Efficient Analysis of Live and Historical Streaming Data and its Application to Cybersecurity

Frederick Reiss[1,2], Kurt Stockinger[1], Kesheng Wu[1],
Arie Shoshani[1], Joseph M. Hellerstein[2]

[1]Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, USA

[2]Computer Science Division
University of California, Berkeley
Berkeley, California, USA

July 13, 2006

Abstract

This paper describes our experiences building a coherent framework for efficient simultaneous querying of live and archived stream data. This work was motivated by the need to analyze the network traffic patterns of research laboratories funded by the U.S. Department of Energy. We review the requirements of such a system and implement a prototype based on the TelegraphCQ streaming query processor and the FastBit bitmap index. The combined system uses TelegraphCQ to analyze streams of traffic information and FastBit to correlate current behaviors with historical trends. We present a detailed performance analysis of our system based on a complex query workload and real network traffic collected at Lawrence Berkeley National Laboratory (Berkeley Lab). Our experiments identify key performance bottlenecks for stream query processing systems that incorporate historical data. We also identify strategies for mitigating these bottlenecks. With these strategies in place, we demonstrate that it is possible for our system to analyze the entire traffic of the DOE lab network on a small cluster of machines.

1 Introduction

Many applications need to process streaming data from devices ranging from temperature sensors to spy satellites [9]. In these applications, it is often essential to correlate

live data with historical trends [4]. Motivated by an important cybersecurity application, we have built a system that combines TelegraphCQ, a stream query processor, with FastBit, a system for mining append-only data. In this paper, we will show that our combined system can overcome a number of challenges and is able to efficiently process the complex queries expected from our target application.

The United States Department of Energy (DOE) operates nine major research labs nationwide. These laboratories conduct classified and unclassified research in areas such as high-energy physics, nuclear fusion and climate research. Researchers at the labs regularly collaborate with major university and industrial research organizations. To support collaborations both between the labs and with outside researchers, each lab maintains high-speed network connections to several nationwide networks. In addition, each lab publishes large amounts of information via its connection to the public Internet.

The security of these network connections is crucial to operations at the DOE labs. As a U.S. government agency, the Department of Energy is a prime target of malicious hackers worldwide. The laboratories need to protect sensitive information and to prevent illegal misuse of their equipment.

To help maintain network security, availability, and performance at its laboratories, DOE is creating a nationwide network operations center. This centralized monitoring station will enable a small team of network administrators to maintain a 24-hour alert for potential problems. Figure 1 illustrates how this system will work. At each laboratory, TCP/IP *flow monitors* collect information about network sessions and stream the resulting *flow records* to the operations center via a backbone link. Software at the operations center aggregates and analyzes the streams of flow records to present a real-time picture to the human operators on the scene.

1.1 Challenges

In this paper, we describe our experiences building a prototype of the analysis software that could be run at the network operations center. We identified three key requirements for this software:

- **Flexibility:** The network operator needs to focus analysis on the machines, patterns, and protocols that are relevant to the problem at hand. He or she will also need to develop and deploy new analyses quickly in response to evolving threats.
- **History:** An effective monitoring system needs access to historical data about the network. This past history allows the operator to weed out false positives by comparing present behavior against past behavior. History is also essential for determining the cause of a malfunction or security breach that occurred in the past.
- **Performance:** During periods of peak load, the DOE networks generate thousands of flow records per second. It is during these peak periods that effective network monitoring is most essential. High traffic puts stresses on the network that make it more likely to fail. Also, knowledgeable adversaries will attempt to hide their attacks inside these large bursts of data.

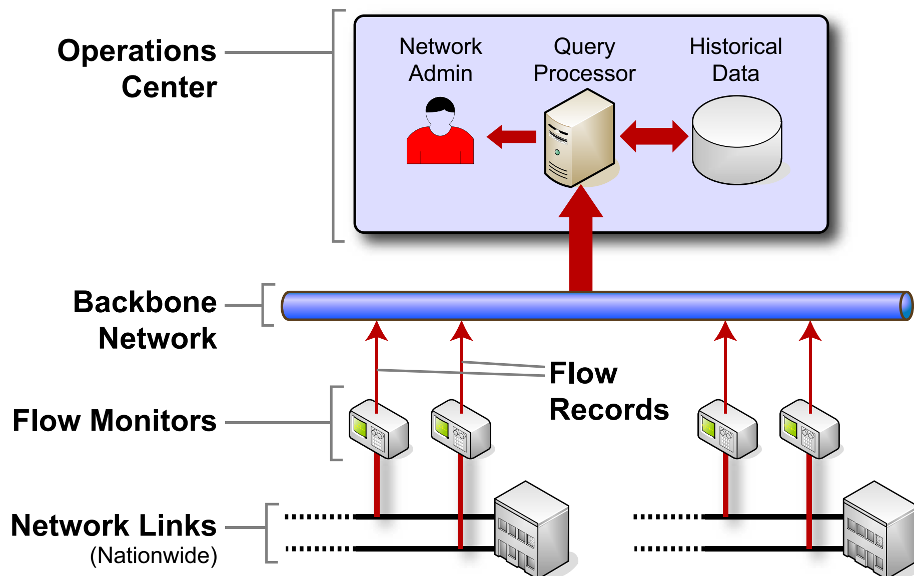

Figure 1: High-level block diagram of the proposed nationwide network monitoring infrastructure for the DOE labs.

The authors of this paper have been working on two late-stage research prototypes, TelegraphCQ and FastBit. Combined, these two systems meet the first two of our requirements. TelegraphCQ[5] is a streaming query processor with a flexible declarative language. FastBit, a bitmap index based query engine for append-only data, has previously demonstrated its effectiveness at searching over large amounts of historical networking data [2, 22].

The main goal of our work was to demonstrate that these two general purpose query processing systems can meet our third requirement: performance. We were concerned about performance both in terms of handling high data rates and in terms of running many complex queries simultaneously. These dimensions of performance represent significant potential problems.

1.1.1 Data Rates

The total traffic on the Department of Energy's networks is classified, but we can make a rough estimate based on unclassified data. We have obtained flow records from Berkeley Lab's unclassified NERSC (National Energy Research Scientific Computing) Center for a 42-week period from August 2004 through June 2005.

Figure 2 shows the number of flows (or network sessions) per week during this period. During the trace, the network generated as many as 250 million flow records in a week, or 500 records/sec on average. However, the rate at which these flows arrived varied significantly, reaching as high as 55,000 flow records per second as shown in

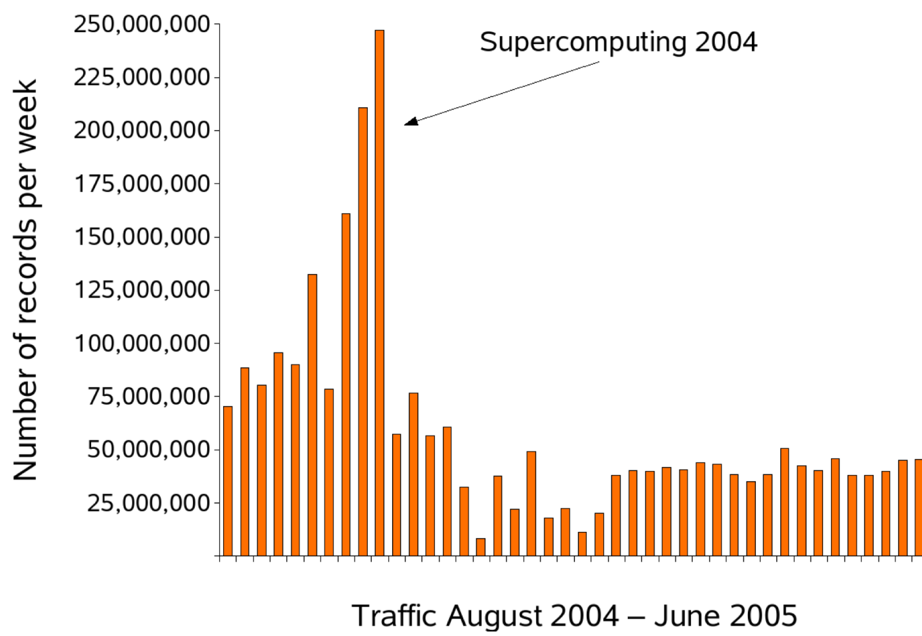

Figure 2: Flow records per week in our 42-week snapshot of Berkeley Lab's connection to the NERSC backbone.

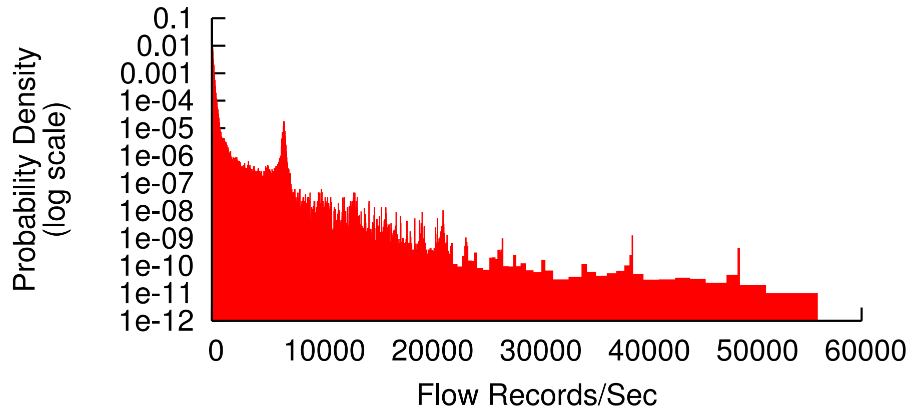

Figure 3: Histogram of the number of flow records per second in the data set in Figure 2. The distribution is heavy-tailed, with a peak observed rate of 55,000 records per second.

Figure 3. This figure shows the histogram of flow record rates. We observed that this distribution is heavy-tailed.

The NERSC traffic represents about one third of network connections at Berkeley Lab, whose employees are about one tenth of all DOE lab researchers. Assuming that other network links have approximately the same traffic levels as the NERSC link and that network usage scales with the number of researchers, we therefore expect the NERSC link to represent approximately 1/30 of the total DOE traffic. So a DOE-wide network operations center would need to process sustained rates of roughly $500 \times 30 = 15,000$ flow records per second, with peaks of as much as $55,000 \times 30 = 1,650,000$ records per second.

1.1.2 Query Complexity

Based on a literature survey and our own conversations with networking researchers, we have developed a sample workload of queries for monitoring networks. We describe this workload in detail in Section 5. Each of our queries has a live streaming component and a historical archive component. The live component of each query monitors the current status of the network. The historical parts analyze previous flow records to trace problems back to their roots or to determine whether potential anomalies are actually normal behavior. Running these queries requires real-time analysis and fast index access to as much as a month of historical data. We anticipate that a real deployment would run a mix of 10 to 100 such queries, monitoring various aspects of the network in real time.

1.2 Contributions

In this paper, we describe our experiences building a prototype network monitoring/-forensics system for monitoring the DOE backbone networks with declarative queries. To our knowledge, ours is the first robust end-to-end system studied in the research literature that performs declarative "live/archive" queries on high-bandwidth streams. Using declarative queries for network traffic analysis is significantly more convenient and productive than the common practice of using the C-like scripting languages supported by most network monitoring systems.

We evaluate our system on traces of real flow records from NERSC and present a detailed performance study that identifies performance bottlenecks and strategies for solving them. Our results demonstrate that, by using these strategies, our system can perform complex analyses of high-speed network traffic incorporating both live and historical data. Though this paper concentrates on the analysis of network traffic data, the system is general-purpose and can be applied easily to other applications that require simultaneous queries on live and historical stream data.

2 Related Work

A good overview of stream query systems can be found in [9]. Common to most of these systems is that they process continuous data streams that are often not stored persistently for subsequent analysis. Typical application areas for these systems are financial market analysis, inventory tracking and network traffic monitoring (see for instance Tribeca [23] or the Gigascope system [7]). The architecture of the TelegraphCQ system that we use in our work was described in CIDR 2003 [5]. Two previous studies have used TelegraphCQ to run simple network monitoring queries [15, 16]. In this paper, we embed TelegraphCQ in an end-to-end system and use a larger workload of more complex queries to get a more realistic view of TelegraphCQ's performance.

The objective of the work presented in [6] is to build a database system for analyzing off-line network traffic data for studying coordinated scan activities. Another recent database effort for analyzing network traffic is described in [21]. Both approaches use open source database systems and index data structures for efficient analysis of off-line network traffic data. However, these systems do not manage streaming data.

A combination of live and historic data processing is presented in [4]. Historic data is managed by a B-tree that is adaptively updated based on the query load. In order to keep up with high query loads, the B-tree updates are delayed for periods with lower traffic. Thus, historic queries can operate on reduced (sampled) data sets during high loads. This leads to approximate answers for queries on historical data, which may affect analysis results. Rather than B-trees, we use bitmap indices for querying historic data. One reason for this change is that the bitmap indices can answer multi-dimensional range queries quickly and accurately [13, 3, 25, 18, 26]. Such queries appear frequently in our network monitoring workload.

One of the motivations given in [4] for their work was that updating B-trees may take too much time to keep up with the arrival of new records. In this paper, we reexamine this assumption in the context of our bitmap indices and show that index insertions

do not need to be a bottleneck. Unlike B-trees, bitmap indices do not require sorting of the input data. This property allows for very efficient bulk append operations. Recent proposals for improving the write performance of B-Trees [10] may narrow this performance gap, but we are unaware of any hard performance numbers for the new designs.

In the network community, two commonly used Intrusion Detection Systems are Bro [14] and Snort [17]. These systems are used to analyze and react to suspicious or malicious network activity in real time. Recently Bro was extended by a concept called the *time machine* [12], i.e. to analyze historic data by traveling back in time. The high-level concept of a time machine is similar to the one described in this paper. However, the authors do not provide any details on the performance of querying historic data. The goal of our paper is to provide a detailed performance analysis on the combination of stream processing and historic data analysis. In addition, the analyses in Bro and Snort are performed through C-like scripting language with manual management of data structures, while our system uses high-level declarative queries. In contrast, similar scripts written in declarative languages such as SQL are much more compact and much easier to create.

Since TelegraphCQ is built on top of PostgreSQL, we briefly note the difference between the use of bitmaps in PostgreSQL and in FastBit. FastBit uses bitmap indices as primary storage, while PostgreSQL (as of version 8.1) uses in-memory bitmaps to store intermediate results, such as the results from scanning B-tree indices. The use of bitmaps to store intermediate results allows PostgreSQL to efficiently combine results from different index scans. However, PostgreSQL does not use bitmaps for on-disk index storage, and previous work [4] has shown that the current PostgreSQL B-Tree indices are inefficient for storing high volumes of append-only data.

FastBit implements a set of compressed bitmap indices using an efficient compression method called the *Word Aligned Hybrid* (WAH) code [25, 26]. In an number of performance measurements, WAH compressed indices were shown to significantly outperform other indices [24, 25]. Recently, FastBit has also been used in analysis of network traffic data and shown to be able to handle massive data sets [22, 2]. This makes it a convenient choice for our prototype.

In this paper, we focus on finding out how much data a single computer can handle. As the number of network connections increases, it would become necessary to use a distributed architecture like that of CoMo [11] or HiFi [8] to overcome resource and bandwidth constraints. There are a number of ways to parallelize TelegraphCQ to handle the stream volume [19], or alternatively the system can use Data Triage [16] to trade off result accuracy for speed. In terms of handling historical data, FastBit has been shown to parallelize well [22]. Overall, we anticipate the system we propose can be parallelized effectively on a modest cluster of computers.

3 Architecture

This section gives a high-level overview of the architecture of our prototype monitoring system and provides references for further reading on individual components of our system. Figure 4 shows how the pieces of our architecture fit together. The major

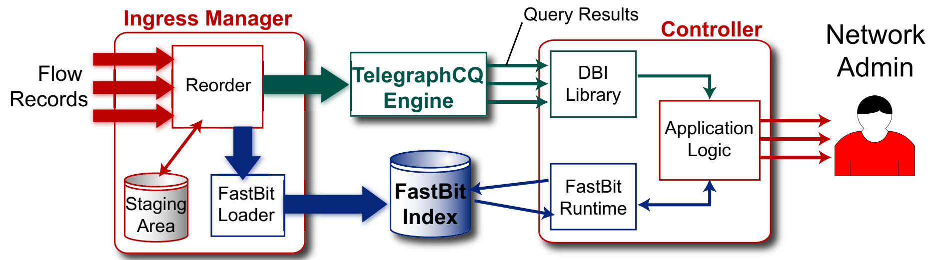

Figure 4: Block diagram of our network monitoring prototype. Our system combines the data retrieval capabilities of FastBit with the stream query processing of TelegraphCQ.

components of the system are as follows:

TelegraphCQ is a streaming query processor that filters, categorizes, and aggregates flow records according to one or more continuous queries, generating periodic reports. TelegraphCQ accepts continuous queries in the CQL query language [1]. TelegraphCQ's query language features include windowed joins, grouped aggregation, subqueries, and top-K queries. TelegraphCQ is based on the open-source PostgreSQL database engine and supports PostgreSQL's user-defined types, user-defined aggregates, storage manager, and application APIs. The work in this paper uses the latest development version of TelegraphCQ, which incorporates a new data ingress layer, additional query language support, and substantial performance improvements.

FastBit is a bitmap index system with an SQL interface. It is designed to work with append-only data sets such as historical records of a network monitoring system. In this work, we mainly use its command line interface to provide fast associative access to archived network monitoring data. The keys to its efficiency are its vertical data organization and the compressed bitmap indices [20, 25, 26].

The **Ingress Manager** is a component that merges incoming streams of flow records, converts data into formats that TelegraphCQ and FastBit understand, and stages data to disk for loading into FastBit.

The **Controller** is a component that receives streaming query results from TelegraphCQ, requests relevant historical data from the FastBit index, and generates concise reports for the network administrator. Each analysis that the controller performs consists of three parts: A TelegraphCQ query template, a FastBit query template, and application logic. The Controller reads TelegraphCQ and FastBit query templates from configuration files and substitutes in runtime parameters. The application logic for an analysis consists of a callback that is invoked for each batch of query results that comes back from TelegraphCQ. Each analysis runs in a separate process, but the Controller's admission control limits the number of concurrent FastBit queries to prevent thrashing.

The controller is currently implemented in the Perl programming language. We chose Perl because it has an efficient interface with TelegraphCQ, dynamic compilation for loading application logic at runtime, and well-developed facilities for generating reports.

The major challenge in the combined system of TelegraphCQ and FastBit is to meet the performance requirement to handle the expected volume of network flow records and complex queries that involve both live and historical stream data. In this paper we analyze the performs of all above four components to identify bottlenecks and study how to mitigate the problems. Ultimately, we may build the intelligence into the controller so that some of the decisions can be made automatically.

4 Data

All our performance benchmarks are based on real network connection data collected on the NERSC backbone connection at Berkeley Lab. The data represents a time period of 5 weeks and consists of 121 million records. Each flow record contains information on the source/destination IP addresses, source/destination ports, time stamp, packet size, etc. In total, each record contains 11 attributes.

Table 1 shows the number of network traffic records per week along with the cumulative number of records. Note that week 1 comprises about 5 times more records than week 4.

Week	Records/week	Cumulative no. of records
1	49,289,726	49,289,726
2	17,826,252	67,115,978
3	22,315,221	89,431,199
4	11,402,761	100,833,960
5	20,256,664	121,090,624

Table 1: Number of network traffic records collected at Berkeley Lab over a period of 5 weeks.

Each flow record in our data set contained 11 attributes. We mapped these attributes into the following TelegraphCQ stream:

```
create stream Flow(
    src_ip      Inet,
    dst_ip      Inet,
    src_port    integer,
    dst_port    integer,
    protocol    integer,    -- Protocol number
    bytes_sent  integer,    -- src->dst
    bytes_recv  integer,    -- dst->src
    outgoing    boolean,
        -- TRUE if this connection was initiated
        -- inside the protected network

    state       integer,
        -- Where in the TCP state machine the
        -- connection ended up.

    duration    interval,
    tcqtime     timestamp TIMESTAMPCOLUMN
)
```

type unarchived;

TelegraphCQ create stream statements are similar to SQL create table statements. The text type unarchived at the end of the statement indicates that the TelegraphCQ is not to archive flow records internally; our system uses FastBit for all archival storage. The Inet data type is a built-in PostgreSQL type for storing IP addresses; other types used in this schema are native SQL92 types.

To store the historical flow records, we created a corresponding FastBit table. FastBit's queries operate over a single vertically-partitioned table, with columns stored in separate files. Individual columns values are stored in native C++ data types. Table 2 summarizes the mapping between our FastBit and TelegraphCQ schemas:

TCQ Column	FastBit Column(s)	C++ Type
src_ip	IPS	int
dst_ip	IPR	int
src_port	SP	int
dst_port	DP	int
protocol	PROT	int
bytes_sent	S_SIZE	int
bytes_recv	R_SIZE	int
outgoing	FLAG	int
state	STATE	int
duration	dur	double
tcqtime	ts	int

Table 2: Relationship between columns of our TelegraphCQ and FastBit schemas

5 Queries

Based on a literature search and discussions with networking researchers, we have created a representative workload of real-time network analyses. Each analysis is comprised of a stream query processing component, expressed as a TelegraphCQ query; and a historical component, expressed as a FastBit query.

In the sections that follow, we present these queries in the native query languages of TelegraphCQ and FastBit, respectively, using the schemas from Section 4. Each query has one or more parameters that are bound at runtime. Following the convention of most SQL databases, we denote these variable parameters with a preceding semicolon. For example, in the first "elephants" query below, the parameter :windowsz is variable. These queries are presented in the forms of parametered templates, which can be built into some high-level interfaces to our system. We list these queries in detail here to show the complexities of the query needed to do real analyses and also to help users understand the performance data to be presented later.

5.1 Elephants

Goal: Find the k most significant sources of traffic (source/destination pairs, subnets, ports, etc.)[1]. TelegraphCQ finds the top k, and FastBit compares these top k against their previous history. Report significant traffic sources that were not significant in the past.

TelegraphCQ Query: Finds top 100 addresses by data sent.

```
select sum(bytes_sent), src_ip, wtime(*) as now
from flow [range by :windowsz slide by :windowsz]
where outgoing = false
group by src_ip
order by sum(bytes_sent) desc
limit 100 per window;
```

FastBit Query: Retrieves the total traffic from the indicated 100 addresses for the same time of day over the past 7 days. The variables :X1, :X2 and so on are the src_ip outputed from the above TelegraphCQ query. Note also that FastBit queries have an implicit *GROUP BY* clause on all selected attributes that are not part of any aggregate functions.

```
select IPS, sum(S_SIZE), sum(R_SIZE)
where IPS in (:X1, :X2, :X3, ..., :X100) and
((ts between [:now − 24 hr] and [:now − 23 hr])
or (ts between [:now − 48 hr] and [:now − 47 hr])
or (ts between [:now − 72 hr] and [:now − 71 hr])
or (ts between [:now − 96 hr] and [:now − 95 hr])
or (ts between [:now − 120 hr] and [:now − 125 hr])
or (ts between [:now − 144 hr] and [:now − 143 hr])
or (ts between [:now − 168 hr] and [:now − 167 hr])
);
```

5.2 Mice

Goal: Find all traffic from hosts that have not historically sent large amounts of traffic. FastBit finds the top k hosts by bytes sent over the past week. Then TelegraphCQ looks for traffic that is *not* from these top k hosts.

FastBit Query: Retrieves the top k hosts by total traffic for the past week. The variable :history, called the length of history, defines how far back in history we look for historical patterns. We usually vary its value from a few days to a month.

```
select sum(S_SIZE), IPS
where ts between [:now − :history] and [:now]
order by sum(S_SIZE) desc
limit :k;
```

TelegraphCQ Query: Produce a breakdown of traffic that is not from the hosts in the FastBit query's results.

[1] The names of this query and the one that follow derive from the networking term "elephants and mice", which refers to the sources of the largest and smallest traffic on a network link, respectively. "Elephants" tend to dominate bandwidth usage, and security problems often involve "mice".

```
select sum(bytes_sent), src_ip, dst_ip, wtime(*)
from flow [range by :windowsz slide by :windowsz]
where
    src_ip != :X1 and src_ip != :X2 ...
    and src_ip != :X100
group by src_ip, dst_ip
order by sum(bytes_sent) desc
limit 100 per window;
```

5.3 Portscans

Goal: Find behavior that suggests port scanning activity. TelegraphCQ flags current behavior, and FastBit is used to filter out hosts that exhibit this behavior as part of normal traffic.

TelegraphCQ Query: Finds external hosts that connect to many distinct destinations and ports within Berkeley Lab.

```
select src_ip,
    count(distinct
        (dst_ip :: varchar || dst_port :: varchar))
            as fanout,
    wtime(*)
from flow [range by :windowsz slide by :windowsz]
where outgoing = false
group by src_ip
order by fanout desc
limit 100 per window;
```

FastBit Query: Finds the 100 distinct hosts a given host normally connects to. FastBit queries with no aggregates have an implicit count(∗) aggregate.

```
select IPR, IPS
where
    IPS in (:X1, :X2, :X3, ... , :X100)
    and (ts between [:now - :history] and [:now])
```

5.4 Anomaly detection

Goal: Compare the current local traffic matrix (traffic by source/destination pair) against a traffic matrix from the past. TelegraphCQ fetches the current traffic matrix, and FastBit fetches the matrix in the past. Then application logic compares the matrices using one of several measures of similarity to determine if traffic patterns have changed.

TelegraphCQ Query: Computes the local traffic matrix, merging the traffic in both directions.

```
select
    (case when outgoing = true
    then src_ip else dst_ip end) as inside_ip,
    (case when outgoing = true
    then dst_ip else src_ip end) as outside_ip,
```

```
    sum(bytes_sent) + sum(bytes_recv) as bytes
from flow [range by :windowsz slide by :windowsz]
group by inside_ip, outside_ip ;
```

FastBit Query: Fetches the incoming portion of the traffic matrix for a single period of time.

```
select
    IPS, sum(S_SIZE)
where
    outgoing = false
    and (ts between [:now − :history] and [:now])
order by sum(S_SIZE) desc
```

5.5 Dispersion

Goal: Find subnets at a IP prefix length that appear in two time windows within a given period, and report the time lag between these two windows. Certain patterns of time lag can indicate malicious behavior[6]. A TelegraphCQ query with two subqueries does the real-time analysis, and a FastBit query summarizes the traffic history for each IP prefix identified. This is based on a query in [6].

TelegraphCQ Query: Produces a breakdown of traffic by subnet. The parameter :prefixlen determines the length of the IP address prefix that defines a subnet.

```
with
WindowResults1 as
(select
    network(set_masklen(src_ip, 8)) as prefix ,
    wtime(*) as tcqtime
from Flow [range by :windowsz slide by :windowsz]
group by prefix)
WindowResults2 as
(
    -- Create a second copy of the stream
    select * from WindowResults1
)
(select
    W1.prefix as prefix ,
    W2.tcqtime − W1.tcqtime as lag,
    count(*),
    wtime(*)
from
    WindowResults1 W1 [range by 10 * :windowsz
                        slide by :windowsz],
    WindowResults2 W2 [range by 10 * :windowsz
                        slide by :windowsz]
where W1.prefix = W2.prefix
    and W2.tcqtime > W1.tcqtime
group by W1.prefix, lag);
```

FastBit Query: Fetches the incoming traffic for the given subnets in the given time window. *NOTE: FastBit currently does not support arithmetic expressions in its* select *lists, so the query breaks down traffic by source address.*

```
select
    IPS, IPR, sum(R_SIZE)
where
    ( prefix (IPS, :prefixlen) = :X1 OR ... = :X2 OR ... )
    and (ts between [:now − :history] and [:now])
    and outgoing = false
order by sum(R_SIZE) desc
```

6 Experiments

In this section, we evaluate the performance of our system in a series of experiments. We start by analyzing the major components of our system separately to determine the peak data throughput of each component. We then benchmark the throughput of the entire system and compare this end-to-end throughput against that of the individual components. Finally, we evaluate the performance of the system with realistic packet arrival rates derived from the timestamps in our traces. All experiments were conducted on a server with dual 2.8 GHz Pentium 4 processors, 2 GB of main memory, and an IDE RAID storage system capable of sustaining 60 MB/sec for reads and writes.

Our current implementations of the Ingress Manger and Controller can process in excess of 100,000 flow records or result tuples per second, respectively. Neither of these parts represent a performance bottleneck to the system. We will not discuss their performances further in this paper.

6.1 TelegraphCQ

Our first experiment examined the performance tradeoffs of the TelegraphCQ component of our system. We focused on two parameters: query type and window size. Recall that each of the five TelegraphCQ queries in our workload produces results for discrete windows of time, and the size of these windows is given as a parameter in the query.

We sent week 5 of the trace through TelegraphCQ and measured the total running time for each of our five TelegraphCQ queries. From this running time, the window size, and the number of tuples in the trace, we computed an average throughput figure for the query processor. We repeated the experiment while varying the window size from 1 to 1,000 seconds. This range of time intervals corresponds to an average of between 38.1 and 38,114 tuples per time window.

Figure 5 shows the measured throughput of TelegraphCQ in our experiments. Each query showed an initial increase in throughput as window size increased. As window size continues to increase, throughput eventually reaches a peak and declines somewhat. The “dispersion” query showed a particularly pronounced instance of this pattern, with throughput increasing from 6,000 to 25,000 tuples across the range of window sizes targeted. The remaining queries had much flatter throughput curves.

We profiled our TelegraphCQ testbed to determine the reason for the changes in throughput we observed. We found that the relatively low throughputs at smaller window sizes were due to the large number of result tuples the queries produce at those

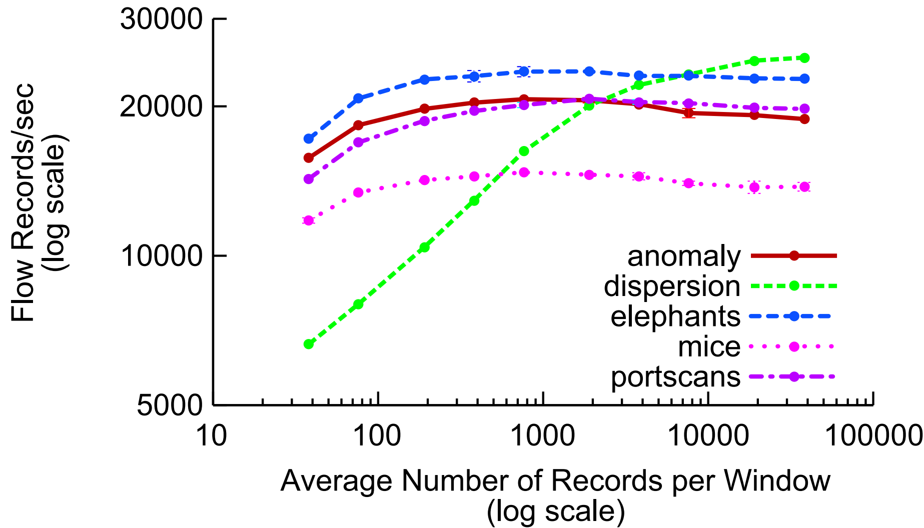

Figure 5: The throughput of TelegraphCQ running our queries with varying window sizes over the 5th week of the NERSC trace. Note the logarithmic scale.

window sizes. This effect was particularly pronounced with the dispersion query because the results of the first subquery went into a self-join, which is a more complex operation than the rest. We traced the falloff in performance at larger window sizes to increased memory footprint, as TelegraphCQ buffers the tuples in the current window in memory.

Our results indicate that, with an appropriate choice of window size, a small cluster of servers running TelegraphCQ can handle the aggregate flow record traffic of all DOE labs while running a 10-20 query workload. Choosing the correct window size automatically remains an interesting open problem.

6.2 FastBit

Next, we examine the performance of indexing and querying the historical data with FastBit. As we indicated before, we would be appending batches of new records in FastBit. We will concentrate on how to adjust the batch sizes to achive the best performance. In terms of querying performances, we will measure the average query response time to find out how much historical data can be incorperated into the analysis.

6.2.1 Index Creation

We first measured the speed for appending new data and building the respective bitmap indices. For these experiments we appended the data and built the indices in batches

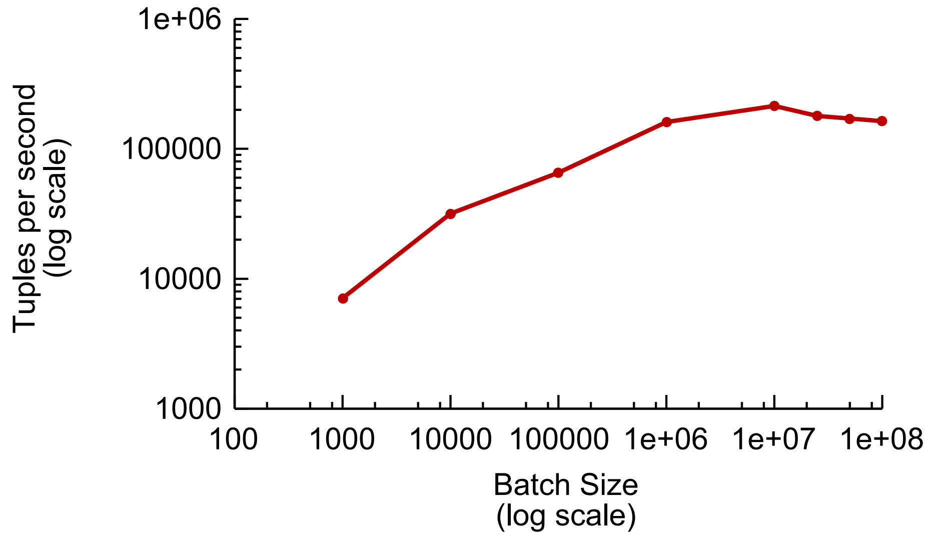

Figure 6: Speed for appending data and building the bitmap index in batches of sizes ranging from 1,000 to 100,000,000. Each tuple contains 11 attributes (48 bytes).

of various sizes to identify the most efficient batch size. In particular, the batch sizes were in the range of 1,000 and 100,000,000 tuples. Each tuple contains 11 attributes.

Figure 6 shows the speed for building the bitmap indices in terms of tuples per second. In our test data, each tuple is 48 bytes, 10 4-byte values plus 1 8-byte value. The maximum speed of 213,092 tuples per second is achieved when the append and index build operations are done in batches of 10,000,000 tuples. The performance graph also shows that the tuple rate decreases for batch sizes above 10,000,000 tuples and later stays about constant at around 170,000 tuples per second.

Let us now analyze this result in more detail. Building the indices consists of two parts: a) appending the data and b) building the indices. During the append operations FastBit copies the streaming data into two index directories called the *active directory* and *backup directory*. The *active directory* is the directory that is accessible while the indices are built in the *backup directory*. Having two separate directories guarantees that the user has always one consistent copy. When the batch size is small, say, 1000 tuples, the overhead such as opening files and closing files domiates the total time needed for copying data. When the batch size is in the millions, the total time required for making two copies is dominated by the read and write speed. As the batch size further increases, it is no longer possible to keep the content of files in memory, so another set of read operations is needed to make the second copy. For this reason, the rates of the append operations are slower for larger batch sizes.

6.2.2 Index Lookup

In the following experiments we measure the throughput of the index lookups (query response time) of the five queries described in Section 5. Apart from the "mice" queries (see Section 5.2), the starting point for all the queries was one day's worth of output data produced by TelegraphCQ. In particular, the TelegraphCQ continuous queries as described in Section 5 were run starting from week 5. Next, the output of these queries was used as input for FastBit to query the historical data. In order to measure the scalability of FastBit, we varied the length of history between 1 and 28 days. All queries were executed with 10 different lengths of history of equal size in the range of 1 and 28 days. By increasing the length of history, the result set (number of records fetched) of the queries increases monotonically and thus allows us to measure the query response time as a function of the result size.

Figure 7 shows the average query response time for all the 5 query types with 10 different lengths of history. In total, 100×10 queries were executed per query type on 100 million tuples with different lengths of history. The input for the 100 queries was randomly selected from the output of the TelegraphCQ streaming queries. In general, we observe a linear query response time with respect to the number of records fetched. We can see that the "elephants" and "portscans" queries have the best query response times between 0.1 and 1 seconds. The "mice", "anomalies" and "dispersion" queries have a higher query response time since they fetch more records. In fact, for large historical window sizes, nearly all of the 100 million records are fetched. Thus, the query response time is dominated by the time spent on fetching the results as opposed to the time spent on evaluating the queries with the bitmap index. For example, it takes about 20 seconds to answer the "anomalies" and "mice" queries when they fetch about 100 million records. Since both these queries select two attributes of 4-byte each, a total of about 800 MB are read into memory and sorted to compute aggregate functions. This leads to a reading speed of about 40 MB/s, which is nearly optimal because the read operations are not contiguous (some records are not needed). This implies that the time spent in evaluating the range conditions in these queries over 100 million records were negligible. The total query processing time is dominated by reading the selected values.

6.3 End-to-end Throughput

Our final experiment combined all the components of our system to measure end-to-end capacity. We connected the Ingress Manager and Controller components to TelegraphCQ and FastBit as described in Section 3, and we fed our trace through the Ingress Manager and measured the amount of time until the Controller produced its last report. We then used this total elapsed time and the number of tuples in the trace to compute the number of flow records per second that the system can consume.

We benchmarked each pair of TelegraphCQ/FastBit queries separately. The ranges of time selected in the FastBit queries were as depicted in Section 5. We varied the window size of the TelegraphCQ queries over the range of time intervals used in Section 6.1. Flow records were appended to the FastBit index in batches of 1 million.

Figure 8 shows the results of this experiment. Each thick line shows the perfor-

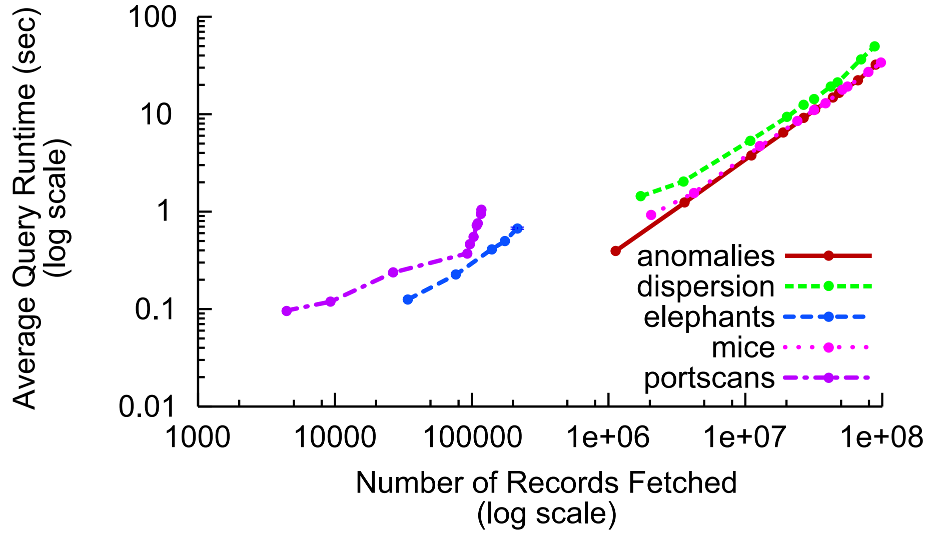

Figure 7: FastBit index lookup time for 5 types of historical queries with various lengths of history denoted by variable `:history`.

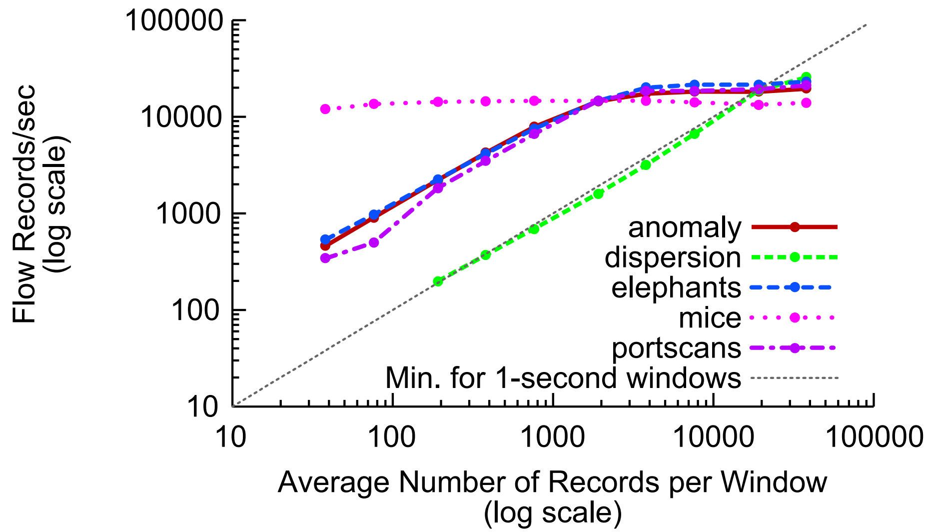

Figure 8: End-to-end throughput with varying window sizes over the 5th week of the NERSC trace. Note the logarithmic scale. The thin dotted line indicates the minimum throughput necessary to deliver results at 1-second intervals.

mance of a TelegraphCQ/FastBit query pair. The thin dotted line indicates the number of flow records per second the system would need to sustain to be able to deliver results at 1-second intervals. Points above this threshold indicate data rates at which the system can support window sizes of 1 second.

At smaller window sizes, FastBit lookup time dominates the combined query processing time. As window size increased, TelegraphCQ's performance came to dominate the system throughput. One exception to this rule was the mice queries where the FastBit component is executed offline prior to starting the TelegraphCQ component. Since we loaded tuples in large batches and ran the FastBit append operations and querying operations sequentially, the time spent on appending 1 million flow records (about 4 seconds), can appear as a noticeable lag in the query response time.

While most of the other queries produced similar throughput curves, the dispersion query showed an especially high degree of performance variation at different window sizes. Throughput for the dispersion queries did not reach comparable levels to that of the other queries until the number of tuples per window approached 20,000. This delayed increase was due to two factors. As we observed in the previous section, the TelegraphCQ component of the dispersion queries runs slowly at smaller window sizes. It turns out that the FastBit query for this analysis is also slower since the size of the result set is very large. Also, as we noted in Section 5, the current version of the query returns traffic broken down by address instead of by subnet. As a workaround, the Controller currently reads in the (much expanded) results of the FastBit query and performs an additional round of aggregation by subnet. We are working to remove this bottleneck by adding support for arithmetic expressions in FastBit's `select` lists.

In Figure 8 we also show the minimum performance needed to support a time window of 1 second. In this application, a window size of 1 second is extremely short. Even in this case, our system can handle between 9,000 and 20,000 flow records per second when running a single analysis on our test machine. Such data rates are in line with our estimates (See Section 1.1.1) of the average combined data rates for all DOE labs. However, our current system cannot handle our estimated *peak* rates of 1.6 million records per second. To handle these bursts of data, the system could either spool flow records to disk for later processing, use Data Triage [16] to trade off query result accuracy for response time, or use parallelism [19] to increase capacity.

Overall, the results of our end-to-end experiment underscore the importance of selecting an appropriate window size to maintain the required throughput.

7 Conclusions

The preceding sections analyzed the performance of our system components, both separately and as a complete system. The experiments identified several potential bottlenecks for monitoring systems that use declarative queries to perform both real-time monitoring and comparison against past history: *(1) Stream query processing*: Using small window sizes that create a large number of output tuples per time unit. *(2) Index creation*: Adding single records or small batches of records to the index. *(3) Index lookup*: Fetching large amounts of data or performing a large number of lookups.

We also identified effective strategies for mitigating these bottlenecks: *(1) Stream*

query processing: Control window size to maintain the proper ratio between input and output tuples. *(2) Index creation*: Periodically bulk-load the index; this contrasts with the strategy proposed in [4]. *(3) Index lookup*: Limit the number of lookups by keeping window size sufficiently large; scale selectivity by controlling how much history the queries consider.

With these strategies in place, our end-to-end experiments demonstrated that the system could handle the DOE labs' aggregate data rates with a relatively small degree of parallelism. Thus, a cluster of 30 to 50 processors should suffice to analyze the entire traffic patterns of tens of medium sized computer networks.

References

[1] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford University, 2003.

[2] E. W. Bethel, S. Campbell, E. Dart, K. Stockinger, and K. Wu. Accelerating Network Traffic Analysis Using Query-Driven Visualization. In *2006 IEEE Symposium on Visual Analytics Science and Technology (to appear)*, 2006.

[3] C. Y. Chan and Y. E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. In *SIGMOD*, 1999.

[4] S. Chandrasekaran and M. Franklin. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *VLDB*, 2004.

[5] S. Chandrasekaran and O. Cooper et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[6] B.-C. Chen, V. Yegneswaran, P. Barford, and R. Ramakrishnan. Toward a Query Language for Network Attack Data. In *NetDB Workshop*, 2006.

[7] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[8] M. Franklin, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-in Systems: The HiFi Approach. In *CIDR*, 2005.

[9] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2), 2003.

[10] G. Graefe. Goetz graefe. *SIGMOD Record*, 18(9):509–516, 2006.

[11] G. Iannaccone. CoMo: An Open Infrastructure for Network Monitoring – Research Agenda, 2005.

[12] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer. Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In *Internet Measurement Conference*, 2005.

[13] P. O'Neil and D. Quass. Improved Query Performance with Variant Indices. In *SIGMOD*, 1997.

[14] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *USENIX Security Symposium*, January 1998.

[15] T. Plagemann, V. Goebel, A. Bergamini, G. Tolu, G. Urvoy-Keller, and E. W. Biersack. Using Data Stream Management Systems for Traffic Analysis - A Case Study. In *Passive and Active Measurements*, 2004.

[16] F. Reiss and J. M. Hellerstein. Declarative Network Monitoring with an Underprovisioned Query Processor. In *ICDE*, 2006.

[17] M. Roesch. Snort-Lightweight Intrusion Detection for Networks. In *USENIX LISA*, 1999.

[18] D. Rotem, K. Stockinger, and K. Wu. Optimizing Candidate Check Costs for Bitmap Indices. In *CIKM*, 2005.
[19] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly Available, Fault-Tolerant, Parallel Dataflows. In *SIGMOD*, 2004.
[20] A. Shoshani, L. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional Indexing and Query Coordination for Tertiary Storage Management. In *SSDBM*, July 1999.
[21] M. Siekkinen, E. W. Biersack, V. Goebel, T. Plagemann, and G. Urvoy-Keller. InTraBase: Integrated Traffic Analysis Based on a Database Management System. In *Workshop on End-to-End Monitoring Techniques and Services*, May 2005.
[22] K. Stockinger and K.Wu et al. Network Traffic Analysis With Query Driven Visualization - SC 2005 HPC Analytics Results. In *Super Computing*, 2005.
[23] M. Sullivan and A. Heybey. Tribeca: A system for Managing Large Databases of Network Traffic. In *USENIX*, 1998.
[24] K. Wu, E. Otoo, and A. Shoshani. A Performance Comparison of Bitmap Indices. In *CIKM*, 2001.
[25] K. Wu, E. Otoo, and A. Shoshani. On the Performance of Bitmap Indices for High Cardinality Attributes. In *VLDB*, 2004.
[26] K. Wu, E. J. Otoo, and A. Shoshani. An Efficient Compression Scheme For Bitmap Indices. *ACM Transactions on Database Systems*, 31:1–38, 2006.