FILTERING TECHNIQUES FOR DATA STREAMS

by

IRINA ROZENBAUM

A Dissertation submitted to

the Graduate School-New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

written under the direction of

Shanmugavelayutham Muthukrishnan

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

[October, 2007]

ABSTRACT OF THE DISSERTATION

Filtering Techniques for Data Streams

By IRINA ROZENBAUM

Dissertation Director:

Shanmugavelayutham Muthukrishnan

With the growth in popularity and complexity of streaming applications, there is a rising need for sophisticated analyses of massive high speed data generated by such applications. Such analyses often need to be performed in near real-time, using limited system resources. Under such conditions, it is very important to find an appropriate balance between the efficiency of processing and the accuracy of the produced results. A common technique is to filter the stream with suitable conditions so that the resulting data size is manageable, and the analyses are still accurate.

The work presented by this thesis focuses on a number of complex filtering techniques that are of interest in data steam processing in general and in network traffic monitoring in particular. These techniques allow the analyst to define a filtering condition that is more appropriate for the particular query at hand than the simpler random uniform sampling.

First, we propose a single operator which captures a common thread of evaluation of sampling queries and can be specialized to implement a wide variety of quite sophisticated stream sampling algorithms within an operational data stream management system and scale in performance to line speeds. Additionally, we propose a solution for flow sampling mechanism, which integrates the logic of flow aggregation as well as flow sampling into

one procedure that works directly on IP traffic.

Next, we introduce the notion of the inverse distribution for massive data streams, and present algorithms that draw a uniform sample from the inverse distribution in the presence of inserts and deletes to the stream; such a sample can be used for a variety of summarization and filtering/mining tasks.

Another contribution of this thesis is the development of a filter join operator, which makes it feasible to evaluate a common type of join query that searches for records matching dynamic criteria on high speed data streams, in an efficient, stable and accurate manner. We also present analyses of query transformations which expose the filter join operator in conventional query join.

Finally, we study the problem of matching regular expression that can span multiple data records in a data stream in the presence of stream quality problems, such as duplicates and out-of-order records; we present a number of algorithms that can match regular expressions over multiple data stream records without stream reassembly, by maintaining partial state of the data in the stream.

The ideas presented in this thesis are motivated by actual practical problems that arise in data stream processing, and are further validated by the presented experimental studies.

To my dear Daniel

Acknowledgements

This dissertation is a result of help, encouragement and support that was given to me by a number of people I have been privileged to have come to know.

Above all, I would like to thank my advisor, S. (Muthu) Muthukrishnan, whose sparkling personality and brilliant mind led me to graduate school in the first place and whose inspiration, support and skillful guidance made me want to stay there during all those years. I am forever in your debt.

My deepest thanks to Theodore Johnson, my supervisor and mentor at AT&T Shannon Labs. His professional advice, continuous patience and encouragement were instrumental in my career development.

I would also like to express my gratitude to Graham Cormode for safety net during difficult times.

Additionally, I would like to thank Oliver Spatscheck for sharing his ideas with me and providing technical support in projects related to Gigascope Data Stream Management System.

Finally, I would like to thank my thesis committee, which, in addition to Muthu, includes Divesh Srivastava, Amelie Marian and Richard Martin, for their time and insightful comments.

Contents

List of Tables

List of Figures

Chapter 1

1 Introduction

1.1 Data Streams and Data Stream Management Systems

Traditional data base management systems (DBMSs) are widely used in applications that require persistent storage for large volumes of data. The data is viewed and processed as an unordered set of records[1] which remain valid until explicitly modified or deleted. Queries posed on data are executed in a timely manner and reflect the state of the database at the time of execution. DBMSs offer their users consistent, persistent and recoverable data storage, a set of well defined operators and a highly optimized query processing engine for efficient management of transactions over the maintained data set.

With the general increase of data rates and volumes in different areas of information technology a new type of applications had emerged in which large volumes of data are being continuously generated in real time, taking the form of an ordered, unbounded sequence of items, or a *data stream*. Such applications include financial data analyses [98, 41, 6], sensor networks [82, 30, 14, 100], IP network monitoring [50, 51, 52, 120], phone records log processing [49] and others. At the same time these applications often require sophisticated processing capabilities for continuous monitoring of the input stream in order to detect changes or find interesting patterns over the data in a timely manner.

[1] We use terms record, item or tuple interchangeably throughout the thesis.

In more general terms, data streams can be characterized by the following:

- A data stream is potentially unbounded in size.
- The data is being generated continuously in real time.
- The data is generated sequentially as a stream. It is most often ordered by the timestamp assigned to each of the arriving records implicitly (by arrival time) or explicitly (by generation time).
- Typically, the volume of the data is very large, and the rates are high and irregular. For example, storing one day's worth of IP network traffic flows in the AT&T IP backbone alone may require as much as 2.5TB [7].

In addition to the above characteristics of data streams, a large number of streaming applications require near real-time sophisticated analyses, as discussed in detail in section 1.2.

The data stream characteristics are clearly very different from those assumed for data stored in traditional DBMSs and thus make traditional DBMSs ill-suited for efficient implementation of many data stream processing applications. New data processing techniques are required to monitor and analyze massive volumes of data streams in real-time.

The emergence of stream-based applications has gaven rise to Data Stream Management Systems (DSMSs), which aim to provide data stream management and processing capabilities similar to traditional data processing, while dealing with the novel requirements posed by high-speed data streams:

- DSMSs are append-only data-systems, which means that the newly arriving data stream records are continuously pushed into the system. Traditional DBMSs, on the other hand, deal with unordered data sets, and rely on random or repeated access to individual data records.
- DSMS queries are *persistent (continuous) queries* [123] (discussed in detail in section 2.1.3) that produce query results continuously over the lifetime of a stream. They are issued once and can remain active over a long period of time. In contrast, a DBMS deals with queries that are computed once, with query results reflecting the current state of the database.

- The volume, rate and high dimensionality [107] of the data arriving in a stream makes the data challenging to store. For instance, the potential number of combinations of values of the source and destionation IP addresses alone of IP network packets is 2^{64}. Consequently, a DSMS strives to compute the results by storing only a small portion of the original stream at any point of the computational process, or its approximate data summary structures (*synopses*) [21], such as samples [127, 15, 61, 102, 70], sketches [17, 63, 46], histograms [86, 112] and wavelets [35, 73], which often allow a good approximation of a number of stream characteristics.

- Due to performance and storage constraints, the input data being processed is not available for random access from disk or memory, but rather needs to be processed and analyzed as it arrives. This makes it infeasible to reevaluate a streaming query at a later time. In contrast, relational query results can be recomputed at any time so long as the data is present in DBMS.

- Timely processing of data is another critical requirement of many data stream applications, e.g. fraud detection in financial industry, anomalous behavior and intrusion detection in IP network monitoring, etc. The results of this type of analyses often need to be reported instantaneously, without delays introduced by the offline processing.

- Stream arrival rates can fluctuate drastically thus making it irregular and bursty in nature, and the query workload may change accordingly. For example, during a distributed denial of service (DDoS) attack on an IP network, the rate of the incoming stream may reach over $500,000$ packets per second [104]. It is critical for DSMS to be stable under those adversarial conditions and to provide the user with the meaningful feedback.

In recent years both industry and academia have developed a number of DSMSs to process and analyze data streams. These include Aurora/Borealis [12, 13, 26], STREAM [19, 18, 106], Nile [16, 81], NiagaraCQ [41], TelegraphCQ [36], Tribeca [120] and others. Commercial systems, such as IPMON [2] and CMON, Streambase [5], Gemfire Real-time Events [4], are being used for monitoring of financial trading applications, telecommunication applications. Gigascope [50, 51, 52] is another commercial DSMS developed and

used at AT&T for high-speed IP network monitoring. Some of the contributions of this work were implemented and evaluated in the setting of the Gigascope system.

The balance between efficiency of processing and accuracy of results for the tasks being evaluated is one of the biggest challenges of data stream processing. The question then becomes: how can we reduce the volume of the stream to a manageable size without compromising the accuracy?

1.2 Problem Statement

As the complexity of applications dealing with data streams grows, so does the complexity of the queries DSMSs should be able to deal with. Such queries might include detection and analyses of extreme events in the traffic and fraud detection (e.g. sudden increase in trading volume in a stock market application, unusual credit card transactions), intrusion detection [10, 59] (e.g. DDoS attacks [91, 103, 104], worms and viruses in IP network monitoring [93, 119, 109]), traffic outliers [132] (e.g. generation of statistics for network provisioning and service level agreements), network application identification using application specific signatures [116, 131, 55] (e.g. P2P file sharing applications), etc. Processing of such tasks often involves evaluation of a number of complex aggregations, joins, matching of various regular expressions, generation of data synopses. Even formulating these tasks in terms of DSMS queries is sometimes not trivial, which in turn may make it difficult to generate an efficient query evaluation plan, and thus hinder performance.

With all these complexities, these tasks still require real-time responses, which puts considerable constraints on the per-item processing time. They also demand accuracy of results, which by itself is a significant challenge when dealing with high-speed, large volume infinite streams with potentially unpredictable behavior patterns. The challenge becomes even more formidable due to the fact that query evaluation needs to take place in bounded space while dealing with unbounded streams over potentially very large domains of multiple attributes [107].

It is therefore natural that many queries on data streams translate to some form of the reduction of the initial amount of data in the stream. Conceptually, while observing an

infinite stream of data, we would like to be able to look at each of its items and quantify whether the item is of interest and should be stored for further evaluation. To this end, a number of data stream filtering techniques were developed in order to reduce the volume of the data being analyzed:

One type of filtering can be referred to as ***implicit*** (described in detail in section 2.3.1); it is done by the system in order to keep it stable under adversary conditions. In the literature this technique is widely referred to as random or semantic *load shedding*. It is usually done by performing uniform random sampling of the stream at different levels of query hierarchy [121, 23, 113]. This type of filtering doesn't take into consideration the query semantic and thus might have implications of the correctness of the query results. To improve this situation, window-aware load shedding [122] and per-group semantic sampling [90] were recently proposed.

In many cases the complexity of the query being issued, or the nature of the query itself, call for use of customized filtering conditions. To distinguish these from the automatic, implicit filtering described above, we will refer to this type of filtering, when the filtering condition is specified by the procedure itself, as ***explicit*** (described in detail in section 2.3.2). We include in this category stream sampling algorithms [61, 102, 70], techniques for fast regular expression matching [94, 131], sketch-based filtering [28] and others. The main goal of explicit filtering is to provide the flexibility of being able to specify the filtering conditions that best fit the problem being analyzed.

1.3 Summary of Contributions

This work focuses on development of a number of explicit filtering techniques that reduce the initial load of the incoming data to a manageable size in a controlled manner according to a procedure specific filtering condition, and provide meaningful and accurate (in some cases within desired error bounds) results to a number of important queries of interest in data stream processing. More precisely, we can categorize the contributions of this thesis according to different types of filtering conditions:

1. ***Filtering condition of a stream item is independent of other items of the same stream or any other data stream.*** The most common example of such filtering is stream sampling, when each item is filtered out with a certain probability and the remaining items form the desired sample. The contributions of this thesis that fall under this category are as follows:

- **A single data stream sampling operator:** Generation of a stream sample is one of the widely used filtering techniques in data stream analyses, when only a small portion of the stream is being saved for further analyses. The past few years have seen the design of many effective stream sampling methods for estimating specific aggregates such as quantiles [76], heavy hitters [102], distinct counts [70], subset-sums [61], set resemblance and rarity [54], as well as generic sampling such as fixed-size reservoir sampling [127], adaptive geometric sampling [25, 83], etc. This dissertation develops a single operator which can be specialized to implement a wide variety of quite sophisticated stream sampling algorithms within an operational data stream management system and scale in performance to line speeds. We perform an experimental study by implementing the sampling operator in the Gigascope DSMS. We use this implementation to present a detailed study of one of the stream sampling algorithms of great interest to IP network management, namely subset-sum sampling [61] that is operationally used for performance monitoring in AT&T's IP backbone and for customer reports. We show that the new operator is a simple and flexible tool for early data reduction of the stream that imposes only a small CPU overhead and scales to line speeds. This work was published in the *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* [88].

- **Summarizing and mining inverse distributions on data streams via dynamic inverse sampling:** Many of the existing methods for summarizing and mining data streams, including those mentioned in the previous paragraph, focus on the forward distribution $f(x)$, for example the number of occurrences of x in the data stream of integer values. In contrast, this work formulates summarization and mining problems on the inverse distribution $f^{-1}(i)$, which is the number of items that appear i times in the stream. We introduce the notion of the inverse distribution for massive data

streams, and presents algorithms that draw a uniform sample from the inverse distribution over the stream which can be used for a variety of summarization tasks (such as building quantiles or equidepth histograms) and filtering/mining (anomaly detection such as finding heavy hitters, measuring the number of rare items), all with provable guarantees on quality of approximations and time/space used by our streaming methods. We also complement our analytical and algorithmic results by presenting an experimental study of the methods over network data streams. This work was published in *Proceedings of the 2005 International Conference on Very Large Data Bases (VLDB)* [48].

2. ***Filtering condition of a stream item is dependent on items from another stream of data.*** A characteristic example of this analysis is evaluation of various types of joins of a number of data streams. In this class of problems this thesis presents the following contribution:

- **Filter join operator:** A large class of queries on data streams search for records matching dynamic criteria, filtering out the rest of the records. For example, a network analyst might want to collect all records in a network flow that start with a suspicious signature; or a financial analyst might want to track trading records of a financial instrument following a suspicious trade. Evaluating these queries requires a join, which might be too expensive to implement on a very high speed stream. In this work, we propose the filter join operator, which makes it feasible to evaluate a common type of join query on high speed data streams in an efficient, stable and accurate manner. The filter join has an inexpensive evaluation algorithm and can be pushed to the data sources in the case of self-join. We provide a relational characterization of the filter join, and a collection of query transformations which can expose the filter join component(s) of a conventional join. We implement approximate filter join algorithms in the Gigascope DSMS and find order-of-magnitude performance improvements when compared to equivalent queries implemented using a conventional join. This work is submitted to the *The International Conference on Data Engineering (ICDE) 2008* [114].

3. ***Filtering condition of a stream item is dependent on other items of the same stream***:

- **Regular expression matching on out-of-order streams:** This work studies the problem of filtering data stream in the presence of data-quality problems, such as duplicates and out-of-order records [99, 56], when the filtering condition is specified in the form of a regular expression. This is a well motivated problem in managing IP networks where regular expressions are signatures that have to be matched against the contents of flows (a set of records sharing identical values on a specified number of record attributes) to detect intrusions, worms or viruses, applications and protocols. Prior work either matched regular expressions against the data segments on individual packets or reassembled the entire flow to match the regular expression using standard methods. Instead, we have proposed streaming algorithms that can be run in software that match regular expressions across segments even in the presence of out-of-order packets and duplicates by carefully optimizing the state maintained on partial flows. Our experimental study with real data shows that the algorithms are successful in limiting the memory used and are efficient. This work was published in *Proceedings of the International Conference on Data Engineering (ICDE) 2007* [115].

1.4 Thesis Organization

The reminder of this thesis is organized as follows: Chapter 2 presents a general overview of the aspects of data streams relevant to this thesis.

Chapter 3 introduces a single data stream sampling operator developed to incorporate a wide variety of sophisticated stream sampling algorithms into an data stream management system.

Chapter 4 introduces the notion of inverse distribution for massive data streams, and present algorithms that draw a uniform sample from the inverse distribution over the stream, which can be used for a variety of summarization and filtering problems on data streams.

Chapter 5 presents a filter join operator, which makes it feasible to evaluate a common type of join query on high speed data streams in an efficient, stable and accurate manner.

Chapter 6 introduces a number of algorithms that filter the stream by matching the specified regular expressions across multiple data segments in presence of out-of-order packets

and duplicates.

Chapter 7 offers concluding remarks and discussion of future work.

Chapter 2

2 Background

This chapter reviews some concepts and issues of data stream processing relevant to this thesis. Section 2.1 provides an overview of streaming models, window specification and persistent queries processing. Section 2.2 focuses on IP network data stream processing, since the work presented by this thesis, although relevant for various types of data streams, was initially developed in the context of network monitoring application. Section 2.3 reviews a number of filtering techniques currently used in data stream processing, including load-shedding techniques, summary data structures and others.

2.1 Overview of Data Streams

2.1.1 Data Stream Models

As it was discussed in section 1.1, data streams are potentially infinite, continuously generated sequences of items that typically arrive at DSMS with high irregular speeds, and comprise very large volumes of data. More formally, a data stream can be defined as follows [73, 107]:

Definition 1. *Input stream* $a_1, a_2, a_3, \ldots$ *arrives sequentially, item by item, and describes an underlying* signal ***A****, when in the simplest case* ***A*** *is a one-dimensional function* $\boldsymbol{A} : [0 \ldots (N-1)] \rightarrow Z^+$.

The stream can describe the underlying signal in various ways, resulting in a number of

different data stream models [73, 107]:

- *Unordered cash register* model - individual items of the stream are domain values that arrive in no particular order and without any pre-processing.
- *Ordered cash register* model - individual items of the stream are not pre-processed and arrive in the increasing (or decreasing) order of the domain values (e.g. in the order of the timestamp attribute values).
- *Unordered aggregate* model - individual items of the stream that belong to the same domain arrive in a pre-processed (range values) format in no particular order.
- *Ordered aggregate* model - individual items of the stream arrive in a pre-processed form and in the order of the domain values.

In addition to this classification, the cash register model distinguishes between *contiguous* and *non-contiguous* domain values of the arriving stream items.

2.1.2 Window Specification

Compared to one-time queries in convectional DBMS's, persistent queries differ significantly in their semantics. Since persistent queries are evaluated on potentially unbounded streams of data and the state of the data is not known in advance, query results depend on a set of records available during query evaluation process. The most naïve implementation of many useful operators, such as aggregation or join, would require maintaining the entire stream history in order to produce the exact result. This, however, is obviously impractical. Moreover, for efficiency it is necessary to retain the streaming data being processed in memory rather than on slow, larger storage, which limits the amount of streaming data available during evaluation. Additionally, the majority of real world streaming applications regard as relevant only the most recent data, rather than the entire past history of the stream.

All these considerations gave rise to various *window based models*, when at any instant a window specifies a finite set of the most recent records from the infinite data stream which is used to evaluate a steaming query and produce results that correspond to the time period spanned by the window.

The structure of the window has the following distinctive properties [111]:

- *upper bound* - the timestamp of the most recent item of the window.
- *lower bound* - the timestamp of the oldest item in the window.
- *size* - can be expressed either in a number of records or as a temporal interval spanning window contents.
- *mode of adjustment* - determines how a window changes its state over time with the arrival of new items.

Further, a window can be categorized depending on definition of its scope as follows:

- *Physical window*: defined in terms of the number of records within the window bounds.
  - **count-based sliding**: this type of window includes the most recent N items of the stream, while continuously expiring the old items as the new arrive.
  - **count-based tumbling**: in this case the stream is divided into non-overlapping partitions of N records each, and only the data that is contained within the current partition is kept for processing.
  - **partitioned**: to apply this type of window, the stream records are first partitioned into a number of substreams according to values of grouping attributes. The union of the most recent N values from each partition defines a window.
- *Logical window*: defined in terms of monotonically increasing (or decreasing) attribute(s) of the data stream records (usually values of the timestamp attribute).
  - **landmark (agglomerative)**: one variation of this type is when the lower bound of the window is fixed at a specific time instant, i.e. the window identifies a certain starting landmark in the stream and processes records from that point with evolution of time. In another variation only the upper bound of the window can be fixed to a future time instant.

  - **fixed-band**: the window is defined by combining both of the landmark window variants, fixing the upper and lower bounds according to a band window function, which allows to express arbitrary time intervals ("bands").
  - **time-based sliding**: this is the most commonly used type of windows over data streams, defined in terms of time units. The window stores only those items of the stream that have arrived in the last T time units and continuously expires the old items as the new arrive (T here is the size of the window).
  - **time-based tumbling**: similarly to the count-based tumbling window definition, the stream is now divided into non-overlapping partitions of fixed time intervals. A new window is created as soon as the old one ceases to exist.

- *Punctuation-based*: this model uses special annotations embedded in data stream, sometimes referred to as *punctuations*, to signify the end of the currently processed window [124].

Window based models are the most commonly used approach to limiting the scope of the input data over which the query is being evaluated. An alternative model is the *time-decay model* [42, 43, 57], also referred to as *fading*, where each of the items is discounted by a scaling, non-decreasing with time factor, such as exponential and polynomial decays.

2.1.3 Persistent Queries

Queries evaluated against unbounded stream of data are called *continuous* or *persistent*: they are issued once and run continuously as the items of the stream continue to arrive [24, 123].

A streaming query is called *monotonic* if its results do not shrink over the time of execution. More formally, if Q is a persistent query at time t_1, then Q is monotonic if $Q(t_1) \subseteq Q(t_2)$ for all $t_1 \leq t_2$. For example, queries that contain a simple selection predicate are monotonic, since the newly arrived tuple either satisfies the predicate and is thus added to the set of the output tuples, or doesn't satisfy the predicate and thus has no effect on the output.

A streaming query is called *blocking* if it is unable to produce the first tuple of its output until it has seen its entire input [21]. Thus aggregation (SUM, COUNT, AVG, MAX, etc.) sorting and joins operators are blocking operators, whereas selections and projections are *non-blocking*. Consider, for instance, calculating MAX of the values of a data stream. Since data stream is unbounded in size, the MAX aggregate will never see its entire input and therefore will never be able to produce any output. The following general techniques are commonly used to make blocking operators suitable to the data stream computation model [21, 74]:

- **limiting the scope**: the scope of the operator is restricted to a finite set of data from the stream by imposing windows [111] or augmenting data stream with punctuations [124], where the finite data set size is small enough to fit into memory. Thus, for example, Symmetric Nested Loop Join(SNLJ) [79] maintains state of the tuples from the most current window, rather than the state of the entire data stream.

- **incremental evaluation**: streaming operators can incrementally update the results of the query with every newly arriving tuple. For example, AVG (average) aggregate can be computed by incrementally maintaining the cumulative sum and item count [97]. Similarly, when computing MAX (maximum) of values of the stream, the maximal value seen so far can be maintained at each step of the query processing.

2.2 IP Network Data Stream Monitoring

In streaming applications large volumes of data are being continuously generated in real time, taking the form of an ordered, unbounded sequence of items. In this section, we look in greater detail at IP network monitoring applications, since the work presented in this thesis was initially developed for this type of analyses.

On the Internet, data is being transmitted by packet switching using the Internet Protocol (IP). It is of crucial importance to such organizations as Internet service providers (ISPs) to be able to analyze traffic patterns, relationships between networks, network problems etc. Tasks in this area include tracking bandwidth usage statistics for traffic engineering and network provisioning, routing system analyses, billing, detection of suspicious activities,

equipment failures, denial-of-service attacks, fraudulent behaviors, and others. This is typically accomplished by monitoring and analyzing the traffic generated by the network.

IP network traffic is basically a data stream that consists of a IP network packets, each containing a number of components [3]:

- **header** - contains instructions about the data carried by the packet.
- **payload** - also called the *body* or *data* of a packet. This is the actual data that the packet is delivering to the destination.
- **trailer** - sometimes called the *footer*, typically contains a couple of bits that signify the end of the packet. It may also have some type of error checking.

A stream of such network packets can be monitored and analyzed at a number of levels [72]. At the *packet level*, certain fields of the protocol headers are particularly useful for data analyses purposes. In the IP, these are the length of a packet in bytes, transport protocol, source and destination IP addresses. In the TCP header, such useful fields include source and destination port numbers, packet sequence number, acknowledgment number, ACK/SYN/FIN/RST flags and others.

At the *flow level* analyses, we look at *flows* - groups of packets with the same source and destination IP addresses and port numbers, where each consecutive packet is not separated by more than a certain duration. Analyses at this level is widely used by network operators, and has been shown to be effective both as a useful research tool and as a practical approach to network usage measurements, detection of suspicious traffic patterns etc.

Some interesting queries on IP traffic logs may include:

- Which TCP connections transmitted more than N bytes of data?
- List the number of bytes and the number of packets for K most frequent destination IP addresses per source IP address.
- How many duplicate sequence numbers occurred per TCP connection over the last minute?
- What are the source IP addresses involved in more than K TCP connections?

- How many flows consist of a single IP packet?
- What is the median, 95th percentile, 98th percentile, and 99th percentile of TCP round trip times?
- Collect detailed flow statistics and every minute, return M samples in such a way as to preserve subset sums on flow size [61].

2.3 Data Stream Filtering

Due to the nature of data streams, stream ***filtering*** is one of the most useful and practical approaches to efficient stream evaluation, whether it is done implicitly by the system to guarantee the stability of the stream processing under overload conditions, or explicitly by the evaluating procedure. In this section we will review some of the filtering techniques commonly used in data stream processing.

2.3.1 Implicit Filtering in Data Stream Management Systems

Data Stream Management Systems cope with the high rates and the bursty nature of streams in a number ways in order to guarantee stability under heavy loads. Some of them employ various *load-shedding* techniques which reduce the load by processing only a fraction of the items from the stream and discarding others without any processing.

The Aurora DSMS employs *random* and *semantic* load shedding techniques [121] to deal with the unpredictable nature of data streams, where semantic load shedding makes use of tuple utility computed based on quality-of-service (QoS) parameters. Intuitively, the system drops tuples that are believed to be less important for stream evaluation that others. QoS of the system is captured by a number functions: latency graph, which specifies the utility of a tuple as a function of tuple propagation through the query plan; value-based graph, which specifies which values of the output are more important than others; and loss-tolerance graph, which describes how sensitive the application is to approximate answers. The work proposes a number of heuristics for determining when, where and how much load to shed with the minimal loss of utility.

The load-shedding mechanism in STREAM [23] DSMS places random sampling operators at various points of the query plan, which uniformly sample the stream, while the sampling rate is dynamically adjusted with respect to the operator selectivity and arriving rate of the data. The objective of this approach is to shed the load while minimizing negative impact on accuracy of query results. It is clear that the strategy of dropping tuples at the early stages of the query plan makes the process of query evaluation more efficient for subsequent operators in the plan. However in case when multiple queries share parts of their plans, the question of where to shed load becomes more complex, since the above strategy might have different effect on results of different queries. This work proposes an algorithm for optimal placement of sampling operators in multi-query plans involving windowed aggregates.

Tuples dropped by the load shedding mechanism in the TelegraphCQ DSMS [113] are collected by the Data Triage components into data synopses and are later combined with the standard query results in order to better capture the properties of the entire input. This mechanism attempts to increase the accuracy of the query results in cases when the load-shedding is necessary for the system to maintain stable operational state. However, data synopses are usually very specialized to the evaluation of a particular stream characteristic and cannot provide valid information to all possible query operators. This makes the proposed load-shedding mechanism effective only for certain type of queries.

All the stream filtering techniques mentioned above reduce load by performing random uniform sampling of items of the stream. Although this type of filtering effectively reduces the amount of data to be processed and hence stabilizes the stream processing system, it might have negative implications on the accuracy of query results. Recent work on *window-aware load shedding* [122] addresses this problem in the context of sliding window aggregation queries. To guarantee that the result of the query is a subset of the correct output, the work proposes a “window drop” operator that can drop the entire window of tuples, rather than performing a per-tuple sampling. The work is limited to aggregate queries on sliding windows and is not easily applicable to arbitrary streaming queries.

Another load shedding technique, *query-aware semantic sampling* [90], was recently

developed in Gigascope DSMS [50, 51, 52]. For every query in a given set of queries this technique automatically infers whether the query should use uniform per-tuple sampling or per-group sampling, depending on the type of aggregate used by the query. Thus queries containing loss-sensitive aggregates, such as OR, Min, Max, Count of duplicates are subjected to per-group sampling to guarantee semantically correct output.

2.3.2 Explicit Filtering of Data Streams

As it was mentioned previously, the implicit filtering techniques may often have a negative impact on a variety of data streams analyses problems, such as, for example, computation of sketches and samples of distinct items for estimation of quantiles, heavy hitters and other properties of a stream. Other problems in this category include estimation of IP network flow sizes, detection of most prevalent substrings in the network for worm signature generation and others. In such cases it is therefore more appropriate to include the filtering conditions that best fit the query being issued as part of the query procedure itself.

Below we give some more detailed examples of such explicit filtering procedures on data streams.

Fine-grained estimation of network traffic (flows) volume is very important in various network analysis tasks. The work presented in [61] offers a solution by proposing the *threshold sampling* algorithm that generates a sample of stream items with guarantees on estimated flow sizes. At each step of sample generation, the procedure maintains a value of a threshold, which is compared to the size of the item, and the decision is made on whether the item of the stream should be filtered out or retained in the sample. Depending on the version of the algorithm, the threshold value can remain constant throughout the process or be dynamically adjusted to ensure the size of produced sample.

Due to the nature of this sampling algorithm, with a number of various parameters, such as a number of items in the final sample, item size, threshold value, count of items larger than the threshold value, etc., playing a role in how the procedure is executed and what results it produces, its results would be seriously compromised if it was to be evaluated on a stream which was a product of random load shedding. On the other hand, because of the specificity of this sampling algorithm it would be infeasible to attempt to incorporate it

implicitly in a DSMS.

In [28] the authors propose a technique of filtering out a fraction of stream items based on “norm” of the stream seen in the process of generation of CM sketch [46]. This technique aims at improving the update time of stream items with the norm-aware procedure, more sophisticated than uniform random sampling. The technique guarantees accuracy of the results and can be used for a number of stream processing problems, such as summarization, heavy hitters and self-join size estimate. The type of filtering described by this work can be applied to IP packet headers as well as to packet payloads, which are typically several tens of factors larger in size than the headers and hence require longer computing time.

Another interesting problem where filtering is performed at the level of packet content is identification of network traffic associated with different network applications using application signatures, where signature is typically described in terms of regular expression. In [116] authors identify applications signatures for a number of P2P file sharing applications, such as Gnutella, eDonkey, BitTorrent, DirectConnect and Kazaa, by examining available documentation and packet-level traces. Then the signatures are used as filters and compared against packet payloads in order to efficiently and accurately identify traffic of desired applications.

The above examples demostrate that the filtering condition can take different forms when analyzing various stream characteristics. Making such filtering procedures efficient while producing accurate results on high speed data streams is an important objective. The work presented by this thesis adresses a number of such problems.

Chapter 3

3 Sampling Algorithms in a Stream Operator

Stream sampling is one of the most common filtering techniques on data streams, with the result of the procedure execution being a *sample* - a small-sized subset of the items from the stream representative of a certain characteristic (or a number of characteristics) of the input. The research community has developed a rich literature on stream sampling algorithms with many of these algorithms providing better properties than conventional random sampling. In this chapter, we abstract the stream sampling process and design a new stream sample operator. We show how it can be used to implement a wide variety of algorithms that perform sampling and sampling-based aggregations. Also, we show how to implement the operator in Gigascope [50, 51, 52] - a high-speed stream database specialized for IP network monitoring applications. As an example study, we apply the operator within such an enhanced Gigascope to perform threshold sampling (also referred to as subset-sum sampling), which is of great interest for IP network management. We evaluate this implementation on a live, high speed internet traffic data stream and find that (a) the operator is a flexible, versatile addition to Gigascope suitable for algorithm tuning and engineering, and (b) the operator imposes only a small evaluation overhead. This is the first operational implementation we know of, for a wide variety of stream sampling algorithms at line speed within a data stream management system.

3.1 Introduction

The body of work that focuses on sampling methods for data streams includes algorithms for approximation of quantiles [76], heavy hitters [102], set resemblance [54], count distinct [70], and so on. Sampling has a rich history in statistics, with several variants: sampling with/without replacement, biased sampling, fixed or variable size sampling etc. There is also extensive use of sampling in databases with many modified methods such as stratified, congressional [15], outliner or distance-based sampling etc. [118]. Sampling in the context of data streams shares some common aspects with sampling in statistics and databases, but has additional constraints. In stream sampling, typically one is interested in sampling in one pass over a high speed data that cannot be stored at its matching rate. As a result, when an item repeats on the stream, it is difficult to sample based on whether or not it has been seen before. So, even uniform sampling of the *distinct* items in the data stream is tricky. Further, one may need to obtain fixed-sized sample when the size of the stream is unknown. Finally, stream input has many attributes and items are often "weighted" and it is difficult to ensure that the sample has desirable properties - such as it captures the heavy hitters or sub-range aggregates - accurately for various subset combinations of attributes and cumulative weights on these combinations. The past few years have seen the design of many effective stream sampling methods for estimating specific aggregates such as quantiles [76], heavy hitters [102], distinct counts [70], subset-sums [61], set resemblance and rarity [54] etc. as well as generic sampling such as fixed-size reservoir sampling [127], adaptive geometric sampling [25, 83], etc.

The focus of this work is not to design new stream sampling methods. Instead, we address the problem of how these widely varied and quite sophisticated filtering methods can be implemented within an operational data stream management system and scale in performance to line speeds in IP network monitoring applications. The problem we address in this work is to incorporate approximate streaming algorithms into a DSMS, specifically sampling-based algorithms.

Possible Approaches: There are several approaches to doing this integration, which we

discuss here:

The first approach is to incorporate the different sampling algorithms directly into the DSMS kernel, and make the option of using them available to the user through several keywords. This approach is attractive when the special techniques being incorporated into the database engine are mature, for example data mining keywords in SQL Server 2005 [11], windowing keywords in SQL 99 [77], and so on. However, stream sampling algorithms is an active research area with new techniques being continually developed. Incorporating new techniques into the kernel is cumbersome and does not promote experimentation. In addition, the query language is burdened with a keyword explosion. Aurora incorporates a DROP operator, which performs random sampling to shed load [33]; STREAM also provides operator-level sampling via a SAMPLE keyword [106].

The second approach is to implement individual stream sampling algorithms with User Defined Aggregate (UDAF) Functions. This approach was explored in [44] for one of the methods, namely, approximating heavy hitter frequency counts by sampling [102]. While the UDAF approach is useful for obtaining point values (e.g., the median packet length), it is cumbersome at best for obtaining set values. For example, to obtain set of destination IP addresses responsible for at least 1% of traffic using the UDAF approach, we could write a query with 100 references to the heavy hitters UDAF (one for each of the possible 100 heavy hitters) in the SELECT clause, *pivot* [78] the result to get the set value, and filter out invalid values. While set results are not inherently better than point results, many applications require set results as their input. In addition, some algorithms, such as subset-sum sampling [61] are better expressed as a sampling query. ATLaS [128] is a system in which a UDAF is specified in SQL. Its set-oriented nature makes set-valued return results possible. As will be evident later, our operator is in some ways a highly structured version of an ATLaS UDAF. The structure we impose enables the simple expression of many algorithms, and a highly efficient evaluation process.

The third and related approach is to provide for User Defined Operators (UDOs), which consume input streams and produce output streams, one for each of the stream sampling methods. Some DSMSs provide a mechanism to incorporate UDOs, including Gigas-

cope [50, 51, 52] and Aurora/Borealis [12, 13, 26]. Aurora is built as a system of interconnecting operators, and by nature supports UDOs. Gigascope has special facilities for incorporating UDOs into a query set. However, writing and supporting a DSMS operator is a difficult and error-prone task and does not scale with the number of different stream sampling methods of interest. Our discussions with the Gigascope users indicate that few ODOs had been written, and only as a last resort.

Our Approach: The approach we take in this work is to develop a *single* operator, which can be specialized to implement a wide variety of stream sampling algorithms. The advantage of this approach is that it encourages experimentation and development of new streaming algorithms and their rapid deployment for practical applications. The functions, which support the streaming algorithm using the operator for different problems, can be written by the algorithmic expert, following a simple API. The developer is not burdened with the details of kernel integration or stream operator development. Our contributions are as follows:

- *We abstract an operator construct and define its semantics.* We show that this generic operator can be used to implement wide variety of stream sampling algorithms including the reservoir sampling [127], subset-sum sampling [61], min-wise hash sampling [54], heavy hitter algorithm [102], and many others.
- *We show how to implement the generic operator in a data stream management system (DSMS).* The sample operator is invoked using special keywords in a grouping and aggregation query. We detail an efficient templatized implementation of the sample operator. These constructs, as well as STATEFUL functions we introduce, may be of independent interest in conventional data warehouse DBMSs because of their ability to support approximation queries.
- *We perform an experimental study by implementing our sampling operator in the Gigascope DSMS.* We use this implementation to present a detailed study of one of the stream sampling algorithms of great interest to IP network management, namely, subset-sum sampling [61] that is operationally used for performance monitoring in

AT&T's IP backbone and for customer reports. To demonstrate the flexibility of the proposed sampling operator we implement two variations of the subset-sum sampling - packet based and flow based. The latter implementation is more stable and thus more resistant to rapid network changes. It allows us to create very informative flow samples on streams of network data. Our implementation works at line speed and is now part of the Gigascope release; it shows that the computational and memory overhead is very small. In addition, our experience with real data revealed its burstyness and led to a small fix in the subset-sum stream sampling algorithm that substantially improved its performance. The ability to do such easy tuning and engineering is one of the attractions of our approach.

Our operator is specifically targeted at stream sampling algorithms and can be used to implement scores of them. In this work, we have chosen to focus on four representatives: reservoir sampling for standard fixed-size sampling on streams, heavy hitter algorithm from the database community, min-wise hash sampling from the algorithms community and subset-sum sampling from the networking community. We believe our work is the first to operational implementation we know of, for a widely variety of stream sampling algorithms at line speed within a DSMS.

Map: In Section 3.2, we discuss related work. In Section 3.3, we provide an overview of the Gigascope DSMS. In Section 3.4, we present an overview of the four stream sampling methods above and describe their common framework. In Section 3.5, we present our operator and show how it can be used generically to implement different stream sampling algorithms. In Section 3.6 we discuss STATEFUL functions, SUPERGROUPs and show how to implement our operator in Gigascope. In Section 3.7, we present our experimental study. Conclusions are in Section 3.8.

3.2 Related Work

Sampling has an extensive history in statistics and relational databases. We focus on stream sampling. As mentioned earlier, a number of specific sampling algorithms have been de-

signed for quantiles [76], heavy hitters [102], distinct counts [70], subset-sums [61], set resemblance and rarity [54], geometric sampling for range counting [25] and adaptive sampling for convex hulls [83], etc. Many of these have been implemented and tested on reasonable streams, but few, to the best of our knowledge, on IP network line speeds at which packets are forwarded. In [44], the authors implemented the heavy hitters' algorithm [102] as a UDAF in line speed. In [61] the threshhold sampling method is implemented at IP flow speeds and not at packet speeds; flows are several orders of magnitude aggregated from packet streams.

There are a number of DSMSs being developed: Aurora [12, 13, 26], STREAM [19, 18, 106], Gigascope [50, 51, 52], TelegraphCQ [36], NiagaraCQ [41], etc. Many of them support random sampling, including the DROP operator of Aurora, the SAMPLE keyword in STREAM, and sampling functions in Gigascope. Still, these are uniform sampling operators. We do not know of prior work on these systems that systematically implemented a variety of sophisticated stream sampling methods.

3.3 Gigascope

Gigascope is a DSMS that has special architecture for performing analysis on high speed data streams [50, 51, 52]. Since our experimental results use Gigascope, we discuss some relevant aspects of its architecture in this section.

Gigascope is a stream-only database which does not support stored relations or continuous queries. This restriction implies that there are no explicit time windows which are critical for evaluation of blocking operators, like aggregation and join. Instead, each tuple of the data stream is labelled with monotonically increasing timestamp. Gigascope uses the timestamp and the results of query analysis to determine when the blocking operator needs to be unblocked. Thus for instance, Gigascope requires in an aggregation query that at least one of the group-by attributes to have monotonically increasing values. When the value of this attribute changes, all existing groups are sent flushed. This process effectively defines *epochs* for data aggregation. Similarly, when performing a symmetric hash join on data streams, the join window is changed when the timestamp label of a tuple changes. In cases

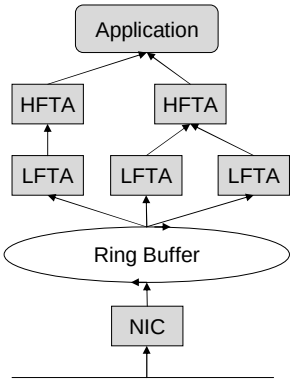


Figure 1: Gigascope architecture: LFTA is a low level query, HFTA is a high level query.

when two joining data streams have significantly different rate, Gigascope uses heartbeat mechanism to unblock one of the streams [89].

Gigascope has a two-level query architecture, where the low level is used for data reduction and the high level performs more complex analysis (see Figure 1). This approach allows the system to keep up with high speed streams in a controlled manner, in contrast to a various load shedding techniques used by other DSMSs [23, 121, 113]. Query nodes which are fed by a source data stream (e.g. packets sniffed from network interface) are called *low level queries* or *LFTAs*, while all other query nodes are called *high level queries* or *HFTAs*. Data from a high speed data stream is first placed into a ring buffer, and low-level queries read records directly from the ring buffer (thus saving a large data copying cost). Significant data reduction performed by LFTAs makes it possible to copy the filtered data stream to the HFTA level of query architecture for more complex processing. The low level queries are intended to be very fast and lightweight data reduction queries. This architecture allows Gigascope to defer expensive processing until data reaches high level of architecture, which makes processing fast and minimizes buffer requirements. Depending on capabilities of the NIC, some or all of the low query processing can be pushed farther down into the NIC itself. Choosing the most effective strategy for query processing is a challenging optimization problem, the goal of which is to maximize the data reduction without overloading LFTAs that might cause packet drops leading to incomplete query results.

3.4 Stream Sampling Algorithms

Recall that while approximate stream algorithms can be implemented as UDAFs, they return point values rather than set values. That is, to return samples $s_1, s_2, \ldots, s_k$ associated with group G, they return data in a schema such as $(G, S_1, S_2, \ldots, S_n)$ rather than as (G, S). When we considered the problem of incorporating stream sampling algorithms which return set results into a DSMS, we observed that a large class of these algorithms has a similar control structure. In this section, we survey a representative selection of stream algorithms to illustrate their common structure

3.4.1 Reservoir Sampling

The reservoir sampling algorithm [127] solves the problem of selecting a random sample of size n from a pool of N records, where the value of N is unknown. Let T be a tolerance parameter, where $10 < T < 40$; t denote the number of data records processed so far. The current set of candidates for the final sample is stored in the array C. The basic idea of reservoir sampling algorithm can be described as follows:

Within each time window:

- Make first n data records candidates for the sample by saving them into reservoir of size $(T * n)$.
- Process the rest of the record within the time window in the following manner:
  - At each iteration generate an independent random variable $\varphi(n, t)$.
  - Skip over the next φ data records.
  - Make the next data record a candidate by replacing one at random. The index of the record being replaced is $(n * random())$, where $random()$ is random number generator that returns a real number in the unit interval.
- If the current number of candidates exceeds n records, randomly choose n samples out of the reservoir of candidates.

An independent random variable φ can be generated in several ways. The fastest version of the algorithm generates φ in constant time, on the average, by a modification of von

Neumann's rejection-acceptance method and runs in average time $O(n(1 + log(N/n)))$, which is optimal, up to a constant factor.

3.4.2 Heavy Hitters

The *heavy hitters* problem is to find the elements in a data stream that account for at least ϵ fraction of the all tuples. A fast and simple heavy hitters algorithm was proposed by Manku and Motwani [102]. Let f_e be the true frequency of element e in the stream. The incoming stream is conceptually divided into buckets of width $w = \lceil 1/\epsilon \rceil$ transactions each, where ϵ is an error bound. Buckets are labeled with bucket id starting from 1. The current bucket id is calculated as $b_{current} = \lceil N/w \rceil$, where N is current length of the stream. The algorithm also uses a parameter s (support): for all collections of transactions, an itemset $X \subseteq I$, where I is universe of all items, is said to have support s if X occurs as a subset in at least a fraction s of all transactions. The data structure D is a set of entries of the form (e, f, Δ) where e is an element in the stream, f is an integer representing its estimated frequency, and Δ is the maximum possible error in f. Initially D is empty. The algorithm works as follows:

- For every new element e check whether it exists in D. If so, increment its frequency f by 1. Otherwise create a new entry in D of the form $(e, 1, b_{current} - 1)$.
- At the boundary of every bucket iterate over all elements of D. An element (e, f, Δ) is deleted if $f + \Delta \leq b_{current}$.
- When a user requests a list of items with threshold s, we output those entries where $f \geq (s - \epsilon)N$.

The algorithm is simple and uses at most $\frac{1}{\epsilon} log(\epsilon N)$ space. Although the output is approximate, the error is guaranteed not to exceed ϵ, in the sense that if $f_e \geq s$ the algorithm will return element e, and if $f_e < s - \epsilon$, the algorithm will not return e.

3.4.3 Min-Hash Computation

The *resemblance*,ρ, of two sets A and B is the size of their intersection divided by the size of their union:

$\rho(A, B) =\mid A \cap B \mid / \mid A \cup B \mid$

A *min-hash signature* [32] is a compressed representation of a set from which one can approximate the resemblance of two sets. Let $h_i(a)$ be a hash function. The signature of set A, $S(A)$, is:

$s_i(A) = min(h_i(a) \mid a \in A)$

$S(A) = (s_i(A), \ldots s_n(A))$

If $S(A)$ and $S(B)$ are two min-hash signatures, then

$\widehat{\rho}(A, B) = \sum_{i=1}^{n} I(s_i(A), s_i(B))$

Where $I(x, y)$ is the indicator function, returning 1 if $x = y$ and 0 otherwise. While any given element $s_i(A)$ can be easily computed in an SQL query, a signature typically contains 100 or more elements, making its expression in SQL quite cumbersome. However, a substitute for the minimum of N hash functions is the N minimum values of a single hash function [32]. In [54], the authors use min-hash to sample uniformly from the set of distinct elements in the stream and use it to estimate rarity (the ratio of the number of items that appear once in the stream to the number of distinct items) as well as set similarity between two windowed streams.

3.4.4 Subset-Sum Sampling

Estimation of sums of sizes of objects sharing a common set of properties is of a particular interest for the network management community. In this context the *threshold sampling* (also referred to as *subset-sum sampling*) algorithm provides a better estimate than random sampling. Like reservoir sampling, the subset-sum sampling can produce fixed size results. Unlike reservoir sampling, subset-sum sampling provides guarantees on sums of a measure attribute.

The subset-sum sampling algorithm [61] collects a sample S of tuples from R in such a way that we can accurately estimate sums from the sample. We phrase the algorithm in database language by assuming that the schema of R is (C, x), where C is an attribute we use for subset selection (the ”color” of a tuple) and x is the measure attribute. Then

$E[\sum(t.x \mid t \in S \wedge t.C = c)] = \sum(t.x \mid t \in R \wedge t.C = c)$

Furthermore, the variance of the subset sum over S is within a factor z (defined below) of

the subset sum over R.

In the basic subset-sum sampling algorithm, the user sets a threshold z, which determines the sample size. Each tuple t is sampled with probability $p(x) = min\{1, t.x/z\}$. In particular, the algorithm uses a *counter*, initialized to zero, and works in the following manner:

- For every new tuple t, check whether $t.x > z$. If yes, sample the tuple. Otherwise, add value of the t.x to the small flow counter.
- If tuple was not sampled, check whether $counter > z$. If yes, subtract z from counter and sample the tuple, setting $t.x$ to z. Otherwise discard the tuple.

The idea behind this algorithm is that tuples with large values of $t.x$ contribute the largest amount to a sum. Therefore all large tuples are sampled; however small tuples cannot be discarded without biasing some subset-sum. The algorithm samples one small tuple every time the combined weight of the small tuples exceeds z. To estimate the sum, the measure $t.x$ of the sampled small tuple is adjusted to z, since it represents a weight of threshold z: $t.x = max\{t.x, z\}$.

The result of the algorithm described above is a sample of arbitrary size, which introduces an element of unpredictability. In many cases we would like a sample of a particular size, say 1000 samples regardless of the distribution of $t.x$ or the size of R. The second version of the algorithm (*dynamic subset-sum sampling*) will produce a consistent number of sampled tuples. The user specifies the desired sample size N and an initial value of the threshold z. In addition to small tuples count (`count`), the algorithm tracks the number of tuples sampled so far (`sample_count`). The algorithm works in the following manner:

- Collect samples according to the basic subset-sum sampling algorithm, keeping a count of the number of sampled tuples in `sample_count`.
- If `sample_count` $> \gamma * N$ (e.g., $\gamma = 2$), estimate a new value of z which will result in N tuples. Subsample S using basic subset-sum sampling and the new value of z, and continue with basic subset-sum sampling.
- When all tuples from R have been processed, if `sample_count` $> N$ then adjust z and

subsample S using basic subset-sum sampling.

When applied to a data stream, subset-sum sampling occurs in successive time windows. In this case, an initial threshold can be estimated for the new time window using the threshold from the old time window, adjusting its value to obtain an estimated N samples during the new time window.

The authors of [61] suggest a variety of strategies for adjusting z. In our implementation, we used the *aggressive* version of the z threshold adjustment (z-threshold, $\mid S \mid$-currently maintained number of samples, M-desired number of samples, B-number of samples for which sample size $>$ threshold):

If $0 \leq |S| \leq M$, then $z_{new} = z_{old}(|S|/M)$

If $|S| \geq M$, then $z_{new} = z_{old}((max\{|S| - B, 1\})/(M - B))$

3.4.5 Summary

We observe that these stream sampling algorithms are quite sophisticated, and far from "pick each item with some probability" that one expects from uniform sampling. They also solve very different problems and each has found many applications. Still they follow a common pattern. First a number of items are collected from the original data stream according to a certain criteria, and perhaps with aggregation in the case of duplicates. If a condition on the sample is triggered (e.g., the sample is too large), a cleaning phase is triggered and the size of the sample is reduced according to another criteria. This sequence can be repeated several times until the border of the time window is reached and the sample is output. This framework fits each of the summarized algorithms as follows:

- **Subset-Sum sampling**: Sample records according to the basic subset-sum sampling algorithm. Trigger the cleaning phase when `count_sample` $> \gamma * N$. In the cleaning phase, adjust z and subsample.

- **Heavy hitters**: Count the frequency of occurrence for every distinct sample. Trigger the cleaning phase every w input tuples. In the cleaning phase, delete samples according to the defined rules.

- **Min-hash**: Sample a hash value whenever it is within the smallest N of hash values seen thus far. Trigger the cleaning phase when the number of samples exceeds $\gamma * N$. In the cleaning phase, remove the hash values larger than the Nth smallest value seen thus far.

- **Reservoir sampling**: repeatedly generate φ, skip that number of records, and select the next record for the reservoir. Trigger the cleaning phase when the sample size exceeds $(T * n)$. In the cleaning phase, randomly choose n records from the reservoir to keep and delete the rest.

The common framework above inspires our operator in the next section.

3.5 The Sampling Operator

From the discussion above, we derive a number of common characteristics for the sampling algorithms in question:

- A "global" state structure.
- A loose predicate for admitting a tuple to the sample.
- A predicate which triggers a sample cleaning phase.
- A predicate for removing samples during the cleaning phase.
- A finishing-off predicate.

The process of sampling is in some ways similar to that of aggregation, as they both collect and output sets of tuples which are representative of the input. Accordingly, our textual representation of the sampling operator is based on the textual representation of aggregation:

SELECT `<select expression list>`
FROM `<stream>`
WHERE `<predicate>`
GROUP BY `<group-by variables definition list>`
[SUPERGROUP `<group-by variable list>`]
[HAVING `<predicate>`]

CLEANING WHEN `<predicate>`

CLEANING BY `<predicate>`

The "global" state structure stores the control variables of the sampling algorithm. For example, in the Manku-Motwani algorithm [102] the state stores variables such as the count of tuples processed since the last cleaning phase and the number of cleaning phases that have been triggered. Since we might wish to obtain a sample on a group-wise basis (e.g., for each source IP address, report the destination IP addresses accounting for at least 10% of the total packets sent from the source IP), we associate the sampling state with *supergroups*, and samples with the groups in a supergroup. The variables in the SUPERGROUP clause must be a subset of group-by variables defined in the GROUP BY clause (thus, supergroups are a specialization of grouping sets [77]). By default, the supergroup is ALL. Along with sampling state variables, the supergroup can compute superaggregates (aggregates of the supergroup rather than the group). One example of a useful superaggregate is `count_distinct$()`, which returns the number of distinct groups in a supergroup (we use the `$` to denote that an aggregate is associated with the supergroup rather than the group).

More concretely, the semantics of a sampling query is as follows:

- When a tuple is received, evaluate the WHERE clause. If the WHERE clause evaluates to false, discard the tuple.
- Else if the condition of the WHERE clause evaluates to `TRUE` then
  - Create and initialize a new supergroup and a new superaggregate structure if needed, otherwise update the existing superaggregates (if any).
  - Create and initialize a new group and a new aggregate structure if needed, otherwise update the existing aggregates (if any).
  - Evaluate the CLEANING_WHEN clause.
  - If the CLEANING_WHEN predicate is `TRUE`
    - Apply CLEANING_BY clause to every group.
    - If the condition of CLEANING_BY clause evaluates to `FALSE`
      - Remove group from the group table, and update any superaggregate

- When the sampling window is finished,
  - Evaluate the HAVING clause on every group.
  - If the condition in the HAVING clause is satisfied, then the group is sampled, else discard the group.

That completes the description of the operator. The discussion thus far is independent of any specific DSMS.

3.6 The Operator in Gigascope

In this section, we discuss how sampling operator interacts with a specific DSMS, namely Gigascope, and is realized in it.

3.6.1 Sampling Operator in Gigascope

The sampling operator in previous section brings up certain details within Gigascope. For example, in the Gigascope DSMS, the sampling window ends whenever any ordered group-by variable changes value, so the sampling operator will produce output once every time window. As a corollary, all ordered group-by variables are part of the supergroup. Also, in some algorithms, e.g., dynamic subset-sum sampling, initial values of a state in a new time window are derived from the state of the old time window. Our implementation of the sampling operator supports this at superaggregate structure initialization time by checking if a supergroup with the same non-ordered group-by variables existed in the previous time window. If so, all states in the new superaggregate are initialized by a function which accepts the equivalent state from the old time window.

For an example, the following Gigascope query expresses the dynamic subset-sum sampling algorithm which collects 100 samples:

```
SELECT uts, srcIP, destIP, UMAX(sum(len),ssthreshold())
FROM TCP
WHERE ssample(len,100) = TRUE
GROUP BY time/20 as tb, srcIP, destIP, uts
HAVING ssfinal_clean( sum(len), count_distinct$(*)) = TRUE
```

CLEANING WHEN `ssdo_clean(count_distinct$(*)) = TRUE`

CLEANING BY `ssclean_with(sum(len)) = TRUE`

where `UMAX(val1, val2)` is a function which returns the maximum of the two values, and `uts` is a nanosecond granularity timestamp (with its timestamp-ness cast away) used to make each tuple its own group.

The `sshthreshold(), ssample(), ssfinal_clean(), ssdo_clean()` and `ssclean_with()` functions are *stateful functions*, which we discuss in the next section.

To complete the description of the sample operator, we need to discuss some working details, which we do in the context of our implementation in Gigascope.

3.6.2 Stateful Functions

To implement some of the algorithms, a number of functions need to access the same global state throughout the execution. For this reason, we call those functions *stateful*. Typically, a collection of functions will share the same *state* structure. Stateful functions are very similar to UDAFs, but with the following differences:

- They can produce output a number of times during the execution.
- The state can be modified only when the functions that share the state are referenced.

A state is declared as follows:

STATE `<type> <name>`;

The declaration of stateful functions ties the function to the state it shares:

SFUN`<type>[modifiers]<state_name><function_name>(<param_list>)`

In case of subset-sum sampling algorithm:

STATE `char[50] subsetsum_sampling_state`;

SFUN `int subsetsum_sampling_state ssample(int,CONST int)`;

SFUN `int subsetsum_sampling_state ssfinal_clean(int, int)`;

SFUN `int subsetsum_sampling_state ssdo_clean(int)`;

SFUN `int subsetsum_sampling_state ssclean_with(int)`;

SFUN `int subsetsum_sampling_state ssthreshold()`;

When the query references a new supergroup, the space for the SFUN state is allocated in the superaggregate structure. The state is initialized with its associated initialization func-

tion. For example, the prototype of the state initialization function in our implementation of the sampling operator is:

```
void_sfun_state_init_<state name>(<pointer to memory for the state>,
                                  <pointer to old state, or NULL>);
```

Stateful functions are implicitly passed a pointer to their associated state. In our implementation, the prototype of the stateful functions has the following form:

```
<return type> <name>(void *s, <param_list>);
```

where s is the pointer to the state. In the case of our subset-sum sampling implementation, some of the functions that we added to the Gigascope runtime library are:

```
void _sfun_state_init_subsetsum_sampling_state( void* n, void* o);
int ssample(void*s, int len, int sample_size);
```

3.6.3 Groups and Supergroups

As discussed earlier, very often there is a need to reference global aggregates, or supergroups. For instance, in subset-sum sampling the cleaning phase is triggered when the number of groups exceeds the threshold (it's important to notice that in the subset-sum sampling implementation every packet needs to be distinctly unique, thus every group consists of a single packet). Another example of the query that uses supergroups is the min-hash problem, when we would like to compute k min-hash destination IP addresses per source IP address; and hence we need a superaggregare which returns the kth smallest value.

There is a difference between regular aggregation and global (super) aggregation. To be able to maintain superaggregate, we need to maintain group aggregate of the same type. When a new group is added or deleted (as a result of the cleaning phase), we need to update the supergroup aggregate by adding or subtracting the group aggregate value. One of the useful superaggregates is `count_distinct$()` which reports the number of groups in the supergroup.

3.6.4 Sampling Operator Implementation

Our implementation of the sampling operator maintains three types of hash tables: one for the groups, one for the supergroups and an additional table that keeps track of all groups

for every supergroup:

- **Group table**: *key* - set of group-by variables; *value* - structure that maintains group aggregates.
- **Supergroup table**: *key* - set of supergroup variables not including ordered variables (when no supergroup is specified, the key is associated with a single time window); *value* - structure that maintains state(s) associated with the supergroup, and any superaggregates.
- **Supergroup-Group table**: *key* - set of supergroup variables (when no supergroup is specified, the key is associated with a single time window); *value* - list of all groups in this supergroup.

Note that the key of the supergroup table is always a subset of elements that represent the key of the group table.

We actually maintain two supergroup hash tables - “old” and “new”. The “old” supergroup hash table maintains all the supergroups that were sampled in the previous window. The evaluation process can be summarized as follows:

- When a tuple is received, compute the key for the supergroup table using group-by variables.
- If at the border of the window, call `final_init()` function for the states in the new supergroup table (to signal to the state that the time window is finished) and apply HAVING clause to every group of the new group hash-table. Clear the group table, the old supergroup table, and the supergroup-group table, and move the new supergroup table to the old supergroup table.
- If the supergroup of the newly arrived tuple exists in the new supergroup table, then apply WHERE condition to the tuple. If the condition evaluates to `TRUE`, update superaggregates of the supergroup, else start processing next tuple.
- If the supergroup doesn’t exist in the new supergroup table, check whether the supergroup with the same key exists in the old supergroup table. If so, initialize the state of

the new supergroup by using `state_init()` function, passing a pointer to the old state as the second argument. If the supergroup is entirely new, pass a `NULL` as the second argument. Create a new supergroup in new hash table. Apply WHERE condition to the tuple. If the condition evaluates to `TRUE`, update superaggregates.

- Compute key for the group table using group-by variables.
- If the group with this key exists in the new group hash-table, update group aggregates.
- If the group doesn't exist, create a new group and new aggregates of the group. Add the key of the group to the supergroups' entry in the supergroup-group table.
- Apply the CLEANING WHEN condition to the supergroup state. If the condition evaluates to `TRUE`, trigger the cleaning phase by applying CLEANING BY clause on every group that belong to the current supergroup (i.e., using the supergroup-group hash-table). If the condition evaluates to `FALSE`, then delete the group from the group hash-table and remove its key from the supergroup's supergroup-group table.
- Stateful functions that appear in SELECT clause will be evaluated last, when the output tuple is created.

3.6.5 Evaluation Example

Let us consider an example of the subset-sum sampling algorithm. The global structure of the algorithm uses a number of parameters, such as the value of the threshold z, the counter of small packets `count`, the counter of large packets `bcount`, value of the cleaning threshold γ, etc. The evaluation process of the query that expresses the algorithm is as follows:

- When the tuple is received, call `ssample()` function:

  The loose predicate for admitting a tuple to the sample is the basic subset-sum sampling predicate using the current value of z. If the function returns false, then the predicate condition had failed and we start processing next tuple. If the function returns true, process the tuple by creating (or updating) appropriate entries for supergroup, group and supergroup-group hash tables.

- Call `ssdo_clean()` function:

  The cleaning phase is triggered when the current sample size exceeds the threshold of the number of samples that can be maintained by currently processed supergroup. If the function returns false, the condition is not met and we start processing next tuple. Otherwise, z is adjusted and the cleaning phase is triggered.

- Call `ssclean_with()` function on every group of currently processed supergroup. The current sample is cleaned by applying the new value of threshold for the size of the data record and deleting those records which don't meet the cleaning condition. The cleaning condition states that if the size of the data record is smaller than the value of the threshold before the most recent adjustment (`z_prev`), then `z_prev` will replace size of the record during the cleaning phase.

- Call `ssfinal_clean()` at the border of every window. If the number of samples still exceeds the desired size of the final sample, do the final subsampling. This function implements the final cleaning condition which is identical to the cleaning condition implemented in `ssclean_with()` function. If the function call returns false, the group is evicted from the hash table. Otherwise, the group is sampled and the output tuple is created.

3.6.6 Query Example

Although we have focused on the dynamic subset-sum sampling implementation, in this section we show how the other three algorithms from our representative four can be implemented using the generic sampling operator.

Query for Heavy Hitters Algorithm: This query will report the 100 most common source addresses within a time window of 1 minute. The function `current_bucket()` returns id of current bucket. The aggregate `first()` returns the first value that was returned by `current_bucket()` function within the current time window. The function `local_count(`N`)` increments `current_bucket` and returns true once every N calls.

```
SELECT tb, srcIP, sum(len), count(*)
```

```
FROM TCP
GROUP BY time/60 as tb, srcIP
CLEANING WHEN local_count(100) = TRUE
CLEANING BY count(*) < (current_bucket()- first(current_bucket())
```

Query for Min-Hash Computation: This query will report 100 min-hash values of destination IP addresses per source IP address. This query does not make use of stateful functions but instead relies on the `count_distinct$(*)` and the `Kth_smallest_value$(HX,100)` superaggregates (`Kth_smallest_value(x,`n`)` returns the nth smallest value of `x`).

```
SELECT tb, srcIP, HX
FROM TCP
WHERE HX <= Kth_smallest_value$(HX, 100)
GROUP_BY time/60 as tb, srcIP, H(destIP) as HX
SUPERGROUP BY tb, srcIP
HAVING HX <= Kth_smallest_value$(HX,100)
CLEANING WHEN count_distinct$(*) >= 100
CLEANING BY HX <= Kth_smallest_value$(HX, 100)
```

Query for Reservoir Sampling Algorithm: This query will return 100 random samples per time window of 1 minute. The function `rsample(100)` implements the sampling condition by returning true for those tuples that should be saved in the reservoir of candidate tuples, and returning false for those that are skipped over. The function `rsdo_clean()` returns true when the number of candidates (`count_distinct$()`) exceeds the threshold value of Tn, and returns false otherwise. The functions `rsclean_with()` and `rsfinal_clean()` randomly subsample n final samples the reservoir of candidates:

```
SELECT tb, srcIP, destIP
FROM TCP
WHERE rsample(100) = TRUE
GROUP_BY time/60 as tb, srcIP, destIP
```

```
HAVING rsfinal_clean() = TRUE
CLEANING WHEN rsdo_clean(count_distinct$()) = TRUE
CLEANING BY rsclean_with() = TRUE
```

3.6.7 Flow Sampling

The following example demonstrates the flexibility of the sampling operator. In network traffic analysis it is often useful to perform network measurements using flow statistics rather than packet statistics, since flows offer a considerable compression of information over packet headers. The straightforward implementation of this approach in terms of the stream sampling operator can be expressed as a set of queries, where basic flow aggregation is performed as a first query, and the result is fed to a higher level sampling query. (Note that the high level query is the sampling query described previously in section 3.6.1).

```
DEFINE query_name 'psample';
SELECT tb, srcIP, destIP, totalBytes,
       packetCnt, UMAX(packetCnt, ssthreshold())
FROM source
WHERE ssample(packetCnt, 100) = TRUE
GROUP BY tb,srcIP, destIP, totalBytes, packetCnt
HAVING ssfinal_clean(packetCnt) = TRUE
CLEANING WHEN ssdo_clean()= TRUE
CLEANING BY ssclean_with(packetCnt) = TRUE;

DEFINE query_name 'source';
SELECT tb , srcIP, destIP, COUNT(*) as packetCnt,
       SUM(len) as totalBytes
FROM TCP
GROUP BY time/60 as tb, srcIP, destIP
```

In the example, the source query is a low-level query that performs aggregation of flows over the time window of 1 minute. The output of the query is fed to the high-level *psample* sampling query, which performs flow sampling. The query outputs 100 samples per time window.

However, this implementation exhibited difficulties under certain network conditions,

in particular when there are a large number of small flows consisting of only a few packets (e.g. during DDOS attacks). Under these conditions, the flow aggregation query requires an enormous number of groups (corresponding to the enormous number of flows), exhausts the available memory, and fails. One way of fixing this problem would be to emulate the sampling process performed by CISCO routers and uniformly sample 1 out of 500 packets at the flow aggregation query (i.e. at the query source). While this approach allows monitoring of the DDoS attacks, it doesn't provide a sufficiently informative flow sample for their analysis.

To overcome this problem we have developed the approach of integrating sampling and flow aggregation and doing them simultaneously on the traffic at the packet level. At the high level, this involves integrating the logic of different sampling algorithms into the semantics of a flow aggregation engine. In databases, there is a well-established principle that selection is "pushed" to the lowest level in the query processors. Our approach may be seen as an instantiation of this principle to data stream management systems in general and to network traffic analysis in particular. This is not only a sound and efficient flow sampling mechanism in general; we believe that this is the only scalable approach to flow sampling at adverse conditions. The key trick is that small flows can be quickly sampled and purged from the group table. The new sampled flows query is a more stable implementation that is resistant to rapid network changes.

We demonstrate our approach using an example. We modified the implementation of the subset-sum sampling algorithm by integrating flow aggregation with sampling into a single query processing phase. Here `P` is the set of conditions which indicate that the flow is closed:

```
DEFINE query_name 'fsample';
SELECT tb, srcIP, destIP, COUNT(*),UMAX(sum(len),ssthreshold())
FROM source
WHERE flow_ssample(100) = TRUE
GROUP BY time/60 as tb, srcIP, destIP
HAVING flow_ssfinal_clean(P) = TRUE
CLEANING WHEN flow_ssdo_clean(max$(time)) = TRUE
CLEANING BY flow_ssclean_with(P) = TRUE;
```

```
DEFINE query_name 'source';
SELECT time, srcIP, destIP, len
FROM TCP
```

Note that in the above set of queries, the low-level query does not do any aggregation and only performs preliminary projection of the incoming data stream. The high-level sampling query uses a new set of stateful functions and is evaluated by the stream sampling operator in the following manner:

- When a tuple is received, evaluate the WHERE clause. Call `flow_ssample()`, which always returns `TRUE` and admits all incoming tuples into the sample without performing any preliminary filtering. As a result, the group table now collects and maintains flow statistics, each group representing a distinct flow of data. The function also provides the algorithm with the information about the size of the desired sample.

- Evaluate the CLEANING WHEN clause. Call `flow_ssdo_clean()`. This function implements two phases of query evaluation - the counting phase and the cleaning phase. The counting phase is triggered every second. During this phase we count the number of "closed" flows which are currently in the group table. The count of "closed" flows is used to trigger the cleaning phase. The cleaning phase is triggered when the current number of "closed" flows, which was obtained during the most recent counting phase, exceeds the threshold for the number of samples. If the function returns `FALSE`, neither of the two conditions is met and we proceed to the next tuple.

- Evaluate the CLEANING BY clause whenever CLEANING WHEN returned `TRUE`. Call `flow_ssclean_with(P)` function, where `P` is a set of conditions which indicate whether a flow is closed. For instance, a flow can be considered closed if we have received `FINISH` or `RESET` or there was no packet from this flow within last 5 seconds:
  `flow_ssclean_with( (Or_Aggr(finish)|Or_Aggr(reset)),5)`
  If the function is called during the counting phase, `P` is applied to every group to determine whether the group is closed. The counter of closed flows that are not evicted from the group hash table is incremented accordingly. The function always returns

`TRUE` during the counting phase When the function is called during the cleaning phase, the current set of closed flows is subsampled by applying to each group the newly estimated value of the threshold for the size of the tuple and deleting those groups which do not meet the cleaning condition. The function returns `TRUE` if the group satisfies the condition.

- The HAVING clause is evaluated only when the sampling window is closed. At that point all flows are considered closed. Call `flow_ssfinal_clean(P)`, which performs the final subsampling of the current sample only if the size of the current sample exceeds the desired size. If the function returns `FALSE`, the group is evicted from the hash table. Otherwise, the group is sampled.

- SELECT is applied to every sampled group while it is being output as the answer to the query. This implementation of the algorithm allows creating very informative flow samples on streams of network data. In addition, the implementation is able to handle heavy loads and is resistant to rapid network changes.

This implementation of the algorithm allows creating very informative flow samples on streams of network data. In addition, the implementation is able to handle heavy loads and is resistant to rapid network changes.

3.7 Experimental Study

We implemented the sampling operator in the Gigascope DSMS in order to experiment with the feasibility and performance of the operator. The Gigascope implementers also provided us with access to several network data streams. We implemented not only the operator, but also amended the parser and query analyzer to instantiate the sampling operator from a query with the textual representation described in Section 3.5.

In our experiments, we focus on the dynamic subset-sum sampling algorithm. The dynamic subset-sum sampling algorithm is used extensively in the AT&T network performance monitoring infrastructure [61], and consequently this algorithm is well understood by the Gigascope developers. In addition, the Gigascope developers indicted that dynamic

subset-sum sampling is a good first algorithm because of the demand for its use. Our implementation of dynamic subset-sum sampling follows the description given in Section 3.5.

We had two network feeds available for experiments. The first is the network connection to our research center. This data stream produces a moderate 5,000 to 15,000 packets per second, with a rate that is highly variable. The second network feed is a data center tap, producing moderately high speed 100,000 packets per second (about 400 Mbits/sec). This data feed is highly aggregated, and hence has a much lower variability in its data rate than the first. When testing accuracy, we generally use the first data feed because its high variability will tend to emphasize estimation problems. When testing performance, we generally use the second data feed because its low variability and high data rate make measurements much more consistent. For our experiments we used an inexpensive dual 2.8 GHz processor server.

3.7.1 Accuracy

We measured the accuracy of the dynamic subset-sum sampling algorithm by running two query sets simultaneously. One computed the sum of packet lengths during successive 20 second intervals, and the other applied dynamic subset-sum sampling to collect 10,000 samples of packets, then computed the sum of (subset-sum sample adjusted) packet lengths for each time interval. We found that on many of the time intervals, the dynamic subset-sum sampling algorithm is inaccurate. This property is illustrated in Figure 2, where the aggregate result is labeled *actual* and the dynamic subset-sum sampling result is labeled `estimated (non-relaxed)`.

The problem lies in the threshold update procedure discussed in 3.4.4. The load during the next interval is estimated to be the load during this interval; if the load drops sharply, dynamic subset-sum sampling collects too few samples and underestimates the sum.

To correct this problem, we made a minor adjustment to the dynamic subset-sum sampling so that it will estimate that the load in the next time period is a fraction $1/f$ of the load during this interval. We call this the relaxed version. In Figure 2 we use $f = 10$ and the relaxed estimates match the actual sum very closely for all time periods. The relaxed algorithm works well because the cleaning phases readily adapt the threshold upward to

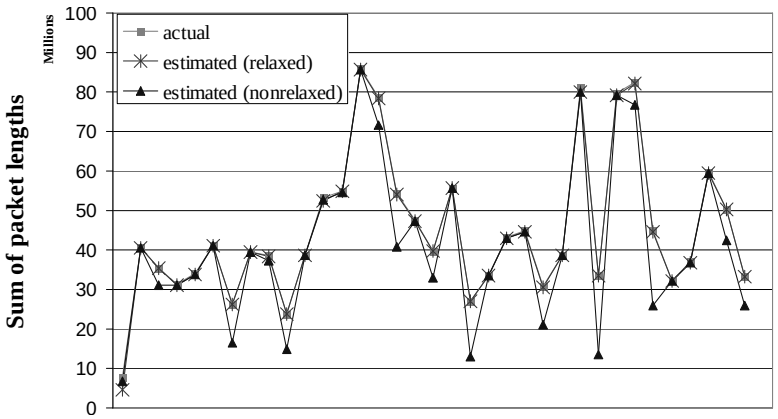


Figure 2: Accuracy of summation of subset-sum sampling

the appropriate value.

Another illustration of the problem with non-relaxed subset-sum sampling is shown in Figure 3 (a). The relaxed algorithm occasionally over-samples, while the non-relaxed algorithm frequently under-samples causing an underestimation.

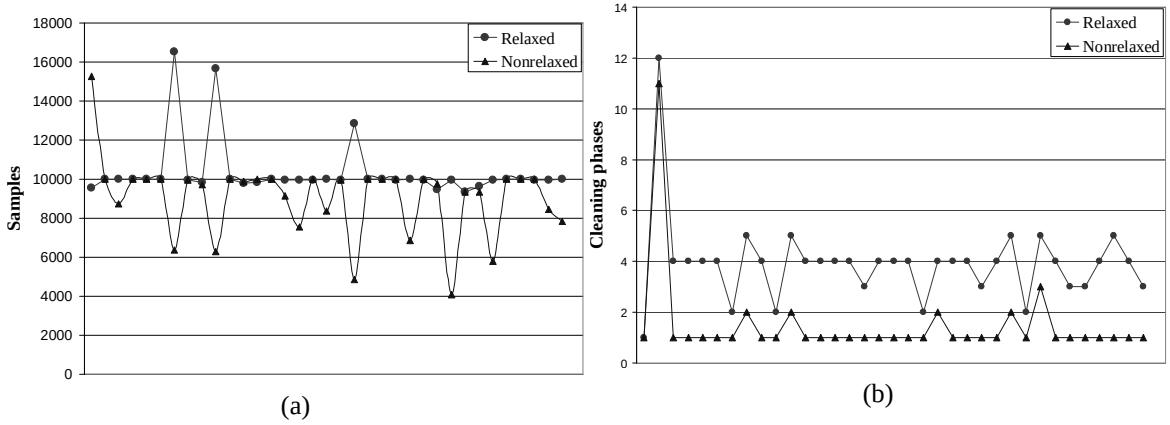


Figure 3: Relaxed vs. Nonrelaxed: (a) Actual number of samples produced per period with the requested sample size of 10000. (b) Number of cleaning phases per period.

The cost of the relaxed algorithm is that the cleaning phase is invoked more frequently. Figure 3 (b) shows the number of cleaning phases for the relaxed and non-relaxed dynamic subset-sum sampling algorithms during the experiment. The first interval was very short (as can also be seen from the other charts). In the second interval, both algorithms used a large number of cleaning phases to identify the appropriate threshold; afterwards the number of cleaning phases stabilized at a low level. The relaxed algorithm consistently used about 4 cleaning phases, as compared to 1 for the non-relaxed algorithm. If the cost of the cleaning phase is small (which we explore in the next section), using the relaxed algorithm incurs

only a small overhead.

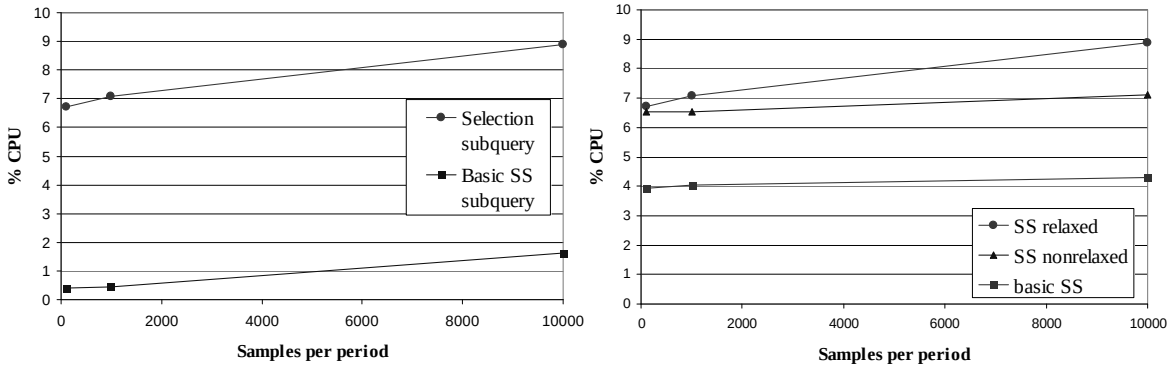
We repeated these experiments to collect 100 and 10,000 samples per period, and obtained nearly identical results (a user will collect a larger or smaller number of samples depending on storage costs and the degree of subsetting during analysis).

3.7.2 Performance

To evaluate the CPU overhead of running adaptive subset-sum sampling using our sampling operator, we ran both the relaxed and the non-relaxed dynamic subset-sum sampling algorithms on the high speed link (100,000 packets/sec), as the CPU utilization of these queries on the moderate speed link is too low to measure accurately. For a comparison, we also ran basic subset-sum sampling using a user-defined function in a selection operator. A comparison of the CPU usage for each of these algorithms is shown in Figure 4 (a). Even when processing 100,000+ packets/sec and producing large outputs, the dynamic subset-sum sampling algorithm implemented using the sampling operator uses only a small fraction of a CPU (two CPUs are available at the server). Compared to the selection query (basic subset-sum sampling), the sampling operator uses only about 3% to 5% additional CPU load. The cost of the additional cleaning phases to support relaxed subset-sum sampling can be seen in this chart.

However the overhead is small, at most about 2% of CPU for this experiment. However, there is a problem with this implementation of dynamic subset-sum sampling. Recall that there are two types of queries nodes in the Gigascope architecture: low level queries which read from the network interface, and high level queries which read from Gigascope-managed query streams. The low-level queries nodes are simple data reduction operators. Currently only selection and (partial) aggregation are supported. Therefore we need to run a low-level selection query to feed the subset-sum sampling queries. In the run of experiments shown in Figure 4 (a), evaluating the low-level query required about 60% of a CPU, due to the cost of memory copies.

Fortunately, it is possible to evaluate part of a subset-sum sampling query at the low-level query. We modified the low-level selection query to have it perform basic subset-sum sampling with a threshold $1/10$th the level used by the dynamic subset-sum sampling al-



(a) Subset-sum sampling CPU usage.

(b) Effect of low level query type.

Figure 4: Performance analysis

gorithm when it returns 10,000 samples per interval. The low-level query load dropped to about 4% of a CPU. In addition, the dynamic subset-sum sampling CPU load dropped significantly, as shown in Figure 4 (b). We ran additional experiments regarding the setting of γ (the trigger to initiate a cleaning phase). Increasing (decreasing) γ decreases (increases) the number of times cleaning is done, but increases (decreases) its cost. We found little dependence of CPU load on γ.

3.7.3 Flexibility

We ran a set of experiments to measure the memory usage of the alternative queries for computing a given sized sample of flows over a 1 minute interval. We used a Gigascope installation consisting of a dual 2.8Ghz processor server with 4 Gbytes of RAM, with two Gigabit Ethernet interfaces connected to a live network feed from a data center tap. Each of the Gigeths carries about 100,000+ packets/sec.

For a given number of desired flow samples (10,000, 50,000, and 100,000) we ran a set of queries consisting of the flow+sample query (*fsample*), and the flow computation, then sampling query (*psample*). Both of the queries were described in detail in section 3.6.7. We modified the *psample* query to first randomly sample the packets, with sampling rates of 100% (no sampling), 20%, 10%, and 2%. To adjust the input rate, we ran experiments with data from one of the interfaces (1 stream) and from both (2 streams). For a given number of output flow samples and input streams, we ran all of the sampling queries simultaneously; hence these results all reflect the same conditions. The six sets of experiments were

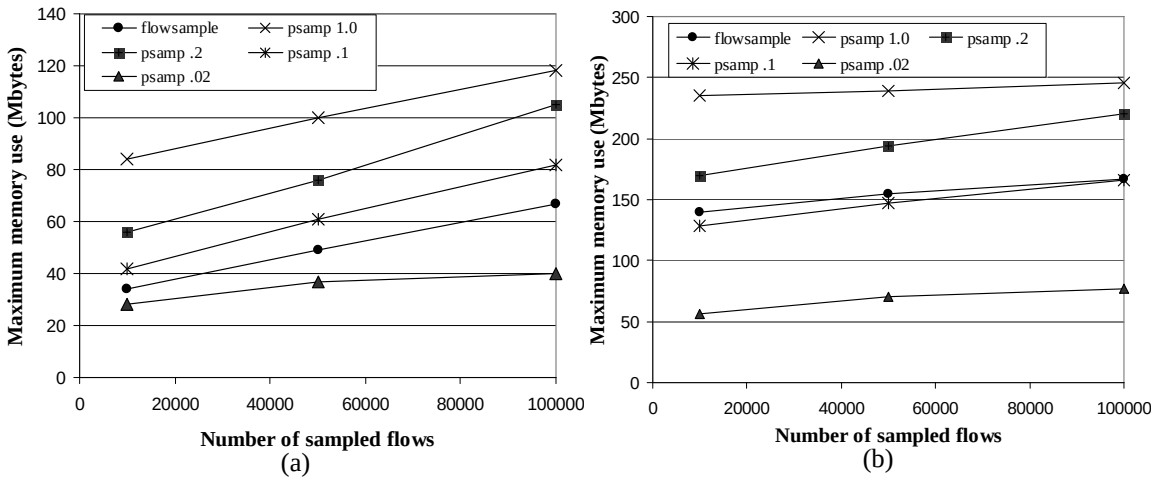


Figure 5: Flow sampling memory usage: (a) One stream (110 packets per second). (b) Two streams (210 packets per second).

all run within a short period of time, so they executed under similar though not identical conditions. The memory usage results are shown in Figures 5.

These results show that computing the sampled flows directly (*fsample*) significantly reduces memory usage as compared to computing flows, then sampling (*psample* 1.0). To obtain an equivalent reduction in memory usage by packet sampling, we need to use a packet sampling rate of 10% or less, rendering individual flows unsuitable for analysis. If the rate of sampling is too low, the *psample* query cannot deliver the desired number of flows. For example, the *psample* 0.02 query given a single stream input could only deliver about 70,000 flows per minute (the memory usage of the *psample* 0.02 curves level out for this reason).

The *fsample* query imposes a moderate maximum CPU overhead: 42% of one CPU for the *fsample* query run under two streams and collecting 100,000 flows per minute, as compared to 33% for the *psample* 0.01 query run under the same conditions. Packet sampling does significantly reduce the CPU utilization of the *psample* query. However, since we are able to run five simultaneous sampling queries on a high speed data stream, CPU use does not seem to be a significant issue.

3.8 Conclusions

Query sets which make use of very high speed data streams must often use approximate data reduction strategies to provide complex statistics while keeping up with the offered

data load. A useful approximation technique is sampling, which reduces the data set into a much smaller and yet representative result. Typical sampling methods are often quite simple: sample each item with some probability, say p. But in streaming context, even uniform sampling from distinct elements on the stream is a challenge. Over the past few years, researchers have proposed very sophisticated sampling algorithms on streams for a variety of problems. Rather than propose new stream sampling methods, we have focused on how to implement the many intricate sampling methods in the literature. Our approach has been to abstract and propose a new stream operator for evaluating sophisticated sampling algorithms, on a data stream. This operator is powerful enough to evaluate many widely different stream sampling algorithms including subset-sum sampling (from networking), reservoir sampling (from databases), min-hash sampling (from theoretical algorithms), etc., as well as sampling-based aggregation algorithms such as the Manku-Motwani heavy hitters' algorithm, and many more. We urge the readers to try modeling other stream sampling algorithms via our stream operator to appreciate its flexibility and generality. Some of our ongoing work consists of cascading one type of stream sampling inside a different type of stream sampling group.

We implemented the sampling operator in the Gigascope DSMS, and implemented dynamic subset-sum sampling on top of that. We made a performance evaluation of dynamic subset-sum sampling on both highly variable and high speed data streams. We found that:

- The accuracy of the dynamic subset-sum sampling algorithm can be greatly improved by *relaxing* the threshold between time windows. This was re-engineering that was a result of experience with the real system.
- The sampling operator imposes only a small CPU overhead, as compared to a simple selection operator. We can readily scale subset-sum sampling to much higher data rates.
- By performing part of the subset-sum sampling at the low level query, we can collect a 1% subset-sum sample on a high speed data stream using less than 6% of a CPU.

Obtaining the best performance from a DSMS such as Gigascope requires a significant amount of early data reduction at the low-level queries. The method for doing this will

depend on the approximation algorithm. For example, the Manku-Motwani heavy hitters algorithm would be best supported by aggregation at the low-level queries. We have not explored operator transforms in this work, but we have gained valuable query optimization tips during our experimental study.

The significance of our results is that we have developed a simple way in which sophisticated streaming algorithms that returns set results can be integrated into a query system. The supporting UDAFs and functions need only follow a simple API. Once written, the user has the power of the query language to explore new combinations. This ease of experimentation allowed us to find the simple upgrade of subset-sum sampling which so improved its accuracy. The relaxed version of subset-sum sampling, along with the sampling operator, has been incorporated into the release version of Gigascope. This implementation is the first one that we know of in an operational DSMS which can handle line speeds.

Our success stems from our observation that a large class of sampling algorithms have an essentially simple communication structure, namely between individual samples and a sample summary only. We have focused on this core aspect of sampling algorithms. We note that it is quite possible to derive sampling-based algorithms that operate on the samples in more complex ways and therefore require a far more complex communication structure. An excellent example is a more-holistic sampling algorithm such as the Greenwald-Khanna quantile algorithm [76]. The *compress* phase of this algorithm merges adjacent samples, and thus requires inter-sample communication. This algorithm (expressed as a UDAF in [44]) and others which may have such computations on samples built into them, are best expressed using a stream UDAF on top of the sampling operator we have developed here. In contrast, all sampling algorithms that work on a per-sample tuple basis can be implemented using our sampling operator.

In addition to capturing capturing a common thread of evaluation of a large variety of sampling algorithms, our sampling operator is able to maintain information about groups and supergroups in terms of aggregates and superaggregates required for implementation and statistical analysis of a sampling algorithm. We believe that this, along with stateful functions, gives the user the level of flexibility required for implementation and cus-

tomization of various sampling-related algorithms. Our work with subset-sum sampling demonstrated this, but we provide another example.

Paradoxically, existing methods to sample flows - 1 in X sample from packets and then aggregate flows, or aggregate into flows and then sample the output - fail when they are most needed, i.e., at times of adverse traffic conditions such as network attacks. We have proposed a solution by integrating the logic of flow aggregation as well as flow sampling into one procedure that works directly on the IP packet data stream. Our solution works at more than 200k+ packets per second, with only moderate load on the CPU and outperforms existing methods. Our approach is not only an efficient way to sample from flows, but may also be the only viable way during adverse traffic conditions when the number of flows increases tremendously. Also, the general principle of “pushing selection operators” to low level in stream processing is likely to find other applications in IP network data analysis.

Chapter 4

4 Summarizing and Mining Inverse Distributions on Data Streams via Dynamic Inverse Sampling

Emerging data stream management systems approach the challenge of massive data distributions which arrive at high speeds while there is only small storage by summarizing and mining the distributions using samples (see chapter 3 for an overview of sampling algorithms) or sketches. However, data distributions can be "viewed" in different ways. A data stream of integer values can be viewed either as the *forward* distribution $f(x)$, i.e., the number of occurrences of x in the stream, or as its inverse, $f^{-1}(i)$, which is the number of items that appear i times. While both such "views" are equivalent in stored data systems, they may be significantly different over data streams that entail approximations. In other words, samples and sketches developed for the forward distribution may be ineffective for summarizing or mining the inverse distribution. Yet, many applications such as IP traffic monitoring naturally rely on filtering methods that require mining inverse distributions.

In this chapter we formalize the problems of managing and mining inverse distributions and show provable differences between summarizing the forward distribution vs the inverse distribution. We present filtering methods for summarizing and mining inverse distributions of data streams: they rely on a novel technique to maintain a dynamic sample over the stream with provable guarantees which can be used for variety of summarization tasks (building quantiles or equidepth histograms) and mining (anomaly detection: finding heavy hitters, and measuring the number of rare items), all with provable guarantees on quality of

approximations and time/space used by our streaming methods. We also complement our analytical and algorithmic results by presenting an experimental study of the methods over network data streams.

4.1 Introduction

DSMSs approach the task of handling and mining massive data streams by *summarizing* the streams in small space. These summaries may be various "samples" (selection of subsets of items by sampling with or without replacement, weighted sampling, deterministic sampling, etc) or "sketches" (inner product or aggregate of subsets of items using different hash functions that compactly describe the subsets in each inner product). Sampling and sketching solutions have been designed for a number of tasks such as finding heavy hitters, change detection, quantiling, histogramming, etc. (See recent surveys and tutorials [107, 67, 21] etc.) For most of these tasks, a precise answer is not paramount and also impossible to obtain within the limited space and time constraints of DSMSs. Therefore, workable approximations are necessary and indeed suffice in these applications. As a result, samples or sketches have proved to be a suitable fit in DSMSs since they provide accuracy guarantees and have small footprint. Both sampling and sketching are used in Gigascope [44] and CMON[2].

The departure of our work from extant literature emerges from our experience with IP traffic stream analysis: input streams can be "viewed" in different ways, and the summaries built to manage and mine one "view" may differ significantly from those used for another.

4.1.1 Motivating Example: Forward and Inverse Views

We will expose the phenomenon of different "views" of the input data stream using an example drawn from the IP traffic analysis case. Consider the IP traffic on a link as packet p representing (i_p, s_p) pairs where i_p is the source IP address and s_p is the size of the packet (there are other attributes of IP traffic on the link—destination IP addresses, port numbers, payload or content—but for exposition, we focus on these attributes).

[2] `http://www.ipmon.sprint.com`

Problem A. Which IP address sent the most bytes? That is, find i such that $\sum_{p|i_p=i} s_p$ is maximum.

Problem B. What is the most common volume of traffic sent by an IP address? That is, find traffic volume W such that $|\{i|W = \sum_{p|i_p=i} s_p\}|$ is maximum.[3]

Both Problem A and B arise naturally in IP traffic analysis. Problem A is a simplification of the problem of finding the "elephant flows" [62]. Problem B is related to estimating the number of "mice" (small flows) and is a generalization of the problem of estimating the number of flows with small number of packets [54]. "Port scanning" attacks, which probe a large number of ports looking for vulnerabilities by trying to open connections on each port have low volume per flow, but show up as small W's in Problem B.

For Problem A, there are many known solutions using samples [102] or sketches [45], and these solutions have even been tested in live DSMSs on IP traffic [44]. In contrast, we are not aware of any solutions for Problem B with strong guarantees.

4.1.2 Formalizing Different Views

We formalize the problems as follows:

- Problem A deals with the *forward* distribution, that is, we work on $f[0 \ldots U]$ where $f(x)$ is the number of bytes sent by IP address x. Each new packet (i_p, s_p) results in $f[i_p] \leftarrow f[i_p] + s_p$. We ask what is the x for which $f[x]$ is the largest.

- Problem B deals with the *inverse* distribution, that is, we work on $f^{-1}[0 \ldots K]$ where each new packet (i_p, s_p) results in $f^{-1}[f[i_p]] \leftarrow f^{-1}[f[i_p]] - 1$ and $f^{-1}[f[i_p] + s_p] \leftarrow f^{-1}[f[i_p] + s_p] + 1$. We ask which i gives the largest $f^{-1}(i)$.

For conventional DBMSs where the input can be stored, both views f and f^{-1} are equivalent. For example, both can be expressed as nested SQL queries. For Problem A:

```
SELECT srcIP
FROM R
GROUP BY srcIP
HAVING sum(bytes) = max(
```

[3]This can be thought of as determining the popular bandwidth requirement for hosts. In more detail, one may group bandwidth into ranges of volume 1—2KB, 2—3KB, etc. and ask this question on such ranges rather than precise volumes.

```
        SELECT sum(bytes)
        FROM R
        GROUP BY srcIP)
```

and Problem B:

```
SELECT S
FROM (
        SELECT sum(bytes) as S
        FROM R
        GROUP BY srcIP)
GROUP BY S
HAVING count(S) = max(
        SELECT count(K)
        FROM (
                SELECT sum(bytes) as K
                FROM R
                GROUP BY srcIP)
        GROUP BY K)
```

So, if the data is stored, we can derive either. However, in DSMSs where we maintain only a summary of the data, f and f^{-1} can not be readily derived, and operating on one from the input data stream is fundamentally different from operating on the other. Of course, summaries of *both* f and f^{-1} are of interest since they give an idea of traffic size distribution in two, quite different ways. Similarly, mining f and f^{-1} for changes or anomalies will show quite different phenomena. However, much of extant literature has developed methods for summarizing and mining f, but not much is known for summarizing and mining f^{-1}.

Methods that have been successful in mining the forward distribution do not obviously apply to f^{-1}. Consider maintaining the popular AMS [17] sketch on f^{-1} on the data stream. Each new packet modifies f^{-1}; because its AMS sketch is based on *precisely* knowing $f[i_p]$, it is provably impossible to know all i_p's in a small space streaming setting. In other words, each new packet changes the "domain" itself in a way we can not track in small space over the stream. Hence, sketch methods that rely on knowing the precise domain value of each new update such as the AMS sketch and all its variations fail directly.

4.1.3 Our Contributions

Our contribution is to introduce the problems of summarizing and mining the inverse distribution, and proposing solutions in full generality for them. More precisely:

1. We formalize the problems of summarizing and mining the inverse data distribution on data streams. We have given intuition why samples and sketches developed for the forward distribution does not solve the inverse distribution problems. We go on to prove concrete lower bounds that separate the performance of algorithms for problems on forward vs. inverse distribution on data streams, no matter what techniques are used.

2. We present a general summary for the inverse distribution based on *dynamic inverse sampling* and an algorithm to maintain such samples dynamically, in presence of *both* inserts and deletes, with provable guarantee. No such dynamic sampling method was previously known. Using such samples, we present algorithms with provable guarantees for a number of inverse distributions problems including heavy hitters, range queries, quantiles, etc.

3. We complement our analytical and algorithmic results by a thorough implementation study on real data and show that our methods are both practical and effective.

Our approach extends to a variety of scenarios, and smoothly handles continuous distributions, fractional counts, working on the sum or difference of two distributions and so on.

Map. The rest of the chapter is laid out as follows. In Section 4.2, we define the inverse distribution and the mining tasks of interest over it, then prove lower bounds on problems in the inverse distribution. In Section 4.3 we introduce our Dynamic Inverse Sampling method for insertions only streams, and extend it to insert and delete streams in Section 4.4. This is applied to tracking properties of the Inverse Distribution in Section 4.5. Our experimental evaluation is in Section 4.6, with prior work discussed in Section 4.7 and concluding remarks in Section 4.8.

4.2 The Inverse Distribution

Let f be a discrete distribution over a large set X, with the semantics that $f(x \in X) = i$ means that item x occurs i times. Let $N = \sum_{x \in X} f(x)$, the total number of items, and $D = |\{x | f(x) > 0\}|$, the number of distinct items. The inverse distribution is defined as follows:

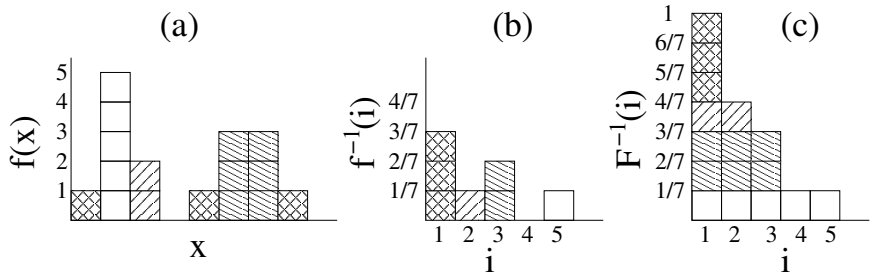


Figure 6: Example distribution, shaded to indicate items with the same count. (a) Forward distribution f where items have counts $\{1, 5, 2, 0, 1, 3, 3, 1, 0\}$ (b) Inverse distribution, f^{-1}. (c) Cumulative inverse distribution, F^{-1}.

Definition 2. *The* inverse distribution, $f^{-1}(i)$ *gives the fraction of items from* X *whose count is* i*. That is,* $f^{-1}(i) = |\{x|x \in X, f(x) = i, i \neq 0\}|/D$.[4]

The cumulative inverse distribution, $F^{-1}(i)$ *is defined as* $\sum_{j \geq i} f^{-1}(j)$.[5]

For clarity and simplicity, we assume that f is a discrete, integer valued distribution, but generalizations to continuous or real valued distributions follow naturally. An example is shown in Figure 6. From this figure, it can be seen that $N = \sum_i i f^{-1}(i) D = \sum_i F^{-1}(i) D$.

4.2.1 Queries on the Inverse Distribution

Queries on the inverse distribution give a variety of information about the distribution itself. We define the following queries on the inverse distribution:

• *Point Queries* on the inverse distribution are, given i, to return $f^{-1}(i)$. This corresponds to finding the fraction of items that occurred exactly i times. For example, finding $f^{-1}(1)$ over a stream of network flows corresponds to finding flows consisting of a single packet — possible indication of a probing attack if $f^{-1}(1)$ is large. This quantity is sometimes known as the *rarity* of the distribution.

• *Range Queries* on the inverse distribution generalize point queries and given a range $[j, k]$ return $\sum_{i=j}^{k} f^{-1}(i) = F^{-1}(j) - F^{-1}(k+1)$. Thus in a database of transactions, one could ask "what percentage of items sold between 10 and 20 units last month" by computing the Inverse Range Query $[10, 20]$ over the appropriate relation.

• *Inverse Heavy Hitters* applies the notion of Heavy Hitters (frequent items) to the inverse

[4]This definition forces $f^{-1}(0) = 0$ so that $\sum_i f^{-1}(i) = 1$.
[5]This definition computes the cumulative distribution of items with counts i or *above*, not i or below, which is equal to $1 - F^{-1}(i)$.

distribution. Given a fraction ϕ, an Inverse Heavy Hitters Query must return $\{i|f^{-1}(i) > \phi\}$. That is, which are the item counts that occur most frequently?

• *Inverse Quantiles* takes a fraction ϕ and returns the ϕ-quantile of the inverse distribution. That is, return the i such that $F^{-1}(i-1) > \phi, F^{-1}(i) \leq \phi$. This allows to pose queries such as, over a stream of connections, what is the median number of connections made by consumers.

4.2.2 Computational Challenge

All the queries we have defined can be answered exactly by taking the original distribution and performing sorting and scanning passes over it. However, we seek solutions that can answer queries on high speed data streams, consisting of an arbitrary mix of insertions and deletions. Deletions arise in many traditional database settings, where records are inserted and deleted; they also occur in the network scenarios we have discussed as flows begin and end. Hence our solutions must consume only small space (much smaller than the number of updates, and also smaller than the size of the domain $|X|$). We analyze the complexity of answering these queries rapidly and using only small space, by allowing approximation and probabilistic methods. In general, several computations over the inverse distribution are strictly harder than their counterparts over the original distribution. We demonstrate this for both exact and approximate query answering:

Lemma 1. Fixed point queries *are point queries where the point is given ahead of the data. Fixed point queries can be answered exactly on the original distribution using constant space (by simply counting the number of times the given item occurs). They require space linear in the number of items, $|X|$ to compute on the inverse distribution. A probabilistic, relative error approximation still requires linear space.*

Proof. We show that answering fixed point queries on the inverse distribution require linear space in general. We focus on the case when the query is to find the number of items that occur two times. We reduce to the problem of testing whether a pair of bit strings are disjoint. Take two bit strings of length $|X|$. For each bitstring, construct a stream of values containing the set of locations where the bitstring is 1, and pass these streams

to the proposed algorithm. We see that there if there is an intersection between the two bit strings then some value must occur twice (once in each stream); however, if no value occurs twice then the bit strings must be disjoint. The communication complexity of the disjointness problem is $\Omega(|X|)$ [96]; a standard reduction shows that this implies an $\Omega(|X|)$ space bound for our streaming algorithm: imagine running the algorithm on the first stream and then sending the memory contents to another player, who then runs the algorithm on the second stream. If the algorithm correctly computes the number of items that occur two times then it solves the disjointness problem, and therefore the size of the communication (and hence the space required) is linear in the length of the bit strings, ie. $\Omega(|X|)$. This holds under probabilistic setting. Even if we allow a relative approximation guarantee, then we must still be able to distinguish between the case where the number of items with count two is zero, and more than zero. This corresponds to the case that the bit strings are disjoint, and the case that they intersect. □

Lemma 2. *The number of distinct values in the original distribution, $F_0(f)$ can be approximated up to a fixed error with constant probability in $O(1)$ space. However, the number of distinct values in the inverse distribution $F_0(f^{-1})$, requires linear space to approximate to a constant factor.*

Proof. We use the same reduction from the disjointness problem to show the hardness of this problem. On our bitstring example, if the two bit strings are disjoint then the number of distinct values in the inverse distribution is one (all items have frequency one); but if they are not disjoint then the number of distinct items is two (we can add a unique item to ensure this). If we can approximate the number of distinct items with relative error less than a third, then we can distinguish the two cases, and hence even a probabilistic approximation of the number of distinct items requires at least linear space. □

We seek good approximations for the queries we have defined over the inverse distribution, with strong guarantees of the quality. To do this, we develop a new technique, *Dynamic Inverse Sampling*, which effectively samples uniformly from the inverse distribution, as the original distribution is modified by insert and delete transactions. We will show how using this sample can give good estimators for the queries over the inverse distribution.

There are two challenges in this approach. First, maintaining random samples in the presence of inserts and deletes in one-pass is quite challenging. All known methods resort to rescanning the past relation for populating the sample when it dwindles under deletes. In order to make our goals feasible, we must disallow the "adversarial" strategy that asks for a sample from the inverse distribution and then deletes the sampled items, and repeats. Clearly, such a strategy can force any sampling method that uses sublinear space to end up with an empty sample. We are able to prove strong guarantees on our dynamic inverse sampling algorithm under the standard assumption in probabilistic algorithms that the randomization (coin tosses) our algorithm uses is not known to the adversary. The adversary may not use the output of queries to affect the stream of updates (equivalently, we assume that the updates are specified in advance). The second challenge is that as we show below, existing techniques of sampling from the original distribution, and sketch summarization, fail to answer our queries; this emphasizes the importance of sampling from the inverse distribution.

Lemma 3. *A uniform sample from the forward distribution based on probing records is insufficient to answer queries on the inverse distribution.*

Proof Sketch. Consider the distribution where one item occurs $N - k$ times, and k items occur once each, for some constant k, e.g. $k = 2$. Unless the sample of items is linear in N, it is unlikely to draw any of the k items which occur once, and so cannot distinguish this distribution from one where one item occurs N times. But in the first distribution, $f^{-1}(1) = 1 - 1/(k + 1)$, whereas in the second it is 0. To correctly distinguish between these two cases, a very large sample is required. □

Lemma 4. *A sketch synopsis of the forward distribution is insufficient to answer queries on the inverse distribution.*

Proof Sketch. Queries to sketch data structures, such as the AMS sketch [17], estimate the count of individual items with additive error related to the L_2 norm of the distribution. To guarantee accurate answers to queries on the inverse distribution, this error must be very small, requiring the sketch to be at least linear in D (number of distinct values). □

4.3 Dynamic Inverse Sampling: Insertions

Our methods to answer queries on the inverse distribution rely on a technique that we call "Dynamic Inverse Sampling" (DIS). The goal of this technique is to process a sequence of insertions and deletions and then be able to draw samples uniformly from the inverse distribution. Each sample is drawn with replacement, and returns a pair uniformly from the set of $\{(i, x)|x \in X, f^{-1}(x) = i\}$. The size of this set is D, the number of distinct items in X, and so the probability of returning any pair is $\frac{1}{D}$.

In order to simplify the exposition, we introduce our dynamic inverse sampling method when the input consists of insertions only. This shows the main structure of the algorithm. In subsequent sections, we will show how to generalize this to our main case of interest, where the input can consist of an arbitrary sequence of insertions and deletions.

4.3.1 Data Structure and Update Procedure

We first describe the main structure, which draws a pair (i, x) from the inverse distribution. We later analyze how many independent copies of this data structure are required to guarantee a sample of sufficiently large size. At a high level, the procedure works by hashing the items to levels such that the likelihood of being hashed to level l is exponentially decreasing in l. So at some level $l \approx \log D$ there is a high probability that only one item hashes there, and we recover this item and its count as the sampled count. In order to prove correctness, we will have to show that this item is selected uniformly, and that there is at least constant probability that there is a level that has a unique item for us to return. Throughout, we assume that $X = [0 \dots m-1]$ for some m such that any $x \in X$ is represented in a single machine word; our approach naturally generalizes to other settings but we focus on this case for simplicity.

Data Structure. Our data structure takes two parameters: (1) a ratio $0 < r < 1$ which is used to partition the input items (2) M, the range of the hash function used to determine where items are stored within the data structure. We fix values for these parameters based on our analysis. The size of the data structure is proportional to $L = \log_{1/r} M$. We keep three arrays of length L: $item$, which stores items from the input; $count$, which stores

item counts; and boolean flags $uniq$. We initialize the array of counts to zero. We keep a hash function h which maps from $[1 \ldots m]$ to $[1 \ldots M]$. For the purposes of the analysis, we require h to be (strongly) universal. Such hash functions are very fast to compute and require only a constant amount of space [34].

Update process. For each insertion of an item x, we use the hash function h to determine where in the data structure it belongs. From h, we define

$$h_l(x) = \lfloor h(x)/(r^l * M) \rfloor$$
$$l(x) = l \iff h_l(x) = 0, h_{l+1}(x) \neq 0.$$

The value $l(x)$ determines the place where x is stored in the data structure (it is the greatest l such that $h_l(x) = 0$). Observe that $l(x)$ can be computed in constant time by solving $h_l(x) = l$, which sets $l(x) = \lceil \log_{1/r}(M/h(x)) \rceil$. We inspect $count[l(x)]$: if it is zero, then no item is stored there, and so we set $item[l(x)] = x$, and set $uniq[l(x)] =$ `true`. If $count[l(x)]$ is not zero, we inspect $item[l(x)]$. If $item[l(x)] \neq x$, then we have a *collision*, and we set $uniq[l(x)] =$ `false`. Lastly, in all cases we increment $count[l(x)]$.

Output process. In order to output an item from the data structure, we search the data structure. We describe two variations, one with guaranteed bounds, and a second, "greedy" approach that extracts as many samples as possible from the data structure. Begin by setting $l = L$. If $count[l]$ is not zero, then we inspect $uniq[l]$: if it is `true` then we output the pair $(count[l], item[l])$. If $uniq[l]$ is `false`, then we do not output an item, since we do not have an accurate count for the item. Else, $count[l]$ is zero, so we decrement l and repeat the process. In the basic output routine, we halt as soon as we find a level where $count[l] > 0$; in the "greedy" version, we process every level. The output routine scans the whole data structure, so the time to run the output process is $O(L)$.

Observe that one outcome is that no item is output from the data structure. In our analysis, we will show that for appropriate settings of the parameters r and M, the probability of this outcome is most a constant, $p < 1$. So by sufficiently many repetitions of this data structure with different hash functions h, we can guarantee high probability of returning a sample of the required size.

Example. We consider the following example sequence of insertions of items:

Time Step	Level 1			Level 2			Level 3		
	item	count	uniq	item	count	uniq	item	count	uniq
1.	4	1	T	0	0	T	0	0	T
2.	4	1	T	7	1	T	0	0	T
3.	4	2	T	7	1	T	0	0	T
4.	4	3	F	7	1	T	0	0	T
5.	4	3	F	7	2	F	0	0	T
6.	4	4	F	7	2	F	0	0	T
7.	4	4	F	7	2	F	2	1	T
8.	4	5	F	7	2	F	2	1	T
9.	4	6	F	7	2	F	2	1	T
10.	4	6	F	7	2	F	2	2	T

Figure 7: Example of state of data structure at each time step on sample input

Input: `4, 7, 4, 1, 3, 4, 2, 6, 4, 2`

Suppose these hash to levels in an instance of our data structure as follows:

x	1	2	3	4	5	6	7	8
$l(x)$	1	3	2	1	1	1	2	1

Figure 7 shows the state of the data structure after each update. For each level we indicate whether there is a unique item at that level that can be recovered as the sampled value. Observe that at timesteps 5 and 6, no such item can be found, but at all other times we can recover a sampled item: at time 1 we return (1,4); between time 2 and 4 we would return (2,7), from time 7 to time 9 we would return item (1,2) and lastly at time 10 we return (2,2). The greedy output routine would also return item 4 at times 2 and 3.

4.3.2 Analysis

We show that the Dynamic Inverse Sampling returns uniform samples from the inverse distribution. First, we show that provided a unique item is found at some level then it is drawn uniformly from the set of items with non-zero counts. The main technical result is given in Lemma 6, which shows that there is at least a constant probability that such an item exists after our hashing procedure. Lastly, we show that repeating this procedure several times over will draw a sample (with replacement) of the desired size.

Lemma 5. *If a pair (i, x) is returned from the output procedure, x is selected uniformly from the inverse distribution and $f(x) = i$.*

Proof. Firstly, we observe that if we return a pair (i, x), then indeed $f(x) = i$, since we have counted the number of occurrences of x exactly. To show that x is drawn uniformly, we rely on the universal properties of the hash functions. The strong universality property of $h(x)$ means

$$\mathsf{Pr}[h(x) = a \wedge h(y) = b] = \tfrac{1}{M^2}$$

Applying this to $h_l(x)$ gives:

$\mathsf{Pr}[h_l(x) = a \wedge h_l(y) = b] = \mathsf{Pr}[\lfloor \frac{h(x)}{r^l M} \rfloor = a \wedge \lfloor \frac{h(y)}{r^l M} \rfloor = b] = \frac{r^l M * r^l M}{M^2} = r^{2l}$

Thus, $h_l(x)$ is also strongly universal over r^{-l}. Hence (over choices of h), $\mathsf{Pr}[h_l(x) = 0] = r^l$, and this is independent of x. □

Lemma 6. *Over random choices of h, there is constant probability of the output process returning a pair (i, x).*

Proof. Let D denote the number of distinct items at output time, and let $B_l = 1/r^l$. The function h_l maps onto values $0 \ldots B_l - 1$. From the previous lemma, h_l is 2-universal onto this set. Let X_l denote the number of distinct items observed that satisfy $h_l(x) = 0$. $\mathsf{E}(X_l) = D/B_l$, and $\mathsf{Var}(X_l) \leq \mathsf{E}(X_l)$, using the pairwise-independence of h_l.

Consider the level l such that $\alpha/r \leq D/B_l \leq \alpha/r^2$ for an appropriate scaling constant $\alpha > r$. We analyze the number of items that satisfy $h_l = 0$, and show that there is constant probability that this is small. By the Chebyshev inequality,

$$\mathsf{Pr}[|\mathsf{E}(X_l) - X_l| \geq \mathsf{E}(X_l)] \leq \mathsf{Var}(X_l)/\mathsf{E}(X_l)^2 \leq 1/\mathsf{E}(X_l) = B_l/D \leq r/\alpha.$$

We use this expression to analyze the probability that X_l is either 1 or 2. The event $|\mathsf{E}(X_l) - X_l| \geq \mathsf{E}(X_l)$ occurs only if $X_l \leq 0$ or if $X_l \geq 2\mathsf{E}(X_l)$. We set $E(X_l) = 3/2$, which fixes $r = \sqrt{2\alpha/3}$. Because $2\mathsf{E}(X_l) \leq 3$, and X_l takes on only integer values, $|\mathsf{E}(X_l) - X_l| < \mathsf{E}(X_l) \Rightarrow X_l \in \{1, 2\}$. Hence, $\mathsf{Pr}[X_l \notin \{1, 2\}] \leq \frac{r}{\alpha} = \sqrt{2/3\alpha}$. This is a constant provided $2/3 < \alpha < 3/2$ (since both this probability and r must be less than 1).

If $X_l = 1$, then there is one item, x, stored at level l or above, and we can easily identify this item and its count. However, if $X_l = 2$, it is possible that both items (say, x and y), are stored at the same level, and we are unable to find the identity of either of them. Assuming

$X_l = 2$, we bound the probability that both x and y are stored at the same level. Using the universality of h_l again,

$$\begin{aligned}
&\mathsf{Pr}[l(x) \geq l + a | l(x) \geq l] = r^a] \Rightarrow \\
&\mathsf{Pr}[l(x) = l + a | l(x) \geq l] = r^a - r^{a+1} = r^a(1-r) \Rightarrow \\
&\mathsf{Pr}[l(x) = l(y) | l(x), l(y) \geq l] = \sum_{a=0}^{l} (r^a(1-r))^2 + \frac{1}{r^{2L}},
\end{aligned}$$

since $\mathsf{Pr}[h(x) = h(y) = 0] = \frac{1}{M^2} = (r^{-l})^2$. Then:

$$(1-r)^2 \sum_{a=0}^{l} (r^2)^a + \frac{1}{r^{2L}} \leq \frac{(1-r)^2}{1-r^2} = \frac{1-r}{1+r}$$

This relies on the fact that the r^{2L} term is dominated by the residue of the infinite sum, which is true if M is chosen sufficiently large. This is achieved provided $M = \Omega(m)$, so we set $M = 2m$.

Using the Markov inequality, $\mathsf{Pr}[X_l \geq 2] \leq \frac{\mathsf{E}(X_l)}{2} \leq \frac{r}{2\alpha}$. So $\mathsf{Pr}[X_l = 2 \wedge l(x) = l(y)] \leq \frac{r(1-r)}{2\alpha(1+r)}$, using (4.3.2).

The probability, p, that the output process does not return a pair (if there are not one or two items at level l or below, or the two items are mapped to the same level) is

$$\begin{aligned}
p &= \mathsf{Pr}[(|X_l - \mathsf{E}(X_l)| > \mathsf{E}(X_l)) \vee (X_l = 2 \wedge l(x) = l(y))] \\
&= \tfrac{r}{\alpha} + \tfrac{r(1-r)}{2\alpha(1+r)} = \tfrac{r}{2\alpha} \tfrac{2(1+r)+(1-r)}{1+r} = \tfrac{1}{3r} \tfrac{3+r}{1+r}
\end{aligned}$$

Which follows since we have set $\alpha = 3r^2/2$. Our constraints on r and α are that r and all probabilities should be strictly less than 1. For concreteness, we set $\alpha = 1$, and find $p = \frac{3\sqrt{3}+\sqrt{2}}{2\sqrt{3}+3\sqrt{2}} = 0.8577\ldots$ Thus there is constant probability that the output function will return a pair. □

Having set $r = \sqrt{2/3}$ and $M = 2m$, the size of the data structure is therefore $O(\log_{1/r} M) = O(\log m)$. This gives constant probability at least $1 - p$ of extracting a sampled item from the data structure. By keeping $\log(1/\delta)/\log(1/p)$ independent copies of the data structure the failure probability is reduced to arbitrarily small δ. If we require a sample of size k and we keep $k/(1-p)$ copies of the data structure, we recover k items in expectation. In general we need a stronger guarantee on the number of items returned.

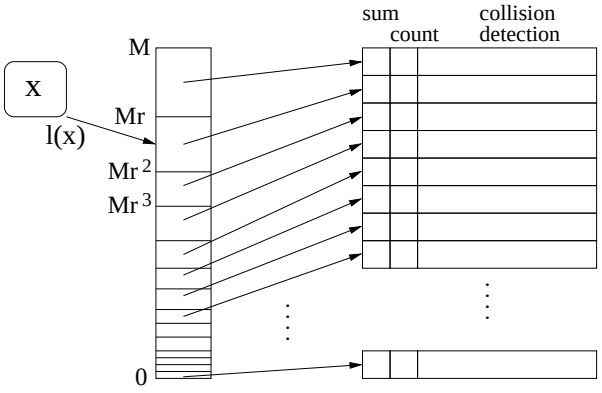


Figure 8: Dynamic Inverse Sampling Data Structure: hash function l maps item x to a level where $count$, sum and collision detection information are updated.

For small k, we can just keep $k \log(k/\delta)/\log(1/p)$ copies of the data structure: each group of $\log(k/\delta)/\log(1/p)$ guarantees probability of $1 - \delta/k$ of returning a sample, so overall, there is probability of $1 - \delta$ of getting k samples. Asymptotically, the cost is $O(k \log k)$ copies. For larger k we can give tighter guarantees, using Chernoff bounds:

Lemma 7. *Let $\epsilon = \sqrt{\frac{2 \log 1/\delta}{k}}$. If $k \geq 8 \log 1/\delta$ and we keep $K = (1 + 2\epsilon)k/(1 - p)$ copies of the data structure, then with probability at least $1 - \delta$ we are able to recover at least k samples.*

Based on the above results, our main theorem follows.

Theorem 1. *We can maintain $O(k)$ independent copies of DIS in $O(k \log m)$ space, and guarantee with high probability to return a uniform sample of size k from the inverse distribution. Each insertion operation takes time $O(k)$; extracting the sample takes time $O(k \log m)$.*

4.4 Dynamic Inverse Sampling: Deletions

In generalizing the data structure to handle deletions, we will perform updates so each deletion precisely counteracts the effect of a previous insertion of the same item, leaving the data structure as if both the deletion and corresponding insertion had never happened. To do this, we make both insertion and deletion *linear* operations on the data structure, which do not inspect the contents of the data structure but rather have the effect of *adding on* or *subtracting off* quantities to various counters, independent of their current values. The

Procedure update(x,tt)
Input: Item x, tt=insert/delete
1. $h = h(x)$;
2. **if** (tt= insert) **then**
3. $a = +1$
4. **else** $a = -1$;
5. $l = \lceil \log(M/h)/\log(1/r) \rceil$;
6. $sum[l] = sum[l] + x * a$;
7. $count[l] = count[l] + a$;
8. collision-update(x, a);

Procedure query(gr)
Input: gr flag for greedy output
Output: Samples from f^{-1}
1. **for** $l = L$ **downto** 0 **do**
2. **if** $count[l] > 0$ **then**
3. $x = sum[l]/count[l]$;
4. **if** (($\lfloor x \rfloor = x$) **and**
5. (collision-test()) **then**
6. **output** $(count[l], x)$;
7. **if** (!gr) **then break**;

Figure 9: Pseudo-code for the dynamic inverse sampling

correctness of this approach then follows immediately from the commutativity of addition and subtraction.

We keep the basic format of the data structure, but make some modifications to how we treat it. Firstly, we replace the $item$ array with an array sum initialized to zero, which will store the sum of item identifiers (which we treat as integers). We also replace $uniq$ with a very small (few bytes in size) "collision detection" data structure, which we will discuss in the next section. The collision detection data structure maintains a distribution of items (which is a subset of the original distribution) under insertions and deletions, and can be queried to find whether there is one distinct item in the distribution or more than one.

Update process. For each insertion of an item x, we compute $l(x)$ as before. We increment $count[l(x)]$, and set $sum[l(x)] \leftarrow sum[l(x)] + x$. We update the collision detection structure with x. For a deletion, we decrement $count[l(x)]$ and set $sum[l(x)] \leftarrow sum[l(x)] - x$, and delete a copy of x from the collision detection structure. Observe that a deletion of x precisely cancels out the effect of a prior insertion of x.

Output process. In order to output an item from the data structure, we search the data structure in a similar way to before, by searching levels l from L down to 0. If $count[l]$ is not zero, then we try to extract an item from the sample. Suppose that x is the only item that is stored at this level in the data structure. Then x can be recovered as $sum[l]/count[l]$. However, we need to be sure that x is the only item stored at this level. So we make use of the collision detection data structure to tell us (either deterministically or with some probability of error) whether there is only one distinct item stored here, or more than one.

The structure of our data structure is shown in Figure 8, and pseudo-code for insert and

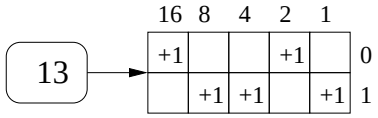


Figure 10: Deterministic collision detection: to insert 13 (represented as a $b = 5$ bit integer) we write $13_2 = 01101$, and so increment $c[1,1], c[2,0], c[3,1], c[4,1], c[5,0]$, corresponding to the 1, 2, 4, 8 and 16 bits.

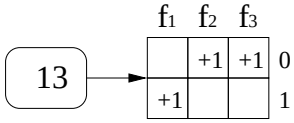


Figure 11: Probabilistic collision detection: to insert 13, with $t = 3$ hash functions, compute $g_1(13) = 1$, $g_2(13) = 0$, $g_3(13) = 0$ and so increment $c[1,1]$, $c[2,0]$, $c[3,0]$.

query operations is given in Figure 9. The cost per update is now dominated by the cost of updating the collision detection mechanism, since the rest of the update can be completed in constant time.

4.4.1 Collision Detection

We require a data structure that can be updated in the presence of insertions and deletions of items so that at query time, we can distinguish between the following two events for a given level: (a) a single item occurs at that level one or more times; or (b) there are a mixture of items at that level. One check we can make is to see that $count[l]$ divides $sum[l]$ exactly: if not, then case (b) must hold. But this is not sufficient: if we observe $sum[l] = 20$ and $count[l] = 2$, the input can be any pair of items that sum to 20, not necessarily two copies of item `10`. To avoid outputting items that did not occur in the input we define three approaches, which trade off speed, space and accuracy.

Deterministic. Suppose $|X| = m = 2^b$ so each $x \in X$ is represented as a b bit integer. We can keep $2b$ counters $c[j,k]$ indexed by $j = 1 \ldots b$ and $k \in \{0,1\}$. Every time we see an insertion of x, we increment the counts one count for each value of j: we add one to $c[j, bit(j,x)]$, where $bit(j,x)$ returns the jth bit of the binary representation of x. Symmetrically, for a deletion of x, we decrement the corresponding counts. At output, we can tell whether there is exactly one item or more than one item stored: if and only if there is one item in the bucket, then for all j exactly one of $c[j,0]$ and $c[j,1]$ is non-zero. The space required is $O(b)$ counters, and the time to process each update is also $O(b)$. An

Procedure deterministic-update(x, val)
Input: Item x, val=+1/-1 for insert/delete
1. **for** $j = 1$ **to** b **do**
2. $\quad bit = x\&1$
3. $\quad c[j, bit] = c[j, bit] + val$;
4. $\quad x = x \gg 1$;

Procedure probabilistic-update(x, val)
Input: Item x, val =+1/-1 for insert/delete
1. **for** $j = 1$ **to** t **do**
2. $\quad bit = g_j(x)$;
3. $\quad c[j, bit] = c[j, bit] + val$;

Procedure heuristic-update(x, val)
Input: Item x, val=+1/-1 for insert/delete
1. **for** $j = 1$ **to** q **do**
2. $\quad sumg[j] = sumg[j] + val * g_j(x)$;

Procedure deterministic-collision-test()
Output: $true$ if no collision else $false$
1. **for** $j = 1$ **to** b **do**
2. $\quad$ **if** $c[j, 0] \neq 0$ **and** $c[j, 1] \neq 0$ **then**
3. $\quad\quad$ **return** $false$;
4. **return** $true$;

Procedure probabilistic-collision-test()
Output: $true$ if no collision, else $false$
1. **for** $j = 1$ **to** t **do**
2. $\quad$ **if** $c[j, 0] \neq 0$ **and** $c[j, 1] \neq 0$ **then**
3. $\quad\quad$ **return** $false$;
4. **return** $true$;

Procedure heuristic-collision-test()
Output: $true$ if no collision else $false$
1. **for** $j = 1$ **to** q **do**
2. $\quad$ **if** $g_j(sum/count) * count$
$\quad\quad \neq sumg[j]$ **then**
3. $\quad\quad$ **return** $false$;
4. **return** $true$;

Figure 12: Pseudo-code for the different collision detection mechanisms.

example update is shown in Figure 10.

Probabilistic. The deterministic approach requires a lot of space for large values of b. We can trade a small probability of error for reduced space. A natural first approach is to use an approximate counter capable of processing insertions and deletions [63]. Such algorithms guarantee $1 \pm \epsilon$-factor approximation to the number of distinct elements with probability at least $1 - \delta$ using space $O(\frac{1}{\epsilon^2} \log m \log 1/\delta)$. Setting $\epsilon < 1/3$, the algorithm can distinguish between 1 item and 2 or more items, in space $O(\log m \log 1/\delta)$. But this space cost is still large.

Instead, a similar method to the deterministic approach uses hashing to give a probabilistic test for collisions. We draw t hash functions, $g_1 \ldots g_t$ which map items uniformly onto $\{0, 1\}$, and use a set of $t \times 2$ counters $c[j, k]$. For every insertion, we increment $c[j, g_j(x)]$, and decrement the same counter for a deletion. We apply the same test as in the deterministic case. If for any j, $c[j, 0] \neq 0$ and $c[j, 1] \neq 0$, then there is more than one distinct item in the bucket. The probability of wrongly declaring a single distinct item in the bucket is at most 2^{-t}. The space used is $O(t)$ counters, and it takes $O(t)$ time per update. An example

update is shown in Figure 11.

Heuristic. The previous method may still consume too much space. A simple heuristic gives faster updates and few errors in practice (we make no formal claims about the error probability here). We compute q new hash functions $g_j[x]$ mapping items x onto $0 \ldots m$ and track the summation of $g(x)$ as $sumg[j, l(x)]$. For every insertion of an item, we add $g(x)$ to $sumg[j, l(x)]$, and for every deletion, we subtract $g(x)$ from $sumg[j, l(x)]$. At query time, we extract x from the bucket as $sum[l]/count[l]$. If x is the only distinct item in the bucket, then $sumg[j, l] = g_j(x) * count[l]$ for all j. We can check this condition and reject if it is not satisfied by any hash. The space required for the heuristic collision detection mechanism is $O(q)$ counters per level, and $O(q)$ time per update.

In all three cases, the collision detection data structures are updated by summing positive and negative values, without examining the contents of the counters. Therefore arbitrary combinations of insertions and deletions can be handled by them. Pseudo-code for the three different collision detection methods is shown in Figure 12. The analysis of Lemma 6 can be applied again, leading to:

Theorem 2. *Using $O(k \log m)$ space, we can maintain $O(k)$ dynamic inverse sampling data structures to process a sequence of insertions and deletions and that guarantee with high probability to return a uniform sample from the inverse distribution of size k. Each update operation takes time $O(k)$; extracting the sample takes time $O(k \log m)$.*

4.4.2 Extensions

We have discussed insertion and deletion of single items. We now observe other ways in which our data structures can be manipulated:

Sliding Window. In many settings, we only want to draw a sample from a recent history of an (insertions-only) stream. The sliding window model [53] specifies that only the most recent W updates (or updates that occurred within W time units) should be considered, for some fixed value of W. When W is too large to buffer the most recent W updates, we can apply a variation of our technique. For each update x, we overwrite the current item stored at level $l(x)$. We can modify the deterministic and probabilistic collision detection

mechanisms so that instead of incrementing a counter, we overwrite the current contents with the *timestamp* of the new item x. At output time, we check the collision detection mechanism to see if there has been any collision within the last W time units: if there is one unique item, then for each pair of counters, exactly one will store a timestamp from within the last W time units. Hence, we can use this modified version of the data structure to draw an item x uniformly from the inverse distribution over the sliding window. However, this does not give us a value for i; and to give the exact value of i is impossible without using $\Omega(W)$ space. Instead, we can use the counting techniques of [53] to approximate the value of i for x, which gives a doubly-approximate answer to queries on the inverse distribution.

Multiple insertions or deletions. We have considered the case where a single item arrives or departs at a time. We can easily generalize this to handle arbitrarily many copies of a single item by appropriate scaling of the counts that we add or subtract.

Fractional and negative item counts. Our analysis does not require the counts of items to be positive integers, hence we can allow counts to become negative and fractional. The interpretation of the sample values returned is that these are selected uniformly from the set of items whose count is non-zero.

Unioning and Differencing of summaries. We can combine two summaries that were created with the same parameters and hash functions by summing the values in their corresponding counters. The result is exactly identical to the result if all updates had been processed by a single summary structure. Hence, the algorithm can easily be carried out in a distributed fashion over a variety of streams, and then the summaries merged to allow investigation of the inverse distribution of the union of all the streams. Similarly, we can compute the difference of two summaries by subtracting corresponding counters; scale all counts by a scalar value; and so on.

4.5 Inverse Distribution Queries

We now show how to use a sample drawn by the Dynamic Inverse Sampling algorithm to answer queries on the inverse distribution.

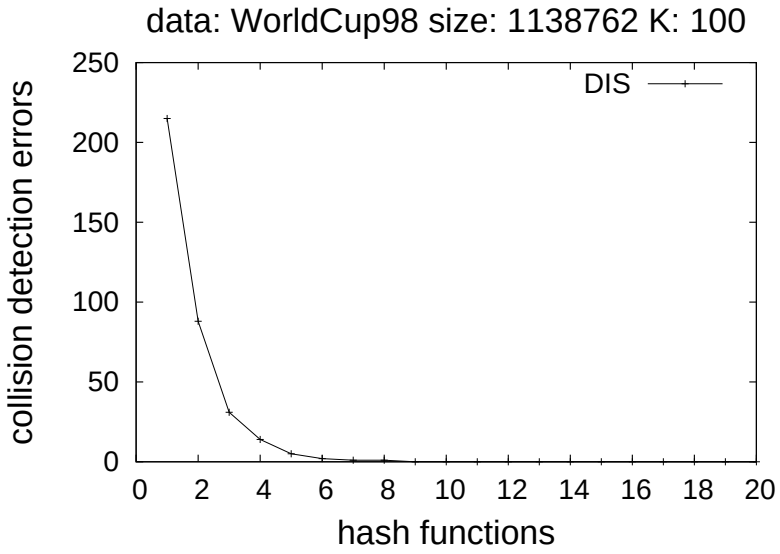


Figure 13: Evaluating number of hash functions required for the probabilistic collision detection.

Theorem 3. *Given a sample from the inverse distribution of size* $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$*, we can answer Inverse Point Queries with additive error less than* ϵ *with probability at least* $1-\delta$*.*

Proof. Let S be the sample drawn from by Dynamic Inverse Sampling, which is a multiset of pairs. We approximate $f^{-1}(i)$ with $\frac{|\{(i,x)\in S\}|}{|S|}$. To analyze this estimator, we set up an indicator variable for each sample in S. Let $Y_j = 1$ if the jth sample in S is a pair (i,x), and $Y_j = 0$ if the jth sample is a pair (i',x') for $i' \neq i$. Since each sample is drawn uniformly, $\mathsf{Pr}[Y_j = 1] = \{x|f(x) = i\}/D = f^{-1}(i)$. So the estimate is correct in expectation. By applying the Hoeffding inequality to $\sum_j Y_j/|S|$, we get $\mathsf{Pr}[|\sum_j Y_j/|S| - f^{-1}(i)| \leq \epsilon] \geq 1-\delta$, as required. □

Corollary 1. *Given a sample from the inverse distribution of size* $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$*, we can answer Inverse Range Queries and queries to the cumulative inverse distribution with additive error less than* ϵ *with probability at least* $1-\delta$*.*

Proof. For an inverse range query $[j,k]$, our estimator is $\frac{|\{(i,x)\in S, j\leq i\leq k\}|}{|S|}$. A similar proof to the above shows that this estimator is correct in expectation, and within ϵ with probability at least $1-\delta$. Queries to the cumulative inverse distribution can be reduced to open-ended inverse range queries $[i,\infty]$, and so the same bounds apply. □

Corollary 2. *Given a sample from the inverse distribution of size* $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$*, Inverse heavy hitters can be answered with additive error* ϵ *with probability at least* $1-\delta$*.*

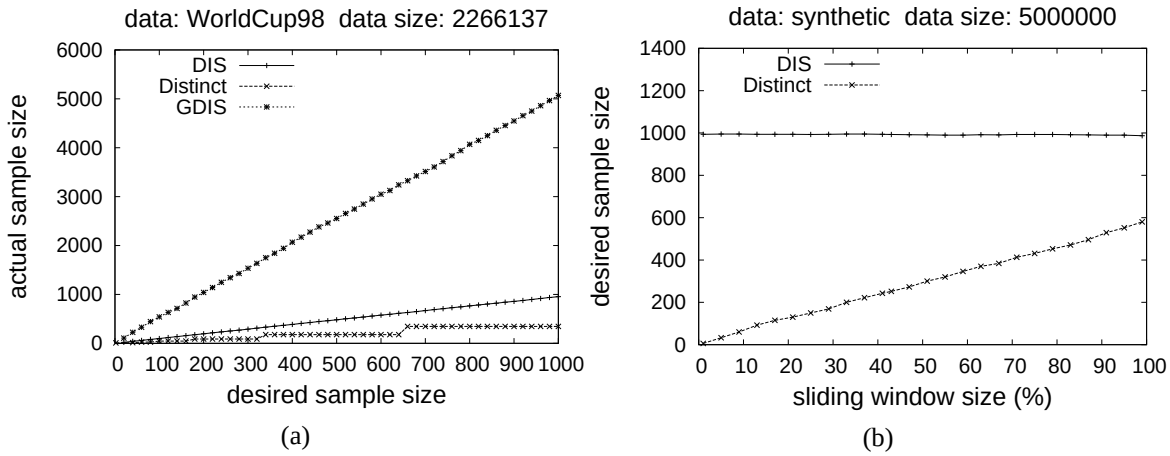


Figure 14: Returned Sample Size: (a) Number of samples returned by the different inverse sampling methods as a function of desired sample size. (b) Number of samples returned by the different inverse sampling methods as a function of deletion frequency.

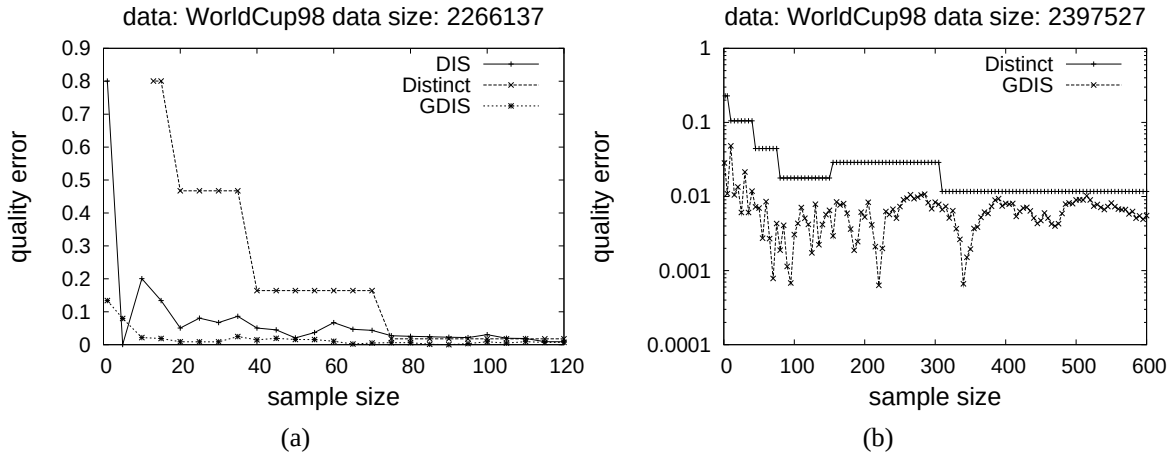
Proof. In order to answer inverse heavy hitter queries, we compute our estimate of $f^{-1}(i)$ for each i that is in the sample, and output those for which $\frac{|\{(i,x)\in S\}|}{|S|} \geq \phi$. By the above theorem, for each i that is output, there is ϵ error in the estimate with probability $1-\delta$, and so we guarantee (with this probability) that every i that is output satisfies $f^{-1}(i) > \phi - \epsilon$. Similarly, since every i that does not appear in the sample is approximated by $f^{-1}(i) = 0$, we conclude that with the same probability, every item with $f^{-1}(i) > \phi + \epsilon$ is output. □

Corollary 3. *Given a sample from the inverse distribution of size $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$, Inverse quantiles queries can be answered with additive error ϵ with probability at least $1-\delta$.*

Proof. For Inverse Quantile Queries, we compute the estimate of $F^{-1}(i)$ for all i in the sample. Observe that this estimate gives a decreasing function as i increases. We output the (unique) i such that the estimate of $F^{-1}(i-1) > \phi$ and $F^{-1}(i) \leq \phi$. By the guarantees on cumulative inverse distribution queries, we have (with probability $1-\delta$) the i that is output has $\phi - \epsilon \leq F^{-1}(i) \leq \phi + \epsilon$. □

4.6 Experimental Study

We implemented our Dynamic Inverse Sampling algorithm, and evaluated it on large sets of network data drawn from HTTP log files from the 1998 World Cup Web Site (stored in the Internet Traffic Archive [8]), as well as on a large synthetic data set of randomly generated distinct values. Each log file consists of several million log records. We used



(a) (b)

Figure 15: Sample quality (range query): (a) Accuracy of sampling methods on inverse range query (linear scale). (b) Accuracy of sampling methods on inverse range query (logarithmic scale).

the size attribute (number of bytes in the response) of the records and the client ID attribute as our target attributes in the data set. The size attribute takes on a wide range of values (from zero bytes to several megabytes), while the range of values for the client ID attribute is more limited with each value of the range occurring more frequently in the data set. Our synthetic data set contains 5 million randomly generated distinct items. To give a data set with a large number of deletions, we built a dynamic transaction set by inserting all the records and then deleting a fraction of these. Since one cannot predict which records will survive the deletions, it gives a challenging test for our methods.

For comparison, we implemented the Distinct Sampling method [70, 69] augmented to handle deletions since this can be used to draw a sample from the inverse distribution under insertions only streams (see the discussion in Section 4.7). The algorithms were implemented in C and were run on a 2.4GHz processor desktop computer.

Collision Detection Experiments. We compared the different collision detection mechanisms for the Dynamic Inverse Sampling (DIS). We ran the algorithm over a data set consisting of insertions only, and counted the number of times that the approximate methods reported no collision (at any level in the data structure), when the deterministic method (correctly) indicated that there was a collision.

We tested the probabilistic collision detection mechanism by gradually increasing the number of hash functions from 1 to 40. The results are shown in Figure 13; it can be observed that the total number of errors over all levels in all data structures drops to 0 when

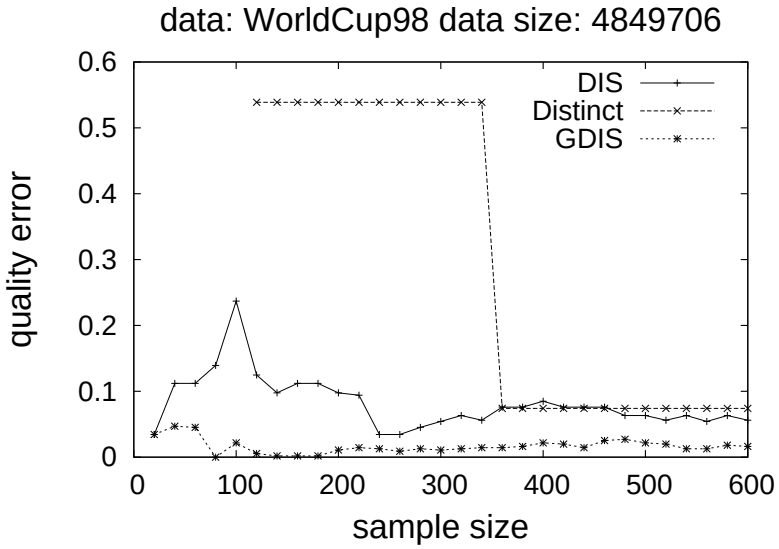


Figure 16: Accuracy of sampling methods on inverse quantiles.

Collision Detection	**Hash Functions**	**Space Factor**	**Time Cost**
None	—	—	96s
Deterministic	—	32	132s
Heuristic	$q = 1$	1	119s
Heuristic	$q = 2$	2	140s
Heuristic	$q = 3$	3	162s
Probabilistic	$t = 5$	5	165s
Probabilistic	$t = 10$	10	225s

Table 1: Timing results and space/time tradeoff for different collision detection methods. 'Space factor' denotes relative space cost of each method.

we use 9 or more hash functions. The heuristic collision detection mechanism was run with the number of hash functions ranging from 1 to 5. With one hash function, there were 3 collision detection errors on a dataset of 1.3 million records. There were no collision detection errors with two or more hash functions.

We compared the time cost of all three methods to process a total of 260 million updates to the data structures. Timing results are showing in Table 1. They show that our method is capable of processing several million updates per second (for comparison, our implementation of Distinct sampling was faster still, processing 9 million items/second). We see that the Deterministic method is quite fast, since it requires no additional hash function computation. But it still requires space for $\log m$ counters. With two hash functions, an undetected collision under the heuristic method is very unlikely, and this requires only two additional counters per level, plus two hash functions per copy of the DIS structure. This

gives a good trade-off of time against space used. For the remainder of our experiments, we worked with the deterministic method only, knowing that for suitable settings of q and t we would get identical results using the heuristic or probabilistic collision detection methods.

Returned Sample Size. We compared the size of sample returned by the different methods over the datasets we used in our experiments. We ran our experiments on the client ID attribute of the HTTP log data. Each network dataset generated a sequence of insertion and deletion transactions, with over 3 million operations in total for each dataset. We measured the actual sample size returned by the algorithms after processing all the insertions and deletions, when 50% of the inserted records were deleted. The results for other network datasets were similar; we show a representative plot in Figure 14 (a). For the desired sample size of 100, the distinct sampling ("Distinct") technique returned a sample of about 45% of the desired size. When the desired sample size was increased to 1000, the size of the sample was only 30% of the desired size. These results support our claim that this approach has difficulty with handling a large number of delete operations.

The Dynamic Inverse Sampling algorithm (DIS) returned a sample of almost 100% of the desired size for all sample sizes (for instance, for $k = 1000$ it returned 998 samples when there were 1% deletions, 981 samples at 10%, 970 for 20% and 955 for 50%) which indicates that in practice the probability of obtaining at least one sampled record from each dynamic inverse sampling structure is close to 1. Using the greedy output routine (GDIS) which extracts all possible sample records from every dynamic structure, returned approximately five items from each data structure. Both variations of the Dynamic Inverse Sampling method are not affected by the order and amount of insert and delete operations.

Next we investigate the dependency between the size of the sample returned by the methods and the fraction of deletions in the data set. We ran our experiment on the synthetic data set of distinct items, when the desired sample size is 1000. The results are shown in Figure 16. For a data set with a large number of deletions, the distinct sampling technique performs poorly. When 80% of the inserted records were deleted from the sampling structure, the sample size was about 12% of the desired sample size. As the number of deletions approaches the number of insertions the sample size returned by the distinct

sampling algorithm decreases linearly. When the number of deleted records was increased to 99% of the number of insertions, the resulting size of the sample was less than 1% of the desired sample size. The Dynamic Inverse Sampling algorithm was stable under any number of deletions and returned a sample (with replacement) of size almost 100% of the desired size.

Sample Quality. Lastly, we measured how well the obtained sample represented the sampled dataset. To calculate this estimate, we posed a series of inverse range queries $F^{-1}(i)$ on the samples (to compute the fraction of records with size greater than i), and compared it to the exact value of this query computed offline. Figure 15 shows experiments on two different network datasets for $i = 1000$, the first on a linear scale and the second on a log scale. In Figure 15 (a), we see that both the regular and the greedy output procedure give very low error for small sample sizes — in particular, the greedy procedure achieves close to zero error for as sample size as small as 15. This shows that this output function seems to do very well in practice. In contrast, for very small sample sizes, Distinct sampling is unable to return any sample at all. In Figure 15 (b), we see that GDIS consistently outperforms Distinct sampling, by up to an order of magnitude, making it the method of choice.

Another set of experiments was performed on the network data set with over 4 million records by posing a series of inverse quantile queries on the samples using the client ID attribute of the records. In particular, we estimated the median (to find i that $F^{-1}(i-1) > 0.5, F^{-1}(i) \leq 0.5$) of the inverse distribution using the resulting sample, and measured how far the true position of the returned item i was from 0.5. Figure 16 shows the results of the experiment ("quality error" is computed as $2|F^{-1}(i) - 0.5|$). We can see that for small desired sample sizes (under 100), the distinct sampling algorithm does not have a large enough sample to give any results. The algorithm's error of median estimation becomes sufficiently small only when the desired sample size is about 350 or higher. In contrast, both versions of the dynamic sampling algorithm are much more accurate in their estimation of the median value even for small sample sizes.

4.7 Related Work

The research community has developed a rich literature on applications of random sampling algorithms in databases and data streams. One of the most common and well studied applications of sampling in large data warehouse environments is to provide fast approximate answers to complex aggregation queries based on statistical summaries which are created and maintained using various sampling techniques [110, 68, 71]. Random sampling is a standard technique for constructing approximate summary statistics, such as histograms, for query optimization and query planning purposes [71, 39]. Random sampling is widely used for distinct-values estimators [70, 69, 37] which play an important part in network monitoring and online aggregation systems.

In today's database systems random sampling is routinely used for a variety of purposes. Microsoft SQL Server 2000 uses sampling to build and maintain histograms which provide various statistics for the query optimizer to choose the most efficient plan for retrieving and processing data. Statistics are maintained by re-sampling column values whenever substantial update activity has occurred[6]. The Oracle database system uses "dynamic sampling" to improve server performance by determining more accurate selectivity and cardinality estimates, which allow the optimizer to produce better performance plans. Oracle determines at compile time whether a query would benefit from dynamic sampling. If so, a recursive SQL statement is issued to scan a small random sample of the table's blocks to estimate predicate selectivities. [7] Thus, while commercial DBMSs need dynamic sampling, they resort to rescanning or re-sampling from stored databases, and therefore, do not work in one pass.

A number of studies address the problem of query optimization by exploiting workload information [66, 38, 110]. The goal is to sample from the *output* of relational operators and queries such as union, difference, and join, etc. These approaches typically use additional statistics such as indexes to weight the sampling toward records which contribute more to the result of the query. A number of general techniques are known for sampling uniformly from relations, we now summarize the most relevant to our study.

[6] http://msdn.microsoft.com/library/en-us/dnsql2k/html/statquery.asp
[7] http://www.dba-oracle.com/art_dbazine_oracle10g_dynamic_sampling_hint.htm

Algorithm	Type	Method	Deletions	Random
Reservoir Sampling [127]	Fwd	WoR	No	Full
Backing Sample [71]	Fwd	WoR	Few	Full
Weighted Sampling [40]	Fwd	WR	No	Full
Concise Sampling [68]	Fwd	CF	No	Full
Count Sampling [68]	Fwd	CF	Few	Full
Minwise-hashing [54]	Inv	WR	No	$\frac{1}{\epsilon}$-wise
Distinct Sampling [70, 69]	Inv	CF	Few	Pairwise
Dynamic Inverse Sampling (here)	Inv	CF, WR	**Yes**	Pairwise

Figure 17: Key features of existing sampling methods.

The well known and widely used technique, reservoir sampling, was introduced by Vitter in [127]. The algorithm solves the problem of selecting a random sample of size n from a pool of N records, where the value of N is unknown. The reservoir is initially filled with the first n records. Each subsequent record is included in the reservoir (and a currently stored record randomly evicted) with appropriately chosen probability to ensure that, overall, the probability of any record surviving in the reservoir is uniform.

The backing sample [71] approach is an extension of the reservoir sampling method, used for incremental maintenance of approximate histograms. This approach attempts to maintain a random sample of a relation undergoing updates. Delete operations are handled by removing the record from the sample, if it is in the sample. The result is a uniform random sample, but in the presence of a large number of deletions, the sample size can become arbitrarily small. In [71], if the sample size drops below a certain threshold, the relation must be rescanned and the sample is repopulated.

One of the important applications of sampling in database systems is optimization of queries that involve join operations on relations. In [39] the authors describe a number of techniques that use a weighted sampling method to improve efficiency of the query by avoiding the need to compute the full join. In weighted sampling each element is sampled with probability proportional to its weight; for joins, these weights correspond to the frequency of the item in the other relation. Hence although the approach takes one pass over relations, it requires certain statistics to be available. The application is not concerned with deletions and so these are not discussed.

The study presented in [68] describes two new sampling summary statistics, concise

samples and counting samples, which provide highly accurate approximate answers to the "most frequently occurring values" queries in warehousing environment and can be incrementally maintained regardless of data distribution. A concise sample is a uniform random sample of the data set such that values appearing more than once in the sample are represented as $< value, count >$ pair. A newly selected record is added to the sample S with probability p (initially, $p = 1$). When the sample exceeds the space allocated for it, p is decreased and the sampled points are sub-sampled to give the probability of a record remaining in the sample to be equal to the new p. Counting samples are a variation of concise samples in which the counts are used to keep track of *all occurrences* of a value inserted into the relation since the value was selected for the sample. When the threshold is reached, a biased coin is flipped for each value in the counting sample, decrementing the count on each flip of tails until either the count reaches zero, or heads is flipped. The advantage of counting samples over concise samples is that they can handle deletions, by decrementing the count of the record if it is in the sample. However, in the presence of many deletions to the data, the counting sample can significantly shrink in size, compromising accuracy of the results.

Estimating the number of distinct values for some target attribute is yet another well-studied problem [39, 37, 85, 80] where a uniform random sample of the data is used to provide a fast approximate answer to distinct values queries. In [70, 69] the authors present a distinct sampling approach that collects a specially tailored sample over the distinct values that can be incrementally maintained. The algorithm uses a hash function to deterministically toss a coin for each record, so that identical records obtain the same outcome. Hence the probability that records are kept is uniform over the number of distinct records. The hash function maps records onto $1 \ldots \log N$ where N is a bound on the number of distinct values, which is called the "die-level" of the record. The probability of mapping to die-level l is approximately 2^{-l}. All records mapping to level L or higher are retained in the distinct sample; if the size of the sample grows larger than the available space then L is incremented, and the sample is pruned of all records mapping to less than the current L. The result is that each distinct record is retained in the final sample with uniform probability.

Our focus is on providing a uniform sample of the inverse distribution which can be used to approximate queries on the inverse distribution. Despite the many works on sampling in databases, there is very little work that directly applies to inverse distributions. Following [40] sampling methods broadly fall into three categories: sampling With Replacement (WR), Without Replacement (WoR), and coin flipping (CF)[8]. All the sampling methods we consider can be classified with one or more of these labels. In addition, two other factors are relevant to our focus:

Processing of Deletions. Existing methods either do not handle deletions (that is, it is unclear how to process a deletion and still retain a uniform sample), or can handle only a limited number of deletions: the result is still a uniform sample, but in the presences of many deletions, the size of the sample shrinks to zero.

Amount of Randomness Required. Early works assume "truly random" numbers, but more recent work considers what strength of randomness is needed. k-wise random hash functions guarantee that any k items collide under the hash function with independent probability [105], and such functions are efficient to compute and store for small k (eg pairwise hash functions with $k = 2$ [34]).

We summarize the relevant sampling techniques that can draw a sample from a stream of updates in Figure 17. We classify them on which distribution they sample from — the forward distribution (fwd) or the inverse distribution (inv); deletion handling; and the randomness required. Although many algorithms maintain a uniform random sample of data items of the forward distribution in the presence of insertions, none handle a significant number of deletions to the data set while guaranteeing a sample of a certain size.

There is a limited prior work that relates to inverse distributions. Some existing techniques can be used to create a sample from the inverse distribution on insert-only streams. The Distinct Sampling technique of Gibbons *et al.* [70, 69] draws a sample based on a coin-tossing procedure using a pairwise-independent hash function on item values. This effectively draws a uniform sample from the inverse distribution, which we can use to answer queries on the inverse distribution, as discussed in Section 4.5. As with all other existing sampling methods, deletions can deplete the sample, and it is not possible to re-

[8] Where the sample size is not fixed but rather each item is chosen to be in the sample with some probability p.

cover a sufficiently large sample—in our streaming scenario, backtracking on the past data for a rescan is simply not possible.

An alternative approach is to make use of Min-wise hash functions, which sample uniformly from the set of items seen. These were applied in [54] in order to estimate rarity, the number of items which occur exactly once. This is precisely $f^{-1}(1)$, and more generally the sample obtained by the procedure obtained there can be used to build a sample of the inverse distribution. Again, deletions were not considered; one can apply a "best effort" approach by decrementing the counts of deleted items in the sample until these fall to zero—but it is not possible to give worst case bounds on the size of the sample stored. Work on estimating the cardinality of set expressions over data streams [65] uses a similar data structure to the one we propose here, and with some amount of modifications can be used to draw a sample from the inverse distribution. However, this is not the goal of that work, and the given analysis requires hash functions that are at least $\log 1/\epsilon$-wise independent. Here, we show that for the purpose of sampling from the inverse distribution, a simpler structure is sufficient, with only pairwise independence. Similar results have been recently obtained by Indyk, and Frahling and Sohler [64]

4.8 Conclusions

Many of the existing methods for summarizing and mining data streams focus on the forward distribution. In contrast, we formulate summarization and mining problems on the inverse distribution. We introduced the notion of the inverse distribution for massive data streams, and gave algorithms that draw uniform samples from the inverse distribution when the data stream consists of insertions only, as well as insertions and deletions. With a sample of size $O(\frac{1}{\epsilon^2})$, we can answer a variety of summarization and mining tasks on the inverse distribution up to an additive approximation of ϵ. These are the first such results known for managing inverse distributions on data streams. In our experiments we saw that the methods we propose can process massive data streams of updates at very high rates, and answer queries on the inverse distribution with high accuracy.

Chapter 5

5 Filter-Join Operator

A common type of filtering query in data streams that identifies “interesting” tuples and filters out the rest requires a *join* for its evaluation. For example, a network analyst might want to collect all records in a network flow that start with a suspicious signature, and a financial analyst might want to track trading records of a financial instrument following a suspicious trade. However, evaluation of join on high speed data streams might bee very expensive.

In this chapter, we propose the *filter join* operator, which makes it feasible to evaluate a common type of join query on high speed data streams in an efficient, stable and accurate manner. The filter join has an inexpensive evaluation algorithm and can be pushed to the data sources in the case of self-joins. We provide a relational characterization of the filter join, and a collection of query transformations which can expose the filter join component(s) of a conventional join. We implement approximate filter join algorithms in the Gigascope Data Stream Management System (DSMS), and find order-of-magnitude performance improvements when compared to equivalent queries implemented using a conventional join.

5.1 Introduction

The applications of filtering queries that identify “interesting” records for reporting or more intensive analysis can be found in various domains:

- In the network traffic analysis domain, the "interesting" packets might be ones belonging to the same TCP flow and matching a signature of a malicious application. The collected packets could later be used for collecting statistics, closer examination of the packet payloads and other types of real-time analysis.

- A financial analyst monitoring securities trading could define as "interesting" an event of detecting a suspicious trading pattern, such as sudden increase in trading volume of a particular security. The analyst could then be interested in collecting the data on all subsequent activities related to the security for intensive analysis, for arbitrage possibilities.

- In a data stream from security video cameras, an event of interest could be a sudden or anomalous movement by an object in view, which would trigger capture of all succeeding frames that contain that object for subsequent human analysis.

The most natural approach to evaluating queries of this class is to perform a self-join on the data stream. In the general case of a join query, we have two data streams, S and R, and a set of attributes which define the join key. In a self-join we duplicate our original data stream into two streams on which a join is performed. For example, when collecting packets belonging to a suspicious TCP flow, the stream S is used to identify the beginning of a flow, and the stream R is used to identify the rest of the packets, which may include either all packets from the flow or only those matching a particular pattern of interest.

However, even if we manage to formulate our inquiry as a join query, executing it on high speed data streams can be very expensive, since it requires that two possibly high volume streams be brought together in an operator and time-synchronized, potentially requiring a large amount of buffering. A self-join query may be less resource intensive, since only one stream needs to be brought to the operator and no synchronization is needed. However, even a self-join in its full generality still remains a resource intensive operation on high speed data stream, as it involves keeping track of a large number of records that expire over time. As a result, it is often the case that neither join nor self-join queries can be evaluated under adversarial conditions in an efficient and stable manner without sacrificing accuracy of the query results.

Rather than employing such measures as load-shedding as is commonly the case nowadays when dealing which high-speed streams, we inspected our target applications more closely and observed that they do not require a full-blown self-join operator. In fact, these problems can be described as self-join problems that adhere to the following pattern of evaluation:

1. Mark a record of a stream as a beginning of a sequence of records of interest
2. Evaluate certain conditions on every subsequent record of interest, and
3. Output only those records of a marked sequence that satisfy the condition.

It is clear that, in addition to solving the queries we described earlier, this pattern of evaluation applies to a large class of similar problems in various domains. While being a specific case of a join operator on two data streams, this particular pattern of evaluation can also be regarded as a filtering procedure which, in contrast with a join or a self-join, can have an inexpensive implementation since its semantics are those of a set membership. We therefore abstract this idea and introduce a join operator which we refer to as *filter join (FJ)*. We can then use a filter join to answer the class of queries we described above on high speed data streams in an efficient, stable and accurate manner.

Our contributions are as follows:

- We introduce a new filter-join operator and discuss its applications in data stream processing. In many cases, the filter join can be considered an inexpensive predicate and pushed to the leaf of the query plan.
- We provide a relational definition of the filter join operator and its semantics. We also show various query transformations that expose filter joins in conventional join queries.
- We design and implement approximate filter-join algorithms in Gigascope DSMS using two different data structures.
- We test and measure the performance of our filter-join algorithms on live network traffic and show that the operator is robust and produces accurate query results.

The rest of the chapter is organized as follows. In Section 5.2 we discuss the large body of work on joins, in particular, on data streams and how they differ from our filter-joins. In Section 5.3, we define our filter-join operator using a relational expression. In Section 5.4 we discuss query transformations that expose filter-joins in queries with conventional joins. In Section 5.5 we introduce a cost model for filter join evaluation. In Section 5.6 we discuss algorithms for implementing filter-joins with approximate set membership data structures. In Section 5.7, we present detailed experimental study of our filter-join implementations with live network traffic. We use for our analysis Gigascope DSMS (described in section 3.3); its architecture and target application of network traffic analysis nicely fit our study of filter-joins for early data reduction in processing data streams.

5.2 Related Work

Data streams in general, and joins in general, are well-researched areas. In this section we will focus only on work that is related to evaluating joins on data streams.

While joins are very important for data stream analysis, their computation is resource-intensive and conceptually requires maintaining an unbounded state for each item in the infinite data stream. To make the computation of the join feasible on data streams, its semantics are usually changed to restrict the number of tuples participating in a join to a bounded-size window that slides over the input stream. The window boundaries can be defined in terms of time units, number of tuples or punctuation marks [22, 124]. Although restricting the amount of data participating in a join operation reduces the resource requirements, the join computation might still exceed resource availability. One of the characteristics of a data stream is its *burstiness*, i.e. the amount of data that arrives to the data analysis center at any point of time can vary greatly. When the window is large and the CPU can keep up with the processing, the main memory might be too small to maintain all the relevant tuples in-memory, thus significantly slowing down the system. Even when the window is reasonably small, the CPU might not be fast enough to process all incoming tuples, which might cause either a system failure or might reduce the accuracy of the results by dropping some of the tuples. In order to balance between these tradeoffs, DSMS's em-

ploy various load shedding techniques [121, 23, 113] , multilevel architectures [50, 51, 52], as well as data processing algorithms and data structures [130, 129, 125, 79, 75, 126, 29] that make it possible for a system to operate within these constraints.

Join algorithms have been studied extensively in the context of data streams. The symmetric hash join (SHJ) [130, 129] was originally designed to allow a high degree of pipelining in parallel database systems. However, SHJ requires in-memory hash tables for both of its inputs during the query evaluation. Thus, the ability of SHJ to sustain large inputs is severely limited. To rectify this situation, XJoin [125] was introduced, which provided an efficient way to spill overflowing inputs to disk and later join them to produce the final query output. In [87] authors present a way of adapting SHJ into hybrid hash join, whenever inputs are too large to fit in memory. The approaches in [125, 87] access disk in cases of large inputs, which is prohibitively expensive when handling high speed data streams and is not feasible if instant query results are required.

A symmetric nested loops join (SNLJ) [79] was proposed for online aggregation. Evaluation of this join requires for each tuple of a stream to scan the entire hash table of another stream in order to produce join tuples. This operation makes per-tuple processing very slow and inefficient when operating on high speed data streams. SHJ was extended to the binary sliding window join (BSWJ) [92], and that work also introduced a cost model for each operator as a function of individual stream arrival rates. This work showed that asymmetric join processing has advantages if the arrival rates of the two joining streams differ. However, it uses combination of hash join and nested loop join to construct BWSJ, thus making this approach subject to the same inefficiencies as mentioned above. Other works on joins for data streams include multi-join processing on a number of data sources [75, 126].

SemiJoin [27] and BloomJoin [31] were developed for distributed query processing in an attempt to minimize the amount of data transmitted over the network. For relations R and S that are stored at different sites and are being joined on a set of attributes key (i.e. $R \bowtie_{key} S$), SemiJoin works as follows: (1) Compute $P_R = \pi_{key}(R)$ at the site that stores R. (2) Send P_R to the site of S and compute $P_S = P_R \bowtie_{key} S$. (3) Transfer P_S to the site of R and compute the final join $R \bowtie_{key} P_S$. BloomJoin transmits Bloom filters [29]

rather than join attribute values. A Bloom filter is generally smaller than the projected join attributes and therefore often results in lower network overhead and join cost at the joining site. Due to hash collisions in the filter, BloomJoin can be viewed as a lossy variation of the SemiJoin. The filter join presented in this work could be viewed as a special case of these two techniques: just as in filter join, both SemiJoin and BloomJoin aim at making the initial data reduction before executing the join operation. However, the work on SemiJoin and BloomJoin does not address or analyze the query semantics and the query transformations for efficient execution of the join on high speed data streams.

The join operator has been implemented in various Data Stream Management Systems. Aurora [13] has a windowed binary join operator that can join streams as well as stored relations. To deal with the unpredictable nature of data streams, Aurora employs random and semantic load shedding techniques, dropping tuples at various locations of the system during query processing in cases of system overloads. The binary join operator in STREAM [19] maintains synopsis for each of the joining streams. To process continuous queries over data streams in an adaptive manner, STREAM also employs load shedding techniques [23]. During the query execution the data stream is uniformly sampled at various points of the query plan, while the sampling rate is dynamically adjusted with respect to the operator selectivity and arriving rate of the data. TelegraphCQ [36] is another DSMS that implements a traditional symmetric join operator using state modules and adaptive routing modules by maintaining hash indexes on both relations. When the speed of the data streams exceeds the capacity of the state modules, the system uses triage queues to collect the dropped tuples and eventually uses synopses to capture their approximate properties [113]. All of the aforementioned load shedding techniques, although effectively reducing the amount of data to be processed, have negative implications on accuracy of query results.

Unlike the perviously discussed DSMSs, in order to process data streams in a controlled manner Gigascope [50, 51, 52] uses a multilevel architecture (details in section 3.3) instead of load shedding techniques. The low level works as a filter for the incoming high-speed data streams performing significant early data reduction, and the filtered data is sent to the

higher levels where more sophisticated data analysis is performed. Similarly to the previously mentioned DSMSs, Gigascope has a high level SHJ operator that requires significant data reduction at the low level of the system architecture to keep up with the rate of the incoming stream. This work presents a new filter join operator that works directly on high-speed streams requiring little, if any, input data reduction but achieving high data reduction in its output.

5.3 Filter Join

A large class of network data stream analysis queries require following a certain pattern of evaluation, where first the beginning of a flow is marked, and then every subsequent packet of the flow is evaluated on a desired condition and only those packets that satisfy the condition are passed through for further analysis. In particular, we will now consider two motivating IP network data analysis problems and demonstrate how they are solved using the described pattern of evaluation:

Problem 1*: Find flows in which the payload of HTTP response packets contains links to audio or video files. Output only packets of those flows that satisfy the condition.*

In order to solve this problem, we need to find and mark the first packet of an HTTP response message. Then we need to examine all the subsequent packets of the response for the presence of links to audio or video files.

Problem 2*: Find flows of Gnutella P2P file sharing application in which the payload of a response packets contains a signature of a particular virus. Output only packets of those flows that satisfy the condition.*

In this problem we first need to find and mark the first packet that belongs to a response of the Gnutella, and then examine all the subsequent packets of the flow for presence of the specified virus signature in the payload of a packet.

The solutions to both of the above problems require performing a self-join on the incoming data stream. However, when we deal with high-speed network data streams, performing such self-join query is prohibitively resource intensive, so we need an alternative approach. What we propose is looking at the problem from a slightly different angle. We are asked to find and output packets that satisfy a certain condition; in other words, this is a data reduction, or data filtering, problem: we are asked to filter out the rest of the packets of a flow. We refer to this particular unidirectional case of a traditional hash join operation as a *filter join*, and we formally define its general case as follows:

Definition 3. *Let R and S be two data streams, A be a set of attributes associated with every tuple $t_R \in R$ and $t_S \in S$, $A_{key} \subseteq A$ be a set of join attributes, and $a_t \in A$ a monotonic increasing attribute. Let c be a positive integer. A **filter join** of the two streams is a subset of R defined by* $\{t_R \in R | \exists_{t_S \in S}\ t_R.A_{key} = t_S.A_{key} \text{ and } t_R.a_t \geq t_S.a_t \text{ and } t_R.a_t \leq t_S.a_t + c \text{ and } t_R \text{ follows } t_S\}$.

The motivation for this definition is two-fold. First, this definition of the filter join matches the needs of our example queries described earlier, as well as of other queries of the similar nature in various analysis domains. In both our sample queries we want to find tuples in a stream that have been marked as "interesting" — HTTP response message in the first example and Gnutella response message in the second. Second, it has an efficient implementation. We only need to store tuples of S to perform the join; tuples of R are streamed out if there is a matching tuple of S. In most cases, S will be much smaller than R, minimizing the memory footprint size. If the join is a self-join, then no tuple buffering is required to synchronize R and S.

With this definition of filter join we can now define a new operator FILTER JOIN that follows the self-filtering pattern of evaluation we described above. This new operator helps us formulate a query for the audio/video links search problem stated above as follows:

Query 1:

```
SELECT R.time, R.srcIP, R.destIP, R.srcPort, R.destPort,
       R.sequence_number, R.ack_number, str_regex_match('.(aac|ac3|aif|
```

```
        aiff|asf|avi|divx|dv|m1v|m2p|m2v|mov|moov|mpa|mpg|mpeg|mp1|mp2|
        mp3|mp4|mpv|ogg|ogm|omf|qt|rm|ram|swf|vob|wav|wma|wmv)',
        R.TCP_data) as header
FILTER JOIN TCP as R, TCP as S
WHERE R.srcIP = S.srcIP AND R.destIP = S.destIP AND
        R.srcPort = S.srcPort AND
        R.destPort = S.destPort AND
        R.protocol = 6 AND
        S.protocol = 6 AND
        R.data_length <> 0 AND
        S.data_length <> 0 AND
        str_starts_with('HTTP', S.TCP_data) AND
        str_regex_match('.(aac|ac3|aif|aiff|asf|avi|divx|dv|m1v|m2p|m2v|
        mov|moov|mpa|mpg|mpeg|mp1|mp2|mp3|mp4|mpv|ogg|ogm|omf|qt|rm|ram|
        swf|vob|wav|wma|wmv)', R.TCP_data) AND
        R.time ≤ S.time + 10
```

The first four conditions of the WHERE clause of the query above specify the join key attributes, while the next four predicates define the two joining streams R and S. Query 1 finds all packets in S that start with a string "HTTP". It also matches the regular expression specified as an argument to the `str_regex_match()` function to every packet of R, thus ensuring that the payload contains at least one reference to a file with a known audio or video file extension. Note that the regular expression starts with an implicit ".*", which specifies that the desired file extension can be matched anywhere within the packet. The last condition of the query specifies the liveliness of the S tuples: in this example, the tuples expire after 10 seconds of their arrival. The `str_regex_match` predicate is expensive, we would prefer to evaluate it on the minimum possible size set of tuples. A valuable optimization is to first perform the inexpensive filter join, and then perform the expensive `str_regex_match` predicate.

Similarly, we can formulate a query for the second problem of finding a virus signature in traffic of Gnutella application:

Query 2:

```
SELECT R.time, R.srcIP, R.destIP, R.srcPort,R.destPort,
       R.sequence_number,R.ack_number,
       str_regex_match('virus_signature',R.TCP_data) as header
FILTER JOIN TCP as R, TCP as S
WHERE R.srcIP = S.srcIP AND
       R.destIP = S.destIP AND
       R.srcPort = S.srcPort AND
       R.destPort = S.destPort AND
       R.protocol = 6 AND
       S.protocol = 6 AND
       R.data_length <> 0 AND
       S.data_length <> 0 AND
       str_starts_with('GNUTELLA', S.TCP_data) AND
       str_regex_match('virus_signature',R.TCP_data) AND
       R.time ≤ S.time + 180
```

When a server in the Gnutella network establishes connection to another server in the network, the response contains the message "`GNUTELLA OK`$\backslash n\backslash n$". Thus the function `str_match_with` tries to match the string `GNUTELLA` at the beginning of a packet in order to identify the first packet of a flow. The string `virus_signature` that is passed as an argument to the function `str_regex_match` should be substituted by the desired signature of the virus being tracked. The above query tries to match "virus_signature" to all of the subsequent packets of the flow appearing within 3 minutes of the connection establishment.

Query 3 below outlines the general form of a join query on two relations (streams) R

and S; in the query, $f()$ and $g()$ are two predicates on a join key attributes of a record, P_{cmp} stands for complex predicate, P_{ch} and P_e stand for cheap and expensive predicates respectively, and P_t signifies predicate on temporal attributes of a record. In this query, $time$ is a monotonic increasing attribute that defines the tuple's timestamp, A is a set of all tuple attributes $(time, a_1, a_2, ...a_n)$ and A_{key} is a subset of A that contains only attributes $(a_i, a_{i+1}, ...a_j)$ that constitute the join key:

Query 3:

```
SELECT R.time, R.a1, R.a2, ..., R.an,
       S.time, S.a1, S.a2, ..., S.an
JOIN TCP1 as R, TCP2 as S
WHERE
       Join key predicates
       f(R) = g(S) AND
       Complex predicates on both relations
       Pcmp(R,S) AND
       Cheap single relational predicates
       Pch(R) and Pch(S) AND
       Expensive single relational predicates
       Pe(R) and Pe(S) AND
       Predicates on temporal attributes
       Pt(R.time, S.time)
```

The initial join key predicates in the WHERE clause of the query can be as simple as defining equality of the two join attributes, e.g. $R.a_i = S.a_i$. They may also be expressed as a function applied to any or all of the join attributes of a tuple. The complex predicates on both relations might include predicates like $R.a_1 > S.a_1+1$. Single relational predicates are often cheap to evaluate; they might be similar to $R.a_1 = constant$. In other cases they might be more expensive to evaluate, for example invoking expensive functions such as

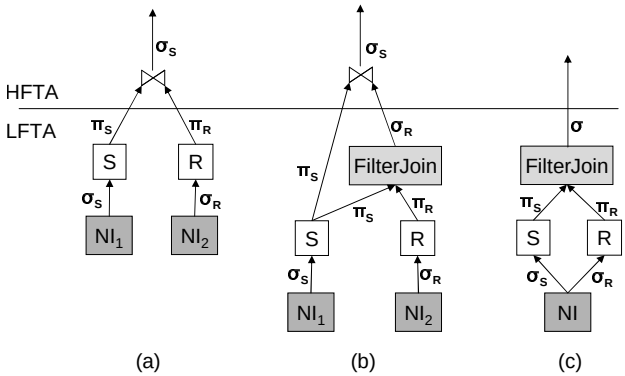


Figure 18: (a) Symmetric hash join on two data streams from two different data sources in Gigascope. (b) Filter join + symmetric hash join on two data streams from two different data sources (c) Filter join on a single data stream

regular expression matching. The last condition of the query on temporal attributes might look like $R.time$ IN $[S.time, S.time + c]$ or $R.time \leq S.time + c$ where c is some constant that defines the lifetime of a tuple; we provide more details on predicates on temporal attributes in section 5.4.3.

Query 3 joins two data streams that have two different data sources TCP_1 and TCP_2 and can be fully evaluated by symmetric join operator at HFTA level of Gigascope. Figure 18(a) shows the query plan of that scenario. The number of tuples that would have to be copied from LFTA to HFTA in order to evaluate the query can be very large, considering that each of the data streams may produce tens or even hundreds of thousands of tuples per second. This query plan is likely to be expensive due to both the join and the tuple copying costs.

If possible, a better execution plan would be the one shown in Figure 18(b), when both streams go through filter join at LFTA and a significantly reduced amount of traffic is channelled to HFTA for completion of the query evaluation. However, in practice it is often possible to further optimize the query plan and make the query evaluation more efficient based on certain additional information about the data source. For example, if we know that both S and R have the same data source, it is possible to complete the query evaluation using only the filter join operator at LFTA level of Gigascope. Figure 18(c) shows the possible query evaluation plan for this case. Pushing a filter join as close as possible to the data source is a critical optimization since doing so minimizes data movement in the DSMS.

The core of the filter join is a set membership test, where elements expire from the set

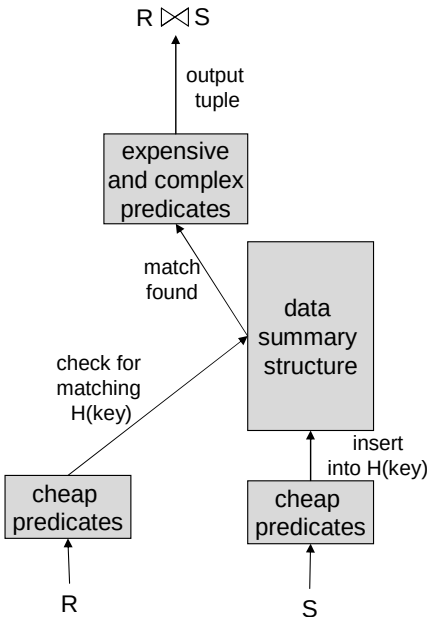


Figure 19: Filter-join procedure

over time. While there are many possible implementations, Figure 19 shows a general schema for filter join query evaluation. When a tuple arrives, it is first evaluated on the cheap single-relational predicate specified by the query. If it does not satisfy the conditions, the tuple is discarded. When a tuple from the S stream passes the filtering predicate, it is hashed into the data summary structure. If the arriving tuple belongs to the R relation, the hashed value of its join attributes is compared with the tuple of S that has an identical hash key within the data structure, if such exists. If a matching S tuple is found, the tuples are evaluated on the expensive single-relational and complex predicates. When all of the conditions are satisfied, the output tuple is produced.

Given a join such as Query 3, how can we determine if it can be evaluated using a filter join? The conditions are:

1. No attributes of S appear in the select clause, except for $S.time$.
2. $P_{cmp}(R, S)$ is empty (i.e. has the value TRUE).
3. The predicate of the temporal attribute is of the form $R.time$ IN $[S.time, S.time + c]$, and either
   (a) $R.time$ and $S.time$ are strictly increasing, or
   (b) There is a predicate equivalent to "R follows S".

To complete the definition of the filter join operator, we define the following set of assumptions used in filter join query evaluation:

Tuple Ordering: We assume that the arriving tuples have synchronized timestamps. When evaluating filter join of the two streams R and S, we consider a tuple from R to be a valid candidate for filter join only if its timestamp is greater than the timestamp of the tuple from S. Thus tuples from S stream can be in advance of tuples from the R stream, however tuples from R streams are never in advance of the tuples from S stream. More formally, $S.time \leq R.time$, if and only if R arrived after S. In the case of a self-join, this synchronization is automatic, otherwise we assume that there is a module which performs any necessary buffering before tuples are processed by the filter join.

Distinct tuples: In the presence of many tuples from S with identical join key attributes and valid timestamps, one possible approach would be to store all such tuples of S in the data summary structure and iterate through this list for every arriving R tuple with the matching hash value. However, this approach could require a considerable amount of memory to maintain tuples from S and would potentially be too slow and unable to keep up with high-speed data streams. Therefore we only store tuples of S with distinct key values in the data summary structure. In other words, queries that perform filter join have an implicit DISTINCT in their SELECT clause and only distinct join tuples are produced in the output.

5.4 Query Transformations

With the definition of filter join given in the previous section, we now explore under which conditions a query can be executed with a filter join operator and define a number of query transformations that can take advantage of filter join at the lower level of a query plan while performing the rest of the data analysis at the higher level.

5.4.1 Referencing Tuple Attributes

In order to achieve better performance, the amount of memory used by the filter join operator must be reasonably small. Therefore, our data summary maintains only a restricted

set of S tuple attributes, which includes all of the attributes that constitute the join key, $a \in A_{key}$, and the timestamp of a tuple $S.time$.

The fact that we maintain only a restricted set of attributes for tuples from S makes the query processing more involved when the query references attributes other than the join key attributes or the time attribute of S tuples in its SELECT clause. Such cases require a query decomposition, when part of the query is processed at the filter join and is completed with another conventional join.

Query 4:

```
SELECT S.a_{j+1}
JOIN R, S
WHERE f(R) = g(S) AND
      P_ch(R) and P_ch(S) AND
      P_e(R) AND
      P_t(R.time, S.time)
```

This query cannot be processed efficiently at LFTA level, since the SELECT clause contains a reference to an attribute which is not a part of the restricted set of attributes of S maintained by data summary structure. In order to process this query efficiently, we need to decompose it. One possibility for the decomposition is as follows:

Query 5.1, HFTA:

```
SELECT S.a_{j+1}
JOIN R_source as R, S_source as S
WHERE f(R) = g(S) AND
      P_t(R.time, S.time)
```

Query 5.2, LFTA:

```
DEFINE query_name R_source
SELECT time, a_i, a_{i+1},..., a_j
FROM R
WHERE P_ch(R) AND P_e(R)
```

Query 5.3, LFTA:

```
DEFINE query_name S_source
SELECT time, a_i, a_{i+1},..., a_j, a_{j+1}
FROM S
WHERE P_ch(S)
```

This query set performs the join of the two streams at the HFTA level, and the only processing that is done at LFTA is simple SELECT filtering. The query plan of this query set is shown in Figure 20(a). However, it is likely that this decomposition is not going to be sufficiently efficient on high-speed streams; as we mentioned before, we'd like to push the join operation as far down the system architecture as possible. A better way of splitting the query is as follows:

Query 6.1, HFTA:

```
SELECT S.a_{j+1}
JOIN R_source as R, S_source as S
WHERE f(R) = g(S)
```

Query 6.2, LFTA:

```
DEFINE query_name R_source
SELECT time, a_i, a_{i+1},..., a_j
FILTER JOIN R, S
WHERE f(R) = g(S) AND
      P_ch(R) and P_ch(S) AND
      P_e(R) AND
      P_t(R.time, S.time)
```

Query 6.3, LFTA:

```
DEFINE query_name S_source
SELECT time, a_i, a_{i+1},..., a_j, a_{j+1}
FROM S
```

This decomposition performs filter join at LFTA level and symmetric hash join at HFTA which outputs the desired value of the $S.a_{j+1}$ attribute. If S_source is highly selective (which is often the case in practice), evaluating Query 5.1 is a low-cost operation.

The decomposition examples described above deal with the case where the SELECT clause of the query references an attribute $S.a_{j+1}$ that is not a temporal attribute or a part of the join key. In cases where such attribute is referenced by any of the single relational

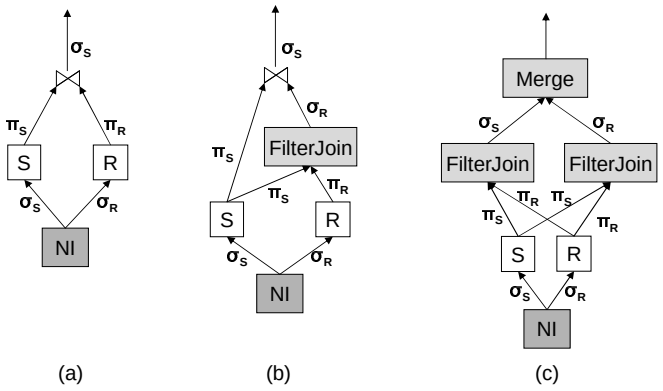


Figure 20: Query Plans

predicates, the query can be evaluated at LFTA level without any decomposition, since the value of $S.a_{j+1}$ is known at the time of S tuple processing and has no dependency on any of the attributes of R.

When a non-temporal, non-join key attribute is referenced in complex predicates on both relations, query evaluation becomes more complex and may also require decomposition. Let's consider the following predicate: $R.a_i > S.a_j$. This predicate references the a_j attribute of S, which belongs to the set of the join key attributes ($S.a_j \in S.A_{key}$). When evaluating a query with this predicate, the value of $S.a_j$ can be retrieved from the data summary structure, and the predicate can be evaluated at the LFTA level before the output tuple is produced. On the other hand, if the predicate is $R.a_{j+1} > S.a_{j+2}$ where $S.a_{j+2} \notin S.A_{key}$, the query needs to be decomposed as follows:

Query 7.1, HFTA:

SELECT `S.a`$_{j+2}$

JOIN `R_source as R, S_source as S`

WHERE `f(R) = g(S) AND`

`P`$_{cmp}$`(R,S)`

Query 7.2, LFTA:

```
DEFINE query_name R_source
SELECT time, a_i, a_i+1, ..., a_j
FILTER JOIN R, S
WHERE f(R) = g(S) AND
      P_ch(R) and P_ch(S) AND
      P_e(R) and P_e(S) AND
      P_t(R.time, S.time)
```

Query 7.3, LFTA:

```
DEFINE query_name S_source
SELECT time, a_i, a_i+1, ..., a_j, a_j+2
FROM S
```

Query plans for the decomposed query sets 6 and 7 are shown in Figure 20(b).

To generalize the query transformation analysis above, a query has to satisfy the following condition to be executed by filter join operator:

The only attributes of S that can appear in the SELECT clause of the query or as a part of its complex predicates on both relations, are either the attributes $S.a \in S.A_{key}$ that constitute the join key of the query, or the temporal attribute(s).

5.4.2 Expensive Single Relational Predicates

The cost of performing an expensive single-relational predicate can be substantial, as we show in the experimental section of the chapter. When a query contains both $P_e(R)$ and $P_e(S)$, in the general case of query execution using filter join operator, the two predicates would be evaluated on every tuple produced by the join. This evaluation can significantly increase the per-tuple time processing. Since we expect the output of a filter join to be much smaller than the input, we push the evaluation of the expensive predicates after the join. Our filter-join operator in Figure 19 has a module for evaluating $P_e(R)$ (Gigascope uses this kind of heavy operator to minimize data movement). To optimize the processing of $P_e(S)$, we can push the evaluation of $P_e(S)$ up the query evaluation plan, thus making the query evaluation faster. If, for example, the evaluation of $P_e(S)$ requires knowing the value of attribute $S.a_{j+1} \notin A_{key}$, the query transformation becomes similar to query set 7:

Query 8.1, HFTA:

```
SELECT S.a_{j+2}
JOIN R_source as R, S_source as S
WHERE f(R) = g(S) AND
      P_cmp(R,S) AND
      P_e(S)
```

Query 8.2, LFTA:

```
DEFINE query_name R_source
SELECT time, a_i, a_{i+2}, ..., a_j
FILTER JOIN R, S
WHERE f(R) = g(S) AND
      P_ch(R) and P_ch(S) AND
      P_e(R) AND
      P_t(R.time, S.time)
```

Query 8.3, LFTA:

```
DEFINE query_name S_source
SELECT time, a_i, a_{i+2}, ..., a_j, a_{j+2}
FROM S
```

To summarize the above, a query has to satisfy the following condition to be executed by filter join operator:

A filer join query may not contain any expensive single relational predicates $P_e(S)$ of S; all such predicates are pushed up in the query evaluation plan.

5.4.3 Predicates on Temporal Attributes

The last condition of the WHERE clause defined by Query 3 in section 5.3 is a predicate on a temporal attribute of the two streams. Such a predicate bounds the range of tuples that can be potentially joined. For example, the predicate $R.time$ *in* $[S.time, S.time + c]$ joins only those tuples of R that arrive within c seconds of the last seen tuple from S. Queries with such a predicate can be fully evaluated by the filter join operator by requiring the filter join procedure to consider only those tuples of S in the data summary structure for which $S.time \leq R.time$, where $R.time$ is the timestamp of the currently processed tuple of stream R.

Another example of predicates on temporal attributes that can be fully evaluated in a very similar manner by the filter join operator include $R.time$ *in* $[S.time + c_1, S.time + c_2]$

where $c_1 > 0$ and $c_2 > 0$ are constants such that $c_1 < c_2$.

A temporal predicate can also be of the form $R.time$ *in* $[S.time - c, S.time + c]$. This case is different from the ones we've just discussed, since now we want to capture all tuples of R such that they appear within $\pm c$ seconds of the matching tuple from S. To evaluate a query with such predicate, we again need to split it between the two levels of Gigascope architecture:

Query 9.1, HFTA:

```
MERGE R.time:S.time
FROM after as R, before as S
```

Query 9.2, LFTA:

```
DEFINE query_name after
SELECT R.time, R.a_i, R.a_i+1, ..., R.a_j
FILTER JOIN R, S
WHERE f(R) = g(S) AND
      P_ch(R) and P_ch(S) AND
      P_e(R) AND
      R.time in [S.time, S.time + c]
```

Query 9.3, LFTA:

```
DEFINE query_name before
SELECT R.time, R.a_i, R.a_i+1, ..., R.a_j
FILTER JOIN S, R
WHERE f(R) = g(S) AND
      P_ch(R) and P_ch(S) AND
      P_e(R) AND
      S.time in [R.time, R.time + c]
```

The query plan for this set of queries is shown in Figure 20(c). The query "`after`" (9.2) takes care of the $[S.time, S.time + c]$ part of the predicate time interval, while the query "`before`" (9.3) captures tuples of R that fall into the $[S.time - c, S.time]$ part of the time interval by transforming the predicate into the form $S.time$ *in* $[R.time, R.time + c]$ acceptable by the filter join operator. The two streams of tuples are later merged together at HFTA level of query processing.

To formalize the above analysis, a query must satisfy the following condition in order to be executable by the filter join operator:

The temporal predicate that defines the liveliness of tuples from S can only refer to time intervals starting with the most recently seen $S.time$. In other words, the time interval such predicate describes must be of a form equivalent to $[S.time, S.time + c]$ where $c > 0$.

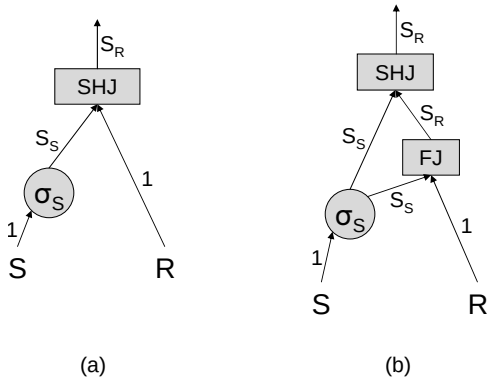


Figure 21: Cost of the query plans (a) Involving Symmetric hash join at HFTA level of Gigascope architecture. (b) Involving filter join at LFTA and Symmetric hash join at HFTA level of Gigascope architecture

In conclusion, the filer join query has the following general form:

Query 10:

SELECT `R.time, R.a1, R.a2, ..., R.an,`

`S.time, S.ai, S.ai+1, ..., S.aj`

FILTER JOIN `TCP as R, TCP as S`

WHERE `f(R) = g(S) AND`

`Pch(R) and Pch(S) AND`

`Pe(R) AND`

`Pt(R.time, S.time)`

5.5 Cost Model

In order to determine when the query transformations presented in Section 5.4 are beneficial, we develop a simple cost model using the two alternative plans in Figure 21 as an example. Our cost model has two components, the data *transmission* cost T and the *processing* cost P. We'll assume that the input rate is 1, and that the selectivities of S and R are S_S and S_R respectively, where $S_S \leq 1$ and $S_R \leq 1$.

Since both FJ and SHJ are hash joins, we can estimate their cost as the the sum of their input and output rates. Further, since FJ is one-sided while SHJ is symmetric, the per-tuple

cost of SHJ is K times the per-tuple cost of FJ, for some appropriate value of K (probably between 1 and 2). Therefore we can derive the following:

$$P_{SHJ} = K(S_S + 1 + S_R) \tag{1}$$

$$P_{SHJ\_FJ} = (S_S + 1 + S_R) + K(S_S + 2S_R) \tag{2}$$

The σ_S and FJ operators can execute as low-level query nodes in Gigascope, while the SHJ operator must execute as a high-level query node, necessitating a data transfer to the SHJ. Therefore we can derive

$$T_{SHJ} = S_S + 1 \tag{3}$$

$$T_{SHJ\_FJ} = S_S + S_R \tag{4}$$

Clearly, T_{SHJ} is always larger than T_{SHJ_FJ}, and is significantly larger if S_R is small. Data transfer costs are often the bottleneck in Gigascope, and hence the filter join plan is usually the best one. However, let us also consider the processing costs. For most reasonable values of S_S and S_R, the filter join plan has a lower cost than the non-filter join plan. To see this, let us determine the value of K for which both plans have equal processing cost. By solving $P_{SHJ} = P_{SHJ\_FJ}$ for K, we get:

$$K = \frac{S_S + 1 + S_R}{1 - S_R} \tag{5}$$

We plot K for different values of S_S and S_R in Figure 22. Even when $K = 1.3$, the filter join plan will have equal or lower processing cost than the symmetric hash join plan S_S and S_R are 0.1 or smaller. Therefore, if the filter join will actually do a significant amount of filtering, it is better than the symmetric hash join plan.

5.6 Implementation

As we emphasized earlier, in order for the filter join operation to be efficient it is essential that it consumes a limited amount of memory and that the per-tuple time processing is small. The filter-join can be implemented using a conventional hash join with excel-

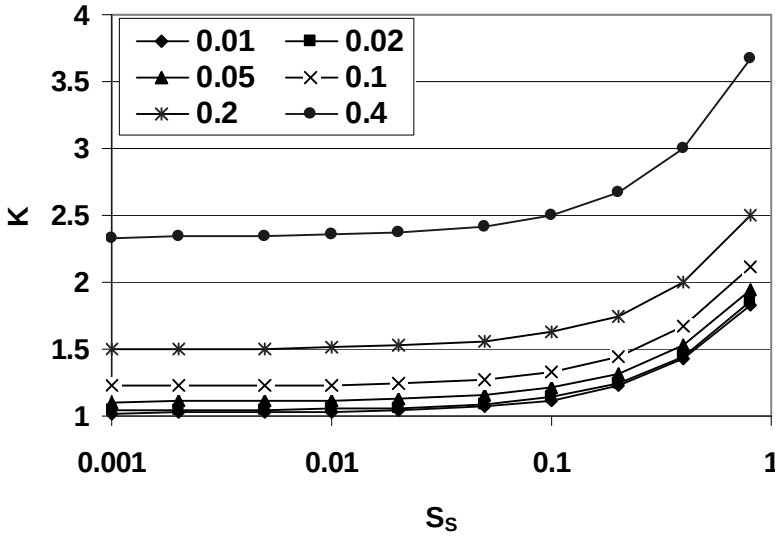


Figure 22: Comparison of SHJ and Filter Join. The values in the legend of the figure are for S_R, and the S_S axis is in log scale.

lent efficiency. However, an advantage of the simple semantics of the filter join (i.e., set membership) is that it readily lends itself to *approximate* algorithms.

In this work, we explore two approximate filter join algorithms, one with negative errors but no positive errors (using a fixed-size hash table) and another with positive errors but no negative errors (using Bloom filters). Negative errors are acceptable in many cases, and positive errors can often be filtered out at the later stages of processing. Also, a technical restriction in Gigascope makes chained hash tables difficult (but not impossible) to implement — query operators at the LFTA level are not supposed to use dynamic memory allocation. Therefore we concentrate on the more interesting approximate implementations.

5.6.1 Hash table

We implemented the first version of the filter join procedure with a hash table used as the data summary structure, where the hash key is the set of the join attributes of a tuple $S.A_{key}$, and the value is the arriving time of the tuple $S.time$. The table structure is shown in Figure 23. We emphasize that since memory reallocation is an expensive operation strongly discouraged at the LFTA levels, the size of the hash table has to be known at the initialization, and therefore the join key attributes must only contain either numeric value attributes or constant size string attributes.

The algorithm uses a second chance probing mechanism in case of hash table insertion

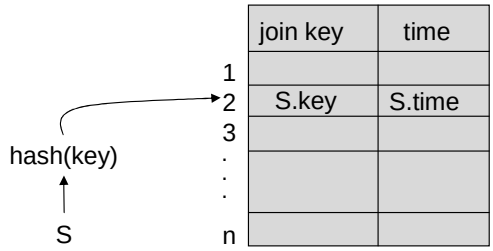


Figure 23: Data summary structure - hash table

collisions: whenever the slot of the hash table if occupied by a valid tuple that has a different set of join key attributes than the one being processed, the next slot of the table is probed for insertion. If that slot is also occupied with a valid tuples, we experimented with two different approaches (the results of our experiments are described later on):

1. Evict the oldest tuple from the two slots considered for insertion, insert the current tuple in its place
2. Drop the current tuple and proceed to the next tuple

The complete procedure of the algorithms is as follows: Let S_c be the tuple of S being currently processed, and let S_h be a tuple of S previously inserted into the hash table.

On the arrival of S_c:

```
if (Sc satisfies Pch(Sc)) then
  slot = hash(Sc.Akey)
  if (slot has valid Sh and Sh.key == Sc.key) then
     Sh.time = Sc.time
  else if (slot contains invalid Sh) then
     replace Sh with Sc in slot
  else if (slot is empty) then
     insert Sc into slot
  else use approach (1) or (2) for collision handling
```

On the arrival of R:

```
if (R satisfies Pch(R)) then
```

```
slot = hash(R.A_key)
if (slot contains valid S_h and S_h.A_key == R.A_key) then
  if (R satisfies P_e(R)) then
    return R
```

This implementation of filter join is prone to producing *false negatives*, i.e. tuples of R and S that were not joined due to hash collisions. However, there are no false positives.

5.6.2 Bloom filter

The other version of the filter join procedure was implemented using a set of Bloom filters [29]. Each Bloom filter is of size n bits, and corresponds to a single time unit of the liveliness time interval of S. In other words, if B is the set of Bloom filters, and the temporal predicate is $R.time$ *in* $[S.time, S.time + c]$, there would be c filters in B, i.e. $|B| = c$. To preserve cache locality and ensure efficient memory access, we arranged all of the Bloom filters into a single bit array in which all *i*th bits of the filters are grouped together, as demonstrated on Figure 24.

A set H of hash functions is used to set bits in the Bloom filter which corresponds to the time unit of each arriving tuple from S. The hash key is the set of the join attributes $S.A_{key}$, and the value of the hash functions is the bit index $[0 \ldots n-1]$. When a tuple from S is inserted, the hash function values are calculated and the appropriate bits in the corresponding Bloom filter are set, as shown on Figure 25.

When a tuple from R arrives, we calculate the corresponding Bloom filter bit numbers and check whether the same bits are set in any of the Bloom filters, in which case (if all other conditions are met) an output tuple is produced.

The Bloom filters are used in a circular manner: with the arrival of the first tuple t (whether it belongs to S or R) in a new time unit, the Bloom filter with the index $[bf = t.time \bmod |B|]$ is first zeroed out, and then the bits corresponding to the tuple are set.

The detailed pseudo code for handling S and R tuples is as follows:

On the arrival of *S*:

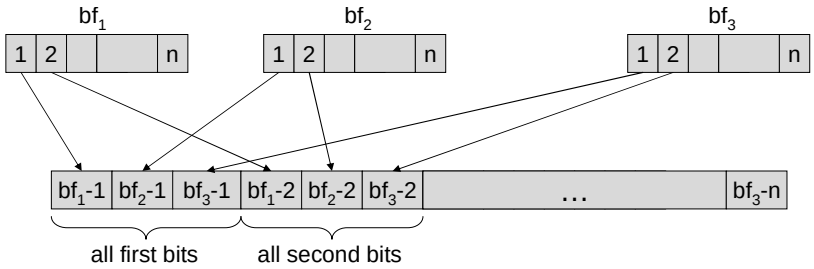


Figure 24: Bloom filters data structure for improved cache locality, $B = 3$

```
bf = S.time mod |B|
zero out Bloom filter B_bf
if (S satisfies P_ch(S)) then
  for each H_i do
     SET_BIT(B_bf, H_i(S.A_key))
```

On the arrival of *R*:

```
bf = R.time mod |B|
zero out Bloom filter B_bf
if (R satisfies P_ch(R)) then
  for each B_i do
     for each H_j do
        if (IS_SET(B_i, H_j(R.A_key))) then
           count_set++
        else
           count_set = 0
           break
     if (count_set == |H|) then
        found = 1
        count_set = 0
        break
     else
        count_set = 0
  if (found == 1) then
```

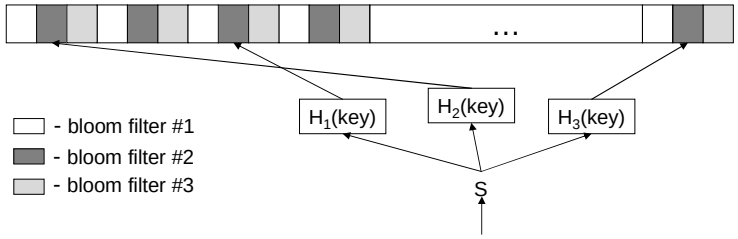


Figure 25: The insertion procedure of S into a single Bloom filter. ($B = 3$, $H = 3$)

```
if (R satisfies Pe(R)) then
  return R
```

Due to the nature of the data structure used, the algorithm is prone to producing *false positives* (but no false negatives) when an output tuple is created as a result of two different join key sets setting the same bit within a Bloom filter, i.e. $H_i(S.A_{key1}) = H_j(S.A_{key2})$. This could be remedied, and in fact may even be considered an advantage over having false negatives as in the case of hash table implementation, when filter join is used as a preliminary filtering procedure whose results are fed into a more sophisticated, heavy-weight analysis at the higher level of query processing.

5.7 Experimental Study

We implemented and tested the filter join operator in the Gigascope DSMS. Gigascope provided us with the ability to experiment with various data structures, configurations of the algorithm parameters as well as with access to high-speed network data feeds. We tested the performance of the operator on a data center network traffic feed which produced moderately high speed traffic of about $60,000$ packets per second (about 250 Mbits/sec). For all experiments, we used an inexpensive dual 2.8 GHz Intel Pentium Xeon processor server.

We chose to evaluate filter join by running Query 1 described in detail in section 5.3, since we are likely to have a large amount of data in the stream matching the predicates specified by the query, making it closer to the worst case scenario in terms of traffic load.

	SHJ-R1	**SHJ-R1 40% smpl**	**SHJ-R1 40% smpl, no regex**	**SHJ-R2**	**SHJ-R2 5% smpl**	**FJ-HT**	**FJ-HT no regex**	**FJ-BF**	**FJ-BF no regex**
HFTA	99%	80%	29%	11%	0.05%	0.05%	4.9%	0.05%	6.9%
LFTA	11%	6.8%	6.8%	99%	8%	65%	7.9%	80%	10.5%

Table 2: CPU utilization statistics for symmetric hash join and filter join queries.

5.7.1 Performance

To evaluate CPU utilization of the algorithm, we ran Query 1 (section 5.3) with the hash table and Bloom filter implementations, and compared it with the symmetric hash join operator implemented at the HFTA level of Gigascope. This query performs CPU intensive processing, performing a regular expression match for every packet in every HTTP response flow. We ran a simple aggregation query at HFTA level that counts the number of tuples produced by filter join at the lower level.

We collected the following statistics about the traffic used in our experiments: approximately $10,000$ out of initial $60,000$ packets satisfied the filter join conditions before regular expression evaluation, and about $3,000$ of them passed the regular expression predicate resulting in an output tuple.

We performed the experiment for a range of hash table and Bloom filter sizes, and we found that there was no significant difference in CPU utilization with respect to the size of the data structure used.

We compared the CPU utilization numbers obtained for the filter join operator with two sets of queries performing symmetric hash join at HFTA and regular expression matching at either HFTA or LFTA levels (the actual queries are not presented here due to space constraints):

- Query set SHJ-R1 performed the symmetric hash join and the regular expression matching at the HFTA level;
- Query set SHJ-R2 performed regular expression matching on tuples of R stream at LFTA level, and symmetric hash join of S and R at HFTA.

The comparison of CPU utilization numbers is shown in Table 2. We can see that

queries SHJ-R1 and SHJ-R2 utilize over 100% of a single CPU on a dual CPU machine, when combining the two levels of query plan. During the evaluation of SHJ-R1, the low level query performs a simple SELECT on a number of tuple attributes; the tuple is then passed to the higher level of the architecture for further processing which includes CPU intensive regular expression matching procedure. The amount of data that needs to be copied and the CPU intensive processing at HFTA cause high CPU consumption. Under these conditions the system was unable to handle the rate of the incoming data stream, and it was forced to drop tuples and produce inaccurate results while overloading the server's CPU. Query SHJ-R2 attempted to perform regular expression matching on every tuple of R causing almost a 100% CPU utilization at LFTA alone, and did not produce any results at all due to system overload.

In an attempt to stabilize the symmetric hash join queries we introduced data reduction by sampling a fraction of flows from the incoming R and S data streams. SHJ-R1 became stable only after filtering out about 60% of the incoming flows, while SHJ-R2 started to produce results in a stable manner only after reducing the number of flows to about 5% of the original traffic. CPU utilization numbers for these two experiments are also shown in Table 2.

In contrast, both the hash table (FJ-HT) and the Bloom filter (FJ-BF) implementations of the filter join operator resulted in a stable execution of the query that contained regular expression matching, without any preliminary data reduction and producing meaningful results at high speeds. To measure CPU utilization of the filter join operation only, we removed the regular expression matching evaluation. Table 2 shows that CPU utilization of the Bloom filter implementation resulted in the cumulative 17.4% on both query evaluation levels and was 4.6% higher than that of the hash table implementation with 12.8% CPU utilization. This increase was due to the operation of cleaning the Bloom filters at every advance of the time unit. However, both filter join implementations use far less CPU time than the SHJ-R1 algorithm without regular expression matching, which consumes a combined 90% utilization when scaled to a 100% sample.

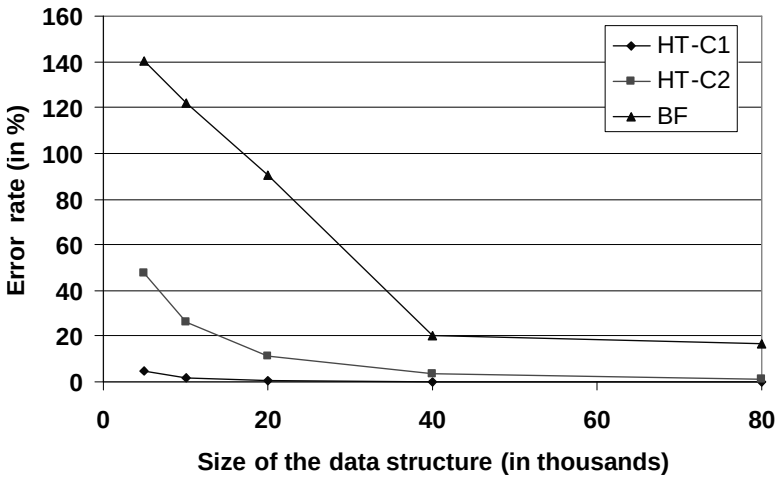


Figure 26: Error rate of the approximate algorithms. (HT-C1 corresponds to the hash table implementation when evicting old tuple on collision, HT-C2 corresponds to dropping the new tuple on collision, and BF is the Bloom filter implementation.)

5.7.2 Accuracy

To measure the accuracy of the algorithms and make it possible for the system to run a number of queries simultaneously, we removed the regular expression evaluation predicate and considered the results of the filter join operation itself. We measured the accuracy of the algorithms by comparing the results to those of a hash table implementation that used a large enough hash table to eliminate collisions and thus produce exact results. We counted the number of tuples produced by the query for every 10 second time interval, and then calculated the average error rate over a number of 10 second time intervals for each of the implementations as follows: let t_{ext} be the number of join tuples produced by the exact filter join query, and let t_{apx} be the number of join tuples produced by an approximate implementation; then *error rate* $= \frac{|t_{ext} - t_{apx}| * 100}{t_{ext}}$.

During the experiments we observed that the number of S tuples needed to be maintained in the data summary structure within a period of 10 seconds was about $15,000$. For the hash table implementation we ran queries with the hash table size of $5,000$, $10,000$, $20,000$, $40,000$ and $80,000$, using the two collision handling techniques described in section 5.6.1. To make it comparable with the Bloom filter implementation, the cumulative size of all Bloom filters used by the algorithm was made to be equal to the corresponding version of the hash table implementation. We also tried different amounts of hash functions used by the Bloom filter algorithm, using $|H|$ of 2, 3, 4 and 5. We observed however that the

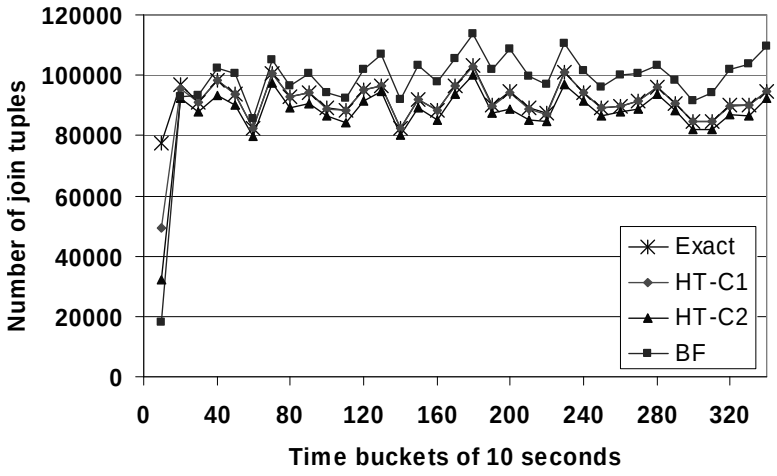


Figure 27: Number of tuples produced by various versions of filter join with hash table size of $40,000$ (HT-C1 corresponds to the hash table implementation when evicting old tuple on collision, HT-C2 corresponds to dropping the new tuple on collision, and BF is the Bloom filter implementation.)

number of hash function did not seem to have a significant effect on the resulting number of join tuples, and thus the analysis presented in Figure 26 shows the accuracy results for $H = 3$. Figure 27 shows the results of the query execution with hash table size of $40,000$.

It can be seen from these graphs that the Bloom filter algorithm produces a large number of false positives (about 20%). We can also observe that the hash table version of filter join that drops the new tuple on collision (HT-C2) is significantly less accurate than the one that evicts the oldest tuple from the hash table and inserts the current one in its place (HT-C1). HT-C1 is the most accurate implementation of the filter join algorithm, resulting in only 0.14% less tuples than the exact version, while using a reasonably small size of the hash table of $40,000$ entries.

5.8 Conclusions

A large class of queries on data streams search for records matching a dynamic criteria. While a relational expression of this kind of query involves a join, these queries can use a faster evaluation algorithm because they are essentially set membership queries. In this work, we propose the *filter join* operator to enable the use of the fast algorithm. Although being a unidirectional case of the traditional hash join, the presented filter join benefits from having an efficient implementation that allows early data reduction of high speed data streams, which is crucial under adversarial conditions.

In this work we have provided a relational expression of the filter join operator; we provided query transformations which expose filter join operator(s) in conventional join queries; we described and implemented two approximate filter join algorithms, one with positive errors and one with negative errors; we presented cost models that clearly demonstrate the advantage of filter join over the traditional symmetric hash join; finally, we have tested and measured our implementations on live traffic streams.

We find that the filter joins provide order-of-magnitude performance improvements when compared to using a regular hash join. A significant contributor to the performance improvement is the early data reduction achieved by pushing the filter join down to the data source. Further, the query we used in our experiments is a difficult case for the filter join, since a large fraction of the stream records pass the filter. Other common uses of a filter join (finding DNS, RTP, or worm flows in network traffic stream analysis) have much more selective filters.

The filer join operator has been implemented in the Gigascope DSMS, and is now part of the production version of the system. It is being actively used in IP network data analysis, and its usefulness for evaluation of queries similar to those presented in this work is being shown in practice.

Chapter 6

6 Regular Expression Matching on Out-of-Order Streams

In data streams, the data normally possesses certain attributes that can be used to define order over the stream elements. However, it is often the case that the data is received out of order, which presents a serious challenge and requires maintaining the state of partial streams for computing order-sensitive queries over such data streams.

A particular instance of this problem is data stream filtering using regular expressions, it is important in such applications as network traffic identification using application signatures. The existing work in this field either simplifies the problem by matching at a single data segment, or reassembles segments in the correct order before applying the regular expression. Neither approach is satisfactory: valid signatures can span multiple data segments, but reassembly is very resource intensive.

We present an efficient algorithm for regular expression matching on streams with out-of-order data, while maintaining a small state and without complete stream reconstruction. We have implemented three versions of the algorithm - sequential, parallel and mixed - and show by experimental study on real network traffic data that the algorithms are highly effective in matching regular expressions on IP packet streams.

6.1 Introduction

Much as in databases, data streams can have *data quality* problems. This may take the form of a duplicate item as is common in practical databases. More characteristically, data

streams may be out of order [101]. In data streams, the data normally possesses certain attributes that can be used to define order over the stream elements. Let us consider two distinct examples:

- The stream of IP packets seen at a network monitor is ordered by time seen and may be loosely ordered based on time sent. However, often the data is received out of order. Consider TCP, a well-known network protocol which guarantees reliable and in-order delivery of data from sender to receiver. In TCP the original message is reconstructed by using the sequence number of the packet payload, which is the offset of the packet's data segment within the originally transmitted message. Due to various transmission delays and network failures, packets might be lost or arrive not in the order they were originally sent. TCP handles those cases with a packet retransmission procedure, which often results in multiple copies of the same packet at the receiver.

- Consider a network of sensors. Due to resource constraints sensors send data when they can, and not necessarily at periodic intervals. Further, for reasons of robustness against loss, sensors often retransmit the same information. Also, the network connecting the sensors use multiple length path for collecting data. Consequently, duplication, out-of-orderness and latency are common in time series data from sensors, and stream management systems that monitor such data have to use order and duplicate insensitive analysis techniques [108, 47].

In the past few years, a number of techniques have been developed for processing and mining data streams Data quality issues such as the ones above present a serious problem for DSMSs because even simple queries on data streams with data quality problems become challenging. For example, computing the average size of distinct packets in a TCP stream now requires one to keep the *state* of the partial stream seen on the link to identify the duplicate packets. The challenge is further exacerbated when one deals with sophisticated streaming queries and the suite of data quality problems includes out-of-order items.

The task we address in this work is a sophisticated query — matching a signature that is regular expression — on an out-of-order stream with duplicates. The motivating problem is as follows.

Motivating Problem. Consider the IP network monitoring application. The TCP protocol sends the content $c_1 \cdots c_n$ to be transferred from the source to the destination in smaller-sized payloads. This involves repacketing $c_i \cdots c_j$ as needed. The set of all packets that is involved in this transfer form a *flow*. The problem we study is to determine which flow, if any, has content $c_1 \cdots c_n$ that matches a profile. The profile is specified as a *regular expression*. For example, a profile for identifying the flow that comprises a download from the popular Kazaa service is `^(GET|HTTP).*[xX]-[Kk][Aa][Zz][Aa][Aa]`—the content should begin with either `GET` or `HTTP`, followed by any series of characters ('.*') before the appearance of `x-kazaa` (case-insensitive).

If the string $c_1 \cdots c_n$ is given altogether, there are well-known methods for matching the regular expression to it that involve traversing the automaton derived from the regular expression, with the string. However, our problem is that we are provided the string in small-sized segments from the payload of various packets that comprise the flow. Any given regular expression has to be matched across these segments. Further, due to mechanisms inherent in TCP, the content may arrive out of order or there may be duplicates and packets with overlapping contents. Considering this, matching the regular expression against $c_1 \cdots c_n$ becomes a serious challenge.

Analysis of network packet contents such as in the problem above at high speeds is crucial to network security and network monitoring applications. It is often required to match the payload of the packet or a number of packets within a stream with a given set of patterns which characterize different applications [116], viruses or worms [10, 93, 109], protocols, etc. For example, it was possible in the past to classify applications based on port numbers, but it has become more and more problematic as applications and protocols have become more sophisticated [60]. Hence, a significant amount of work has been done in the past few years on using signatures to identify different applications [116]. Now the patterns which identify them (such as in the Kazaa example above) often constitute not just an explicit string, but rather a regular expression due to their expressive power and flexibility. Developing these regular expression profiles has its own challenges: a polymorphic worm is hard to characterize since it changes its payload in successive infection attempts.

The problem of application identification is solved in practice in one of two ways.

1. Restrict the regular expression and use simple profiles that will match a segment found inside a *single* packet. This approach severely limits the applicability of the problem because even simple profiles such as the one above for Kazaa often has to be matched across multiple data segments (i.e. multiple packets).

2. Reassemble all the segments of the flow into the content string $c_1 \dots c_n$ and use the well-known regular expression matching methods. The difficulty is that full reassembly of the content is prohibitively resource intensive. [58] points out that existing intrusion detection systems become highly inefficient while maintaining the full state of all open connections. To be practical, existing systems perform random load shedding which is not ultimately accurate or effective in finding intrusions

We have described the regular matching problem for IP traffic streams, and this will be the running motivation throughout. The same problem however arises in time series monitoring and other applications over out-of-order and duplicated data streams, in sensor data streams, email and text streams, and elsewhere. □

In this work, we address the problem of matching a regular expression over real world streams. Our contributions are as follows:

- We formalize the problem of regular expression matching over a data stream with data quality problems such as out-of-order and duplicate items.

- We present an algorithm for regular expression matching without reassembling the entire stream. The algorithm maintains potential start and end states for each stream segment in tracing the finite state automaton that represents the regular expression. The states are pruned as needed so the algorithm maintains only a limited memory. We introduce the concept of "equivalent states" and present three variations of the algorithm depending on how they are identified and pruned.

- We perform a detailed experimental study of our algorithm with real networking data and show that the algorithm is highly effective in matching regular expressions against

TCP flows of IP packets. Our algorithms achieve over 100:1 compression over methods that reassemble the TCP flows, with comparable running time. Our method makes the regular expression matching task eminently practical on live streams.

6.2 Regular Expressions and Application Signatures

A regular expression is a powerful language to describe a set of strings. In standard regular expressions, starting with the alphabet symbols, we compose a set of strings using the following operators and metacharacters: "|" - alternation, "." - any character, "*" - matches the preceding character zero or more times, "+' matches one or more times,"?" matches zero or one times, "^" and "$" match the beginning and the end of the string respectively. It is typical to further enhance the language with ranges of characters ("[a-z]") or sets of characters ("[ABC]"). In what follows, we give a few examples of application signatures that are used in network monitoring applications.

Gnutella p2p protocol signature [116]:

`^(GNUTELLA|(GET|HTTP).*(X-Gnutella|((Server:|User-Agent:)[\t]*(LimeWire|BearShare|Gnucleus|Morpheus|XoloX|gtk-gnutella|Mutella|MyNapster|Qtella|AquaLime|NapShare|Comback|PHEX|SwapNut|FreeWire|Openext|Toadnode|Shareaza))))`

This regular expression is a signature for *Gnutella* P2P network protocol, and can be used to detect *Gnutella* data downloads . It is read as follows:

- The first string following the TCP/IP header is `GNUTELLA`, `GET` or `HTTP`.
- If the first string is `GET` or `HTTP`, it can be followed by one or more arbitrary characters, followed by `X-Gnutella`. The strings `GET` or `HTTP` can also be followed by any number of arbitrary characters, followed by either `Server:` or `User-Agent:` headers, followed by a number of TAB symbols, followed by one of the strings from the list `LimeWire`, `BearShare`, etc.

Kazaa P2P protocol signature [116]:

`^(GET|HTTP).*[xX]-[Kk][Aa][Zz][Aa][Aa]` This regular expression is designed to identify *Kazaa* P2P network downloads [116]. It requires that the data following the TCP/IP

header starts with either GET or HTTP, followed by an arbitrary string with X-Kazaa appearing anywhere in it.

Yahoo traffic [1]:

^(ymsg|ypns|yhoo).?.?.?.?.?.?.?[lwt].*\xc0\x80 This regular expression appears in the open source collection of application signatures included with the l7-filter system [1] and identifies Yahoo traffic. It matches any packet payload that starts with ymsg, ypns or yhoo followed by seven or fewer arbitrary characters, then followed by a letter *l*, *w* or *t* and some arbitrary characters of any length, and finally the ASCII letters *C0* and *80* in the hexadecimal form.

Counter Strike game traffic [131]:

cs.*dl.www.counter-strike.net This rule is also mentioned in [131] and used to detect packets of a an online game "Counter Strike". The expression will match any packet that contains a string cs followed by zero or more arbitrary characters, followed by dl.www.counter-strike.net.

HTTP request:

((OPTIONS|GET|HEAD|POST|PUT|DELETE|TRACE|CONNECT)[ ]+[-~]+[ ]+HTTP/1.[01]([-~]+ \r \n)+\r\n) This regular expression can be used for extraction of HTTP request headers. It matches any packet payload that starts with the key words OPTIONS, GET, etc., followed by one or more space, followed by one or more printable ASCII characters, followed by one or more spaces, followed by HTTP/1.1 or HTTP/1.0, followed by one or more lines with one or more printable ASCII characters (\r\n signify 'carriage return' and 'line feed' at the end of a line), and ending with an empty line.

HTTP response: (HTTP/1.[01][ ]+[0-5][0-1][0-9]([-~]+ \r\n)+\r\n) This regular expression can be used for extraction of HTTP response headers. It matches any packet payload that starts with HTTP/1.1 or HTTP/1.0, followed by one or more spaces, followed

by a 3 digit HTTP response code, with the first digit between 0 and 5, the second either 0 or 1, and the third between 0 and 9.

6.3 Formal description of our problem

We define the problem in the context of TCP flow analysis, but it can be similarly defined for other applications, including text and sensor data analysis.

A stream corresponding to a single TCP flow consists of a number of individual network packets, each packet containing the protocol header and the data segment. Say the data to be transmitted is $c_1, \ldots, c_n$. When n exceeds certain packet size limit, the data is split among multiple packets, and each packet is transmitted independently. The stream seen by a network monitor consists of data *segments* $d_1, d_2, ..., d_i, ...$, where each segment d_i represents a portion of the original data being transmitted. A segment $d_i = c_{s_i} \cdots c_{e_i}$ is described by the *start offset* s_i and *end offset* e_i within the original data. The length of segment d_i is $l_i = e_i - s_i + 1$. We define d_j as the *predecessor* of d_i if $s_i = e_j + 1$ and and d_j as the *successor* of d_i. On the receiving end, the received data segments need to be reassembled in the correct order, so that the original message can be reconstructed. We use D_m to refer to a reassembled portion (sometimes refereed to as "partial flow") of the original data $c_{S_m} \cdots c_{E_m}$.

Due to the nature of computer networks, there can be a number of anomalies in the way the stream segments arrive at the receiver. For a newly arriving data segment d_i, and the reassembled data portion D_m, we have the following anomalies:

Duplicates and Overlaps: The TCP protocol guarantees reliable information delivery. If receipt of a packet is not acknowledged within a certain period of time, the packet is retransmitted, possibly more than once, until the acknowledgement is received. This can lead to the same data segment being received more than one time on the receiving end. Duplicates can occur in a number of ways:

1. $s_i \geq S_m$ and $e_i \leq E_m$, i.e. D_m wholly contains d_i.

2. $s_i \leq S_m$ and $e_i \geq E_m$, i.e. d_i wholly contains D_m.

3. $s_i < S_m$ and $e_i \geq S_m$ and $e_i < E_m$, i.e. start of D_m overlaps with the end of d_i.

4. $s_i > S_m$ and $s_i \leq E_m$ and $e_i > E_m$ start of d_i overlaps with the end of D_m.

Out of order packets: Due to various delays in the network communication, packets may arrive out of order, so that for a newly arriving data segment d_i and the reassembled data portion D_m, there can be a case that $e_i < S_m$ or that $s_i > E_{m+1}$.

Given the situation above, a regular expression $\mathcal{R}$ and the content $c = c_1 \cdots c_n$, our problem is to determine if c matched $\mathcal{R}$, given the series of packets d_i's.

6.4 Overview of Our Algorithms

In our problem described earlier, a string c is presented as a series of packet segments $d_1, d_2, ...$ Matching each d_i against $\mathcal{R}$ will be incorrect when the matching string spans more than one packet. Collecting all the d_i's, reassembling them into c and matching $\mathcal{R}$ using the basic algorithm would require waiting until all data segments of the flow are received, and is therefore slow; it is also resource-intensive.

A more efficient solution would be to match the regular expression with portions of the data received thus far reassembled into "partial flows" and wait until a decision (match/no match) is reached. This would be ideal if the reassembled partial flow represented a prefix of c. However, the fact that some of the data arrives out of order effectively fragments the reassembled data into a number of partial flows D_m's.

A simplistic approach to dealing with fragmented data would be to store all out-of-order partial flows, until they can be merged with the reassembled portion of the original data that represents a prefix of c. The major disadvantage of this approach, to which we refer in the rest of the chapter as *buffering*, is that the size of those disconnected partial flows can be quite large and to maintain them in memory during the matching process can be very costly, as we shell see in the experimental section 6.7.

If we wish to not store the partial flows, we need to simulate the DFA on the D_m's. In order to do this for partial flows which are not a prefix of c, we need to know which state in the DFA to start the simulation from. Our key idea is very simple: to simulate the DFA on D_m's with all potential beginning states for D_m in the DFA (which in the worst case could be all non-accepting states of the automaton). This will lead to a number of

potential end states for each D_m. We hope to extract savings in this "state" we store by merging partial flows when possible, pruning the potential beginning states for D_m and further exploiting the structure of "equivalence classes" of states reached by simulating the DFA from different begin states.

Our algorithm implements this approach and optimizes the state saved and the execution time. We present three algorithms: a *sequential* algorithm, a *parallel* algorithm that aggressively collapses equivalent states (defined later) and a *mixed* algorithm that tries to balance the tradeoffs.

As an aside, notice that given a regular expression it is possible to construct an NFA (nondeterministic finite automaton) with fewer states than the corresponding DFA, which could reduce the state maintained. However, the number of state to state transitions in NFA is significantly larger and it is much more expensive to traverse. Since our focus is on real time analysis, we preferred the DFA-based method which has better update cost per packet. So, we present our results only for the DFA representation, but our algorithms and the concept of equivalent states we use to prune, can be easily generalized to NFAs.

6.5 The Sequential Algorithm

The algorithm maintains the information about the received partial flows in the form of a linked list R of objects $D_1, D_2, ..., D_i, ..., D_n$. Each $D_i = (S_i, E_i, L_i)$ describes a reassembled partial flow, and contains the following:

- (S_i, E_i) - the starting and ending offset of the reassembled data within the original data transmitted within the flow.
- L_i - a linked list of pairs (q_s, q_e) describing the starting and ending states of paths within the automaton representing the regular expression that can be traversed with the data corresponding to D_i.

Figure 28(a) demonstrates a single object D_i of the list R. Each pair of states (q_s, q_e) in the list L_i of the object D_i is such that $q_e = \delta(q_s, D_i)$.

At various stages of the algorithm we will attempt to find partial flows that either precede or succeed the newly arrived segment in the original data, and merge them into one list

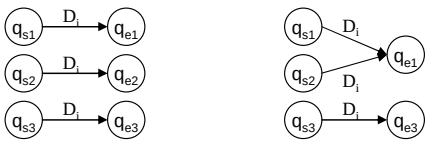


Figure 28: Structure of the object D_i for (a) the sequential and (b) the parallel version of the algorithm

entry. If, as a result, we obtain two entries D_i and D_{i+1} in the list such that D_i precedes D_{i+1} in the original data, we will merge them into one entry as well.

6.5.1 Traversing DFA

As part of the algorithm, we will need to traverse the automaton representing the regular expression with the data contained in the currently processed data segment d, beginning from a given state q_i within the automaton. The automaton traversal stops when an accepting state is reached, the end of the data is reached, or when there's no transition on the current data character from the current automaton state.

The return value of the traversal process is a pair of states (q_s, q_e), designating the starting and ending states of the path traversed, as well as flags indicating whether the q_s is the starting state of the automaton, and whether q_e is an accepting state. The process can also return a null value if there is no useful path that can be traversed with the given input, which can happen in one of the two cases:

- we reach a state during the traversal process from which there is no transition with the next data character, or
- both the beginning and ending state of the traversal process is the starting state of the automaton.

As an example, consider the DFA shown in Figure 29 for the regular expression `^(GET|HEAD|POST).*HTTP`. This regular expression is a simplified version of the regular expression for HTTP request message described in section 6.2.

If the contents of the first packet received is `'GET'` and we run this string though the automaton starting at state 1, the pair of states that will be recorded is (1, 4). If the next packet of the stream contains `'HTTP/1.1'` and we run it through the automaton starting

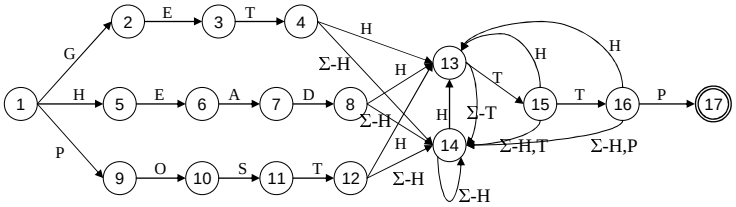


Figure 29: DFA for "^(GET|HEAD|POST).*HTTP"

from the state (4), the pair of states that will be recorded for this data segment is (4,17). The two pairs are merged resulting into the pair (1,17) where 1 is the starting state of the automaton and 17 is an accepting state.

6.5.2 Detecting Start of the Flow

The algorithm begins with *R* empty. The beginning of a flow is detected by inspecting the value of the SYN (synchronize) bit in the TCP header of the arriving packets, with 1 signifying the flow start. When processing the first packet of the flow, we distinguish between two types of regular expressions: those that start with the starting anchor '^' and require the first packet to match starting from the starting state of the automaton, and those that start with '.*' and imply that the regular expression can be matched anywhere within the flow.

Thus the first data segment $d_1 = (s_1, e_1)$ of the flow is processed as follows:
Traverse the DFA beginning from the starting state of the automaton. If the regular expression starts with '^':

- If the traversal process returned null, we label the flow as "not matching", and no further processing is done on the flow's data.
- If the traversal process returned a pair of states (q_s, q_e), with q_s marked as the starting state of the automaton, create a new entry $D_1 = (s_1, e_1, L_1)$ in R, where L_1 contains the pair (q_s, q_e).

If the regular expression does not start with '^':

- If the traversal process returned null, create $D_1 = (s_1, e_1, < empty\ list >)$ in R.
- If the traversal process returned a pair of states (q_s, q_e), with q_s marked as the starting

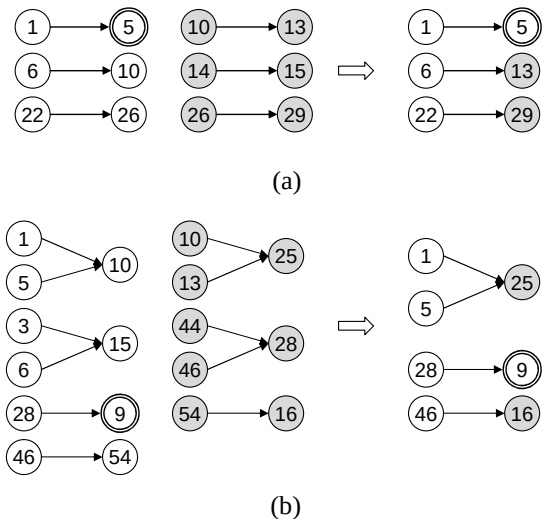


Figure 30: Merging (a) pairs of states of the predecessor and the successor (sequential version) (b) equivalence classes of the predecessor and the successor (parallel version)

state of the DFA, create a new entry $D_1 = (s_1, e_1, L_1)$ in R, where L_1 contains the pair (q_s, q_e), and proceed to the next data segment.

6.5.3 Processing Subsequent Segments

Any other data segment $d_i = (s_i, e_i)$, $s_i > 1$, is processed as follows. For each object D_m in list R:

Duplicate handling:

- If d_i is fully contained in D_m, ignore d_i and proceed to the next segment.
- If D_m is fully contained in d_i, delete D_m from R.
- If d_i and D_m partially overlap, chop off the overlapping section of d_i by adjusting its (s_i, e_i) offsets accordingly. Formally, either $s_i = E_m + 1$ or $e_i = S_m - 1$ depending on whether S_m is smaller than s_i or otherwise.

Predecessor processing: Let $D_p = (S_p, E_p, L_p)$ be a predecessor of d_i, i.e. $E_p = s_i - 1$. If L_p is not empty, for each pair (q_s, q_e) in L_p:

- Traverse the DFA with d_i starting at q_e.
- If the traversal returns a pair (q_e, q_{e1}), delete the pair (q_s, q_e) from L_p, store the pair (q_s, q_{e1}) in L_p and update $E_p = e_i$.

#	**Data Segment** $d_i = (s_i, e_i)$	**Reassembled Segment** $D_i = (S_i, E_i)$	**Pairs of States in** L_i (q_s, q_e)	**Notes**
1	'G'$= (0, 0)$	$D_1 = (0, 0)$	(1,2)	first data segment of the flow
3	'T'$= (2, 2)$	$D_2 = (2, 2)$	(3,4)(11,12)(13,15) (15,16)(4,14)(8,14) (12,14)(14,14)(16,14)	data segment arrives out-of-order
4	' file.html '=(3,13)	$D_1 = (0, 0)$ $D_2 = (2, 13)$	(1,2) (3,14)(11,14)(13,14) (15,14)(4,14)(8,14) (12,14)(14,14)(16,14)	after merging the data segment with its predecessor $D_2 = (2, 2)$
2	'E'$= (1, 1)$	$D_1 = (0, 1)$ $D_1 = (0, 13)$	(1,3) (1,14)	the missing data segment arrives: after merging it with its predecessor $D_1 = (0, 0)$ after merging $D_1 = (0, 1)$ with its $D_2 = (2, 13)$ successor into a single partial flow
5	'HTTP/1.0'=(14,21)	$D_1 = (0, 21)$	(1,17)	declare a **match** since 1 is the starting state of DFA and 17 is a final accepting state

Table 3: Using the sequential algorithm to match the regular expression `"^(GET|HEAD|POST).*HTTP"` (see Figure 29) to a flow containing the data 'GET file.html HTTP/1.0' split into 5 data segments: 'G', 'E', 'T', ' file.html ', 'HTTP/1.0'.

- If the traversal returns null, delete (q_s, q_e) from L_p. If this renders L_p empty, label the current flow as "not matching"

If L_p is empty:

- Traverse the automaton with d_i beginning at the automaton's start state.
- If the traversal returns a pair (q_s, q_e), insert the pair (q_s, q_e) in L_p, and update $E_p = e_i$.
- If the traversal returns null, update $E_p = e_i$; L_p remains empty.

If there is no predecessor for d_i in R:

- Create a new entry $D_p = (S_p = s_i, E_p = e_i, L_p =< empty\ list >)$ in R.
- Traverse the automaton with d_i starting at every non-accepting state, and insert all non-null pairs returned by the traversal process in L_p.

Successor Processing: At the end of predecessor processing part of the algorithm, we have either merged d_i in an existing D_p, or created a new D_p for the newly arrived segment. At this stage of the algorithm we check whether D_p has a successor in R.

#	**Data Segment** $d_i = (s_i, e_i)$	**Reassembled Data Segment** $D_i = (S_i, E_i)$	**Equivalence classes in** L_i (l_i, q_{ei})	**Notes**
1	'G'$= (0, 0)$	$D_1 = (0, 0)$	(1,2)	1st data segment of the flow
3	'T'$= (2, 2)$	$D_2 = (2, 2)$	(3,4)(11,12)(13,15)(15,16) ((4,8,12,14,16),14)	data segment arrives out of order
4	' file.html '=(3,13)	$D_1 = (0, 0)$ $D_2 = (2, 13)$	(1,2) ((3,4,8,11,12,13,14,15,16),14)	after merging the data segment with its predecessor $D_2 = (2, 2)$
2	'E'$= (1, 1)$	$D_1 = (0, 1)$ $D_1 = (0, 13)$	(1,3) (1,14)	the missing data segment arrives: after merging it with its predecessor $D_1 = (0, 0)$ after merging $D_1 = (0, 1)$ with its successor $D_2 = (2, 13)$ into a single partial flow
5	'HTTP/1.0'=(14,21)	$D_1 = (0, 21)$	(1,17)	declare a **match** since 1 is the starting state of DFA and 17 is an accepting state

Table 4: Using the parallel algorithm to match the regular expression `''^(GET|HEAD|POST).*HTTP''` (see Figure 29) to a flow containing the data `'GET file.html HTTP/1.0'` split into 5 data segments: `'G', 'E', 'T', ' file.html ', 'HTTP/1.0'`.

If a successor $D_s = (S_s, E_s, L_s)$, such that $S_s = E_p + 1$, is found (else, proceed to the next arriving data segment):

- If both L_p and L_s are non-empty, update $S_s = S_p$, merge L_p into L_s and delete D_p from R.
- If Ls is empty, update $S_s = S_p$, merge L_p into L_s and delete D_p from R.
- If L_p is empty, update $S_s = S_p$ and delete D_p.

The merging procedure of the lists is as follows:

- For any pair of states (q_{sp}, q_{ep}) in L_p, if q_{ep} is an accepting state, copy (q_{sp}, q_{ep}) to L_s
- For each pair of states (q_{ss}, q_{es}) in L_s, not including those just copied from L_p:

  If there is a pair (q_{sp}, q_{ep}) in L_p such that $q_{ep} = q_{ss}$, delete (q_{ss}, q_{es}) from L_s and insert (q_{sp}, q_{es}) to L_s. Else delete (q_{ss}, q_{es}) from L_s.

Match detection: At any step of the algorithm, if a pair of states (q_s, q_e) such that q_s is the

starting state of the automaton and q_e is an accepting state is found in any of the L lists, label the flow as matching the regular expression.

Figure 30(a) shows an example of the merging procedure outlined above, and Table 3 demonstrates how the algorithm works on a very simple example.

6.6 The Parallel Algorithm

In the algorithm description above, that if we find no predecessor for the newly arrived data segment, we traverse the automaton with the segment, starting at each non-accepting state. This can be a performance bottleneck since the automaton can have a large number of states. In addition, the traversal process can result in a large number of pairs (q_s, q_e), and a significant number of those pairs can be duplicates ($q_{s1} = q_{s2}$ and $q_{e1} = q_{e2}$) stored in the different lists, or pairs with different starting states but identical ending states ($q_{s1} \neq q_{s2}$ and $q_{e1} = q_{e2}$).

Definition 4. *An **equivalence class** is a list of automaton state pairs that have different starting states but identical ending state, and is described as $Q = (l_s, q_e)$, where l_s is a list of starting states $(q_{s1}, q_{s2}, ..., q_{sk})$.*

In the example from Table 3, after traversing DFA with the content of packet 3 we have 5 pairs of states with identical ending state 14. The notion of equivalence classes allows us to replace those 5 pairs with a single equivalence class with the list of starting states $(4, 8, 12, 14, 16)$ and the ending state 14.

Thus, we improve the sequential algorithm by storing automaton state equivalence classes instead of state pairs. This would entail several changes as shown below.

6.6.1 Data Structure

For each element D_i of the list R we maintain the following information: (S_i, E_i) - the starting and ending offset of the reassembled data within the original data transmitted within the flow; L_i - the list of equivalence classes, describing the starting and ending states of paths within the automaton representing the regular expression that can be traversed with the data corresponding to D_i.

Figure 28(b) demonstrates a single object D_i of the list R. Each entry in the list L_i of the object D_i is an equivalence class $Q = (l_s, q_e)$ such that for each $q_s \in l_s, q_e = \delta(q_s, D_i)$.

6.6.2 Traversing DFA

Given a list of automaton states and a data segment d_i containing characters $x_1x_2...x_n$:

1. Attempt to make a transition from each of the states q_j with the first character x_1. Store all pairs of states (q_j, q_k), where $q_k = \delta(q_j, x_1)$, in a temporary list.

2. Find all pairs in the list with identical end states, delete them from the list and replace them with the corresponding equivalence class. As a result, we obtain a list of equivalence classes $Q_1 = (l_{s1}, q_{e1}), Q_2 = (l_{s2}, q_{e2}), ...,$with $|l_{si}| \geq 1$.

3. For each Q_i, attempt to make a transition $\delta(q_{ei}, x_2)$ unless q_{ei} is a final accepting state. If such transition exists, update $Q_i = (l_{si}, \delta(q_{ei}, x_2))$. Repeat the equivalence class merging procedure.

4. Repeat steps (2) and (3) until one of the following:

   - No new transition can be made on the next x_i.
   - End of the data segment d_i is reached. Return the resulting list of equivalence classes.
   - An equivalence class Q_i is obtained such that one of the states in l_{si} is the start state of the automaton, and q_{ei} is a final accepting state. Label the flow as a match of the regular expression.

6.6.3 Processing Data Segments

The procedure (both dealing with the first segment of the flow and the subsequent segments) is mostly identical to the sequential version of the algorithm, storing equivalence classes instead of pairs of states. The important difference in the parallel version is in the predecessor handling part of the algorithm, when the segment d_i arrives out of order:

Predecessor Processing: If there is no predecessor for d_i in R, create a new entry $D_p = (S_p = s_i, E_p = e_i, L_p =< empty\ list >)$ in R; Traverse the automaton using the modified

traversal procedure, with d_i and the list of all non-accepting states as an input. If the flow is not declared "matching", store the returned list of equivalence classes in L_p. A similar optimization can be applied for the case when a predecessor is found, but $|L_p|$ is large.

Successor Processing: We need to revise the merging procedure of two non-empty L lists when a successor is found. Here is a succinct description of the changes in the algorithm.

At the end of predecessor processing part of the algorithm, we have either merged the newly arrived segment d_i in an existing partial flow D_p, or created a new D_p based on d_i. If a successor $D_s = (S_s, E_s, L_s)$, such that $S_s = E_p + 1$, is found in R, and $|L_p| > 0$ and $|L_s| > 0$, we merge the predecessor and the successor into one partial flow by updating $S_s = S_p$, merging L_p into L_s and deleting D_p from R. The merge procedure of the L lists works as follows:

- For each equivalence class in the successor $Q_j = (l_{s_j} = (q_{sj1}, q_{sj2},), q_{ej}) \in L_s$, find all predecessor equivalence classes that end at one of the starting states in Q_j, that is $Q_k = (l_{sk}, q_{ek}) \in L_p$ such that $q_{ek} \in l_{sj}$. Merge such classes into L_s: for each such Q_k, delete q_{ek} from l_{sj}, and merge l_{sk} to l_{sj}. Delete Q_k from L_p.

- For each Q_j in L_s, delete all such starting states in l_{sj} that do not match any of the ending states in any of the predecessor equivalence classes.

- If there is a successor equivalence class $Q_j \in L_s$ and a predecessor equivalence class $Q_k \in L_p$ such that they both end at the same accepting state $q_{ej} = q_{ek}$, replace the starting list l_{sj} with the preceding class starting list l_{sk}. Delete Q_k from L_p.

- If, after completing all previous steps, there is an equivalence class $Q_k \in L_p$ such that it ends at a final accepting state, copy it to L_s and delete it from L_p.

Figure 30(b) shows an example of the merging procedure outlined above and Table 4 demonstrates how the algorithm works on the example from the previous section. The packet processing described in Table 3 now changes as follows: after processing packet 3, a number of state pairs with the end state 14 is replaced by a single equivalence class $((4, 8, 12, 14, 16), 14)$. Similarly, after processing packet 4, D_2 contains a single equivalence class $((3, 4, 8, 11, 12, 13, 14, 15, 16), 14)$.

6.6.4 The Mixed Version

The parallel version of the algorithm significantly reduces the amount of states that needs to be maintained at each step of the algorithm. However, the structure that maintains the states - a list of equivalence class objects - is now more complex, and therefore the overhead of accessing and updating an equivalence class in the list is more significant. To achieve a better tradeoff, we have developed a simple hybrid that integrates both the sequential and the parallel versions of the algorithm. The mixed algorithm will still take advantage of the equivalence classes while improving the parallel algorithm's overall performance.

- For any out of order data segment d_i, run the parallel version of the algorithm for k steps, processing k first characters in d_i and obtaining a list of equivalence classes.
- Run the sequential version of the algorithm with the remaining characters in d_i, starting from every equivalence class' ending state q_e .

In this approach, we assume that running the parallel version of the algorithm for the first k input characters will yield a limited amount of equivalence classes, thus reducing the amount of states starting from which we apply the sequential version of the algorithm.

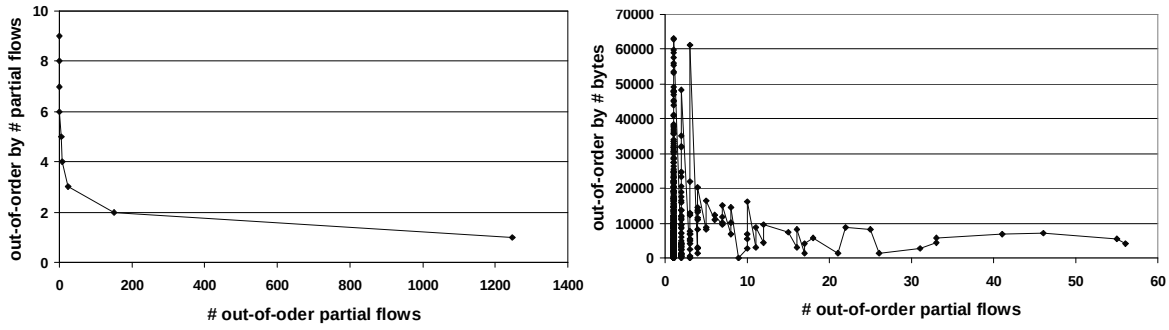
6.7 Experimental Study

6.7.1 Out-of-order Packets: Statistics

The algorithm we developed aims at dealing with out-of-order data segments and strives to minimize the amount of space used to store the information about partial flows with minimal CPU overhead. To demonstrate that the buffering approach described in section 6.4 may be prohibitively expensive, our first experiment attempted to estimate the memory requirements for buffering. We collected a set of $283,139$ distinct TCP flows ($4,154,108$ packets in total). The set contained $1,439$ out-of-order packets, for which we calculated the size and the number of the partial flows needed to be stored until they are merged with the prefix partial flow.

The results on figures 31 (a) and 31 (b) show that the maximal number of buffered partial flows observed is 9 consisting of $46,920$ bytes. We also observe that the largest buffered

partial flow is of size $63,090$ bytes, which is a quite significant amount of memory to be used during the matching process. Average size of the buffered partial flows was $7,860$ bytes.



(a) (b)

Figure 31: Statistics of buffered partial flows: (a) Number of buffered partial flows to maintain while waiting for the arrival of an out-of-order segment. (b) Size of the buffered partial flows (in bytes) to maintain while waiting for the arrival of an out-of-order segment.

6.7.2 Comparing Algorithms

In order to compare the three versions, we collected a set of data from our research center's network connection sent in TCP packets with either the source or the destination port 80, with the total of $5,565$ data segments. We simplified the study by supporting only a limited subset of regular expression language, and by simply replacing every occurrence of '.*' with a set of all supported characters. We have not invested significant efforts in DFA minimization; neither have we concentrated on the optimization of the automaton data structures and related code. With these optimizations performance of our algorithms will be still better he reasoning behind this decision was that even with the extensive research done in the area of DFA minimization and optimization, the amount of states in automata representing complex regular expressions is still very large. Therefore the questions we focused on in the presented work are those of initial evaluation and comparison of the three versions of the proposed algorithm. The objectives of automata minimization and optimization are being pursued in further study of this subject.

We tested our implementation on four regular expressions, chosen in part to match some of the data segments in the two data segment sets we worked with:

	Seq	**Par**	**Mix** $k=1$	**Mix** $k=2$	**Mix** $k=3$	**Mix** $k=4$	**Mix** $k=10$	**Mix** $k=100$	**Mix** $k=200$
Regex 1	1:00	1:30	0:04	0:04	0:04	0:05	0:06	0:16	0:25
Regex 2	2:26	1:32	0:05	0:05	0:06	0:06	0:07	0:20	0:30
Regex 3	19:18	9:23	0:19	0:17	0:17	0:18	0:21	1:30	2:44
Regex 4	20:00	9:25	0:19	0:16	0:17	0:17	0:21	1:30	2:52

Table 5: Out-of-order DFA traversal time of the different versions of the algorithm (in minutes and seconds).

Regex 1: `^HTTP/1.[01].*[0-5][0-1][0-9]` - match an HTTP response message.

Regex 2: `^(OPTIONS|GET|HEAD|POST|PUT|DELETE|TRACE|CONNECT).*HTTP/1.[01]` - an HTTP request message.

Regex 3: `HTTP/1.[01].*User-Agent: Mozilla/[45].0` - messages generated by Mozilla versions 4.0 or 5.0.

Regex 4: `HTTP/1.[01]Host:.*google.co.uk` - messages with the Host header matching `google.co.uk`.

It is important to notice that the last two regular expressions start with the implicit '.*' and have another '.*' in the middle.

The DFA's built for each of these regular expressions contained 109, 134, 214 and 212 states respectively. There were 451 data segments within the data set that matched the first regular expression, 454 that matched the second, 356 the third and 119 the forth.

The timing experiments described below were performed on a 2.8GHz processor server.

6.7.3 Out-of-order DFA Traversal Time

In this experiment we compare the running time of the out-of-order traversal procedure of the three versions of the algorithm, when traversing the DFA for each of the regular expressions as if every data segment of the set had arrived out-of-order. The motivation behind the test is that the out-of-order traversal procedure is the bottleneck of the algorithm and the algorithm with the minimal out-of-order traversal time is the most efficient one. For the mixed version, we ran it with different values of k in order to find its optimal value. The results are presented in Table 5. The results for values of k from 1 through 10 are also plotted in Figure 32.

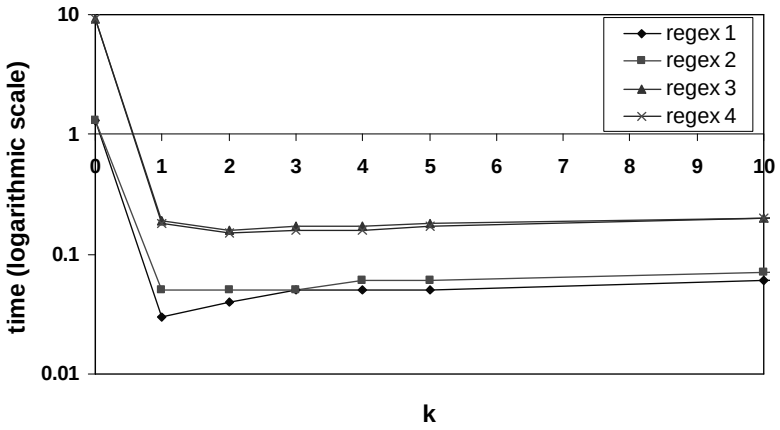


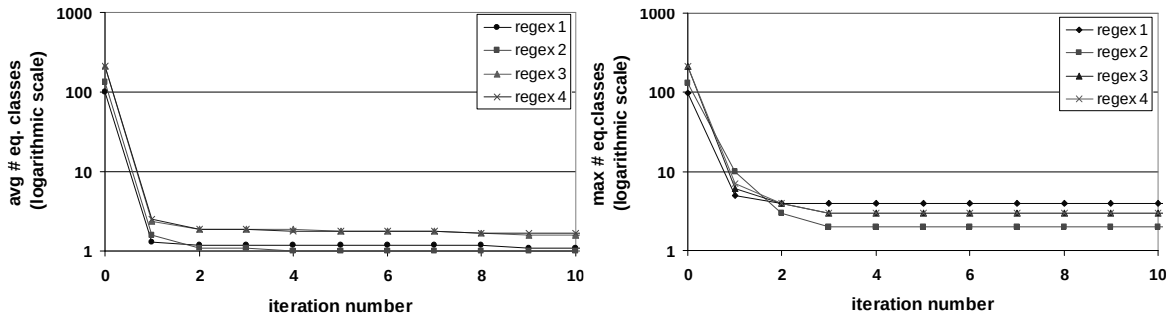
Figure 32: Running time of the out-of-order traversal procedure of the mixed version for various values of k.

The results demonstrate that the parallel version of the algorithm outperforms the sequential version by more than 50%, and that the mixed version is exceedingly faster than the sequential or the parallel for any value of k we used, with $k = 1$ yielding the best results for the two regular expressions with the starting anchor and a single '.*', and $k = 2$ or 3 for the two regular expressions that contained two '.*'s. We observe that as we increase the value of k, the traversal time grows as well. Thus the optimal value of k roughly equals the number of '.*'s within the regular expression being matched.

6.7.4 Size of the Equivalence Classes

To investigate the convergence rate of the number of equivalence classes we need to maintain on each step of the parallel version of the DFA traversal procedure for an out-of-order packet, we collected this statistics while matching the data segment set with each of the four regular expressions.

The graph on Figure 33 (a) shows the average number of equivalence classes at every step of the automaton traversal procedure. It can be seen that the number drops from hundreds to one or two, with the convergence rate for regular expressions starting with '.*' being slightly slower. Again, we can see that the average number of equivalence classes roughly equivalent to a number of '.*'s within a regular expression. The graph on Figure 33 (b) shows the maximal number of equivalence classes at each iteration. The number drops from hundreds to at most 10 after the first iteration and to at most 4 after the second iteration. These results confirm the observation from the previous experiment that $k = 1$



(a) Average number of equivalence classes. (b) Maximal number of equivalence classes.

Figure 33: Convergence rate of equivalence classes

yields the best results in the mixed version of the algorithm for regular expressions with the starting anchor, and $k = 2$ or 3 for regular expressions starting with '.*'.

It is important to note that since each partial flow contains at least one equivalence class, the number of equivalence classes that needs to be maintained at each step of the algorithm can be thought of as corresponding to the number of partial flows that the algorithm maintains at each step. We can therefore see from the experiment above that the average number of maintained partial flows is very low.

6.7.5 Rate of Data Processing

In order to estimate the rate at which the different versions of the algorithm can process data and to calculate the maximal number of partial flows during the matching process of a flow, we collected a large set of data from $14,142$ flows with $193,412$ number of packets in total. We ran the data on the four regular expressions described in section 6.7.2, with $3,624$, $3,644$, $3,568$ and 66 flows matching the four regular expressions respectively. To make a rough comparison of the three versions of the algorithm with the simplistic buffering version that maintains partial out-of-order flows in memory, we reconstructed the original message of each flow into a single data segment and ran it through the matching process as well. The results of the experiment are presented in Table 6

The results show that the mixed version gives by far the best results. It is slower than the buffering algorithm, however the difference is not significant, considering that in our experiment we did not take into account the cost of partial flows reassembly. It is clear, however, that the mixed version has significant advantage over buffering in the amount of

	Seq	**Par**	**Mix** $k=1$	**Mix** $k=2$	**Mix** $k=3$	**Buff**
Regex 1	0:04	0:02	0:01	0:01	0:01	0:01
Regex 2	0:47	0:03	0:01	0:01	0:01	0:01
Regex 3	1:39:26	5:07	2:23	2:23	2:23	1:47
Regex 4	1:37:57	5:08	2:26	2:26	2:26	1:49

Table 6: Time of regular expression matching (in hours, minutes and seconds) on a data set of $14,142$ flows consisting of $193,412$ packets.

memory needed to process out-of-order packets. We discuss a method for further memory use reduction next.

6.7.6 Memory Requirements

The motivation for this work is to avoid the need to store the payloads of out-of-order segments. However to do so, we need to store a summary of the state-to-state transitions after processing a packet. So, we need to quantify this space overhead.

We have two options for storing the state-to-state transition summaries. Let S be the number of states in the DFA, and E be the (expected) number of equivalence classes left after processing a packet.

1. Assuming no more than 2^{16} DFA states, we can store an array of S short integers, indicating the ending state for each start state. This approach requires $2S$ bytes.

2. Since there are usually very few equivalence classes after processing a packet, we can try a different approach. For each equivalence class, we can record the ending state, and a bitmap of the starting states in the equivalence class. This approach requires $E(2 + \lceil S/8 \rceil)$ bytes.

Option 2 is better than option one as long as $E < 16$, which is true for all but the most complex regular expressions. After processing a packet, regex's 1 and 2 had an average of 1.1 equivalence classes, while regex's 3 and 4 had an average of 2.1 equivalence classes. Using 109, 134, 214 and 212 states for the four regex's respectively, we obtain memory requirements of 16, 19, 61, and 61 bytes, respectively. Using the average size of the buffered partial flows obtained in the experiment from Section 7.1 ($7,860$ bytes) we achieve a space reduction of more than 130 to 1 over the naive buffering approach. Actual savings will

be considerable higher, since we can use a single summary to represent an out-of-order segment, which consists of several consecutive out-of-order segments.

6.8 Related Work

Matching regular expressions to strings is a classical area of study from the beginnings of Computer Science and has applications in parsers and text editors. The theory of regular expressions and their relationship to automata can be found in textbooks [84]. In particular, the algorithms for converting a regular expression to a DFA and its minimization is in [84]. A number of the most recent studies introduces an even more compact representation for regular expressions, called the Delayed Input DFA (D^2FA) [94] and Content Addressed Delayed Input DFA (CD^2FA) [95].

Application of regular expressions as signatures in monitoring IP contents is recent. In [116], authors studied various networking protocols and applications in depth to determine suitable signatures for them. They did not solve the problem of matching signatures across segments. We have used their application signatures in this study. Snort [10] is an intrusion-detection application that has a compiled list of several regular expression signatures to match attacks and intrusions. Snort systems use Perl Compatible Regular Expressions (pcre) [9] for regular expression matching which is performed on reassembled packet streams. In networking community, there is significant amount of work on matching regular expression signatures to IP packet streams, using specialized hardware like FPGAs [117, 20, 55]. Even these systems rely on full TCP reassembly. There is added focus on matching multiple regular expressions, but the focus has been on grouping multiple regular expressions to eliminate common states [131]. We are not aware of any Snort systems or specialized hardware solutions in networking that matches regular expression signatures within the network in presence of out-of-order packets on the stream, without reassembly.

In [99] the effect of out-of-order packets on window aggregation queries have been studied. The issues of data quality problems with IP packet streams, ie., out-of-order and duplicate packets, are well-known. A recent work [56] presented statistics on the occur-

rences of this phenomena, that matches our experience with real data.

6.9 Conclusions

We studied the problem of matching a regular expression to a data stream in presence of data quality problems such as duplicates and out-of-order packets. This is a well-motivated problem in managing IP networks where regular expressions are signatures that have to be matched against the contents of flows to detect intrusions, worms or viruses, applications and protocols. Prior work either matched regular expressions against the data segments on individual packets (which misses regular expressions that match across the segments) or reassembled the entire flow to match the regular expression using standard methods (which is highly resource -intensive). In fact, in networking, prior work has involved solving this problem in specialized hardware. Instead, we have proposed streaming algorithms that can be run in software that match regular expressions across segments even in presence of out-of-order packets and duplicates by carefully optimizing the state maintained on partial flows. Our experimental study with real data shows that the algorithms are successful in limiting the memory used and are efficient. These algorithms are more generally applicable for other data streams that produce duplicated or out-of-order data such as time series in sensor networks and tex streams.

The ending anchor “$” is analogous to the starting anchor and forces the match of the end of the regular expression to the end of the string and is not common in regular expressions applied to IP network monitoring. Support for it on data streams would require the ability to detect the end of the flow, which is a nontrivial task. We can accomplish that by using the heartbeat mechanism [89], but it is beyond the scope or need of the motivation for the work here.

Chapter 7

7 Concluding Remarks and Future Work

7.1 Summary of Contributions

With the growth in popularity and complexity of streaming applications, there's a rising need for more sophisticated analyses of massive high speed data generated by such applications. Such analyses often needs to be performed in near real-time, using limited system resources. Under such conditions, it is very important to find appropriate balance between the efficiency of processing and the accuracy of the produced results. A common technique is to filter the stream with suitable conditions so that the resulting data size is manageable, and the analyses are still accurate.

The work presented in this thesis focused on a number of complex filtering techniques that are of interest in data steam processing in general and in network traffic monitoring in particular. These techniques allow the analyst to define a filtering condition that is going to be more appropriate for the particular query at hand than the simpler random uniform sampling.

Data stream sampling is one of the widely used filtering techniques and in chapter 3 we design a single streaming operator which can be specialized for a wide variety of sophisticated stream sampling algorithms. The operator was implemented and tested in the Gigascope DSMS. It imposes only a small CPU overhead compared to a simple selection operator and scales in performance to line speeds. The significance of this contribution is

that this operator is a simple way in which sophisticated streaming algorithms that return set values can be integrated in to the query processing system. In addition to capturing a common thread of query evaluation, the operator is quite elaborate and is able to maintain information about groups, supergroups, aggregates and superaggregates, which gives the analyst the level of flexibility required for implementation and customization of various stream sampling algorithms.

Additionally, we have proposed a solution for flow sampling mechanism, which integrates the logic of flow aggregation as well as flow sampling into one procedure that works directly on the IP traffic. This solution works at speeds of more than 200k+ packets per second with only moderate load on the CPU and may also be the only viable way during adverse traffic conditions when the number of flows increases significantly.

The sampling algorithms described in chapter 3 offer solutions for evaluation of various properties of data streams in terms of forward distribution. In contrast, in chapter 4 we introduced and formalized the notion of the inverse distribution for massive data streams. The main contribution of this work is a novel technique that draws and dynamically maintains a uniform sample of items in the presence of not only insertions, but also deletions, with provable guarantees. Such sample can provide solution to a number of inverse distribution problems, such as heavy hitters, quantiles and range queries. In more general terms, this filtering technique provide a different insight on into various aspects of data streams than techniques that work with forward distributions.

A common type of query on data streams that searches for records matching a dynamic involves a self-join, which might be hard to evaluate in an efficient and stable manner under adversary conditions without significantly compromising accuracy of the results. In chapter 5 we addressed this problem by introducing a filter join operator which, unlike conventional join and self-join, has inexpensive implementation, and can be used to answer this class of filtering queries. We also presented analyses of query transformations which expose the filter join operator in conventional query join. We implemented the operator in Gigascope DSMS, tested it on live network streams and found order-of-magnitude performance improvements when compared to traditional hash join.

Finally, in chapter 6 we studied the problem of matching regular expression that spans multiple data records in a data stream in the presence of duplicates and out-of-order records. Prior work in this area only addresses this problem by either matching regular expressions against data that fits into a single record, reassembling the entire message before matching, or by utliizing specialized hardware. We presented a number of algorithms that can match regular expressions over multiple data stream items without reassembly, by maintaining partial state of the data in the stream. Our experimental study showed that the algorithms are efficient, achieving a high compression of active state by comparison to other methods that reassemble the entire message, while being comparable the running time.

7.2 Directions for Future Research

Work presented by this thesis can be further extended as follows:

- **Pushing down the sampling operator**: In the study presented in chapter 3, the sampling operator is placed at the higher level of processing, while the initial simple filtering of the data is performed at the lower level. This is done because the operations performed by the sampling procedure can be quite elaborate and very expensive to evaluate on every imcoming record of a high speed data stream. However, it's worthwhile researching whether it is possible to push a part of the sampling procedure evaluation into the lower levels of processing. To answer this question, more extensive study is required in understanding sampling procedures and their common pattern of evaluation.

- **Other types of algorithms with sampling framework**: The operator presented in chapter 3 was developed to handle a large class of data stream sampling algorithms that follow a particular common pattern of evaluation. An interesting question is - are there other types of algorihtms that would fit into the presented sampling framework or framework similar to that?

- **Complex queries on inverse distribution**: The study of inverse distribution presented by this thesis shows generation of a sample of distinct items, which can be used in answering a number of inverse distribution queries, including heavy hitters, range

queries and quantiles. However, it remains open to answer more complex queries over inverse distribution, such as computing frequency moments or detecting anomalies.

- **Forward distribution with insertions and deletions**: A fundamental question that arises is to design algorithms to maintain a uniform sample of the *forward* distribution under *both* insertions and deletions over data streams or show that this is impossible - as noted in the chapter 4, no existing algorithms guarantee to return a non-empty sample in this setting.

- **Extending filter join to full join**: The filter join procedure described in chapter 5 was developed with the assumption that all flow records appear on a single link and therefore no tuple synchronization is necessary. One natural continuation of this work is to extend filter join procedure to a full join of two independent streams. In the latter scenario a more complex procedure for tuple synchronization is required.

- **Optimization of regular expression matching**: Work described in chapter 6 focused on the development of an efficient procedure of DFA traversal and state maintenance, however no effort was invested in other optimizations of the procedure. For instance, it would be interesting to explore integration of newly proposed compact DFAs, such as D^2FA [94] (Delayed Input DFA) or CD^2FA [95] (Content Addressed Delayed Input DFA) with the algorithms described by this thesis.

References

[1] Application layer packet classifier for linux: http://l7-filter.sourceforge.net.

[2] http://ipmon.sprint.com.

[3] http://www.cisco.com.

[4] http://www.gemstone.com.

[5] http://www.streambase.com.

[6] http://www.traderbot.com.

[7] Informal discussions with AT&T network analyst.

[8] Internet traffic archive: http://ita.ee.lbl.gov/.

[9] PCRE - Perl compatible regular expressions: http://www.pcre.org.

[10] Snort network intrusion detection system: http://www.snort.org.

[11] SQL server 2005: http://www.microsoft.com/technet/prodtechnol/sql/2005/evaluate/dwsqlsy.mspx.

[12] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Rasin, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. *CIDR*, pages 277–289, 2005.

[13] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB*, 12(2):120–139, 2003.

[14] D. Abadi, W. Lindner, S. Madden, and J. Schuler. An integration framework for sensor networks and data stream management systems. *VLDB*, 2004.

[15] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. *ACM SIGMOD*, pages 487–498, 2000.

[16] M. Ali, W. Aref, R. Bose, A. Elmagarmid, A. Helal, I. Kamel, and M. Mokbel. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. *VLDB*, pages 1295–1298, 2005.

[17] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[18] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. The stanford stream data manager. *IEEE Quarterly Bulletin in Data Engineering*, 26(1):19–26, 2003.

[19] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB*, 15(2):121–142, 2006.

[20] M. Attig and J. W. Lockwood. SIFT: Snort intrusion filter for TCP. *Hot Interconnects*, pages 121–127, 2005.

[21] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. *PODS*, pages 1–16, 2002.

[22] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. *SODA*, pages 633–634, 2002.

[23] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. *ICDE*, pages 350–361, 2004.

[24] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD*, 30(3):109–120, 2001.

[25] A. Bagchi, A. Chaudhary, D. Eppstein, and M. T. Goodrich. Deterministic sampling and range counting in geometric data streams. *Symposium on Computational Geometry*, pages 144–151, 2004.

[26] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB*, 13(4):370–383, 2004.

[27] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.

[28] S. Bhattacharyya, A. Madeira, S. Muthukrishnan, and T. Ye. How to scalably skip past streams. *WSSP Workshop with ICDE*, 2007.

[29] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[30] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. *MDM*, pages 3–14, 2001.

[31] K. Bratbergsengen. Hashing methods and relational algebra operations. *VLDB*, pages 323–333, 1984.

[32] A. Broder. On the resemblance and containment of documents. *IEEE Compression and Complexity of Sequences*, pages 21–29, 1997.

[33] D. Carney, U. etintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. *VLDB*, pages 215–226, 2002.

[34] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[35] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *VLDB*, 10(2-3):199–223, 2001.

[36] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous data processing for an uncertain world. *CIDR*, pages 269–280, 2003.

[37] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. *ACM PODS*, pages 268–279, 2000.

[38] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. Overcoming limitations of sampling for aggregation queries. *ICDE*, pages 534–544, 2001.

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

[40] S. Chaudhuri, R. Motwani, and N. Narasayya. On random sampling over joins. *ACM SIGMOD*, pages 263–274, 1999.

[41] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. *ACM SIGMOD*, pages 379–390, 2000.

[42] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network: Model and algorithms. *ACM SIGMOD*, pages 707–718, 2004.

[43] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. *PODS*, pages 223–233, 2003.

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

[45] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM PODS*, pages 296–306, 2003.

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

[47] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. *PODS*, pages 271–282, 2005.

[48] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. *VLDB*, pages 25–36, 2005.

[49] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. *ACM SIGKDD*, pages 9–17, 2000.

[50] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: High performance network monitoring with an SQL interface. *ACM SIGMOD*, page 262, 2002.

[51] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. *ACM SIGMOD*, pages 647–651, 2003.

[52] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. The Gigascope stream database. *In IEEE Data Engineering Bulletin*, 26(1):27–32, 2003.

[53] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *ACM SIAM*, 2002.

[54] M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. *ESA*, pages 323–334, 2002.

[55] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[56] S. Dharmapurikar and V. Paxson. Robust TCP stream reassembly in the presence of adversaries. *USENIX Security Symposium*, pages 65–80, 2005.

[57] F. Douglis, J. Palmer, E. Richards, D. Tao, W. Hetzlaff, J. Tracey, and J. Lin. Position: short object lifetimes require a delete-optimized storage system. *ACM SIGOPS*, 2004.

[58] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. *Conference on Computer and Communications Security*, 2004.

[59] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. *ACM Conference on Computer and Communications Security*, pages 2–11, 2004.

[60] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. *ACM Conference on Computer and Communications Security*, pages 2–11, 2004.

[61] N. G. Duffield, C. Lund, and M. Thorup. Learn more, sample less: control of volume and variance in network measurement. *IEEE Transactions on Information Theory*, 51(5):1756–1775, 2005.

[62] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *ACM SIGCOMM*, 32(4):323–338, 2002.

[63] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[64] G. Frahling and C. Sohler. Coresets in dynamic geometric data streams. *ACM Symposium on Theory of Computing*, 2005.

[65] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. *ACM SIGMOD*, pages 265–276, 2003.

[66] V. Ganti, M.-L. Lee, and R. Ramakrishnan. ICICLES: Selftuning samples for approximate query answering. *VLDB*, pages 176–187, 2000.

[67] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. *ACM SIGMOD*, 2002.

[68] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. *ACM SIGMOD*, pages 331–342, 1998.

[69] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. *ACM Symposium on Parallel Algorithms and Architectures*, pages 281–290, 2001.

[70] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. *VLDB*, pages 541–550, 2001.

[71] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *VLDB*, pages 466–475, 1997.

[72] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. QUICKSAND: Quick summary and analysis of network data. *DIMACS TR 2001-43*.

[73] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. *VLDB*, pages 79–88, 2001.

[74] L. Golab and M. T. zsu. Issues in data stream management. *SIGMOD*, 32(2):5–14, 2003.

[75] L. Golab and M. T. zsu. Processing sliding window multi-joins in continuous queries over data streams. *VLDB*, pages 500–511, 2003.

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

[77] P. Gulutzan and T. Pelzer. SQL-99 complete, really, CMP books. 1999.

[78] M. Gyssens, L. V. S. Lakshmanan, and I. N. Subramanian. Tables as a paradigm for querying and restructuring. *PODS*, pages 93–103, 1996.

[79] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *ACM SIGMOD*, pages 287–298, 1999.

[80] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. *VLDB*, pages 311–322, 1995.

[81] M. Hammad, M. Mokbel, M. Ali, W. Aref, A. Catlin, A. Elmagarmid, M. Eltabakh, M. Elfeky, T. Ghanem, R. Gwadera, I. Ilyas, M. Marzouk, and X. Xiong. Nile: a query processing engine for data streams. *ICDE*, page 851, 2004.

[82] J. M. Hellerstein, W. Hong, and S. Madden. The sensor spectrum: Technology, trends, and requirements. *ACM SIGMOD*, 32(4):22–27, 2003.

[83] J. Hershberger and S. Suri. Adaptive sampling for geometric problems over data streams. *PODS*, pages 252–262, 2004.

[84] J. Hopcroft and J. Ullman. Introduction to automata theory, languages, and computation. *Addison-Wesley, New York*, 1979.

[85] W. C. Hou, G. Ozsoyoglu, and B. Taneja. Statistical estimators for relational algebra expressions. *ACM PODS*, pages 276–287, 1988.

[86] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. *VLDB*, pages 174–185, 1999.

[87] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. *ACM SIGMOD*, pages 299–310, 1999.

[88] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. *SIGMOD*, pages 1–12, 2005.

[89] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in Gigascope. *VLDB*, pages 1079–1088, 2005.

[90] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. Query-aware sampling for data streams. *WSSP Workshop with ICDE*, 2007.

[91] T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Streams, security and scalability. *DBSec*, pages 1–15, 2005.

[92] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. *ICDE*, pages 341–352, 2003.

[93] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. *USENIX Security Symposium*, pages 271–286, 2004.

[94] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. S. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM*, pages 339–350, 2006.

[95] S. Kumar, J. S. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. *ANCS*, pages 81–92, 2006.

[96] E. Kushilevitz and N. Nisan. Communication complexity. *Cambridge University Press*, 1997.

[97] Y. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. *VLDB*, pages 492–503, 2004.

[98] A. Lerner and D. Shasha. The virtues and challenges of ad hoc + streams querieng in finance. *IEEE Quarterly Bulletin in Data Engineering*, 26(1):49–56, 2003.

[99] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. *SIGMOD*, pages 311–322, 2005.

[100] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. *ICDE*, pages 555–566, 2002.

[101] D. Maier, P. Tucker, and M. Garofalakis. Filtering, punctuation, windows and synopses. *In Stream Data Managenement, Chapter 3. Springer*, 2005.

[102] G. Manku and R. Motwani. Approximate frequency counts over data streams. *VLDB*, pages 346–357, 2002.

[103] J. Mirkovic and P. Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *Computer Communication Review*, 34(2):39–53, 2004.

[104] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2):115–139, 2006.

[105] R. Motwani and P. Raghavan. Randomized algorithms. *Cambridge University Press*, 1995.

[106] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. *CIDR*, pages 245–256, 2003.

[107] S. Muthukrishnan. Data streams: Algorithms and applications.

[108] S. Nath, P. Gibbons, S. Seshan, and Z. Anderson. Synopsis diffusion for robust aggregation in sensor networks. *SenSys*, pages 250–262, 2004.

[109] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. *IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[110] F. Olken. Random sampling from databases. *PhD thesis, Berkeley*, 1997.

[111] K. Patroumpas and T. K. Sellis. Window specification over data streams. *EDBT Workshops*, pages 445–464, 2006.

[112] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. *SSDBM*, pages 24–33, 1999.

[113] F. Reiss and J. M. Hellerstein. Data triage: An adaptive architecture for load shedding in TelegraphCQ. *ICDE*, pages 155–156, 2005.

[114] I. Rozenbaum, T. Johnson, and S. Muthukrishnan. Filter join on data streams. *ICDM workshop DSMM*, 2007.

[115] I. Rozenbaum, T. Johnson, and S. Muthukrishnan. Monitoring regular expressions on out-of-order streams. *ICDE Poster*, 2007.

[116] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of P2P traffic using application signatures. *WWW*, pages 512–521, 2004.

[117] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[118] A. Singh. http://www.cs.ucsb.edu/ ambuj/courses/multimediadb/sampling.pdf.

[119] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. *OSDI*, pages 45–60, 2004.

[120] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. *USENIX*, 1998.

[121] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. *VLDB*, pages 309–320, 2003.

[122] N. Tatbul and S. B. Zdonik. Window-aware load shedding for aggregation queries over data streams. *VLDB*, pages 799–810, 2006.

[123] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *ACM SIGMOD*, pages 321–330, 1992.

[124] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, 2003.

[125] T. Urhan and M. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *In IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[126] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. *VLDB*, pages 285–296, 2003.

[127] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[128] H. Wang, C. Zaniolo, and C. Luo. ATLAS: A small but complete SQL extension for data mining and data streams. *VLDB*, pages 1113–1116, 2003.

[129] M. S. Wei Hong. Optimization of parallel query execution plans in XPRS. *PDIS*, pages 218–225, 1991.

[130] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[131] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. *ANCS*, pages 93–102, 2006.

[132] Y. Zhang, S. Singh, S. Sen, N. G. Duffield, and C. Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. *Internet Measurement Conference*, pages 101–114, 2004.

Curiculum Vita

Irina Rozenbaum

Education:

09/2003 - 10/2007	Rutgers University, NJ, Computer Science Ph.D. program. Research adviser: S.Muthukrishnan.
09/2000 - 01/2003	Rutgers University, NJ, B.Sc. in Computer Science. Highest Honors. GPA: 3.94

Work Experience:

06/2006 - 09/2006	Research Intern, AT&T Shannon Labs, Florham Park, NJ. Developed and implemented a data stream management system operator to enable efficient processing of join queries on data streams (C, FreeBSD).
06/2005 - 09/2005	Summer Intern, AT&T Shannon Labs, Florham Park, NJ. Developed a graphic user interface to enable real time visualization of data stream query results (Perl CGI, Linux).
07/2003 - 12/2003	Web/SQL Software Developer, Global eProcure, Clark, NJ. Developed new features and customized a web-based auction tool (ASP, JavaScript, MS SQL Server 2000).

Publications:

- ”Applying Link-based Classification to Label Blogs”, S.Bhagat, G.Cormode, I.Rozenbaum. WebKDD/SNA-KDD 2007.
- ”No Blog is an Island - Analyzing Connections Across Information Networks”, S. Bhagat, G. Cormode, S. Muthukrishnan, I. Rozenbaum, H. Xue, International Conference on Weblogs and Social Media (ICWSM) 2007.
- ”Monitoring Regular Expressions on Out-of-Order Streams.” T. Johnson, S. Muthukrishnan, I.Rozenbaum, poster at International Conference on Data Engineering (ICDE)

2007.

- ”Summarizing and Mining Inverse Distributions on Data Streams via Dynamic Inverse Sampling”. G. Cormode, S. Muthukrishnan, I.Rozenbaum. International Conference on Very Large Databases (VLDB) 2005.
- ”Sampling Algorithms in a Stream Operator”, T. Johnson, S. Muthukrishnan, I.Rozenbaum, Special Interest Group on Management of Data (ACM SIGMOD) 2005.