results are shown in Figure 3.3a for infinite window and Figure 3.3b for sliding window. We observe that the communication cost of algorithm A decreases slightly with the increase in the width of time slot. The reason is that the distinct number of time slots decreases with the increase in the width of time slot, hence, algorithm A has to maintain a smaller datastructure. However, interestingly, the communication costs of our algorithm and of Algorithm B increase with the width of time slot, if we keep the threshold for persistence the same. This is due to the fact that as the width of the time slot increases, the number of persistent items for a given persistence threshold increases, and the data structures become larger. Though the number of messages decreases, the size of the messages increases, leading to an overall increased communication cost.

We have also included a graph showing the change in the total number of elements tracked across the distributed system when the width of the time slot is varied. The results are shown in figure 3.4a for infinite window and in figure 3.4b for sliding window. The number of elements tracked does not vary for algorithm A as it tracks all the elements in the distributed dataset. However, for our algorithm and for algorithm B, the number of elements tracked increases with the increase in the width of time slot.

We also compared the space cost, which is defined as the total space taken by the data structures at the sites and the coordinator. In general, the space taken by our algorithm is much smaller than that of Algorithms A and B, for most parameter settings. In Table 3.1, we show the space cost of each algorithm for the sliding windows scenario, on the zipfian data on a distributed system of 10 nodes, for different values of ϵ. The space cost of A is constant, since

www.manaraa.com

it is unaffected by the setting of ϵ, while that of B is rather large due to the need to maintain the exact set of distinct time slots where the tracked elements appeared. The results for the other data sets, and for the infinite windows version are similar.

Table 3.1: Space cost for Zipfian data on system of 10 nodes for algorithms A, B and our algorithm (SS) for sliding windows

Epsilon	Space taken by SS (in MBytes)	Space taken by B (in MBytes)	Space taken by A (in MBytes)
.01	149.129	1089.92	1860.38
.02	50.3261	1068.78	1860.38
.03	24.7846	1045.55	1860.38
.04	14.4616	1032.57	1860.38
.05	9.01553	1024.79	1860.38
.06	6.40722	1021.77	1860.38
.07	4.57717	1017.46	1860.38

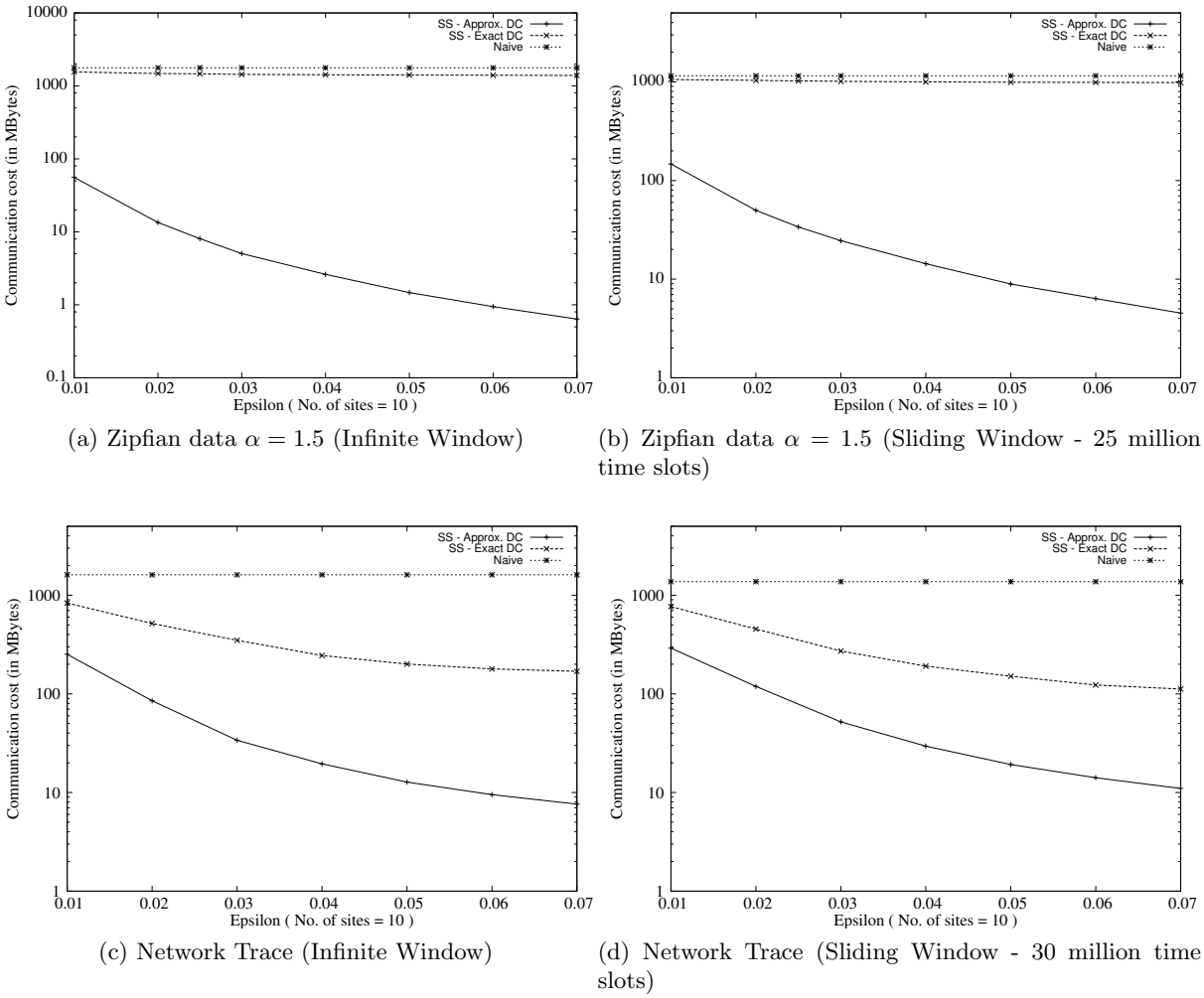
Accuracy We measure the actual false negative rate and the false positive rate of our algorithm for different values of δ, using 10 sites, keeping the value of ϵ fixed at 0.025. We use real network trace and zipfian dataset for our experiments. For the experiments corresponding to the sliding window version of our algorithm, we consider the window size of 30 million time slots for the network trace and 25 million time slots for zipfian.

According to our paper, we report an item d as persistent if its approximate persistence $\hat{p}_d$ is at least αn. Also, an item d is not reported as persistent if its estimated persistence $\hat{p}_d$ is less than $(\alpha n - \epsilon n)$. Note that for the infinite window version of the algorithm, n denotes the total number of time slots in the entire dataset, and for the sliding window version of the algorithm, it denotes the maximum number of time slots in the current window.

We define the false negative rate as a fraction of the persistent items which were not reported as persistent, and the false positive rate as a fraction of the non-persistent items which were reported as persistent. Per Theorem 2, the false negative rate and the false positive rate given by our algorithm is bounded by error probability δ.

The false negative rate for the zipfian and the network trace is shown in Figure 3.5a for infinite window and Figure 3.5b for sliding window. For this experiment, we vary δ from 0.001

www.manaraa.com



(a) Zipfian data $\alpha = 1.5$ (Infinite Window)

(b) Zipfian data $\alpha = 1.5$ (Sliding Window - 25 million time slots)

(c) Network Trace (Infinite Window)

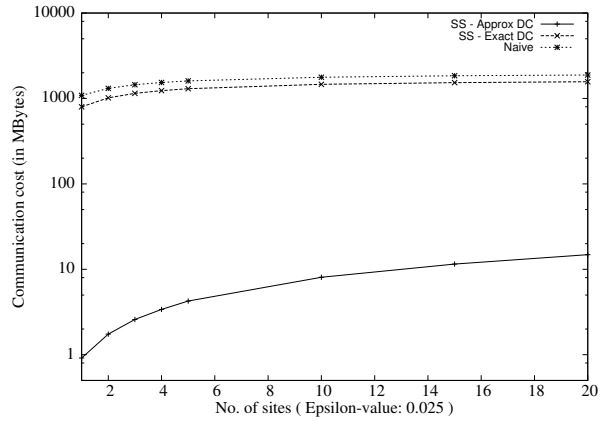
(d) Network Trace (Sliding Window - 30 million time slots)

Figure 3.1: Communication Overhead on varying relative error 'ϵ', for 10 sites

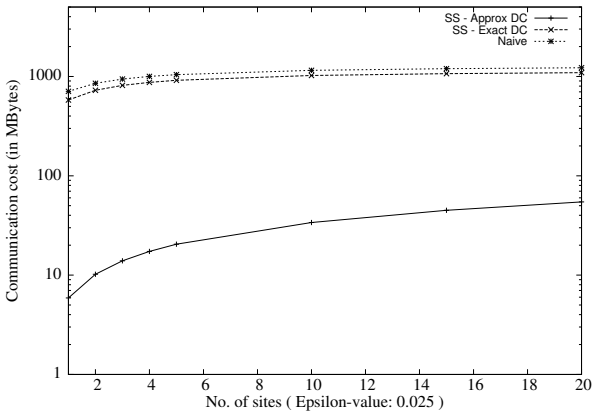
to 0.1. The plot named "Error Threshold" plots the maximum expected error for each value of δ. We observe that the false negative rate for both datasets is always less than the error probability δ. In fact, for our experiments, the false negative rate of network trace and zipfian did not exceed 0.025 for any value of δ. We also observe that for $\delta = 0.001$, there are no false negatives.

Similarly, we observe that the the false positive rates for the zipfian and the network trace, shown in Figure 3.5c for infinite window and in Figure 3.5d for sliding window, is much below

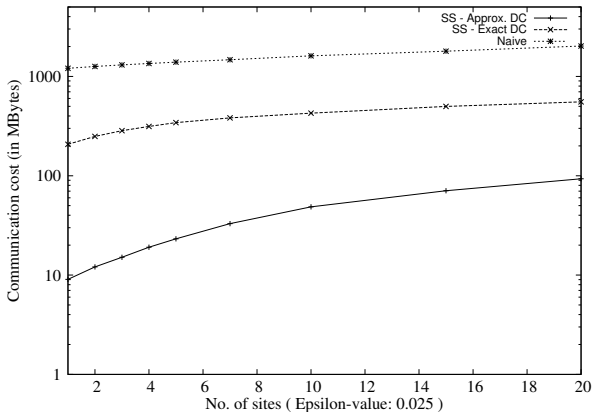
www.manaraa.com



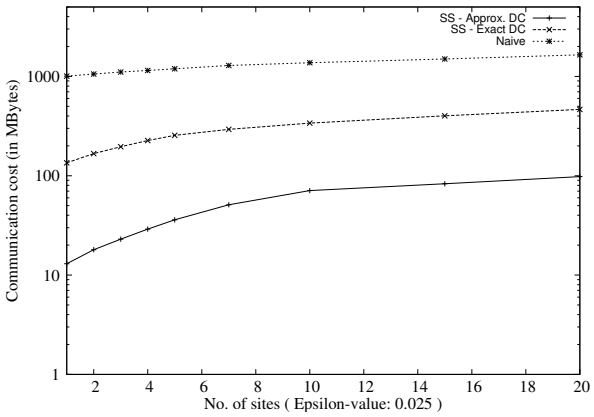
(a) Zipfian $\alpha = 1.5$ (Infinite Window)



(b) Zipfian $\alpha = 1.5$ (Sliding Window - 25 million time slots)



(c) Network Trace (Infinite Window)



(d) Network Trace (Sliding Window - 30 million time slots)

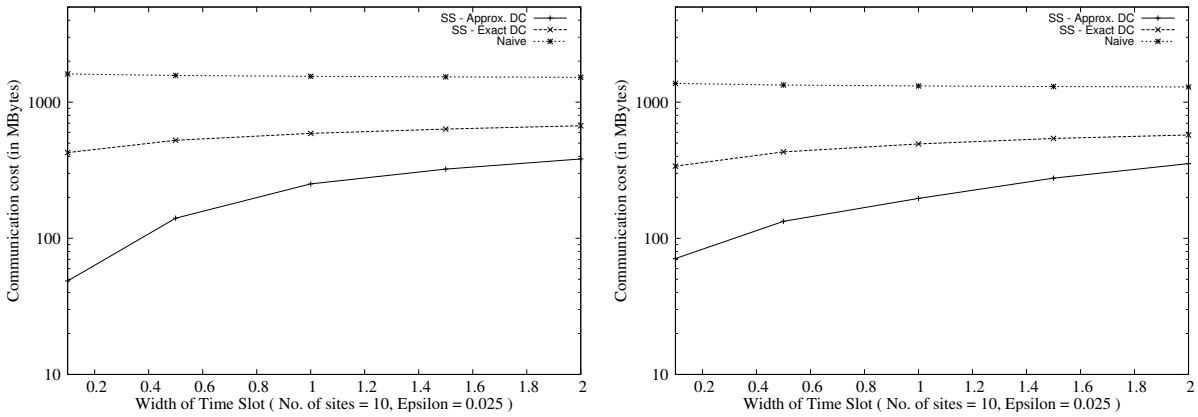
Figure 3.2: Communication Overhead on varying number of sites for $\epsilon = 0.025$

the error threshold. The false positive rate of network trace and zipfian given by both infinite window and sliding window version of our algorithm, are in the order of 10^{-5} to 10^{-7}.

3.8 Conclusion

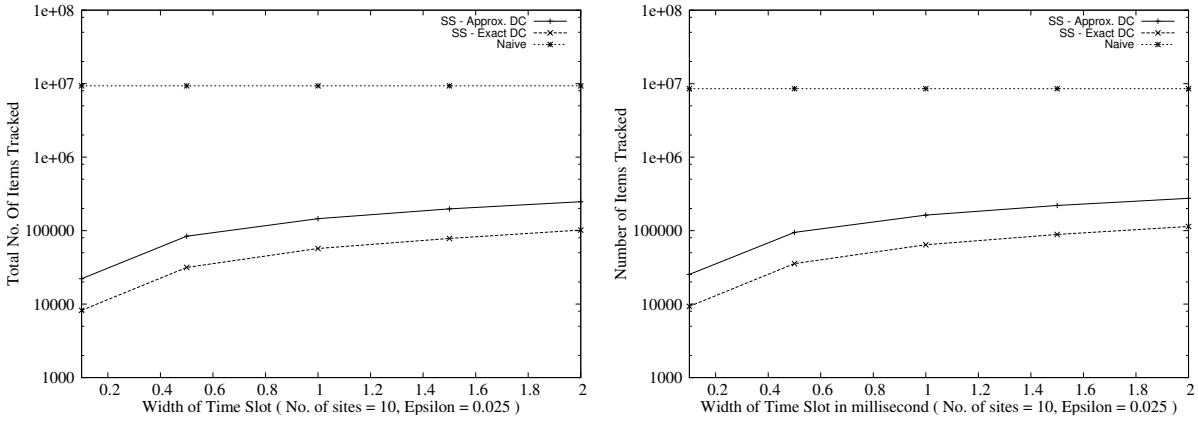
We presented algorithms for communication-efficient monitoring of persistent items in a distributed stream of events. These can help detect situations such as when a malicious adversary is establishing a regularly spaced connection to a remote entity, but is trying to evade detection through keeping the volume of communication low and by having the communication

www.manaraa.com



(a) Network Trace (Infinite Window) (b) Network Trace (50 minutes Sliding Window)

Figure 3.3: Communication Overhead as a function of the width of time slot (in milliseconds)

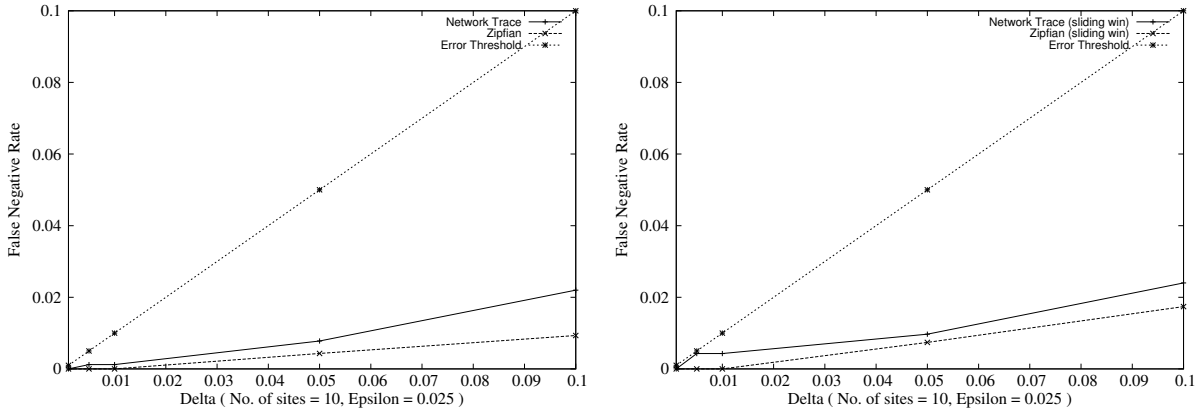


(a) Network Trace (Infinite Window) (b) Network Trace (50 minutes Sliding Window)

Figure 3.4: Total number of items tracked across all sites as a function of the width of the time slot (in milliseconds)

originate from different sites. The total distributed state maintained by our algorithms is far less than the number of distinct items observed in the stream, and the communication overhead is also small compared with the number of events and the number of items observed. Our experimental evaluations show that the communication cost and memory cost of our algorithms are much smaller than those of straightforward algorithms, and their false positive and false negative rates are typically much lower than theoretical predictions.

www.manaraa.com



(a) False Negative Rate - Infinite Window

(b) False Negative Rate - Sliding Window (30 million time slots)

(c) False Positive Rate - Infinite Window

(d) False Positive Rate - Sliding Window (30 million time slots)

Figure 3.5: False Negative Rate and False Positive Rate as a function of δ for Network Trace and Zipfian

www.manaraa.com

CHAPTER 4. QUANTILE ESTIMATION FROM THE UNION OF STREAMING AND HISTORICAL DATA

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 $\texttt{rank}(e, D)$, is defined as the number of elements in D that are less than or equal to e.

Definition 2. *For $0 < \phi < 1$, a ϕ-quantile of D is defined as the smallest element e such that $\texttt{rank}(e, D) \geq \phi n$.*

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.

We consider a setup where a live 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 to as the "streaming data" or "data stream". Data that has been archived in the warehouse is called the "historical data". The historical data is typically larger than the data stream by a factor of the order of thousands. See Figure 4.1 for an illustration of the setup for data processing.

Let U denote the universe that has a total order among all elements. Let $\mathcal{H}$ denote the historical data that has already been loaded into the warehouse and $\mathcal{R}$ denote the live streaming data. Let n denote the size of $\mathcal{H}$ and m denote the size of $\mathcal{R}$. Let $\mathcal{H}[1] < \mathcal{H}[2] < \ldots, < \mathcal{H}[n]$ be the elements of $\mathcal{H}$ according to their total order in U and $\mathcal{R}[1] < \mathcal{R}[2] < \ldots < \mathcal{R}[m]$ be

www.manaraa.com

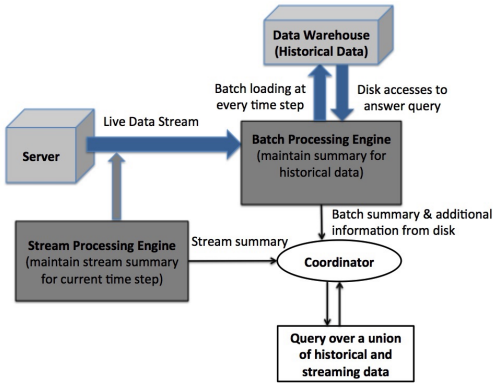


Figure 4.1: Setup for Integrated Processing of Historical and Streaming data

the elements of $\mathcal{R}$ according to their total order. Note that the elements of $\mathcal{H}$ and $\mathcal{R}$ are not necessarily stored in a sorted order, and the notation $\mathcal{H}[i]$ and $\mathcal{R}[i]$ are only to help with the explanation. Let $\mathcal{T} = \mathcal{H} \cup \mathcal{R}$, and let N denote the size of $\mathcal{T}$.

The problem 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 [63], 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.

4.1 Goal

Given an approximation parameter $\epsilon \in (0, 1]$, and a constant $\phi \in (0, 1]$, design a method that identifies an approximate ϕ-quantile, e, from $\mathcal{T}$ such that $|rank(e, \mathcal{T}) - \phi N| < \epsilon m$, where N is the total number of elements in the union of historical and streaming data, $\mathcal{T}$, and m is the number of elements in the streaming data (or the data stream).

The amount of main memory available is much smaller than either $\mathcal{R}$ or $\mathcal{H}$, but secondary storage is abundant. At query time, the processor can answer quantile queries using a combination of data structures stored in main memory as well as by making queries to the disk resident data.

www.manaraa.com

Note that the approximation error in our formulation is only ϵm. Typical streaming algorithms for quantiles, when applied to our problem, yield an approximation error of ϵN, which can be much larger than ϵm, since $N \gg m$.

4.2 Contribution

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, which is typically much smaller than N, the size of $\mathcal{T}$. The answer becomes increasingly accurate as the size of the historical data under consideration increases.
- We provide a theoretical upper bound on the memory requirement of the algorithm. We show (both theoretically as well as in practice) that the resulting accuracy-memory trade-off 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 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 (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. The number of disk accesses required to load a batch into the warehouse is typically not much more than what is required to simply write the batch to disk.

4.3 Related Work

Quantile computation on large data is a well-studied problem [63, 2, 58, 15, 46, 36, 73], both in the context of stored data [63] and streaming data [2, 58, 15, 46, 36, 73]. To compute

www.manaraa.com

quantiles from stored data from 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 [63] 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 [58] 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, [58, 59] that identify ϵ-approximate ϕ-quantiles with probability at least $(1-\delta)$, $0 < \delta < 1$, using $O\left(\frac{1}{\epsilon}\log^2(\frac{1}{\epsilon}\log^2(\frac{1}{\delta}))\right)$ memory.

Greenwald and Khanna [46] present a deterministic single pass streaming algorithm for ϵ-approximate quantiles with worst case space requirement $O\left(\frac{1}{\epsilon}\log(\epsilon N)\right)$. and Shrivastava et al. [73] 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. [83] performed an experimental evaluation of different streaming algorithms [46, 73, 59]. They concluded that MRL99 [59] and Greenwald-Khanna [46] 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 [59] called RANDOM, which performs slightly better than [59] in terms of the processing time.

Current literature on integrated processing of historical and streaming data has focused on developing efficient architectural models for data integration [68, 23, 24, 81, 5, 1, 11, 91]. In particular, [44] proposes a framework called DataDepot designed to store streaming data, thus allowing analysis of massive amount of historical data over a time frame of many years. [91]

www.manaraa.com

proposes a model which enables data analysis over the union of historical and streaming data. Our work is complementary to these in that we investigate query processing techniques that are applicable to these architectural models.

4.4 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 [46] or the QDigest algorithm [73] to $\mathcal{T}$. The streaming algorithm runs continuously and processes each batch of data, before the 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, the new batch 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 an error significantly smaller than ϵN, using a similar amount of main memory as the pure streaming algorithm and limited number of disk I/Os.

www.manaraa.com

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 not 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 that allows for fast updates, but still has only a small number of sorted partitions.

In addition to the on-disk structure, we maintain an in-memory summary $S(\mathcal{H})$, for $\mathcal{H}$ that provides us quick access to elements at different ranks within each sorted partition. This summary of historical is updated periodically 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, $S(\mathcal{R})$, 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, $S(\mathcal{R})$ is reset. Quantile queries are answered by by using a combination of $S(\mathcal{H})$ and $S(\mathcal{R})$ 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 needs to make significantly fewer disk I/Os compared to the strawman approach.

4.4.1 Processing Historical Data

Data is archived into the warehouse at the end of every time-step. When a new data set $\mathcal{D}$ is added to $\mathcal{H}$, $\mathcal{D}$ is sorted and stored as a separate data partition, instead of merging with an existing data partition of $\mathcal{H}$. This step ensures that the entire historical data is not accessed at every time step. This also enables us to maintain data from different time steps separately,

www.manaraa.com

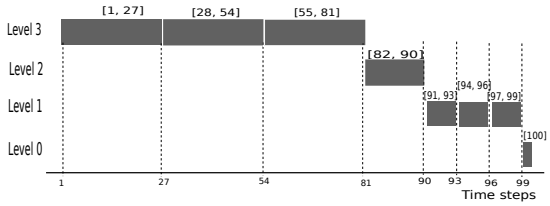


Figure 4.2: Structure of the data partitions for historical data $\mathcal{H}$, over 100 time steps, for $\kappa = 3$. Each segment in the picture is a data partition, and is labeled with the range of time steps it spans.

hence allowing for windowed queries restricted to a specific time window of a certain number of time steps.

Each data partition of $\mathcal{H}$ has a logical "level", a small positive integer, associated with it. Let $\kappa > 1$ be a small integer parameter chosen before the algorithm begins execution. 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 $\mathcal{D}_\ell^0$, $\mathcal{D}_\ell^1$, ..., $\mathcal{D}_\ell^{j-1}$, $j \leq \kappa$. Suppose a newly arrived dataset $\mathcal{D}$, of size η needs to be added to $\mathcal{H}$. Then, $\mathcal{D}$ is first sorted and stored at level 0 of $\mathcal{H}$; the sorting can be performed in-memory, or using an external sort [63, 45], depending on the size of $\mathcal{D}$.

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. 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. Partitions at higher levels contain aggregated information about data 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 $\mathcal{H}$, a common operation is to determine the number of elements in $\mathcal{H}$ that are lesser than a given value. To answer this query, our structure needs to consult a few (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. Figure 4.2 shows the organization of data partitions of the historical data after 100 time steps, for $\kappa = 3$.

www.manaraa.com

The parameter κ that determines the maximum number of partitions at each level, is called the"merge threshold". The maximum number of logical levels in $\mathcal{H}$ is $\log_\kappa T$, where T is the total number of time steps.

Construction of $S(\mathcal{H})$ In addition to the on-disk structure for $\mathcal{H}$, we maintain an in-memory summary for each data partition of $\mathcal{H}$. The in-memory structure $S(\mathcal{H})$ consists of the summaries of every partition. Naturally following a parallel structure to the on-disk organization, $S(\mathcal{H})$ consists of a set of independent data structures at $\lambda = \lceil \log_\kappa T \rceil$ different logical levels, one corresponding to the different logical levels of $\mathcal{H}$, $S_0(\mathcal{H})$, $S_1(\mathcal{H})$, $S_2(\mathcal{H})$, ..., $S_{\lambda-1}(\mathcal{H})$.

Let $\epsilon_1 = \frac{\epsilon}{2}$ and $\beta = \lceil \frac{1}{\epsilon_1} + 1 \rceil$. For $0 \leq \ell \leq \lambda - 1$, each data structure $S_\ell(\mathcal{H})$, is a set of no more than κ independent summaries, $S_\ell^0(\mathcal{H})$, $S_\ell^1(\mathcal{H})$, ..., $S_\ell^{j-1}(\mathcal{H})$, $j \leq \kappa$. The data partition in $\mathcal{H}$ corresponding to $S_\ell^i(\mathcal{H})$ is denoted $\mathcal{D}_\ell^i$. Each summary, $S_\ell^i(\mathcal{H})$, is an array of length β.

When a new data partition$\mathcal{D}$ is created, either due to adding a new dataset to $\mathcal{H}$ at level 0, or due to merging smaller partitions, a new summary is generated for this partition as follows. After $\mathcal{D}$ is sorted, it is divided into β equal subsequences, and the first element of each subsequence is chosen into the summary. 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 the batch summary, beyond those taken for generating the new data partition. Algorithm 8 shows the steps to process $\mathcal{H}$ and compute $S(\mathcal{H})$ at each time step. This uses a subroutine Algorithm Merge-Partitions(ℓ) (described in Algorithm 9) for merging smaller partitions at level ℓ into a single partition at level $(\ell + 1)$.

Algorithm 7: Initialize Data Structures

Input parameters: error parameter ϵ
$\epsilon_1 \leftarrow \epsilon/2$, $\epsilon_2 \leftarrow \epsilon/4$
$T \leftarrow 0$ `// T - number of time steps`
$S(\mathcal{H}) \leftarrow \emptyset$ `// Initialization of batch summary`
$S(\mathcal{R}) \leftarrow \emptyset$ `// Initialization of stream summary`

www.manaraa.com

Algorithm 8: Batch-Update($\mathcal{D}$)

```
/* Update H and S(H) with a new dataset D of size η.                    */
T ← (T + 1);
Sort D and add as a new partition to level 0;
ℓ ← 0;
while (More than κ partitions at level ℓ) do
    Merge all partitions at level ℓ into partition D′, and compute its summary S′ using
    Algorithm Merge-Partitions(ℓ);
    Add partition D′ to level (ℓ + 1) and S′ to S_{ℓ+1}(H);
    S_ℓ(H) ← ∅;
    ℓ ← (ℓ + 1);
```

Algorithm 9: Merge-Partitions(ℓ)

```
/* Merge all partitions at level ℓ into a single partition and recompute
   its summary.  There must be κ partitions at level ℓ.                 */
Multi-way merge the sorted partitions ∪_{j=1}^{κ} D_ℓ^j into a single partition D′ using a single
pass through the partitions;
// Create a summary for D′
η ← size of D′;
S′ ← ∅;
for i from 0 to β − 1 do
    Add element at rank (iϵ1η) within D′ to S′;
return D′ and S′;
```

4.4.2 Processing the Data Stream

We process the data stream using an ϵ_2-approximate streaming quantiles algorithm such as [46, 73], where $\epsilon_2 = \epsilon/4$. Given a desired rank r, such an 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 have used Greenwald-Khanna [46] algorithm.

When a query is received, the streaming algorithm is used to generate a summary $S(\mathcal{R})$, an array of length $\beta' = \lceil \frac{1}{\epsilon_2} + 1 \rceil$, using steps shown in Algorithm 10. Algorithm 10 uses the streaming algorithm to find elements of approximate rank $i\epsilon_2 m$ from $\mathcal{R}$, for i ranging from 0

www.manaraa.com

to $1/\epsilon_2$, and add these elements to $S(\mathcal{R})$. Due to the guarantee provided by the streaming algorithm, each of these β' elements are identified with a maximum error of $\epsilon_2 m$ in their ranks.

4.4.3 Answering Quantile Query over a 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 and (2) a slower, but more accurate response using the in-memory summaries as well as disk accesses. The steps to compute a Quick Response and an Accurate Response has been described in Algorithm 10.

When a query is received, Algorithm 10 generates constructs $S(\mathcal{R})$ using the streaming algorithm, as described in Section 4.4.2. After the construction of $S(\mathcal{R})$, Algorithm 10 takes a union of $S(\mathcal{H})$ and $S(\mathcal{R})$, $S(\mathcal{T})$, such that the elements in $S(\mathcal{T})$ are stored in ascending order. Let $S(\mathcal{T})[i]$ be the i-th element of $S(\mathcal{T})$, $0 \leq i < |S(\mathcal{T})|$.

Both the Quick Response and the Accurate Response requires us to determine the minimum and maximum possible ranks of each element $S(\mathcal{T})[i]$ of $S(\mathcal{T})$, in $\mathcal{T}$, for $i = 0, 1, \ldots |S(\mathcal{T})| - 1$. Let the minimum and maximum possible ranks of element $S(\mathcal{T})[i]$ be denoted as L_i and U_i respectively. We compute L_i and U_i in the following manner.

Since $S(\mathcal{T}) = S(\mathcal{H}) \cup S(\mathcal{R})$, $S(\mathcal{T})[i]$ either comes from $S(\mathcal{H})$ or from $S(\mathcal{R})$, i.e. $S(\mathcal{T})[i] \in S(\mathcal{H})$ or $S(\mathcal{T})[i] \in S(\mathcal{R})$.

If $S(\mathcal{T})[i] \in S(\mathcal{H})$, then element $S(\mathcal{T})[i]$ has to be from one of the data partitions, say $\mathcal{D}^x_\ell$ of size η^x_ℓ, from $\mathcal{H}$. Note that since each element in $S(\mathcal{H})$ is indexed, the rank of $S(\mathcal{T})[i]$ in $\mathcal{D}^x_\ell$ is known exactly. On the other hand, if $S(\mathcal{T})[i] \in S(\mathcal{R})$, then the element $S(\mathcal{T})[i]$ has to be from the data stream $\mathcal{R}$. Since ϵ_2 approximation was used to generate $S(\mathcal{R})$, each element in $S(\mathcal{R})$ has a maximum error of $\epsilon_2 m$ in its rank. Hence, the rank of $S(\mathcal{T})[i]$ in $\mathcal{R}$ is known approximately, with a maximum error in its rank being $\epsilon_2 m$.

Since elements in $S(\mathcal{T})$ are arranged in ascending order, the 0-th element of $S(\mathcal{T})$ is evidently the smallest element in $\mathcal{T}$. Hence, $L_0 = U_0 = 0$. We compute L_i from L_{i-1} as follows: as follows: 1) If $S(\mathcal{T})[i] \in S(\mathcal{H})$,

www.manaraa.com

Suppose $S(\mathcal{T})[i]$ is from data partition $\mathcal{D}_\ell^x$ of size η_ℓ^x

$$L_i = L_{i-1} + 1, \qquad \text{if } (rank(S(\mathcal{T})[i], \mathcal{D}_\ell^x) = 0)$$

$$L_i = L_{i-1} + \epsilon_1 \eta_\ell^x, \qquad \text{otherwise}$$

2) If $S(\mathcal{T})[i] \in S(\mathcal{R})$,

$$L_i = L_{i-1} + 1, \qquad \text{if } (rank(S(\mathcal{T})[i], \mathcal{R}) = 0)$$

$$\text{OR } (rank(S(\mathcal{T})[i], \mathcal{R}) = \epsilon_2 m)$$

$$L_i = L_{i-1} + \epsilon_2 m, \qquad \text{otherwise}$$

Similarly, U_i is computed from U_{i-1} as follows:

1) If $S(\mathcal{T})[i-1] \in S(\mathcal{H})$,

Suppose $S(\mathcal{T})[i]$ is from data partition $\mathcal{D}_\ell^x$ of size η_ℓ^x, then

$$U_i = U_{i-1} + \epsilon_1 \eta_\ell^x - 1$$

2) If $S(\mathcal{T})[i] \in S(\mathcal{R})$,

$$U_i = U_{i-1} + 2\epsilon_2 m - 1 \qquad \text{if } rank(S(\mathcal{T})[i-1], \mathcal{R}) = 0$$

$$U_i = U_{i-1} + \epsilon_2 m - 1, \qquad \text{otherwise}$$

Observation 1. *The following guarantees hold for L_i and U_i:*

1. *For the i-th element of $S(\mathcal{T})$, $S(\mathcal{T})[i]$, $0 < i \leq |S(\mathcal{T})|$,*

   $U_i - L_i < \epsilon_1 n + 3\epsilon_2 m < \epsilon N$

2. *For two consecutive elements in $S(\mathcal{T})$, when arranged in ascending order, $S(\mathcal{T})[i-1]$ and $S(\mathcal{T})[i]$,*

   a. $Li - L_{i-1} \leq \max(\epsilon_1 \max\limits_{1 \leq j \leq \kappa, 0 \leq \ell \leq \log T} \{|\mathcal{D}_\ell^j|\}, \epsilon_2 m)$, and,

   b. $Ui - U_{i-1} \leq \max(\epsilon_1 \max\limits_{1 \leq j \leq \kappa, 0 \leq \ell \leq \log T} \{|\mathcal{D}_\ell^j|\}, \epsilon_2 m)$

4.4.3.1 Quick Response

On receiving a quantile query, our algorithm provides a quick and rough answer that has a similar accuracy as that of a pure streaming approach. Algorithm 10 describes the steps for providing a quick answer to the query.

www.manaraa.com

Algorithm 10: Quantile-Query(r, $\mathcal{T}$)

```
/* Answer query for finding element of rank r from T                    */
// Construct S(R)
for i from 0 to β' − 1 do
    Find element e at approximate rank iε2m using the streaming algorithm
    S(R) ← S(R) ∪ e
S(T) ← S(H) ∪ S(R), arranged in sorted order
// Quick Response to the query
for i = 0, 1, ..., |S(T)| − 1 do
    if L_i > r then
        return e_{i−1}
// Accurate Response to the query
(u, v) ← Generate-Filter(S(T), r)
return Query(S(R), S(H), u, v)
```

Lemma 7. *Given a query to identify an element of rank r in $\mathcal{T}$, Algorithm 10 returns an element of rank $\hat{r}$ such that $\hat{r} \leq |r - \epsilon N|$.*

Proof. Let the algorithm return $S(\mathcal{T})[i]$ as a quick response to the query for finding element of rank r from $\mathcal{T}$. For simplicity, let $S(\mathcal{T})[i]$ be denoted as e_i. The Algorithm 10 returns an element e_i as a quick answer, such that $L_i \leq r < L_{i+1}$. In this situation, the maximum possible rank, U_i, of the returned element e_i in $\mathcal{T}$ can have two possible conditions : 1) $U_i \geq r$ or 2) $U_i < r$.

1. **If $U_i \geq r$:**

   We know $U_i \geq r$ and $L_i \leq r < L_{i+1}$, or $L_i \leq r \leq U_i$. Now,

   $$U_i - L_i < \epsilon N, \qquad \text{from Observation 1,}$$
   $$\implies r - L_i \leq \epsilon N \leq U_i - r, \qquad \text{since } L_i \leq r \leq U_i$$

2. **If $U_i < r$:**

   Since $r < L_{i+1}$, therefore $r < U_{i+1}$. Hence, we have, $L_i \leq r < U_{i+1}$.

www.manaraa.com

Now, from Observation 1,

$$L_{i+1} - L_i \leq \max(\epsilon_1 \max_{j,\ell}(\log T\{|\mathcal{D}_\ell^j|\}), \epsilon_2 m), \quad \text{and,}$$

$$U_{i+1} - L_{i+1} \leq \epsilon_1 n + 3\epsilon_2 m = \epsilon(n/2 + 3m/4)$$

$$\implies U_{i+1} - L_i \leq \epsilon N$$

$$\implies r - L_i \leq \epsilon N \leq U_{i+1} - r, \qquad \text{since } L_i \leq r < U_{i+1}$$

We observe that in both the conditions above, the maximum error in the answer given by Quick Response is ϵN. □

4.4.3.2 Accurate Response

The algorithm for Accurate Response has been described in Algorithm 10. As a first step, we find a pair of elements u and v from $S(\mathcal{T})$ such that the element of desired rank r is guaranteed to lie between these elements, i.e $rank(u, \mathcal{T}) \leq r \leq rank(v, \mathcal{T})$. We refer to this pair as Filters. We generate the filters by calling the Algorithm Generate-Filters(X, r) (described in 11), in the following manner:

For simplicity, let $e_i = S(\mathcal{T})[i]$. We then find elements u and v using Algorithm 11 such that:

$$u = e_i, \qquad U_i = \max_{0 \leq j < |S(\mathcal{T})|} \{U_j\}, \qquad U_i \leq r$$

$$v = e_i, \qquad L_i = \min_{0 \leq j < |S(\mathcal{T})|} \{L_j\}, \qquad L_i \geq r$$

Lemma 8. *Given rank r and summary $S(\mathcal{T})$, Algorithms 10 and 11 can find elements $u \in S(total)$ and $v \in S(\mathcal{T})$, such that $rank(u, \mathcal{T}) \leq r \leq rank(v, \mathcal{T})$ and $(rank(v, \mathcal{T}) - rank(u, \mathcal{T})) \leq 2\epsilon N$, where $N = |\mathcal{T}|$.*

www.manaraa.com

Algorithm 11: *Generate-Filters($S(\mathcal{T})$, r)*

```
/* Find filters u and v from S(T) such that element of rank r in T is
   guaranteed to lie between the filters                              */
for i from 0 to |S(T)| − 1 do
    Let e_i be the i-th element of S(T)
    if U_i ≤ r then
        u ← e_i
    if L_i ≥ r then
        v ← e_i; break
return (u, v)
```

Proof.

$$\text{We know,} \qquad u = S(\mathcal{T})[x], \qquad U_x = \max_{0 \le j < |S(\mathcal{T})|} \{U_j\}, \qquad U_x \le r$$

$$v = S(\mathcal{T})[y], \qquad L_y = \min_{0 \le j < |S(\mathcal{T})|} \{L_j\}, \qquad L_y \ge r$$

$$\text{Hence, } U_x \le r < U_{x+1}, \qquad L_{y-1} < r \le L_y$$

From Observation 1, since,

$$U_{x+1} - U_x \le \max(\epsilon_1(\max_{j,\ell}\{|\mathcal{D}_\ell^j|\}), \epsilon_2 m), \text{and,}$$

$$U_x - L_x < \epsilon_1 n + 3\epsilon_2 m$$

We can conclude that,

$$U_x \ge r - \max(\epsilon_1(\max_{j,\ell}\{|\mathcal{D}_\ell^j|)\}, \epsilon_2 m)$$

$$\implies L_x \ge r - \max(\epsilon_1(\max_{j,\ell}\{|\mathcal{D}_\ell^j|)\}, \epsilon_2 m) - (\epsilon_1 n + 3\epsilon_2 m)$$

www.manaraa.com

Similarly, we can conclude,

$$U_y \leq r + \max(\epsilon_1(\max_{j,\ell}\{|\mathcal{D}_\ell^j|)\}, \epsilon_2 m) + (\epsilon_1 n + 3\epsilon_2 m)$$

Hence,

$$\begin{aligned} U_y - L_x &\leq (2\epsilon_1 n + 6\epsilon_2 m) + \max(2\epsilon_1(\max_{j,\ell}\{|\mathcal{D}_\ell^j|)\}, 2\epsilon_2 m) \\ &\leq 2\epsilon N, \qquad \text{since, } \epsilon_1 = \epsilon/2 \text{ and } \epsilon_2 = \epsilon/4 \end{aligned}$$

which proves the lemma. □

Once we have obtained u and v, we make a series of recursive calls to the function Query($S(\mathcal{R})$, $S(\mathcal{H})$, u, v) (described in Algorithm 12) to narrow down the range of elements between u and v, by finding a new pair of filters with smaller interval size recursively. The objective is to narrow down the range of elements between the pair of filters to a point where all the consecutive elements between the filters in $\mathcal{H}$ can be loaded into memory. These consecutive elements from $\mathcal{H}$ are used in combination with $S(\mathcal{R})$ to accurately answer the quantile query.

4.4.4 Correctness

Lemma 9. *Given a query to identify an element of rank r from $\mathcal{T}$, Algorithm 10 returns an element whose rank in $\mathcal{T}$ is $\hat{r}$ such that $|r - \hat{r}| \leq O(\epsilon m)$, where $0 < \epsilon < 1$ is an error parameter*

.

Proof. Per Algorithm 10, we start with finding a pair of filters u and v from $S(\mathcal{T})$ such that $rank(u, \mathcal{T}) \leq r \leq rank(v, \mathcal{T})$ and $rank(v, \mathcal{T}) - rank(u, \mathcal{T}) \leq 2\epsilon N$. Using Algorithm 12, we recursively revisit the disk to find a new pair of filters u' and v' such that $rank(u, \mathcal{T}) \leq r \leq rank(v, \mathcal{T})$ still holds true though the difference in the ranks of the new filters u' and v' decreases compared to the previous filter, i.e $rank(v', \mathcal{T}) - rank(u', \mathcal{T}) < rank(v, \mathcal{T}) - rank(u, \mathcal{T})$. We recursively compute a new pair of filters u' and v' till we reach a point where maximum number of elements between v' and u' do not exceed $\frac{1}{\epsilon}$ i.e $rank(v', \mathcal{T}) - rank(u', \mathcal{T}) \leq \frac{1}{\epsilon}$ for each summary in $S(\mathcal{H})$.

www.manaraa.com

Algorithm 12: *Query($S(\mathcal{R})$, $S(\mathcal{H})$, u, v)*

```
/* Find accurate quantile from S(R) and S(H) using the filters u and v to
   make disk operations in H                                              */
Let z ← (u+v)/2
// Compute ρ_H - rank of z in H
for each summary S' in RS do
    // Let D' be the corresponding data partition
    Find largest element x in S' such that x ≤ u
    Find smallest element y in S' such that y ≥ v
    Let ranks of x and y in D' be l and r resp. ρ_H ← ρ_H + Rank-in-Batch(z, D', l, p)
// Compute ρ_R - rank of z in R
for i from 0 to |S(R) − 1| do
    if z ≥ S(R)[i] then
        ρ_R ← iε_2 m
ρ ← ρ_H + ρ_R
if (r < (ρ − εm)) then
    return Query(S(R), S(H), u, z)
else if (r > (ρ + εm)) then
    return Query(S(R), S(H), z, v)
else
    return z
```

Algorithm 13: *Rank-in-Batch(element e, partition $\mathcal{D}'$, l, p)*

```
/* Find rank of element e in partition D', given rank of e lies between l
   and p                                                                  */
if (D[(l + p)/2] ≤ e < (D[(l + p)/2 + 1]) then
    return (l+p)/2
else if (e < D^i_ℓ[(l + p)/2]) then
    Rank-in-Batch(element e, D', l, (l + p)/2)
else
    Rank-in-Batch(element e, D', (l + p)/2, p)
```

This implies that we have obtained all the consecutive elements from $\mathcal{H}$ between u and v into the query data structure Q, since we allocate a query workspace of $\frac{1}{\epsilon}$ for every summary. Q now becomes an exact data structure because the element in $\mathcal{T}$ of rank r is guaranteed to lie between filters u and v, and all the elements in $\mathcal{T}$ that lies between u and v and are originally

www.manaraa.com

from $\mathcal{H}$ is contained in Q. This also implies that had there not been the streaming data $\mathcal{R}$, Q would have returned the element of rank r with error 0.

However, there is an error of $\epsilon_2 m$ because $\mathcal{T}$ also includes element from $S(\mathcal{R})$. Hence the maximum error given by our algorithm is that due to the streaming part, i.e $O(\epsilon m)$ □

4.4.5 Complexity Analysis

4.4.5.1 Disk Accesses

Lemma 10. *The number of disk accesses required to update $\mathcal{H}$ and maintain the indices of $S(\mathcal{H})$ when a new dataset is added to $\mathcal{H}$ at each time step is amortized $O(\frac{n}{BT}\log(\frac{n}{B}))$, 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 $\mathcal{D}$ of size η arrives, disk I/Os are made for at most two reasons: 1) to sort and add $\mathcal{D}$ into a new separate partition in $\mathcal{H}$, and 2) if needed, to merge a few data partitions according to the algorithm.

Adding a new dataset $\mathcal{D}$, of size η, to $\mathcal{H}$ at a time step: A dataset to be added to $\mathcal{H}$ at a given time step is generally large, hence in many cases, it might not be feasible to sort the dataset using in-memory sorting methods. In such cases, an external sorting method is used. Suppose, we use a maximum memory size M for external sorting. Then, $\log(NB/M^2)/\log(M/B)+1$ complete scans of $\mathcal{D}$ is required to perform an external sorting over the dataset. In the first scan the following steps are taken: 1) A small chunk of data of size M is loaded in memory, where M is smaller than the size of the memory, 2) The chunk is then sorted using in-memory sorting techniques, 3) The sorted chunk of data is then written back to disk, 4) Steps 1, 2 and 3 are repeated. After the first scan of the dataset is completed, the disk has $\frac{\eta}{M}$ sorted chunks. Since, $\frac{M}{B}$ blocks can be loaded in memory at a time, the number of disk accesses required for the first scan is :

$$\sum_{i=1}^{\frac{\eta}{M}} \frac{M}{B} = \frac{\eta}{M}.\frac{M}{B} = \frac{\eta}{B}$$

After the first scan through the dataset, a $\frac{\eta}{M}$-way merge should be performed over the dataset. Since, at least one block from each of the $\frac{\eta}{M}$ chunks are loaded in memory for the

www.manaraa.com

merge, M must be greater than $\frac{\eta B}{M}$. If $M < \frac{\eta B}{M}$, only $\frac{M}{B}$ chunks can be merged at a time, using a $\frac{M}{B}$-way merge. Hence, after the first round of merge, the dataset is divided into $\frac{\eta B}{M^2}$ chunks. After the second round of merge, the number of chunks is reduced to $\frac{\eta B^2}{M^3}$. Hence, we can generalize and say that after j-th round, the number of chunks is reduced to $\frac{\eta B^j}{M^{j+1}}$. Thus, the number of rounds it takes for the number of chunks to reduce to $\frac{M}{B}$ so that a single $\frac{M}{B}$-way merge can be performed to finally produce one sorted dataset is obtained by solving for x in:

$$\frac{\eta B^x}{M^{x+1}} = \frac{M}{B}$$

which leads to $x = \log(\eta B/M^2)/\log(M/B)$.

Hence the, total number of disk accesses required for merging all the small sorted chunks of dataset produced after the first scan is $\log(\eta B/M^2)/\log(M/B).\eta/B$ or $\frac{\eta \log(\eta B/M^2)}{B \log(M/B)}$. Clearly, $M > B$, so we can conclude the following:

$$\begin{aligned}
\frac{\eta \log(\eta B/M^2)}{B \log(M/B)} &< \frac{\eta \log(\eta B/B^2)}{B \log(M/B)} \\
&< \frac{\eta \log \eta/B}{B \log M/B} \\
&< \frac{\eta}{B} \log(\eta/B) \qquad \text{if } M > B,\ \log(\tfrac{M}{B}) > 1
\end{aligned}$$

From above, we can conclude that sorting the datasets require $\frac{\eta}{B} \log(\eta/B) + \frac{\eta}{B}$ disk accesses or $O(\frac{\eta}{B} \log(\eta/B))$ disk accesses.

Merge κ datasets of higher level: The second kind of disk access is due to merging κ datasets together, for one or more higher levels, if required, as per the algorithm. Since, merging κ datasets of one level is independent of merging κ datasets of another level, this merge process can take place in parallel for each level whose datasets are required to be merged together. We use κ-way merge to merge the datasets of a level together. Since, κ is a constant parameter, a κ-way merge can be performed in a single round by loading one or more blocks from each of the κ datasets in memory. Hence, for a single merge, each block of κ datasets is accessed only once. After T time steps, the maximum number of logical levels across which all the data partitions are distributed is $\log_\kappa T$.

We can say that each of the T datasets that was added to $\mathcal{H}$ in the last T time steps is merged with other datasets into a larger data partition at most $\log_\kappa(T)$ times. We can say that

www.manaraa.com

each element of the dataset went through the merge procedure at most $\log_\kappa T$ times. Since, the sum of the size of the T datasets is n, total number of disk accesses required to merge the datasets as per the algorithm is at most $O(\frac{n \log_\kappa T}{B})$. Hence the amortized disk accesses for this operation is $O(\frac{n \log_\kappa (T)}{BT})$.

From above, we conclude that the amortized number of disk accesses per time step to update $\mathcal{H}$ and $S(\mathcal{H})$ is
$O(\frac{\eta}{B} \log (\eta/B) + \frac{N \log_\kappa (T)}{BT})$. Average size of data at every time step is n/T. Hence, replacing η with n/T, we get the amortized number of disk accesses as $O(\frac{n}{TB}(\log (n/TB) + \log_\kappa (T)))$. Considering κ is a constant, the number of disk accesses can be written as $O(\frac{n}{BT} \log (\frac{n}{B}))$. □

Lemma 11. *Given a query to identify an element of rank r from $\mathcal{T}$, the number of disk accesses required by the above algorithm to answer the query in the worst case is*
$O\left(\log^2 (\frac{\epsilon n}{BT}) \log (T)\right)$*, where ϵ is the error parameter of the algorithm.*

Proof. It takes $O\left(\frac{\log (\epsilon n)}{BT}\right)$ disk accesses to find the exact rank of elements u and v in the historical dataset $\mathcal{H}$, where n is the size of $\mathcal{H}$. We find the rank of element $(u+v)/2$ in each of $\log T$ datasets of H. Let us denote $(u+v)/2$ as z. We check if the quantile lies between elements u and z. If the quantile lies between u and z, then we find the rank of element $(u+z)/2$ and repeat the same process iteratively to narrow down the range. Else if the quantile lies between v and z, we find the rank of element $(v+z)/2$ and repeat the above process of iteratively narrowing down the range. The above method requires $O(\log (\frac{\epsilon n}{BT})$ iterations for each of $\log_\kappa T$ datasets.

In each of these iterations, we have to identify the rank of an element (requiring $O(\log (\epsilon n))$ disk accesses) per dataset, hence, the total disk access is $O\left(\log^2 (\frac{\epsilon n}{BT}) \log_\kappa (T)\right)$. □

4.4.5.2 Space Complexity

Lemma 12. *Total memory required by the summary for historical data, $S(\mathcal{H})$ is $O\left(\frac{\kappa \log_\kappa (T)}{\epsilon}\right)$.*

Proof. If we consider that number of time steps elapsed so far is T, and the number of data partitions merged together into a single partition is κ, then the maximum number of levels

www.manaraa.com

that $S(\mathcal{H})$ can have is $\log_\kappa(T)$. Each level has at most κ summaries, each of size $\frac{1}{\epsilon_1} = O\left(\frac{1}{\epsilon}\right)$. Hence, $\mathcal{H}$ requires a total space of $O\left(\frac{\kappa \log_\kappa(T)}{\epsilon}\right)$ to maintain summary $S(\mathcal{H})$. □

Lemma 13. *Total memory required by the summary of streaming data to maintain $S(\mathcal{R})$ 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 $O\left(\frac{\log(\epsilon m)}{\epsilon}\right)$ to return an element with an approximate rank r from $\mathcal{R}$. This algorithm is used by Algorithm 10 to construct $S(\mathcal{R})$. $S(\mathcal{R})$ requires $\lceil \frac{1}{\epsilon_2} + 1 = O\left(\frac{1}{\epsilon}\right)$ words of memory to store $ceilfrac1\epsilon_2 + 1$ elements from data stream.

Hence total memory required to construct and maintain $S(\mathcal{R})$ is $O\left(\frac{\log(\epsilon m)}{\epsilon}\right)$. □

Lemma 14. *The total main memory required by our algorithm to find an element of rank r from $\mathcal{T}$ is $O\left(\frac{1}{\epsilon}(\log(\epsilon m T))\right)$.*

Proof. We conclude from Lemmas 12 and 13 that the total memory required by our approach to maintain $S(\mathcal{H})$ and $S(\mathcal{R})$ is $O\left(\frac{1}{\epsilon}\left(\log(\epsilon m) + \frac{\kappa \log(T)}{\log(\kappa)}\right)\right)$. Since κ is a constant, the above space bound reduces to $O\left(\frac{1}{\epsilon}(\log(\epsilon m T))\right)$. Hence our overall memory requirement is $O\left(\frac{1}{\epsilon}\left(\log(\epsilon m) + \frac{\kappa \log(T)}{\log(\kappa)}\right)\right)$. □

We present the guarantees provided by our algorithm in Theorem 9.

Theorem 9. *Our algorithm, when given an integer $r \in [0, N)$, returns an element $e \in \mathcal{T}$ such that $(r - \epsilon m) \leq rank(e, \mathcal{T}) \leq (r + \epsilon m)$. The total main memory requirement of our algorithm is $O\left(\frac{1}{\epsilon}(\log(\epsilon m T))\right)$. The amortized number of disk accesses required to update $\mathcal{H}$ at each time step is $O(\frac{n}{BT}\log(\frac{n}{B}))$ and the number of disk accesses to answer a query is $O\left(\log(T)\log^2(\frac{\epsilon n}{BT})\right)$, where B is the block size, T is the number of time steps, m the size of the streaming data $\mathcal{R}$, n the size of historical data, and N the size of $\mathcal{T}$.*

We 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

www.manaraa.com

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 [53]. 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.

4.5 Experiments

We report on the results of experiments to evaluate the practical performance of our algorithm.

4.5.1 Experimental Setup

We used a 64-bit 4-core processor Red Hat Linux Machine, with a processor speed of 3.5GHz and 16GB RAM to run our experiments. We implemented all the algorithms using Java 7. We assumed a block size B of 100 KB.

Datasets We used two synthetic datasets "Normal" and "Uniform Random", and one real world dataset, derived from Wikipedia page view statistics.

- 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 (data that is not leaded into the warehouse yet) is 500MB. 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.
- 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 500MB. With 100 time steps, the total size of historical data is 50GB, with 500MB per time step.
- 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

[1] http://dumps.wikimedia.org/other/pagecounts-raw/

www.manaraa.com

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 58.5 GB.

Performance Metrics The two main performance measures of our algorithm are accuracy and number of disk accesses. While keeping the total main memory used across all algorithms, we compare the accuracy of different approaches, including the "quick response", and the more accurate response, both described in Algorithm 10.

The accuracy of an algorithm is measured using the *relative error*, defined as $\frac{|\phi N - r|}{\phi N}$ where ϕN is the rank desired by the quantile query, and r is the actual rank of the element returned by the algorithm. The performance of our algorithm is mainly measured in terms of number of disk accesses, since these tend to dominate the CPU costs. The quantities measured are: (1) the number of disk accesses required per time step to add a new dataset to the historical data, and (2) the number of disk accesses required to answer a query for the ϕ-quantile.

4.5.2 Algorithm and Optimizations

We used the Greenwald-Khanna streaming algorithm to process streaming data, in conjunction with out on-disk and in-memory batch structures. For comparison, we implemented the "pure-streaming" approach using two prominent deterministic streaming quantile algorithms - Greenwald-Khanna [46] and QDigest [73]. Given a memory budget of m bytes, we allocate $m/2$ bytes to the stream summary, and $m/2$ bytes to the batch summary. Since the size of the data at each time step is smaller than the memory size, we used in-memory sorting to sort new data.

When compared with Algorithm 10, we made an optimization in our program to reduce the number of disk accesses for query processing. As described, when a quantile query is posed, we use the Algorithm 11 to find a pair of elements within which the quantile is guaranteed to lie. Following this, we recursively call Algorithm 12 to narrow down the range $[u, v]$, always making sure that the quantile lies within the range. The optimization is that the search and narrowing of the range 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

www.manaraa.com

disk operations, and store the block in memory for further iterations. This yielded a significant reduction in the number of disk accesses; for example, on the "Uniform" dataset, it reduced the number of disk accesses from approximately 1500 to less than 300.

4.5.3 Results

Accuracy: Overall, we found that the accuracy of our algorithm is significantly better (by a factor of 100 for a dataset with 100 time steps) than that of a pure streaming algorithm, given the same amount of main memory. Our algorithm also provides a "quick response" using only in-memory structures, whose accuracy is comparable to that of the pure streaming algorithms. In all the Figures, we label our algorithm with the accurate response as "Our Algorithm", and the algorithm with the quick response as "Quick Response".

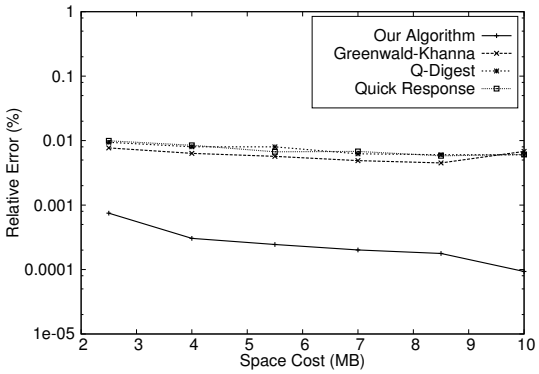
We measured the relative error of different approaches with a memory budget ranging from 2.5MB to 10 MB, and the results are shown in Figures 4.3a, 4.4a and 4.5a. We observed that the performance of the "quick response" is close to the QDigest algorithm, and the accuracy of our "accurate response" is significantly better than the rest.

In Figures 4.3b, 4.4b and 4.5b we show the relationship of the accuracy with κ. Keeping memory fixed at 7MB, we vary the value of κ from 2 to 40. We observe that the accuracy of the algorithm does not depend on the merge threshold κ, and this is consistent with Theorem 9 which says that the accuracy depends only on the input parameter ϵ and the size of the stream.

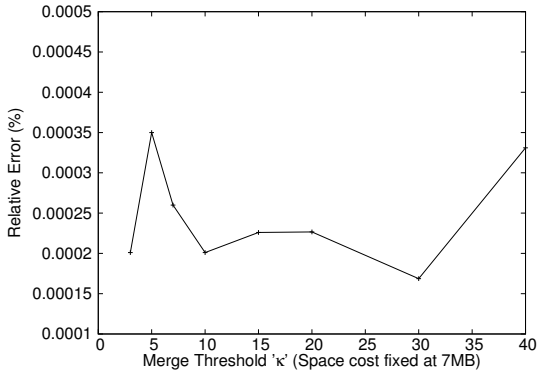
Disk Accesses for Adding a New Batch. Figures 4.6a, 4.7a and 4.8a show the number of disk accesses required to add a new dataset to the warehouse. This takes into account the number of disk operations to insert the new batch, merge older partitions (if needed) and to update the in-memory summaries of historical data. We measure the number of disk accesses for different values of κ. We observe that the number of disk accesses to load a new batch decreases as the value of κ increases. The reason is that as κ increases, the number of merges of partitions decreases overall, resulting in fewer additional disk accesses after the data has been merged.

While the average number of disk accesses per time step for uniform random data is 23000 for $\kappa = 9$ and 14000 for $\kappa = 10$ (Figure 4.6a), Figure 4.9 shows that for $\kappa = 9$, the number of

www.manaraa.com

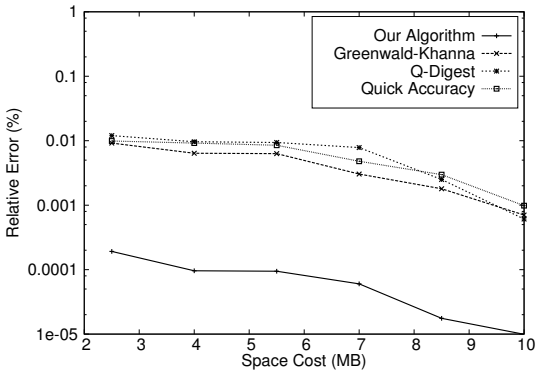


(a) Dependence of accuracy on memory budget

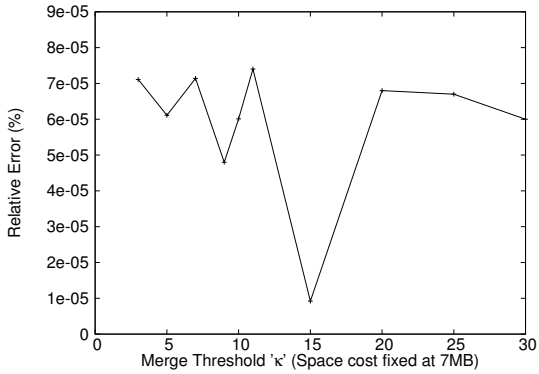


(b) Dependence of accuracy on κ

Figure 4.3: Accuracy for Uniform Random

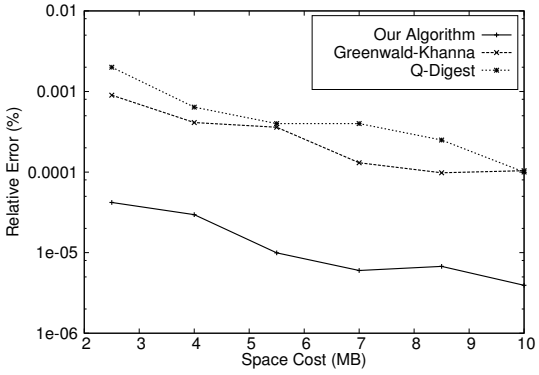


(a) Dependence of accuracy on memory budget

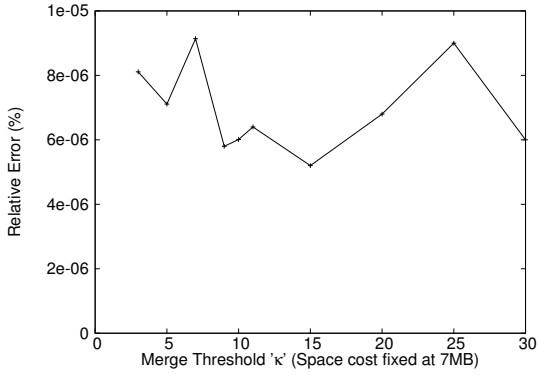


(b) Dependence of accuracy on κ

Figure 4.4: Accuracy for Normal



(a) Dependence of accuracy on memory budget

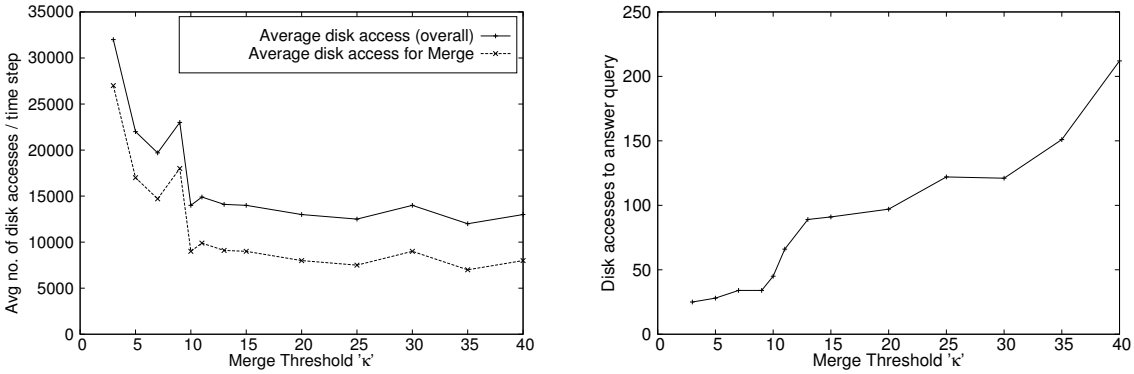


(b) Dependence of accuracy on κ

Figure 4.5: Accuracy for Wikipedia

www.manaraa.com

disk accesses per time step is as low as 5000 for 89 percent of all time steps. To understand this number, we note that with a 100KB block size, it takes 5000 disk accesses to write to disk a single new batch of 500MB. Hence, most of the time, the disk accesses are only to write the new partition to disk (after sorting in memory). According to Algorithm 8 every few time steps, it is required to merge different partitions into a single partition (using Algorithm 9), and this causes additional disk accesses. Similarly, we observe that the number of disk accesses per time step for $\kappa = 10$ is 5000 for for 91 percent of all time steps.



(a) Average Disk Accesses per time step for Update vs κ

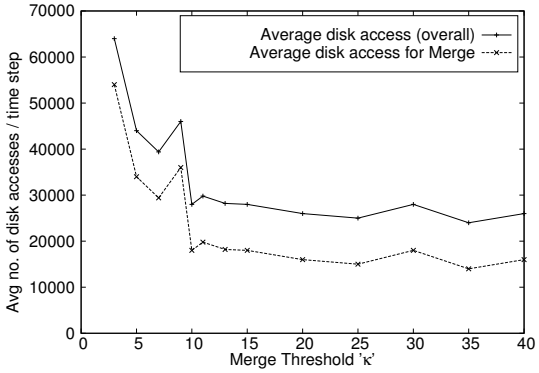
(b) No. of Disk Accesses for Query vs κ

Figure 4.6: Dependence of number of disk accesses over κ, for Uniform Random

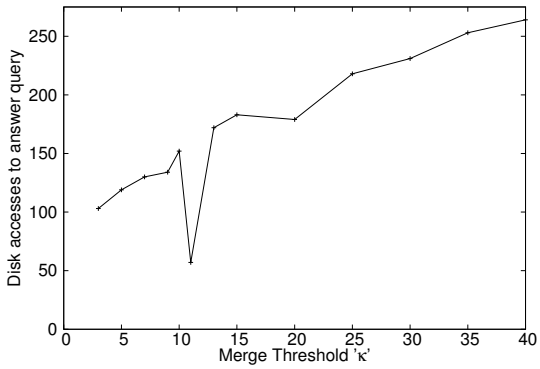
Disk Accesses for a Query. Figures 4.6b, 4.7b and 4.8b show the number of disk accesses required to answer a query, keeping the memory budget fixed at 7MB. We evaluated using different values for the memory budget and the results are similar. We observe that as κ increases the number of disk accesses increases. One reason is that as κ is increased, the number of data partitions per level increases. Since the total memory is fixed, the size of the summary per data partition decreases. A larger interval size in the summary leads to a larger number of disk accesses to answer a query accurately.

Queries over a Timed Window. Our algorithm support queries over a window consisting of a range of time steps. This is possible because historical datasets are stored across different

www.manaraa.com

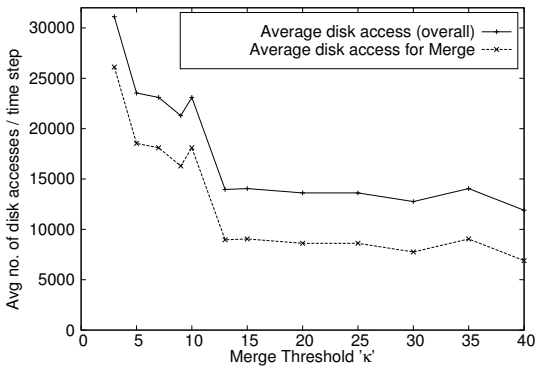


(a) Average Disk Accesses per time step for Update vs κ

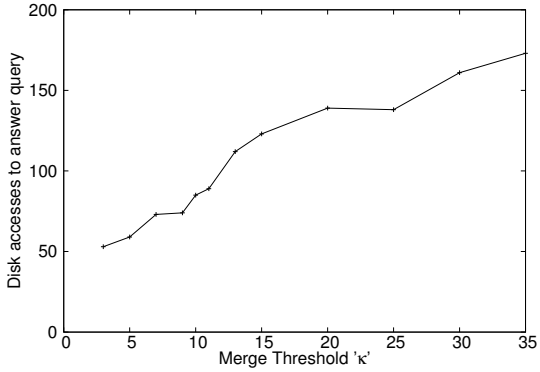


(b) No. of Disk Accesses for Query vs κ

Figure 4.7: Dependence of number of disk accesses over κ, for Normal



(a) Average Disk Accesses per time step for Update vs κ



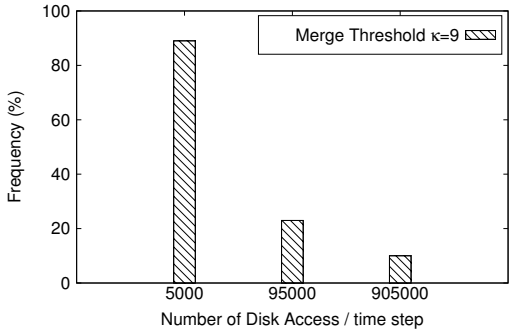
(b) No. of Disk Accesses for Query vs κ

Figure 4.8: Dependence of number of disk accesses over κ, for Wikipedia

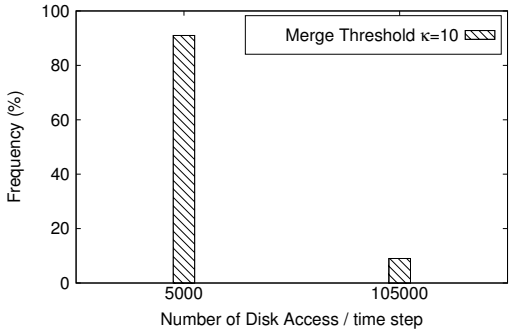
data partitions based on their time steps, and each partition consists of data in a contiguous range of time steps. In order to answer a query over a window, the algorithm accesses only those data partitions that corresponds to the time steps of the current window. Since older data partitions are merged together, it is easiest to answer queries over windows whose boundaries are aligned with the partition boundaries.

Figure 4.10 shows the permissible window sizes, in terms of time steps, over which a query can be answer, for Normal dataset with 100 time steps. We have shown the graphs for $\kappa = 3$ and $\kappa = 10$. We observe that for $\kappa = 3$, a query can be made over window of sizes 1, 4, 7, 10,

www.manaraa.com



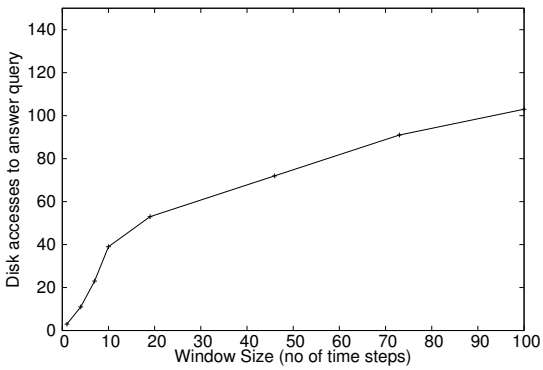
(a) Frequency of Disk Accesses for $\kappa = 9$, mean of all disk accesses



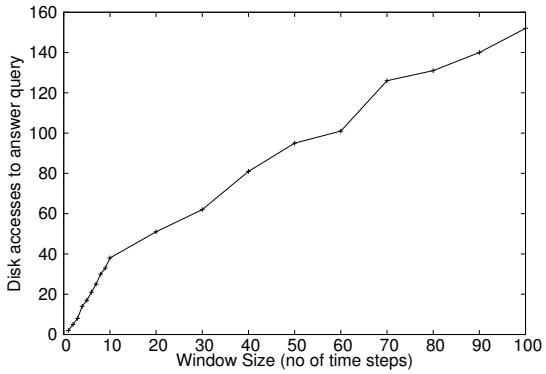
(b) Frequency of Disk Accesses for $\kappa = 10$

Figure 4.9: Frequency of Disk Accesses over all time steps, for Uniform Random

19, 45, 72, 100, 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. 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 query over different window increases with the size of the window due to an increase in the size of the data within the scope of the window.



(a) Query Disk Accesses vs Window size for $\kappa = 3$



(b) Query Disk Accesses vs Window size for $\kappa = 10$

Figure 4.10: Dependence of number of disk accesses over window sizes, for Normal

4.5.4 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 a processing archi-

www.manaraa.com

tecture for such an analysis, query processing strategies 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 trade-offs that are significantly better than current solutions that are based solely on disk-resident indexes or solely on streaming algorithms. The issues involved in a solution are how to combine a streaming algorithm that depends on in-memory summaries with an on-disk index for the historical data. Our theory and experiments indicate that ours is a practical algorithm, potentially scalable to very large historical data.

There are some natural directions for future work. First, can we improve the trade-off between accuracy, memory and disk accesses through improved data structures? Another direction is to consider other classes of aggregates in this model of integrated processing of historical and streaming data.

www.manaraa.com

CHAPTER 5. SUMMARY AND DISCUSSION

5.1 Conclusion

As real-time data analytics is becoming increasingly important, data stream analytics play a key role in finding patterns in various scenarios like internet traffic monitoring, user behavior analysis, Internet of things and so on. We considered three user requirements for data analytics : distributed streaming, sliding window and integration of historical and streaming data. We solved the following prominent streaming problems for each of these user requirements : 1) performed detailed experimental evaluation of distinct counting streaming algorithms over sliding window, 2) designed a communication efficient algorithm to identify persistent elements from distributed streams over both infinite and sliding window, provided strong theoretical guarantees and performed detailed experimental evaluation of the proposed algorithms, and 3) designed a low cost algorithm to find quantiles from a union of historical and streaming data, provided strong theoretical guarantees and performed detailed experimental evaluation.

Though we solved the above described problems, there are many important database problems that remain open for these user requirements in the context of large distributed data, such as finding distinct count or heavy hitters from a union of historical and streaming data. These problems constitute an interesting future course of research.

www.manaraa.com

Bibliography

[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 *Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] R. Agrawal and A. N. Swami. A one-pass space-efficient algorithm for finding quantiles. In *Proceedings of the 7th Intl. Conf. Management of Data (COMAD-95)*, 1995.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (STOC)*, pages 20–29, 1996.

[4] M. M. Astrahan, M. Schkolnick, and K.-Y. Whang. Approximating the number of unique values of an attribute without sorting. *Inf. Syst.*, 12(1):11–15, jan 1987.

[5] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee. Moirae: History-enhanced monitoring. In *Conference on Innovative Data Systems Research (CIDR)*, pages 375–386. www.cidrdb.org, 2007.

[6] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques (RANDOM)*, pages 1–10, 2002.

[7] V. Braverman and R. Ostrovsky. Smooth histograms for sliding windows. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 283–293, 2007.

www.manaraa.com

[8] V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling from sliding windows. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 147–156, 2009.

[9] C. Busch and S. Tirthapura. A deterministic algorithm for summarizing asynchronous streams over a sliding window. In *STACS*, pages 465–476, 2007.

[10] L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

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

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 668–668, 2003.

[13] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 268–279, 2000.

[14] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 693–703, 2002.

[15] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 436–447, New York, NY, USA, 1998. ACM.

[16] A. Chen and J. Cao. Distinct counting with a self-learning bitmap. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE)*, pages 1171–1174. IEEE Computer Society, 2009.

[17] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2):1530–1541, aug 2008.

www.manaraa.com

[18] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic udafs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 35–46. ACM, 2004.

[19] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[20] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. Optimal sampling from distributed streams. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 77–86, 2010.

[21] G. Cormode, S. Muthukrishnan, and W. Zhuang. What's different: Distributed, continuous monitoring of duplicate-resilient aggregates on data streams (icde). In *Proceedings of the 22nd International Conference on Data Engineering*, pages 57–, 2006.

[22] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, June 2002.

[23] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, pages 243–254, New York, NY, USA, 2011. ACM.

[24] N. Dindar, P. M. Fischer, and N. Tatbul. Dejavu: A complex event processing system for pattern matching over live and historical data streams. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, pages 399–400, New York, NY, USA, 2011. ACM.

[25] M. Durand and P. Flajolet. Loglog counting of large cardinalities (extended abstract). In *Proceedings of ESA 2003, 11th Annual European Symposium on Algorithms*, pages 605–617, 2003.

[26] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high-speed links. *IEEE/ACM Trans. Netw.*, 14(5):925–937, oct 2006.

[27] D. Eyers, T. Freudenreich, A. Margara, S. Frischbier, P. Pietzuch, and P. Eugster. Living in the present: On-the-fly information processing in scalable web architectures. In *Proceedings*

www.manaraa.com

of the 2nd International Workshop on Cloud Computing Platforms (CloudCP), pages 6:1–6:6, 2012.

[28] U. Fiedler and B. Plattner. Using latency quantiles to engineer qos guarantees for web services. In *Proceedings of the 11th International Conference on Quality of Service*, IWQoS'03, pages 345–362. Springer-Verlag, 2003.

[29] P. Flajolet. On adaptive sampling. *Computing*, 43(4):391–400, feb 1990.

[30] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, sep 1985.

[31] C. Fraleigh, S. Moon, C. Diot, B. Lyles, and F. Tobagi. Architecture of a passive monitoring system for backbone ip networks. Technical Report TR00-ATL-101-801, Sprint Labs, 2000.

[32] E. Fusy and F. Giroire. Estimating the number of active flows in a data stream over a sliding window. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithmics and Combinatorics*, pages 223–231, 2007.

[33] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 13–24, 2001.

[34] E. Gelenbe and D. Gardy. The size of projections of relations satisfying a functional dependency. In *Proceedings of the 8th International Conference on Very Large Data Bases*, pages 325–333, 1982.

[35] P. B. Gibbons. Distinct-values estimation over data streams. In *In Data Stream Management: Processing High-Speed Data*. Springer, 2009.

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

[37] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 281–291, 2001.

www.manaraa.com

[38] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures(SPAA)*, pages 63–72, 2002.

[39] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. *Theory of Computing Systems*, 37:457–478, may 2004.

[40] F. Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Appl. Math.*, 157(2):406–427, Jan. 2009.

[41] F. Giroire, J. Chandrashekar, N. Taft, E. Schooler, and D. Papagiannaki. Exploiting temporal persistence to detect covert botnet channels. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 326–345, 2009.

[42] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement (IMC)*, pages 173–178, 2003.

[43] L. Golab and T. Johnson. Consistency in a stream warehouse. In *Online Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 114–122, Asilomar, CA, USA,, January 9-12 2011.

[44] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 847–854, New York, NY, USA, 2009. ACM.

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

[46] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 58–66, 2001.

[47] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB)*, pages 311–322, 1995.

www.manaraa.com

[48] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*, pages 683–692, New York, NY, USA, 2013.

[49] T. Johnson and V. Shkapenyuk. Data stream warehousing in tidalrace. In *Online Proceedings of the Seventh Biennial Conference on Innovative Data Systems Research (CIDR)*, January 9-12 2015.

[50] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 41–52, 2010.

[51] R. Kannan, S. Vempala, and D. P. Woodruff. Principal component analysis and higher correlations for distributed data. In *Proceedings of the 27th Annual Conference on Learning Theory (COLT)*, pages 1040–1057, 2014.

[52] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB)*, 2011.

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

[54] B. Lahiri, J. Chandrashekar, and S. Tirthapura. Space-efficient tracking of persistent items in a massive data stream. In *Proceedings of the 5th ACM international conference on Distributed event-based system (DEBS)*, pages 255–266, 2011.

[55] B. Lahiri and S. Tirthapura. Finding correlated heavy-hitters over data streams. In *Proceedings of the IEEE 28th International Performance Computing and Communications Conference (IPCCC)*, pages 307–314, 2009.

[56] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 767–778, 2005.

www.manaraa.com

[57] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB)*, pages 346–357, 2002.

[58] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 426–435, New York, NY, USA, 1998. ACM.

[59] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 251–262, New York, NY, USA, 1999. ACM.

[60] A. Metwally, D. Agrawal, and A. E. Abbadi. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *11th International Conference on Extending Database Technology (EDBT)*, pages 618–629, 2008.

[61] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th international conference on Database Theory (ICDT)*, pages 398–412, 2005.

[62] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

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

[64] S. Muthukrishnan. Data streams: Algorithms and applications. *Found. Trends Theor. Comput. Sci.*, 1(2):117–236, Aug. 2005.

[65] H. Nasgaard, B. Gedik, M. Komor, and M. Mendell. Ibm infosphere streams: Event processing for a smarter planet. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, pages 311–313, Riverton, NJ, USA, 2009.

www.manaraa.com

[66] O. Papapetrou, M. Garofalakis, and A. Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *Proceedings of the VLDB Endowment*, 5(10):992–1003, June 2012.

[67] K. Patroumpas and T. Sellis. Window specification over data streams. In *Proceedings of the 2006 international conference on Current Trends in Database Technology (EDBT)*, pages 445–464, 2006.

[68] S. Peng, Z. Li, Q. Li, Q. Chen, W. Pan, H. Liu, and Y. Nie. Event detection over live and archived streams. In *Proceedings of the 12th International Conference on Web-age Information Management*, WAIM'11, pages 566–577, Berlin, Heidelberg, 2011. Springer-Verlag.

[69] A. Rabkin and R. Katz. Chukwa: A system for reliable large-scale log collection. In *Proceedings of the 24th International Conference on Large Installation System Administration*, pages 1–15, 2010.

[70] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling real-time querying of live and historical stream data. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, SSDBM '07, pages 28–, Washington, DC, USA, 2007. IEEE Computer Society.

[71] M. Resvanis and I. Chatzigiannakis. Experimental evaluation of duplicate insensitive counting algorithms. In *Proceedings of the 2009 13th Panhellenic Conference on Informatics*, pages 60–64, 2009.

[72] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[73] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, pages 239–249, 2004.

[74] S. A. Singh and S. Tirthapura. Monitoring persistent items in the union of distributed streams. *Journal of Parallel and Distributed Computing*, 74(11):3115 – 3127, 2014.

www.manaraa.com

[75] S. A. Singh and S. Tirthapura. An evaluation of streaming algorithms for distinct counting over a sliding window. *Frontiers in ICT*, 2(23), 2015.

[76] S. Tirthapura and D. P. Woodruff. Optimal random sampling from distributed streams revisited. In *Proceedings of the 25th international conference on Distributed computing*, pages 283–297, 2011.

[77] S. Tirthapura and D. P. Woodruff. A general method for estimating correlated aggregates over a data stream. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE)*, pages 162–173, 2012.

[78] S. Tirthapura, B. Xu, and C. Busch. Sketching asynchronous streams over a sliding window. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 82–91, 2006.

[79] S. Tirthapura, B. Xu, and C. Busch. Sketching asynchronous streams over a sliding window. pages 82–91, 2006.

[80] A. S. Tosun. Space-efficient structures for detecting port scans. In *Proceedings of the 18th international conference on Database and Expert Systems Applications*, pages 120–129, 2007.

[81] K. Tufte, J. Li, D. Maier, V. Papadimos, R. L. Bertini, and J. Rucker. Travel time estimation using niagarast and latte. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1091–1093, New York, NY, USA, 2007. ACM.

[82] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *Proceedings of Network and Distributed System Security Symposium*, NDSS '05', pages 149–166, 2005.

[83] L. Wang, G. Luo, K. Yi, and G. Cormode. Quantiles over data streams: An experimental study. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 737–748, New York, NY, USA, 2013. ACM.

[84] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, jun 1990.

www.manaraa.com

[85] K.-Y. Whang, G. Wiederhold, and D. Sagalowicz. Separability - an approach to physical data base design. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 320–332, 1981.

[86] D. Woodruff. Optimal space lower bounds for all frequency moments. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 167–175, 2004.

[87] D. P. Woodruff and Q. Zhang. Tight bounds for distributed functional monitoring. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 941–960, 2012.

[88] D. P. Woodruff and Q. Zhang. When distributed computation is communication expensive. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, pages 16–30, 2013.

[89] K. Youssefi and E. Wong. Query processing in a relational database management system. In *Proceedings of the fifth international conference on Very Large Data Bases - Volume 5*, VLDB '79, pages 409–417, 1979.

[90] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013.

[91] 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 *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[92] L. Zhang and Y. Guan. Detecting click fraud in pay-per-click streams of online advertising networks. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems (ICDCS)*, pages 77–84, 2008.

[93] W. Zhang, Y. Zhang, M. A. Cheema, and X. Lin. Counting distinct objects over sliding windows. In *Proceedings of the Twenty-First Australasian Conference on Database Technologies - Volume 104*, ADC '10, pages 75–84, 2010.

www.manaraa.com

[94] Q. Zhou, Y. Simmhan, and V. Prasanna. Towards hybrid online on-demand querying of realtime data with stateful complex event processing. In *IEEE International Conference on Big Data (BigData)*, 2013.

www.manaraa.com