Data Triage

Frederick Ralph Reiss

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-79
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-79.html

June 1, 2007

Data Triage

by

Frederick Ralph Reiss

A.B. (Dartmouth College) 2000
M.S. (U.C. Berkeley) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Joseph M. Hellerstein, Chair
Professor Michael Franklin
Professor Charles Stone

Spring 2007

Data Triage

Abstract

Data Triage

by

Frederick Ralph Reiss
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

Enterprise networks are becoming more complex and more vital to daily operations. To cope with these changes, network administrators need new tools for troubleshooting problems quickly in the face of ever more sophisticated adversaries. Passive network monitoring with declarative queries can provide the combination of responsiveness, focus, and flexibility that administrators need. But networks are subject to high-speed bursts of data, and keeping the cost of passive monitoring hardware under control is a major problem.

In this dissertation, I propose an approach to passive network monitoring in which the monitor is provisioned for the average data rate on the network. This average rate is generally an order of magnitude or more lower than the peak rate. I describe Data Triage, an architecture that wraps a general-purpose streaming query processor with a software fallback mechanism that uses approximate query processing to provide timely answers during bursts. I analyze the policy issues that this architecture exposes and present Delay Constraints, an API and associated scheduling algorithm for managing Data Triage. I then describe my work on novel query approximation techniques to make Data Triage's fallback mechanism work with an important class of monitoring queries. Finally, I describe a deployment study of Data Triage in the context of a prototype end-to-end network monitoring system at Lawrence Berkeley National Laboratory.

To my parents.

Acknowledgments

I would like to thank my advisor, Joe Hellerstein, who took me on during my first year at Berkeley. Joe's support and guidance have been essential in helping me to succeed in grad school.

I would also like to thank my other committee members, Professors Michael Franklin, Chuck Stone, and Scott Shenker, for taking time out of their extremely busy schedules to provide valuable feedback on drafts of my qualifying exam proposal and my dissertation.

During my summer 2002 internship at IBM, Tapas Kanungo taught me all about writing papers and helped point me in the right direction for getting my dissertation underway.

My fellow grad students in the Berkeley Database Group have been an invaluable resource, not only for comraderie and helpful feedback, but also as role models when I needed direction. Special thanks to Mehul, Shawn, Boon, Sailesh, Amol, David, David, and everyone else.

My friends outside of school have been especially instrumental in helping me to weather the emotional tides of grad school. Many thanks to Tim, Meryl, Ally, Roehl, Mark, Helen, Lara, and Susan. And of course to my fiance Dorit, who more than anyone else has helped me get through the frustrating final slog to the end of the dissertation.

I would also like to acknowledge the American Society for Engineering Education and the U.S. Department of Defense for funding three years of my graduate study through the NDSEG Fellowship Program. Seibel Corporation also provided generous support during my first year at Berkeley.

Finally, I would like to thank my parents, Steven and Loretta Reiss, without whose love and support I would not have made it through the last 22 years of school.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The future will be better tomorrow.

— Dan Quayle (1947 -)

1.1 Overview

Networks are becoming more important, more complex, and more difficult to manage. This growth leads to the need for improved management infrastructure for network operators. In particular, operators need up to date status information, detailed data about the network, and the flexibility to focus on just the information that is relevant. Passive network monitoring with declarative queries is a promising paradigm for meeting these needs, but the hardware costs of deploying passive monitoring are currently prohibitive, due to the bursty nature of network traffic.

In this introductory chapter, I describe recent trends in enterprise networking and how these trends motivate the use of passive network monitoring and declarative queries. I also analyze the costs of deploying such a monitoring infrastructure with current technology and explain the factors that make such a deployment prohibitively expensive. Finally, I give an overview of my solution to this problem and outline how the remaining chapters will describe this solution in detail.

1.2 The Growth of Enterprise Networks

Current trends in information technology are making computer networks a more integral part of large businesses. New technologies leverage Internet protocols and expanded network capacity to make existing processes more efficient and to create new opportunities for growth. These technologies span a wide variety of functions:

- **Virtual organizations with global VPNs:** Large organizations are using Virtual Private Networks (VPNs) to give employees network access from home and from WiFi Internet access points. At AT&T,

  > "30 percent of AT&T managers are full-time virtual office workers and an added 41 percent work frequently out of the office. By increasing productivity, reducing real estate costs and cutting employee turnover, enterprise mobility initiatives are delivering significant business benefits." [23]

- **Sales force automation:** Companies are using Customer Relationship Management (CRM) software and hosted services to provide continuous up-to-date information to salespeople on the road. According to Aberdeen Group,

  > "hosted CRM is emerging as a major force likely to shape the future of CRM and the software industry in ways that cannot be fully appreciated today" [42]

  Increasingly, salespeople are using handheld devices and third-generation wireless modems to access CRM portals from remote locations [68].

- **IP telephony:** By using existing IP infrastructure to route voice calls, businesses are providing high-end telephony services to their employees while saving significant amounts of money. According to BusinessWeek, "11.5 million IP lines were shipped to businesses in 2005, a 46% increase . . . from 2004" [26].

- **Service oriented architecture:** Companies are increasingly using web services and open standards to tie together software components both within and between organizations. This service-oriented architecture, or SOA, allows for greater collaboration between different groups and facilitates code reuse. According to analyst Tom Dwyer of the Yankee Group,

  > "SOA is heading toward broad implementation in only a matter of a few years in the United States, regardless of organization size or vertical industry. 2006 will be the year of initial SOA project completion on a broad basis: not on a hit-or-miss trend, but through a rising tide of broad and deep adoption of SOA across the market" [34]

- **E-commerce and online marketing:** The World Wide Web is becoming more and more important to marketing and selling products in today's global marketplace. According to eMarketer,

  > "Figures released by the Internet Advertising Bureau (IAB) and PricewaterhouseCoopers (PwC), show that U.S. spending on Internet advertising set a record in the first quarter of 2006, reaching $3.9 billion. That represents a healthy 38% increase over Q1 2005, when the total was $2.8 billion, and a 6% increase over the Q4 2005 total of $3.6 billion." [31]

These ongoing trends are leading to several kinds of growth in the complexity of managing enterprise networks. To make use of new technologies, companies are adding thousands of new telephones, sensors, wireless handhelds, servers, and other devices to their networks. These new devices in turn communicate with a broad mix of different protocols that can be difficult to analyze and can interfere with each other. As vital functions like telephone communications and parts procurement start requiring network connectivity, network downtime is becoming far more costly than ever before. Finally, the increased reliance on networking is drawing a new breed of sophisticated adversary, making it more difficult both to keep networks available and secure.

1.3 Monitoring Enterprise Networks

One of the important tasks of a network administrator is to know the status of the network: Are all network services running smoothly? What hardware or software failures are occurring? Is the network under attack?

The current generation of network monitoring tools has difficulty providing these answers in the face of increasing network complexity. In particular, monitoring technology needs to grow in three areas:

- **Up-to-date status:** To maintain high availability, network administrators need instant notification of network problems. Quick response time is especially important when dealing with security breaches such as worm attacks [110]. New closed-loop control systems that react automatically to problems [4, 1, 32] make latency even more important.
- **Detailed information:** Today's multi-protocol, multi-site networks are prone to subtle failures and misconfigurations that can only be detected with detailed analysis of the network traffic itself. For example, to detect the failure of a web service piggybacked on the HTTP protocol, the administrator needs a tool that can decode HTTP sessions and track track information about the web service invocations inside of them.
- **Flexibility:** While a detailed view of the network is important for detecting problems, it can easily lead to information overload. Administrators need an efficient mechanism to filter out the details that are not relevant to the task at hand. For example, the administrator might need to debug a problem with one web service among several hundred on a single machine, or to trace the source of dropped calls between a particular pair of IP-enabled telephones.

The networking community has been developing several types of tools to meet these needs, and one of the more promising solutions is *passive network monitoring*. A passive network monitor is a device that monitors the traffic on a network link, parsing and analyzing the packet stream, as illustrated in Figure 1.1.

Early passive network monitoring systems relied on hard-coded software modules to analyze network traffic. Recent generations have moved towards a more flexible approach that uses declarative continuous queries to decode and analyze traffic. Passive network monitoring with continuous queries provides a way to meet the expanded monitoring needs of today's network administrators. Continuous queries provide an up-to-date picture of the network because they constantly stream query results back to the user. At the same time, these queries can expose a fine-grained picture of the network's traffic to the user, allowing him or her to extract important information about the status of important areas of the network. By

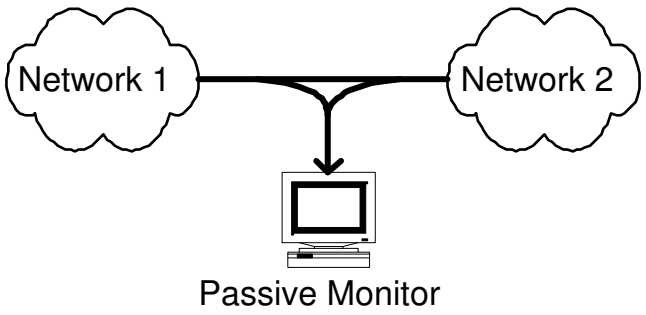

Figure 1.1: A typical passive network monitoring setup.

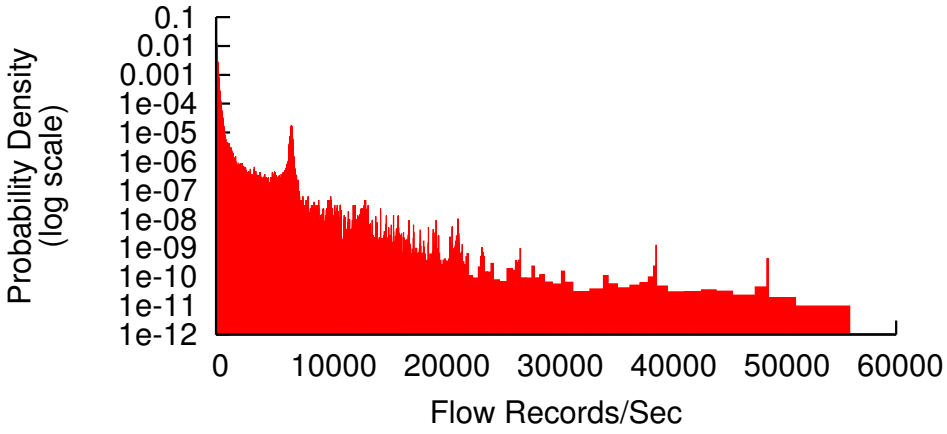

Figure 1.2: Distribution of the arrival rate of TCP sessions in 1-second windows in a trace of traffic on the NERSC supercomputing backbone.

altering query parameters and adding new predicates, administrators can avoid information overload, extracting just those details that are relevant to the problem at hand.

1.4 The Hardware Cost Problem

Researchers in the networking community have long studied the aggregate characteristics of network traffic. One of the more interesting and well-documented of these characteristics is the self-similar, bursty rate of packet arrival. Packet arrival rates in many different kinds of networks follow heavy-tailed distributions, with high-speed *bursts* occurring over a broad range of time scales [61, 79, 89]. Figure 1.2 shows an example distribution, drawn from a trace of TCP flows on the NERSC backbone link at Lawrence Berkeley National Lab.

To avoid outages during bursts, network equipment is generally provisioned for the peak expected load on a network link. Modern enterprise networks often have peak data rates of one gigabit per second. With the speed of modern computing hardware, a number of common events can lead to bursts that saturate such a network link. Some of these events are normal occurrences; for example, a server undergoing a period of heavy load, a client backing up its files, or a cluster of users logging in at 9 am. Other events that can cause these bursts are serious problems; for example, a malfunctioning piece of network equipment, a denial of service attack, an errant software process, or an intruder downloading many confidential files. Network administrators need to be able to distinguish between the different events that can saturate a gigabit link, so it is important for network monitors to remain operational in the face of such events.

Previous work has explored the feasibility of passive network monitoring at gigabit rates. The general conclusion of this work has been that passive monitoring of such traffic requires server-class hardware [25] or special-purpose processors [94]. Such hardware can cost more than the network itself: A high-end managed Gigabit Ethernet switch currently costs about $80 per port, while a low-end rack-mount server costs at least $750[1]. Until a single low-end server can monitor 10 or more network links, it will be difficult to justify deploying such a server as a passive monitor. This *hardware cost problem* is a major factor holding back the broader adoption of passive network monitoring technology.

[1] Estimates based on prices of Dell PowerConnect 6024 24-port managed Layer 3 gigabit switch ($1959) and Dell PowerEdge 850 entry-level server with an Intel Celeron CPU and 512 MB RAM ($769), as of July 21, 2006.

1.5 Solving the Hardware Cost Problem with Data Triage

In this dissertation, I propose a novel solution to the hardware cost problem for passive network monitoring. I have developed an architecture called *Data Triage* that provides a software fallback mechanism for dealing with high-speed bursts. Data Triage uses approximate query processing techniques to increase effective throughput without seriously affecting the quality of query results. With this fallback mechanism in place, the network administrator can provision a monitor for the the average rate of traffic, reducing hardware requirements by one or more orders of magnitude.

The primary benefit of my approach is that it allows the monitoring of fast network links with affordable hardware, surmounting a major barrier to the broader adoption of passive network monitoring. Additionally, my solution maintains the expressivity and flexibility of the streaming query processor and returns results in a timely manner. In between bursts, the system provides completely accurate results to queries. The system can absorb small bursts with no ill effects; during large bursts, result quality degrades gracefully and returns to normal as soon as the burst ends.

This dissertation describes and analyzes the components of my solution. Chapter 3 describes the Data Triage architecture and its implementation in the TelegraphCQ streaming query processor. Chapter 4 analyzes the scheduling and policy tradeoffs inherent in Data Triage and presents my solutions for navigating these tradeoffs. Chapter 5 points out weaknesses in existing work for approximating queries over network data and presents novel classes of histogram that address these weaknesses. Finally, Chapter 6 describes a deployment study at Lawrence Berkeley National Laboratory that demonstrates the feasibility of my approach in a real-world setting.

1.6 Background

Here I present background information that is important for an understanding of the rest of this dissertation.

1.6.1 The TelegraphCQ Streaming Query Processor

TelegraphCQ is a streaming query processor developed by the Telegraph project at U.C. Berkeley from 2002 through 2006. The system extends the PostgreSQL database system [83] to support continuous queries over streaming data. TelegraphCQ maintains the core feature set of PostgreSQL, including the transaction manager, type system, predicate logic, and stored procedures. In addition to these capabilities, TelegraphCQ adds support of the CQL query language (See Section 1.6.1.1) with subqueries and recursive queries; high-performance data ingress for streaming large numbers of tuples into the system; and continuous multi-query optimization for running many queries simultaneously. The initial beta release of TelegraphCQ occurred in late 2003, with version 2.0 released in 2004 and version 2.1 in 2005. Version 3.0 is targeted for a late 2006 release and includes a new data ingress architecture and enhanced query support. The TelegraphCQ software can be downloaded from the Telegraph web site at <http://telegraph.cs.berkeley.edu/>.

1.6.1.1 Query Model

The work in this dissertation uses the query model of the current development version of TelegraphCQ [19]. Queries in TelegraphCQ are expressed in CQL [6], a stream query language based on SQL. Data streams in TelegraphCQ consist of sequences of timestamped relational tuples. Users create streams and tables using a variant of SQL's Data Definition Language, as illustrated by the following sample schema:

```
-- Stream of IP header information.
-- The "inet" type encapsulates a 32-bit IP address
create stream Packets ( src_addr inet, dest_ddr inet,
                        length integer, ts timestamp)
type unarchived;

-- Table of WHOIS information
create table Whois (min_addr inet, max_addr inet, name varchar);
```

In addition to traditional SQL query processing, TelegraphCQ allows users to specify long-running *continuous queries* over data streams and/or static tables. In

this dissertation, I focus on such continuous queries.

The basic building block of a continuous query in TelegraphCQ is a SELECT statement similar to the SELECT statement in SQL. These statements can perform selections, projections, joins, and time windowing operations over streams and tables.

An overview of the SQL query language can be found in [84]. The SELECT statement in CQL has a similar structure to that of SQL:

```
select <columns and aggregates>
from <stream(s) with window clauses>, <table(s)>
where <predicate(s)>
group by <columns>
order by <columns>
limit <num> per window
```

The entries for streams in CQL's FROM clause take the form:

```
<stream name> [range '<interval>' slide '<interval>' start '<interval>']
```

where the RANGE, SLIDE, and START parameters specify a time window over the stream. Every time any stream's time window advances, the query result is updated. As in SQL, optional GROUP BY, ORDER BY, and LIMIT clauses act over the result tuples for each update interval.

TelegraphCQ can combine multiple SELECT statements by using a variant of the SQL99 WITH construct. The implementation of the WITH clause in TelegraphCQ supports recursive queries, but I do not consider recursion in this dissertation.

The specifications of time windows in TelegraphCQ consist of RANGE and optional SLIDE and START parameters. Different values of these parameters can specify sliding, hopping (also known as tumbling), or jumping windows.

The following listing gives several example network monitoring queries that demonstrate the utility of this query model.

```
-- Fetch all packets from Berkeley
select *
from Packets P [range by '5 seconds' slide by '5 seconds'], Whois W
where P.src_addr ≥W.min_addr and P.src_addr < W.max_addr
    and W.name like '%berkeley.edu';
```

```
-- Compute a traffic matrix (aggregate traffic between source-destination
-- pairs), updating every 5 seconds
select P.src_addr, P.dest_addr, sum(P.length)
from Packets P [range by '30 sec' slide by '5 sec']
group by P.src_addr, P.dest_addr;

-- Find all <source, destination> pairs that transmit more than 1000
-- packets for two 10-second windows in a row.
with Elephants as
    select P.src_addr, P.dest_addr, count(*), wtime(*) + '10 sec'
    from Packets P [range '10 sec' slide '10 sec']
    group by P.src_addr, P.dest_addr
    having count(*) > 1000
(select P.src_addr, P.dest_addr, count(*)
    from Packets P [range '10 sec' slide '10 sec'],
         Elephants E [range '10 sec' slide '10 sec']
              -- Note that Elephants is offset by one window!
    where P.src_addr = E.src_addr and P.dest_addr = E.dest_addr
    group by P.src_addr, P.dest_addr
    having count(*) > 1000);
```

1.6.1.2 Architecture

The architecture of TelegraphCQ version 0.2 was the subject of a CIDR 2003 paper [19]. Figure 1.3 shows the architecture of TelegraphCQ version 3.0. Here I briefly summarize aspects of the system that are relevant to the understanding of this dissertation.

The architecture of TelegraphCQ consists of three major components: The *Front End*, the *Back End*, and the *Triage Process*. Each of these components runs in a separate operating system process; the processes communicate with a combination of shared memory and Berkeley sockets. The system currently runs on recent Linux and MacOS platforms.

Front End Process: The TelegraphCQ Front End process has all the components of a standard PostgreSQL "Back End" process; in particular, the TelegraphCQ Front End exports the same client APIs as PostgreSQL, and it has full access TelegraphCQ's buffer pool and transaction manager. The system spawns a separate Front End process for each client connection, and conventional SQL queries run entirely within the Front End. The Front End forwards CQL continuous queries to the Back End process and uses a stub "mini-executor" to fetch results from the Back End.

To run a continuous query on TelegraphCQ, clients connect to TelegraphCQ us-

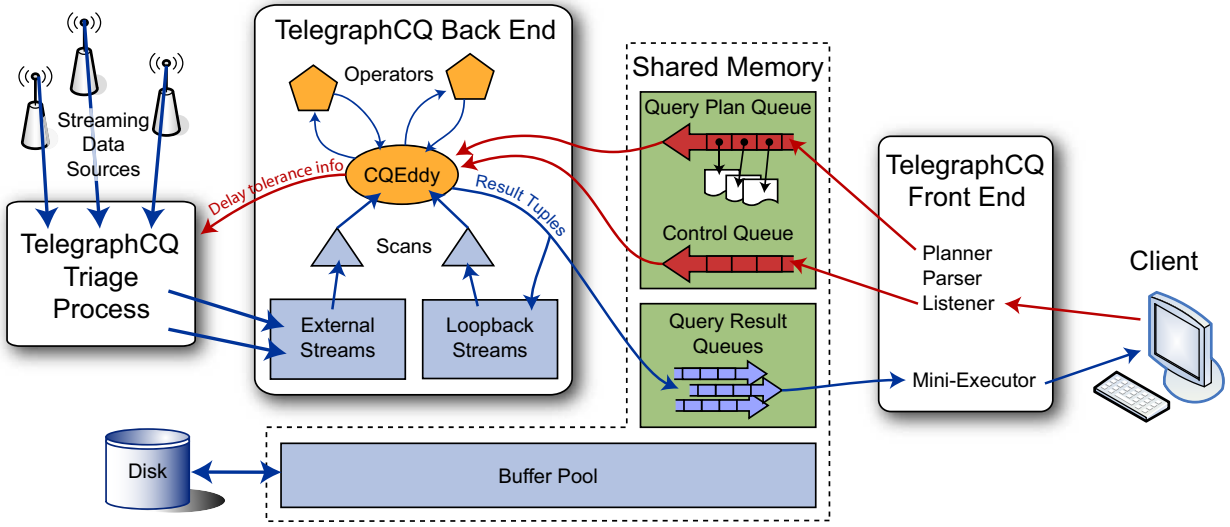

Figure 1.3: The architecture of TelegraphCQ.

ing a standard PostgreSQL interface such as ODBC, JDBC, or `libpq`. To fit a continuous query model into these SQL-based interfaces, the client submits the query using a DECLARE CURSOR statement and retrieves query results with FETCH statements. For example, to run the query:

```
select * from Stream;
```

the client would issue the following command:

```
declare results cursor for
select * from Stream;
```

followed by a sequence of fetch from results statements to retrieve result tuples. Clients can cancel continuous queries by terminating the current transaction.

When the Front End receives a continuous query, it parses and plans the query as if it were a conventional SQL query. Then the Front End sends the resulting query plan to the Back End via a shared memory queue and creates a stub operator tree to fetch results from the Back End.

Back End Process: Inside the TelegraphCQ Back End, continuous queries execute in a single shared execution environment similar to that of the CACQ system [65]. Streaming joins are executed by nonblocking operators known as SteMs, or "State Modules". Each SteM is responsible for maintaining an in-memory index

of one side of a join. Selections and projections are handled by conventional PostgreSQL operators and a special *Grouped Selection Filter* operator that can perform multiple selections at once. Aggregation occurs in a special operator that uses tree of PostgreSQL operators to compute aggregate values for each window. Tuples enter the back end via modified PostgreSQL scan operators. These scans can read tuples from relations, live data streams, or the result streams of existing queries. A special operator known as a CQEddy acts as a nexus of control, coordinating the movement of tuples among different relational operators. The CQEddy adaptively routes tuples among operators, using operator selectivities to determine an efficient execution order.

When a query plan enters the Back End, the CQEddy decomposes it into a collection of operators and merges this collection with the current set of "live" operators, removing duplicates. If multiple queries share a single operator, only a single instance of the operator is created. Where possible, compatible predicates are merged into a single Grouped Selection Filter operator. The CQEddy maintains state about which queries each tuple satisfies and forwards query results to the Front End as they are produced.

As noted in the previous section, TelegraphCQ supports subqueries and recursion via a SQL99-like WITH syntax. Internally, the backend uses "stub" streams to implement the internal queries of these WITH clauses. Instead of sending the results of these subqueries to the Front End, the system forwards the result tuples to scan operators, where the tuples re-enter the CQEddy's collection of operators.

Triage Process: The Triage Process is the component of TelegraphCQ that provides buffering and latency management for incoming data streams. This component is the main focus of this dissertation, and subsequent chapters describe the internal structure of the Triage Process in detail. Streaming data sources send data to the Triage Process, which forwards this data to the Back End as a mixture of PostgreSQL tuples and compact summaries. The Back End sends information about the delay tolerance of queries to the Triage Process, which uses this information to determine the necessary mix of buffering and summarization. Internally, the Triage Process keeps tuples in their native binary or text format, converting them to a format that PostgreSQL (and hence TelegraphCQ) can recognize when sending tuples to the Back End.

Chapter 2

Related Work

Everything of importance has been said before by somebody who did not discover it.

— Alfred North Whitehead (1861 - 1947)

2.1 Introduction

This chapter presents a survey of previous work that is relevant to this thesis. The chapter breaks down this previous work according to which chapter of the thesis is most relevant to it.

2.2 Architecture

The problem of monitoring bursty network traffic with inexpensive hardware has received attention in the networking literature. My solution to this problem, the Data Triage architecture, makes use of with a query processor that is provisioned for the data stream's average data rate. The networking community, in contrast, has taken a different approach to the problem, working to lower the CPU requirements of query processing itself.

The Gigascope system [25, 24] is one of the more well-known examples of this approach. Rather than build a general-purpose query processor, the designers of Gigascope designed a system that is closely tied to a small set of network monitoring applications. Gigascope operates directly on raw network packets, using a query language with primitives for composing hard-coded modules. These modules perform the "inner loops" of query processing and are coded in low-level languages by experts. Some modules are implemented on special-purpose hardware [86]. Because the low-level portions of Gigascope are so specialized and heavily optimized, the system can monitor high-speed network links with relatively inexpensive hardware. The major disadvantage of this approach is that it requires hours of time for experts to implement new queries. My architecture, in contrast, uses a general-purpose streaming query processor, so that relatively unsophisticated users can write new queries quickly.

The CoMo system [48] represents another approach to keeping hardware costs under control: Limit the real-time functionality of the system. The streaming component of CoMo is hard-coded by experts for "line speed" operation, but its functionality is limited to maintaining an on-disk circular buffer of recent network traffic. Basic query language allows remote sites to extract data from the buffer for offline analysis with data-mining tools. Compared with my work, CoMo is a much simpler architecture to implement, but it provides limited real-time analysis and alert generation.

The SNORT intrusion detection system [92] addresses the hardware cost problem by using a very simple query language. SNORT queries, known as "rules," can express simple selections and projections over the packet stream. A highly optimized execution engine with multi-query optimization takes advantage of the simplicity of the rules However, complex queries such as joins are difficult or im-

possible to express in SNORT's query language.

The Bro intrusion detection system [30] uses a C-like scripting language for expressing analyses procedurally, with common "inner-loop" operations hard-coded in hand-optimized C++. Sophisticated users can use the scripting language to write high-level operations, including "contingency plans" for dealing with various overload situations. A combination of hand-optimized low-level code and sophisticated algorithms in the high-level code allows Bro to identify complex patterns on relatively inexpensive hardware. However, the use of a procedural language instead of a declarative one makes modifying Bro's queries difficult and limits the ability of unsophisticated users to write new analyses.

2.2.1 Overload Handling

Overload handling is a natural concern in stream query processing, and several pieces of previous work have proposed solutions to the problem. The Aurora continuous query system sheds excess load by inserting *drop* operators into its dataflow network [104]. My work differs from this approach in two ways: First of all, I use fixed end-to-end delay constraints, whereas Aurora's *drop* operators minimize a local cost function given the resources available. Secondly, my system adaptively falls back on approximation in overload situations, while Aurora handles overload by dropping tuples from the dataflow.

Other work has focused on choosing the right tuples to drop in the event of overload [27, 53, 100]. My work is complementary to these approaches. In my work, I do not focus on choosing "victim" tuples in the event of overflow; rather, I develop a framework that sends the victim tuples through a fast, approximate data path to maintain bounded end-to-end latency. Choosing the right victim tuples for Data Triage is an important piece of future work.

Other stream processing systems have focused on using purely approximate query processing as a way of handling high load [15, 60]. Load shedding systems that use this approach lossily compress sets of tuples and perform query processing on the compressed sets. The STREAM data manager [33] uses either dropping or synopses to handle load. In general, this previous work has focused on situations in which the steady-state workload of the query processor exceeds its capacity to process input streams; I focus here on provisioning a system to handle the steady state well and to degrade gracefully when bursts lead to temporary overload.

Techniques for providing fast, approximate answers to queries are a well-studied area in database research. My Data Triage architecture uses these approximate query processing techniques as a fall-back mechanism for overload handling, and the third part of my proposed thesis presents new techniques for

approximating a class of query that is important for network monitoring.

Approaches to approximate query processing generally involve either *random samples* or *summary data structures*. Random sampling has a long history in the literature; Olken and Rotem [74] summarize work in the area prior to 1995. As recent examples, Chaudhuri *et al.* [21] analyze the effects of random sampling of base relations on the results of joins of those relations. Acharya *et al.* present a technique called *join synopses* that involves sampling from the join of the tables in a star schema rather than from the base tables themselves [3]. Congressional Sampling [2] and Dynamic Sample Selection [8] collect biased samples to improve the accuracy of grouped aggregation.

Summary data structures are also well-studied. Garofalakis and Gibbons give a good overview of work in this area [36]. Chakrabarti *et al.* [17] describe methods of performing query processing entirely in the wavelet domain using multidimensional wavelets. Getoor *et al.* [38] use probabilistic graph models as a compressed representation of relations. Pavlov, Smith, and Mannila [77, 76] investigate methods for lossy summarization of large, sparse binary data sets. Wang *et al.* have studied methods of performing query processing and selectivity estimation using wavelets [67, 66, 109]. Lim *et al.* use query feedback to tune a materialized set of histograms to the workload of the database [62].

2.3 Policy

Chapter 4 of this thesis explains the Delay Constraints API for controlling the tradeoff between query result latency and accuracy in Data Triage. I develop a scheduling algorithm for the Triage Process that ensures that it meets its constraints on query result delay. Scheduling algorithms that meet constraints on delay are common in multimedia scheduling and in real-time systems, but this previous work differs from my work in several important respects.

For "hard" real-time applications like avionics and nuclear power plant control, various algorithms can compute processor schedules to ensure that a set of well-understood tasks meet a specified set of deadlines with 100 percent probability. The most well-known of these approaches are Rate Monotonic and Earliest Deadline First (also known as Deadline Driven) scheduling, published by Liu and Layland in 1973 [63]. Rate Monotonic scheduling precomputes a static schedule for a fixed set of periodic tasks, guaranteeing that all tasks meet specified deadlines. The Earliest Deadline First approach schedules a stream of incoming tasks, giving highest priority to Follow-on work has extended these algorithms in a number of ways. For example, the Spring scheduling algorithm adds support for admission control and distribution and has been implemented in hardware [85, 16]; and Lu *et*

*al.*add closed-loop control to support tasks with unpredictable runtimes [64].

In "soft" real time applications like multimedia scheduling, the scheduler provides probabilistic guarantees that tasks will be finished on time. Lottery scheduling [107] and stride scheduling [108] schedule tasks in fixed ratios to ensure that every task receives at least the requested amount of CPU time per second. Fair Queueing algorithms [72, 28, 9, 97, 102] provide similar guarantees when routing packets from multiple flows across a network. The Dynamic Window-Constrained Scheduling algorithm [111] divides network bandwidth between multiple streams with different "deadlines" (maximum end-to-end latency of any packet) and "loss-tolerances" (percentage of packets in a given period of time that the network is allowed to drop).

In my work, I study a significantly different scheduling problem than previous work in real-time systems and networking. Existing work on processor and network scheduling focuses on determining the correct order in which to perform a collection of different tasks in order to ensure that all the tasks meet individual deadlines with high probability. The Data Triage scheduling problem, in contrast, involves scheduling the processing of a stream of very small, substantially identical tuple processing tasks, with every tuple from a given window having the same deadline. Since the tuples in a streaming query processor must be processed in order, the individual tasks in Data Triage occur in a very constrained order; choosing an order of operations is not an important factor in my work. The chief decision that my scheduling algorithm must make is whether to triage or to process fully the tuple currently at the head of the Triage Queue.

2.4 Histograms

Chapter 5 of this thesis describes a new class of histogram that I have developed. Histograms have a long history in the database literature. In recent years, researchers have developed many heuristics for constructing histograms [82, 69, 51, 12, 29, 103]. Other work has focused on optimal algorithms for histogram construction [49, 50, 51]. Poosala *et al.*[81] give a good overview of work in one-dimensional histograms, and Bruno *et al.*[12] provide an overview of existing work in multidimensional histograms.

Previous work has identified histogram construction problems for queries over hierarchies in data warehousing applications, where histogram buckets can be arbitrary contiguous ranges. Koudas *et al.*first presented the problem and provided an $O(n^6)$ solution [57]. Guha *et al.*developed an algorithm that obtains "near-linear" running time but requires more histogram buckets than the optimal solution [44]. Both papers focus only on *Root-Mean-Squared (RMS)* error metrics. In my work, I consider a different version of the problem in which the histogram buckets consist of nodes in the hierarchy, instead of being arbitrary ranges; and the selection ranges form a partition of the space. This restriction allows us to devise efficient optimal algorithms that extend to multiple dimensions and allow nested histogram buckets. Also, I support a wide variety of error metrics.

The STHoles work of Bruno *et al.*introduced the idea of nested histogram buckets [12]. The "holes" in STHoles histograms create a structure that is similar to my longest-prefix-match histogram buckets. However, I present efficient and optimal algorithms to build my histograms, whereas Bruno *et al.*used only heuristics (based on query feedback) for histogram construction. My algorithms take advantage of hierarchies of identifiers, whereas the STHoles work assumed no hierarchy.

Bu *et al.*study the problem of describing 1-0 matrices using hierarchical Minimum Description Length summaries with special "holes" to handle outliers [13]. This hierarchical MDL data structure has a similar flavor to the longest-prefix-match partitioning functions I study, but there are several important distinctions. First of all, the MDL summaries construct an exact compressed version of binary data, while my partitioning functions are used to find an approximate answer over integer-valued data. Furthermore, the "holes" that Bu *et al.*study are strictly located in the leaf nodes of the MDL hierarchy, whereas my hierarchies involve nested "holes".

Wavelet-based histograms [67, 66] are another area of related work. The error tree in a wavelet decomposition is analogous to the UID hierarchies I study. Also, recent work has studied building wavelet-based histograms for distributive error metrics [37, 54]. My overlapping histograms are somewhat reminiscent of wavelet-based histograms, but my concept of a bucket (and its contribution to the his-

togramming error) is simpler than that of a Haar wavelet coefficient. This results in simpler and more efficient algorithms (in the case of non-RMS error metrics), especially for multi-dimensional data sets [37]. In addition, my histogramming algorithms can work over arbitrary hierarchies rather than assuming the fixed, binary hierarchical construction employed by the Haar wavelet basis.

My longest-prefix-match class of functions is based on the technique used to map network addresses to destinations in Internet routers [35]. Networking researchers have developed highly efficient hardware and software methods for computing longest-prefix-match functions over IP addresses [73] and general strings [14].

2.5 Deployment

Two previous studies have used TelegraphCQ to run simple network monitoring queries [80, 87]. In the deployment study I describe in Chapter 6, I 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 [22] 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 [98]. 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 [20]. 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 [75, 18, 114, 93, 115]. Such queries appear frequently in my network monitoring workload.

One of the motivations given in [20] for their work was that updating B-trees may take too much time to keep up with the arrival of new records. In my work, I reexamine this assumption in the context of 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 [40] may narrow this performance gap, but I am unaware of any hard performance

numbers for the new designs.

In the network community, two commonly used Intrusion Detection Systems are Bro [78] and Snort [91]. 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* [56], 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 Chatper 6. However, the authors do not provide any details on the performance of querying historic data. The goal of my study 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 the system described in Chapter 6 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.

FastBit implements a set of compressed bitmap indices using an efficient compression method called the *Word Aligned Hybrid* (WAH) code [114, 115]. In an number of performance measurements, WAH compressed indices were shown to significantly outperform other indices [113, 114]. Recently, FastBit has also been used in analysis of network traffic data and shown to be able to handle massive data sets [101, 11]. This earlier proof-of-concept made FastBit a convenient choice for the prototype described in Chapter 6.

Chapter 3

Architecture

3.1 Introduction

This chapter describes the Data Triage architecture and its implementation in the TelegraphCQ streaming query processor. Data Triage provides a software fallback mechanism to isolate the query processor from high-speed bursts. The architecture shunts excess data through a *shadow query* that uses approximation to summarize missing query results.

The chapter starts with a brief overview of approximate query processing, followed by a description of the architecture. Then I describe techniques for implementing approximate query processing in a general-purpose query processor like TelegraphCQ. Finally, I describe the query rewrite process that my system uses to produce shadow queries.

3.1.1 Approximate Query Processing with Summaries

Much work has been done on approximate relational query processing using lossy set summaries. Originally intended for query optimization and interactive data analysis, these techniques have also shown promise as a fast and approximate method of stream query processing. Examples of summary schemes include random sampling [106, 3], multidimensional histograms [82, 51, 29, 103], and wavelet-based histograms [17, 67].

Later in this dissertation, in Chapter 5, I will present my work in approximating an important class of queries in the network monitoring space. However, the work in the current chapter centers on leveraging *any* kind of query approximation, including my histograms and the broad body of previous work. No single summarization method has been shown to dominate all others for all query types, so my system supports a variety of techniques. To simplify this support, I have developed a generic framework that allows my system to implement most existing approximations.

My framework divides an approximation scheme into four components:

- A *summary data structure* that provides a compact, lossy representation of a set of relational tuples
- A *compression function* that constructs summaries from sets of tuples.
- A set of *operators* that compute relational algebra expressions in the summary domain.
- A *rendering function* that converts from the summary domain back to the relational domain by generating tuples containing either aggregates over the input stream or representatives of that stream.

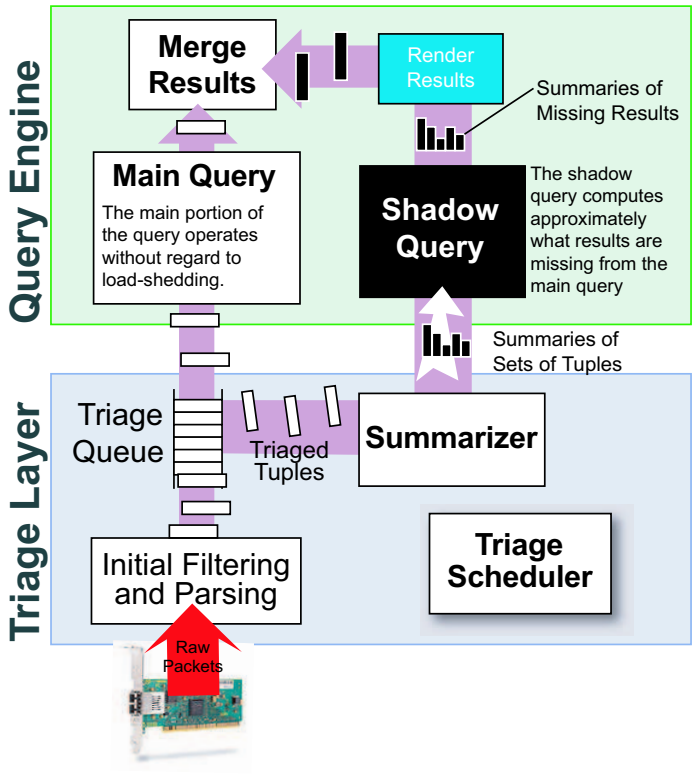

Figure 3.1: An overview of the Data Triage architecture. Data Triage acts as a middleware layer within the network monitor, isolating the real-time components from the best-effort query processor to maintain end-to-end responsiveness.

These primitives allow one to approximate continuous queries of the type described in Section 1.6.1.1. First, summarize the tuples in the current time window, then run these summaries through a tree of operators, and finally render the approximate result.

In addition to the primitives listed above, each approximation technique also has one or more *tuning parameters*. Examples of such parameters include sample rate, histogram bucket width, and number of wavelet coefficients. The tuning parameters control the tradeoff between processing time and approximation error.

3.2 Data Triage

Figure 3.1 gives an overview of the Data Triage architecture. This architecture consists of several components:

- The *initial parsing and filtering* layer of the system decodes network packets or flow records and produces streams of binary data tuples containing detailed

information about the packet stream.

- The *Main Query* operates inside a stream query processor; my prototype uses the TelegraphCQ query processor. The query processor converts tuples from the initial parsing and filtering layer into its internal format and passes them through the Main Query. Architecturally, the key characteristic of this query is that it is tuned to operate at the network's *typical* data rate. Data Triage protects the main query from data rates that exceed its capacity.

- A *Triage Queue* sits between each stream of tuples and the main query. Triage Queues act as buffers to smooth out small bursts, and they provide a mechanism for shunting excess data to a software fallback mechanism when there is not enough time to perform full query processing on every tuple.

- The *Triage Scheduler* manages the Triage Queues to ensure that the system delivers query results on time. The Scheduler manages end-to-end delay by *triaging* excess tuples from the Triage Queues, sending these tuples through the Shadow Query. I give a detailed description of my scheduling algorithm in Section 4.4.

- The *Summarizer* builds *summary data structures* containing information about the tuples that the Scheduler has triaged. The Summarizer then encapsulates these summaries and sends them to the query engine for approximate query processing.

- The *Shadow Query* uses approximate query processing over summary data structures to compute the *results that are missing* from the main query. The outer clause of his query then *renders* the summary into a set of relational tuples that represents the approximate result.

- The *Merge Query* merges the results of the Main and Shadow queries to produce a unified query answer for each time window. For GROUP BY queries, this merge involves a join on the grouping columns. For queries that produce a set of base tuples, the merge involves a UNION ALL clause.

I have implemented Data Triage in the TelegraphCQ stream query processor. The sections that follow describe the approach I used to construct approximate shadow queries efficiently.

3.2.1 Approximate Query Processing Framework

An important challenge of implementing Data Triage is adding the approximate query processing components to the query engine without rearchitecting the system. As I noted earlier in this chapter, most query approximation techniques can be divided into four components: a summary data structure, a compression function, a set of operators, and a rendering function. To perform approximate

query processing in TelegraphCQ, I have mapped these four components onto TelegraphCQ's object-relational capabilities. This implementation permits the use of many different summary types and minimizes changes to the TelegraphCQ query engine.

At the core of an approximation scheme is the summarization data structure, which provides a compact, lossy representation of a set of relational tuples. I used the user-defined datatype functionality of TelegraphCQ to implement several types of summary data structures, including reservoir samples, two types of multidimensional histograms, and wavelet-based histograms.

The second component of a summary scheme is a *compression function* for summarizing sets of tuples. My implementation actually divides the compression function into three parts, a design similar to the user-defined aggregates in PostgreSQL. The first part of a compression is a routine for creating an empty summary; the second part is a routine for adding a tuple to an existing summary; and the third part is a routine that finalizes the summary and prepares it to be sent to TelegraphCQ.

The Summarizer takes a compression function as an argument, with different C++ classes implementing different summarization schemes. This summary is packed into the external representation (currently text) of a user-defined type and sent to TelegraphCQ as a field of a tuple. The constructor for the user-defined type converts this external representation into an internal binary representation of the summary. In my experiments, the overhead of this conversion represented a very small part of overall running time. If necessary, a production implementation of the Summarizer could avoid this conversion step entirely by generating the internal binary representation of the summary directly.

The third component of a summary scheme is a set of relational operators that operate on the summary domain. Once summaries are stored in objects, it is straightforward to implement relational operators as functions on these objects. In TelegraphCQ, the syntax to define user-defined function over a user-defined type is derived from that of PostgreSQL:

```
CREATE FUNCTION function_name(argument list)
    RETURNS return_type AS
    '<path_to_shared_object>', '<function_name>'
    LANGUAGE C;
```

where function_name is a name used for calling the function from within CQL queries, return_type is the built-in or user-defined type that the function returns, and the shared object and function name tell TelegraphCQ where to find the

function implementation at runtime.

If, for example, MHIST multidimensional histograms [82] are encapsulated in a user-defined type, the common relational operators on this type are easily declared in TelegraphCQ syntax:

```
-- Project S down to the indicated columns.
create function project(S MHIST, colnames cstring)
    returns MHIST as ...

-- Approximate SQL's UNION ALL construct.
create function union_all(S MHIST, T MHIST)
    returns MHIST as ...

-- Compute approximate equijoin of S and T.
create function equijoin( S MHIST, S_colname cstring,
                          T MHIST, T_colname cstring)
    returns MHIST as ...
```

The C implementation of each function performs the appropriate operations on the binary representation of the summary. The binary representation of the summary also tracks the mapping between dimensions of a multidimensional summary and the columns of stream tuples. Shadow queries can use calls to these functions to compute relational expressions; for example:

```
(select b,c from A) UNION ALL (select d,e from B)
```

becomes

```
select union_all( project(A.sum, 'b,c'), project(B.sum, 'd,e')) from A, B;
```

The final component of a summary scheme is a rendering function that computes aggregate values from a summary. TelegraphCQ contains functionality from PostgreSQL for constructing functions that return sets of tuples. I use this set-returning function framework to implement the rendering functions for the different datatypes:

```
-- Convert S into tuples, one tuple for each bucket
create function mhist_render(S MHIST)
    returns setof record as ...
```

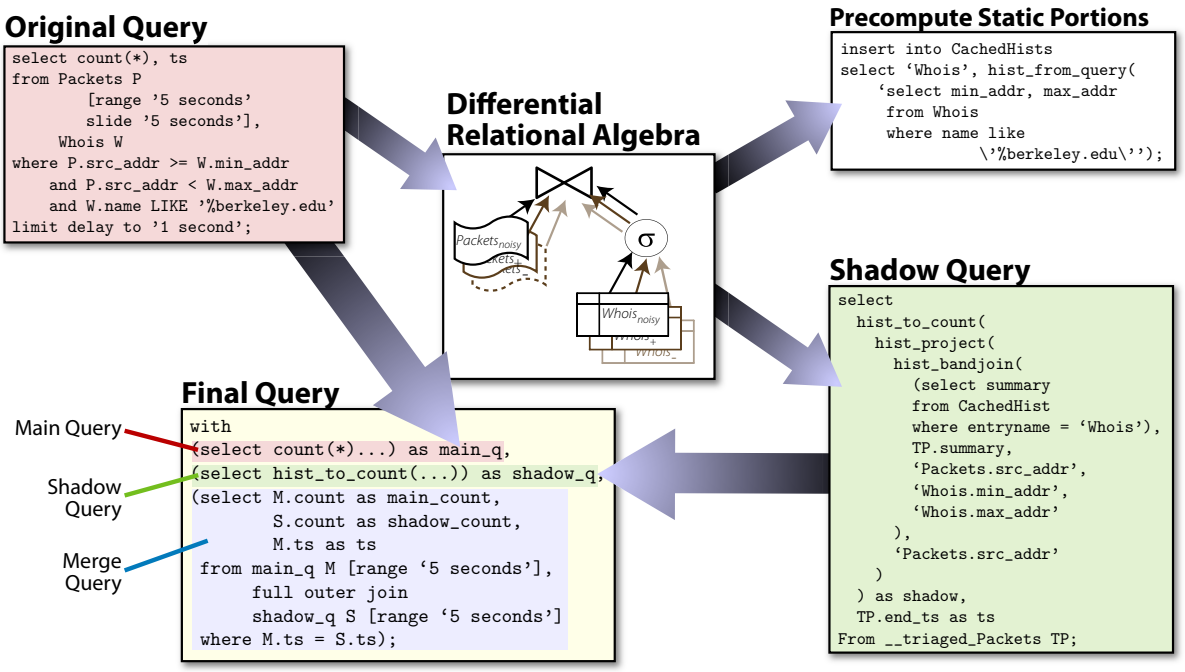

Figure 3.2: An illustration of the Data Triage query rewrite algorithm as applied to an example query. The algorithm produces a main query, a shadow query, and auxiliary glue queries to merge their results. This example uses multidimensional histograms as a summary datatype.

Having implemented the components of a given summarization scheme, I can construct Data Triage's shadow queries and merging logic using query rewriting. Section 3.3 demonstrates this process on a simple example query.

3.3 Lifetime of a Query

To illustrate the methods I use to construct shadow queries and how these methods interact with my implementation of Data Triage, I will now describe the query rewrite and execution process as it applies to the query in Figure 4.2. The query reports the number of packets coming from Berkeley domains every 5 seconds.

3.3.1 Summary Streams

Recall the sample CQL schema from Section 1.6.1.1. This schema contains a table of WHOIS information and a stream, Packet, of information about network packets.

To use Data Triage in TelegraphCQ, the user adds an ON OVERLOAD clause to each CREATE STREAM statement:

```
create stream Packets ...
    on overload keep MHIST;
```

This clause specifies the type of summary that Data Triage will construct on excess tuples in the stream. An alternate syntax could have the user choose the summary type at query execution time, though I did not expose such a syntax in my implementation.

The ON OVERLOAD clause causes TelegraphCQ to generate an auxiliary *summary stream* for summaries of triaged tuples:

```
create stream
 __triaged_Packets (summary MHIST,
                    earliest  timestamp,
                    latest  timestamp);
```

The two Timestamp fields in this stream indicate the range of timestamps in the tuples represented by the summary field. The summary stream will serve as an input to all shadow queries that operate on the Packets stream.

TelegraphCQ supports *stream-stream join* queries, such as:

```
select *
from S [range '10 sec'  slide  '10 sec' ],
     T [range '10 sec'  slide  '10 sec' ]
where S.a = T.a;
```

A more complex example of such a query is my "dispersion" query in Chapter 6. As the derivation in Section 3.3.2 indicates, the shadow query for a stream-stream join requires joining the triaged tuples for each stream the non-triaged tuples from the other stream. The system could join the summaries directly with the non-triaged tuples, but it is generally more efficient to summarize these tuples and then perform the join. To cover the case when a stream-stream join query operates directly on base streams, my system also creates a second summary stream that summarizes *non*-triaged tuples:

```
create stream
 __nontriaged_Packets (summary MHIST,
                       earliest  timestamp,
                       latest  timestamp);
```

This second stream and its associated summarizer are only activated when a stream-stream joins query is present in the system.

3.3.2 Query Rewrite

Figure 3.2 shows the query rewrite process as applied to my sample query. My query rewriting methodology is based on an algebraic formalism that allows my system to correctly rewrite queries into shadow queries. I use relational algebra to build a set of *differential relational algebra operators*. My approach here resembles past work in maintaining materialized views [45], though my setting is different.

Each differential operator propagates changes from the inputs to the outputs of the corresponding relational algebra operator. In naming the components of the algebra, I borrow a convention from the field of signal processing. When analyzing a noisy analog channel, it is a common practice to break the channel into "signal" and "noise" components and to model these components separately. Similarly, the differential relational algebra divides each input relation S into *noisy*, *additive noise*, and *subtractive noise* components S_{noisy}, S_+ and S_-, such that:

$$S_{noisy} \equiv S + S_+ - S_- \tag{3.1}$$

where $+$ and $-$ are the multiset union and multiset difference operators, respectively. Each differential operator propagates changes to these components separately.

Note that relational operations like negation and set difference can cause additional tuples to appear in an expression's output when tuples are removed from its inputs. The *additive noise* component in the differential relational algebra compensates for these additional tuples. For base relations and subexpressions that do not contain negation or set difference operators, the *additive noise* component of the shadow query is always empty. In general, the empty parts of a rewritten shadow query can be pruned by substituting the empty set ($\emptyset$) for these parts and simplifying the resultant relational algebra expression. Appendix A gives detailed definitions of several important differential operators, including the set difference operator.

My query rewriter starts by constructing a differential relational algebra expression for each SELECT clause in the original query. Each differential operator is defined in terms of the basic relational operators. The query rewriter recursively applies these definitions to the differential relational algebra expression to obtain a relational algebra expression for the tuples that are missing from the main query's output. Then the query rewriter removes empty relations and translates the relational algebra expression into the object-relational framework described in Section

3.2.1 to produce the shadow query.

Certain portions of the shadow query reference static tables that do not change during the lifetime of the user's continuous query. The query rewriter precomputes these expressions and stores the resulting summary objects in a system table. At runtime, the shadow query fetches these cached summaries instead of recomputing the corresponding subexpressions.

Finally, the query rewriter generates a single WITH statement that will run the main and shadow queries and merge their results. The application or GUI front end submits this rewritten query to the query engine, which begins executing the main and shadow queries.

3.3.3 Query Execution

When TelegraphCQ receives the rewritten query, it passes query parameters to the Triage Process and begins pulling a mix of full tuples and summaries from the Triage Queue and Summarizer. The Scheduler monitors the Triage Queue on the Packets stream and summarizes tuples that the system does not have time to process fully. Once per time window, the Scheduler sends a summary to the `__triaged_Packets` stream that serves as an input to the shadow query. The Scheduler throttles the flow of full tuples until the query engine has consumed the current summary.

The original query returns a single count of packets per time window. In place of this single count, the rewritten query will instead return two counts per time window — one from the main query and one from the shadow query. The application can add these two counts to obtain an estimate of the true query results for presentation via the appropriate interface. Keeping the results of the main and shadow queries separate provides feedback as to how much approximation went into the overall result.

In some applications, it is useful for the shadow query to return error estimates or confidence intervals, in addition to the counts, aggregate values, or tuples that the queries normally produce. For example, the designers of a graphical visualization may wish to add error bars to indicate a 90% confidence interval. For random samples, a set of recursive formulas exist for computing the error in estimating the results of a relational query with aggregation [46] Most other types of approximation can produce similar, albeit looser, bounds on error.

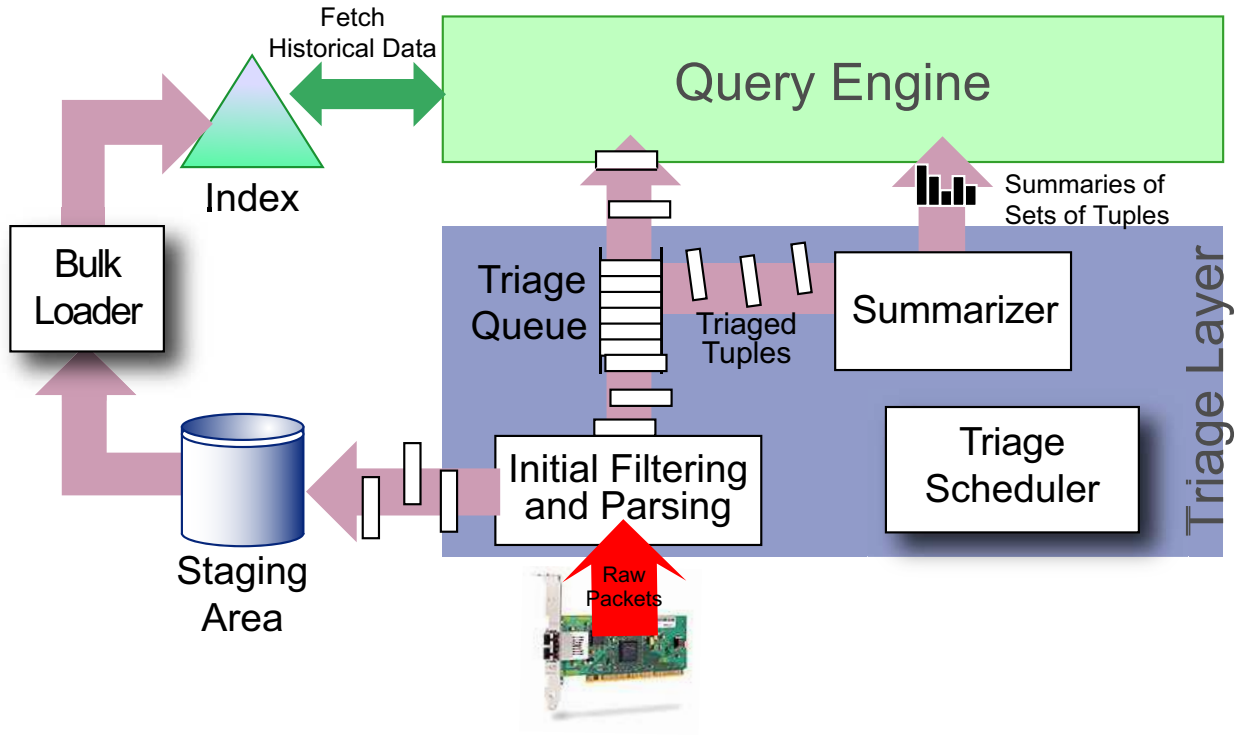

Figure 3.3: Illustration of how Data Triage can be extended to handle archived streams.

3.4 Extending Data Triage for Archived Streams

The Data Triage architecture as presented in this chapter provides an overload handling mechanism for query processors that answer queries over live streams. In many network monitoring applications, it is useful to maintain an archive of streaming data from the past. Administrators often need to examine past history to trace back the source of a problem that a "live" monitoring system detects. Also, historical data can provide a baseline for comparison against current network behavior, helping to eliminate false positives.

Data Triage can be extended in a straightforward way to handle applications that require keeping an archive of historical streaming data. Figure 3.3 illustrates how this extension of the architecture works. Archived stream data is stored in an on-disk index. In addition to being sent to the Triage Queue, streaming data is also written to an on-disk staging area. During periods of low system load, a background process bulk-loads data from the staging area into the index.

This design is partially based on Sirish Chandrasekaran's work on "live/archive" query processing [20]. Unlike this previous work, my design takes advantages of "economies of scale" by bulk-loading indexes. My approach

leads to a simplified design and higher indexing throughput at the expense of somewhat delayed index generation. In Chapter 6, I present an implementation of this design as well as experiments that characterize its throughput benefits and index loading delays.

Chapter 4

Policy

I love deadlines. I like the whooshing sound they make as they fly by.

— Douglas Adams

4.1 Introduction

The Data Triage architecture creates a tradeoff between the latency and the accuracy of query results. Different applications fit at different points along this tradeoff. This chapter presents the *delay constraints* API that I have developed to control latency and accuracy in Data Triage, along with my techniques for meeting the constraints this API imposes.

The chapter starts by analyzing the latency-accuracy tradeoff in detail. Then it presents the API, followed by my scheduling algorithm for implementing this API. I then analyze the problem of provisioning a system with Data Triage, paying particular attention to the performance the system requires from it approximate query processing implementation. I describe the results of a microbenchmark study that shows that my current implementations of several approximation schemes perform sufficiently well to provide an order of magnitude of additional data processing capacity to TelegraphCQ. Finally, I present an end-to-end experimental study on my full implementation of Data Triage.

4.2 The Latency-Accuracy Tradeoff

The Data Triage architecture uses a combination of buffering and approximation to handle overload. If tuples enter the Triage Process faster than the main query can consume them, the Triage Scheduler has two choices: buffer the excess tuples in the Triage Queue; or triage tuples in the Queue, adding them to the current approximate summary. Each of these choices has significant ramifications for the latency and accuracy of the streaming query results that the user sees, as the benchmarks in the following paragraphs demonstrate. Choosing the right mix of buffering and approximation is crucial to maintaining acceptable application performance with Data Triage.

Figure 4.1 shows the results of an experiment to measure the latency of query results in TelegraphCQ when the system uses buffering to absorb excess data. The data source for this experiment was a packet trace from the Lawrence Berkeley National Laboratories web server. I played back this trace through TelegraphCQ, using a schema similar to that in Section 1.6.1.1. I configured TelegraphCQ to run the query:

```
select count(*) from Packets [range '10 sec' slide '10 sec'];
```

My software used the timestamps in the trace to determine when to send tuples to TelegraphCQ. To simulate a less-powerful machine, I increased the playback rate of the trace by a factor of 10 and reduced the query window by a factor of 10. At these settings, the query processor was provisioned for the 90th percentile of

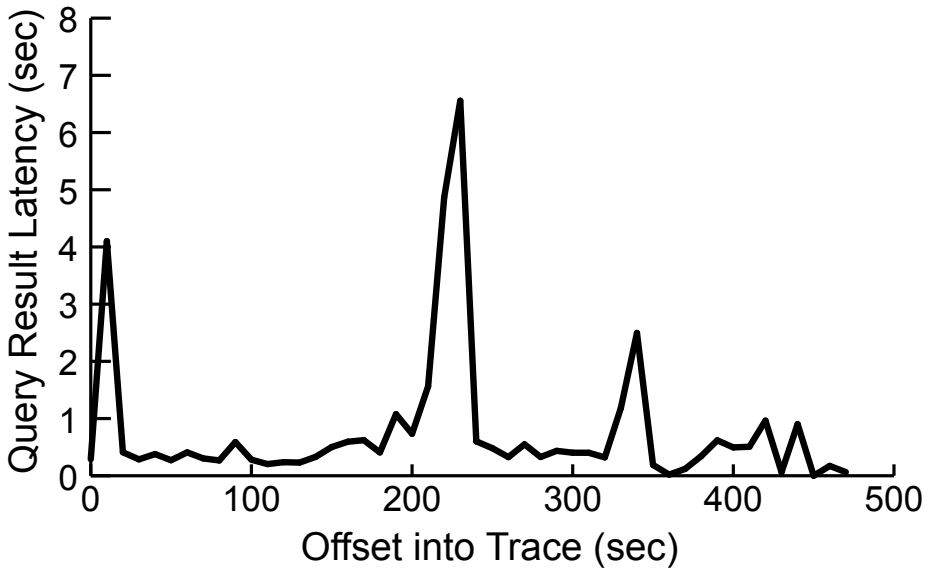

Figure 4.1: Query result latency for a simple aggregation query over a 10-second time window. The query processor is provisioned for the 90th percentile of packet arrival rates. Data is a trace of a web server's network traffic.

packet arrival rates. The graph shows the observed latency between query result generation and the end of each time window.

About 200 seconds into the trace, a burst of high-speed data overwhelms the capacity of the main query to consume data. As the amount of data in the Triage Queue increases, the buffering creates a rapidly-increasing delay between data arrival and query result generation. When the burst ends, this high latency persists for several time windows as the query processor gradually drains the buffer. Note that the heavy-tailed distribution of burst lengths means that this latency is potentially unbounded in practice.

Of course, the Triage Scheduler can limit query result latency by triaging tuples, removing them from the Triage Queue and incorporating them into a compressed summary. As long as there is enough time to summarize the entire input stream, the Scheduler can impose arbitrary limits on query result latency by limiting the number of tuples in the Triage Queue. However, this latency reduction comes at a significant cost: As the system triages larger fractions of the input data, the accuracy of query results will necessarily go down.

The Triage Scheduler must choose the right mix of buffering and approximation to ensure that the system meets application requirements for latency and accuracy. Unfortunately, different network monitoring applications have vastly dif-

```
select count(*)
from Packets P [range by '5 seconds' slide by '5 seconds'], Whois W
where P.src\string_addr ≥W.min\string_addr
    and P.src\string_addr < W.max\string_addr
    and W.name LIKE '\%berkeley.edu'
limit delay to '1 second';
```

Figure 4.2: Sample query with a delay constraint

ferent requirements for these two metrics. For example, systems that detect and respond to security breaches [110] require very low latency but can tolerate false positives. At the other extreme, systems that produce digests of network traffic behavior for offline analysis can tolerate several hours of delay but need to produce detailed, accurate results.

In the applications that I have studied, latency requirements are more strict and show significantly more variation across applications, compared with requirements for result accuracy. Because of this asymmetry, my research has focused on maintaining strict limits on query result latency while delivering the most accurate query results possible. In the sections that follow, I propose an API called *delay constraints* that allows applications to inform the Triage Scheduler of their tolerance for query result latency on a per-query basis. Later in the chapter, I present a scheduling algorithm that allows the Triage Scheduler to meet a delay constraint while minimizing the system's reliance on approximation.

4.3 Delay Constraints

A *delay constraint* is a user-defined bound on the latency between data arrival and query result generation. Figure 4.2 shows an example of my syntax for delay constraints. My modification to CQL adds the optional clause

```
limit delay to [interval]
```

to the end of the SELECT statement.

If the SELECT clause does not involve windowed aggregation, the delay constraint bounds the delay between the arrival of a tuple and the production of its corresponding join results. When the SELECT clause contains a windowed aggregate, the delay constraint becomes what I call a *windowed delay constraint*. A windowed delay constraint of D seconds means that the aggregate results for a time window are available at most D seconds after then end of the window.

Most of the monitoring queries I have studied contain windowed GROUP BY

Variable	Units	Description
D	sec	The delay constraint
W	sec	Size of the query's hopping window
C_{full}	sec (CPU)	Incremental CPU cost of sending a tuple through the main query in Data Triage
C_{shadow}	sec (CPU)	Cost of sending a tuple through the shadow query
C_{tup}	sec (CPU)	Overall cost of processing a tuple
C_{sum}	sec (CPU)	CPU cost of adding a tuple to a summary
R_{peak}	$\frac{tuples}{sec}$	The highest data rate that Data Triage can handle
R_{exact}	$\frac{tuples}{sec}$	The highest data rate at which Data Triage does not use approximation

Table 4.1: Variables used in Sections 4.4 through 4.7

and aggregation, so I concentrate here on delay constraints for windowed queries.

Table 4.1 summarizes the variable names used in this section and the ones that follow. Consider a query with a hopping time window of size W and a windowed delay constraint of D seconds. Let w_t denote the time window to which a given tuple t belongs, and let C_{tup} denote the marginal cost of processing a tuple. We assume that C_{tup} is constant across all tuples; I discuss relaxing this assumption in the Future Work section. Let $\text{end}(w)$ denote the end of window w.

The delay constraint defines a *delivery deadline* for each tuple t of

$$\text{deadline}(t) = \text{end}(w_t) + D - C_{tup} \tag{4.1}$$

It can be easily shown that, if the query processor consumes every tuple before its delivery deadline, then the query engine satisfies the delay constraint.

Note that every tuple in a hopping window has the same deadline. During the remainder of this chapter, I denote the deadline for the tuples in window w by $\text{deadline}(w)$

4.4 The Triage Scheduler

The Triage Scheduler is the control component of Data Triage. The Scheduler's primary purpose is to ensure that the system meets the current delay constraint. The Triage Scheduler meets this goal by controlling three important decisions:

- Whether to send a tuple from the Triage Queue to the main query
- Whether to "triage" a tuple from the Triage Queue, by adding it to a summary
- Whether to transfer the current summary from the Summarizer to the shadow query.

Sending tuples to the summarizer supports significantly higher data rates than full query processing, but compression operations do not have nonzero cost. As I will show in my experiments, summarizing a large number of triaged tuples requires a relatively small but still significant amount of CPU time. Likewise, relational operations on summaries can take a significant amount of time, though they only occur once per time window. In order to satisfy the user's delay constraints, the Triage Scheduler needs to take these costs into account when deciding which tuples to triage and when to triage them.

Recall from the previous section that a delay constraint of D defines a tuple delivery deadline of $\text{deadline}(t) = \text{end}(w_t) + D - C_{tup}$ for each tuple t, where C_{tup} is the time required to process the tuple.

In the Data Triage architecture, the value of C_{tup} depends on which datapath a tuple follows. Let C_{full} denote the CPU time required to send a tuple through the main query, and let C_{sum} denote the CPU time to add a tuple to the current summary. Then: $C_{tup} = \begin{cases} C_{full} & \text{(for main query)} \\ C_{sum} & \text{(for shadow query)} \end{cases}$

The Triage Scheduler also needs to account for the cost of sending summaries through the shadow query. Let C_{shadow} denote the cost *per time window* of the shadow query, including the cost of merging query results. I assume that C_{shadow} is constant regardless of the number of tuples triaged. I assume that the system uses a single CPU. Under this assumption, an increase in C_{shadow} decreases the amount of processing time available for other operations.

Incorporating the costs of the approximate datapath into the deadline equation from Section 4.3, I obtain the new equation:

$$\text{deadline}(w) = \text{end}(w) + D - C_{shadow} - \begin{cases} C_{full} & \text{(for main query)} \\ C_{sum} & \text{(for shadow query)} \end{cases} \tag{4.2}$$

In other words, the deadline for a tuple depends on whether the tuple is triaged. Of course, whether the tuple is triaged depends on the tuple's deadline.

One can satisfy all the above requirements with a single scheduling invariant. Intuitively:

$$\begin{matrix}\text{Time to process remaining} \\ \text{tuples in window}\end{matrix} \leq \begin{matrix}\text{Time before delay} \\ \text{constraint violated}\end{matrix}. \tag{4.3}$$

More formally, letting n denote the number of tuples in the Triage Queue, W the window size, O the real-time offset into the current window, and C_{full} the cost of sending a tuple through the main query:

$$nC_{full} \leq W + D - C_{shadow} - O, \tag{4.4}$$

or equivalently

$$n \leq \left(\frac{W + D - C_{shadow}}{C_{full}}\right) - \frac{O}{C_{full}}. \tag{4.5}$$

As long as the Triage Scheduler maintains this invariant (by triaging enough tuples to keep n sufficiently low), the query processor will satisfy its delay constraint. I note that the Scheduler must maintain the invariant simultaneously for *all* windows whose tuples could be in the Triage Queue.

It is important to note that n, the number of tuples that can reside in the Triage Queue without violating this invariant, *decreases linearly throughout each time window*. One could imagine using a fixed queue length to satisfy the invariant, but doing so would require a queue length of the *minimum value of n* over the entire window. In other words, using a fixed-length queue causes the system to triage tuples unnecessarily. In keeping with my philosophy of using approximation as a fallback mechanism, my scheduler avoids triaging tuples for as long as possible by continuously varying the number of tuples from the window that are permitted to reside in the Triage Queue. Figure 4.3 illustrates this variation in effective queue length.

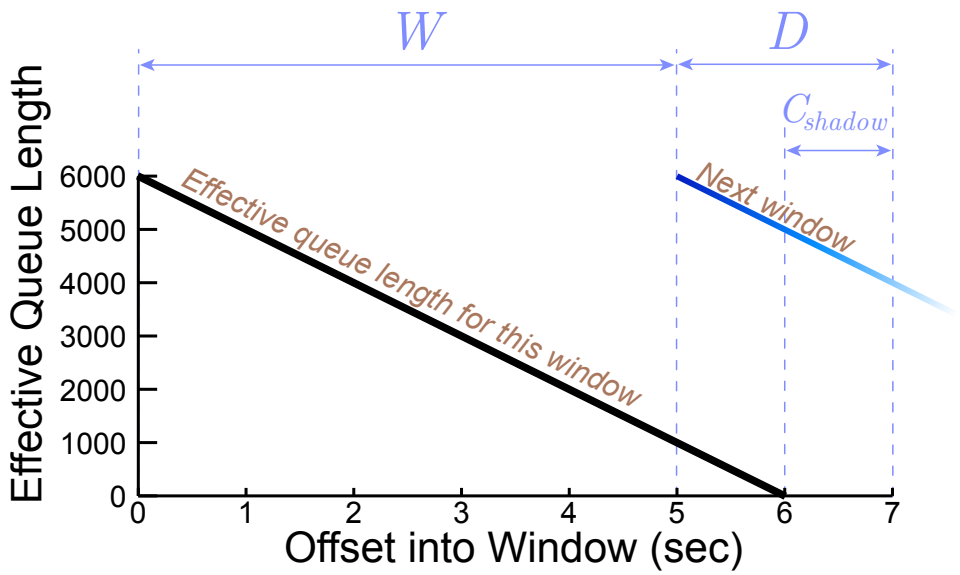

Figure 4.3: The effective length of the Triage Queue for tuples belonging to a 5-second time window, as a function of offset into the window. The delay constraint is 2 seconds, and C_{shadow} is 1 second.

4.5 General Sliding Windows

The previous sections have described my implementation of the Triage Scheduler as it applies to nonoverlapping, or "hopping," time windows. Extending this work to multiple queries with general sliding windows is straightforward.

Briefly, the framework described in this chapter can accommodate arbitrary combinations of TelegraphCQ window clauses with the following changes:

- Use the method described in [58] to convert overlapping time windows to a repeating sequence of nonoverlapping windows.
- Compute delay constraints for each of the nonoverlapping windows by determining the query delay constraints that apply at a given point in the sequence.
- When constructing shadow queries, use user-defined aggregates to merge summaries from adjacent windows as needed.

In the remainder of this section, I explain these steps in greater detail.

4.5.1 Converting overlapping windows

The main challenge in supporting general sliding windows is that the windows can overlap. In particular, a tuple could belong to any number of time windows.

If a tuple can be a member of n windows, the system should to avoid adding the tuple to n summaries.

Krishnamurthy *et al.* have developed a general method of converting any combination of overlapping window specifications into a sequence of nonoverlapping time windows [58]. The idea behind this approach is to create a new nonoverlapping window whenever one of the overlapping windows advances. The transformation produces a repeating sequence of overlapping windows of varying widths. The Triage Scheduler produces a summary for each of these nonoverlapping windows.

4.5.2 Computing delay constraints

Once the original set of sliding windows has been converted into a sequence of nonoverlapping windows, the system assigns a separate delay constraint to each window in the sequence. To compute the delay constraint for a given nonoverlapping window, the system starts by computing the result delivery deadline for every sliding window that contains the nonoverlapping window. The delay constraint is set to the time from the end of the nonoverlapping window to the earliest such deadline.

This approach guarantees that the Triage Scheduler will not triage a tuple unless at least one query requires triage in order to meet its delay constraint. At the same time, the approach is somewhat conservative in that not every query may require that a given tuple be triaged. If there is a large difference in delay tolerance among queries, the system can divide the queries into clusters and create multiple "copies" of the input stream, one for each cluster. This approach is similar to the clustering technique used by Krishnamurthy *et al.*[59].

4.5.3 Using aggregates to merge summaries

Once the system has converted a sliding window to a set of nonoverlapping windows, it needs an efficient way to merge sets of adjacent nonoverlapping windows into a single sliding window. To perform this merging, I create a user-defined aggregate function that computes the approximate `UNION ALL` of a set of summaries:

```
create aggregate Summary_agg(
    sfunc = union_all,
    basetype = Summary,
    stype = Summary
);
```

I can now write a subquery that fetches a "custom" summary of the dropped tuples in a particular window. For ease of exposition, I present this subquery as a view:

```
-- Where <fraction in window> and
-- <window number> are SQL expressions.
create view R_triaged_windowed as
select
    Summary_agg( R.summary ) as summary
    from __triaged_R  [ original  window spec]
```

4.6 Provisioning Data Triage

Data Triage uses approximate query processing as a fallback mechanism for isolating a streaming query processor from high-speed bursts of data. The effectiveness of this approach of course depends on the summary implementation being faster than the general-purpose query processor.

In particular, one would like to know:

- If the approximation method in Data Triage's shadow query performs at a given level, what will be the maximum throughput of the entire system?
- How quickly can different query approximation methods process data?

In this section, I address both of these questions. I start with a theoretical analysis of approximate query processing performance as it applies to Data Triage, then I apply my theory to a microbenchmark study of several approximation methodologies. As in previous sections, my analysis assumes that the time windows in the user's query are hopping windows.

4.6.1 Data Triage and System Capacity

Data Triage operates in two regimes: Up to a certain data rate R_{exact}, the system can perform exact query processing, sending all tuples through the main query. Above R_{exact}, Data Triage must resort to approximation to handle data rates up to a maximum of R_{peak}.

As in Section 4.4, I characterize the CPU cost of a summarization/approximation scheme by two parameters, C_{shadow} and C_{sum}. I assume for ease of exposition that these parameters are constants; similar conclusions can be reached by treating C_{shadow} and C_{sum} as random variables.

In the sections that follow, I derive the relationship between the summarization parameters, C_{shadow} and C_{sum}, and the system capacity parameters, R_{exact} and R_{peak}.

4.6.1.1 C_{shadow}

C_{shadow} represents the CPU cost incurred by sending a summary through the shadow query.

The maximum possible value of R_{exact} is $\frac{1}{C_{full}}$, the rate at which the main query can consume tuples. If the user's query involves hopping windows of length W and it takes C_{full} to process a tuple in the main query, then the number of tuples that the main query can process in a single window is

$$\frac{W - C_{shadow}}{C_{full}}. \tag{4.6}$$

Time spent processing the shadow query reduces R_{exact} by a factor of $1 - \frac{C_{shadow}}{W}$. Additionally, since the system cannot send a summary to the shadow query until the end of a time window, C_{shadow} serves as a lower bound on the delay constraint.

In summary, C_{shadow} constrains the *query* parameters D and W. In order for Data Triage to work effectively, the value of C_{shadow} needs to be less than the delay constraint D and small relative to the window size W.

4.6.2 C_{sum}

C_{sum} represents the incremental CPU cost of adding a single tuple to a summary. In contrast to C_{shadow}, C_{sum} is a *per-tuple* cost. The value of C_{sum} limits R_{peak}, the maximum instantaneous rate at which tuples can enter the system.

The system must be able to summarize incoming tuples quickly enough to meet its delay constraint. The Triage Queue can contain tuples from up to $\lfloor \frac{W}{D} \rfloor + 1$ windows at once, and the number of tuples from each window that can reside in the Triage Queue decreases at a rate of $\frac{1}{C_{full}} \frac{\text{tuples}}{\text{sec}}$. Since the system must be able to handle a sustained load of R_{peak} without dropping any tuples, the peak sustainable rate is:

$$R_{peak} = \left(1 - \frac{C_{shadow}}{W}\right)\left(\frac{1}{C_{sum}} - \left(\left\lfloor \frac{W}{D} \right\rfloor + 1\right)\frac{1}{C_{full}}\right) \tag{4.7}$$

Note that, if $C_{shadow} \ll W$ and $C_{sum} \ll C_{full}$, then $R_{peak} \approx \frac{1}{C_{sum}}$.

4.7 Performance Analysis of Approximation Techniques

As the previous sections showed, the ability of Data Triage to handle tight delay constraints and high data rates depends on the performance of the query approximation technique in use. The C_{sum} parameter affects the rate at which the system can triage excess tuples, while the C_{full} parameter affects the minimum window size and delay constraints that the system can support, as well as having a secondary effect the maximum data rate. This section describes a microbenchmark study that measures the value of these two parameters on prototype implementations of several query approximation techniques. The goal of this study is to determine whether these approximation schemes have sufficiently high performance to support a useful Data Triage implementation on TelegraphCQ.

I have implemented several summary types within the framework I described in Section 3.2.1:

- Multidimensional histograms with a fixed grid of buckets
- MHIST multidimensional histograms [82]
- Wavelet-based histograms [66]
- Reservoir sampling [106]

All of the approximation schemes I studied allow the user to adjust the tradeoff between speed and accuracy by changing a *summary granularity* parameter. For example, reservoir sampling uses a sample size parameter, and wavelet-based histograms keep a fixed number of wavelet coefficients.

I conducted a microbenchmark study to determine the relationship between summary granularity and the parameters C_{sum} and C_{shadow} for my implementations.

4.7.1 Measuring C_{sum}

My first experiment measured C_{sum}, the CPU cost of inserting a tuple into each of the data structures. The experiment inserted randomly-generated two-column tuples into the summaries. I measured the insertion cost across a range of summary granularities.

Figure 4.4 shows the results of this experiment; note the logarithmic scale on the Y axis. The X axis represents summary granularity, measured by the number of histogram buckets, wavelet coefficients, or sampled tuples.

The insertion cost for reservoir sampling was very low compared with the others, though it did increase somewhat at larger sample sizes, probably due to caching effects. Fixed-grid histograms provided low insertion times across a wide

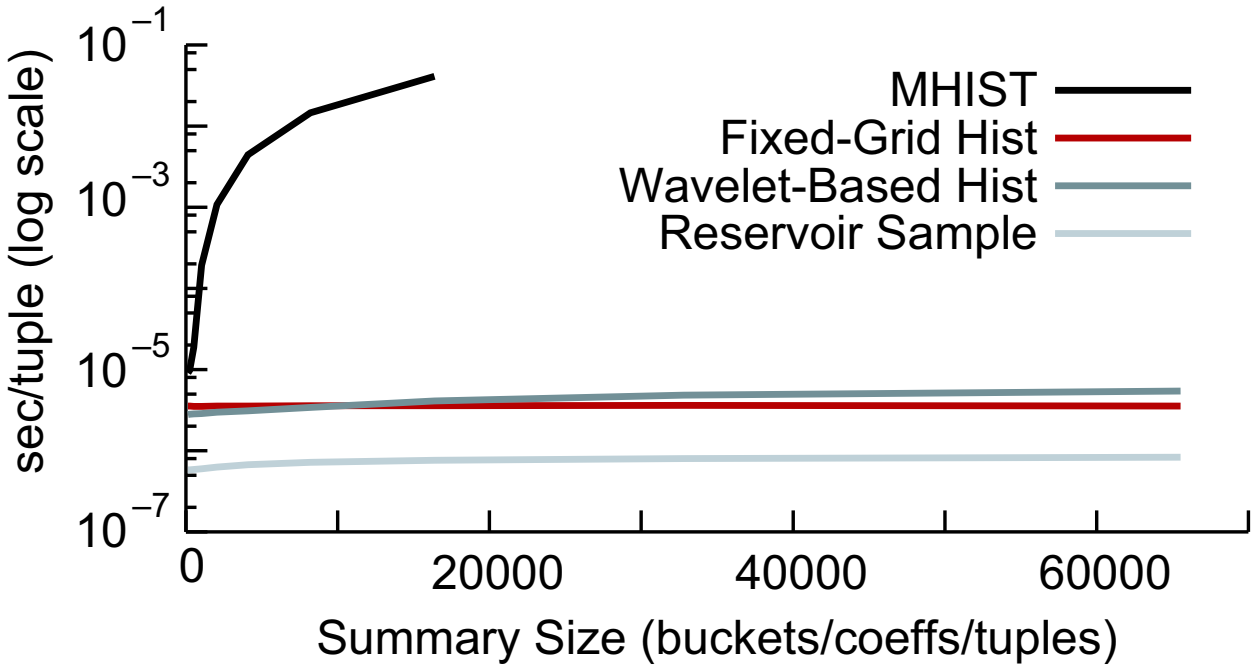

Figure 4.4: The CPU cost of inserting a tuple into the four types of summary we implemented. The X axis represents the granularity of the summary data structure.

variety of data structure sizes. The insertion operation on such a histogram is a simple index into an array, and cache effects were not significant at the summary sizes I examined. The insertion cost for wavelet-based histograms increased somewhat with summary size, primarily due to the cost of sorting to find the largest wavelet coefficients. This increase was only a factor of 2 across the entire range of wavelet sizes.

MHISTs exhibited a relatively high insertion cost that became progressively worse as the number of buckets was increased. For more than a few hundred buckets, inserting tuples into my MHIST implementation would be slower than sending these tuples through a full query processor. The high insertion cost of my implementation stems mostly from the lack of an efficient data structure for mapping tuples to the appropriate MHIST buckets. Using a kd-tree [10] to map tuples to buckets would rectify this problem. Even with this optimization, my MHIST implementation would still have a higher insertion cost than the other summaries, as evidenced by the leftmost point on the curve.

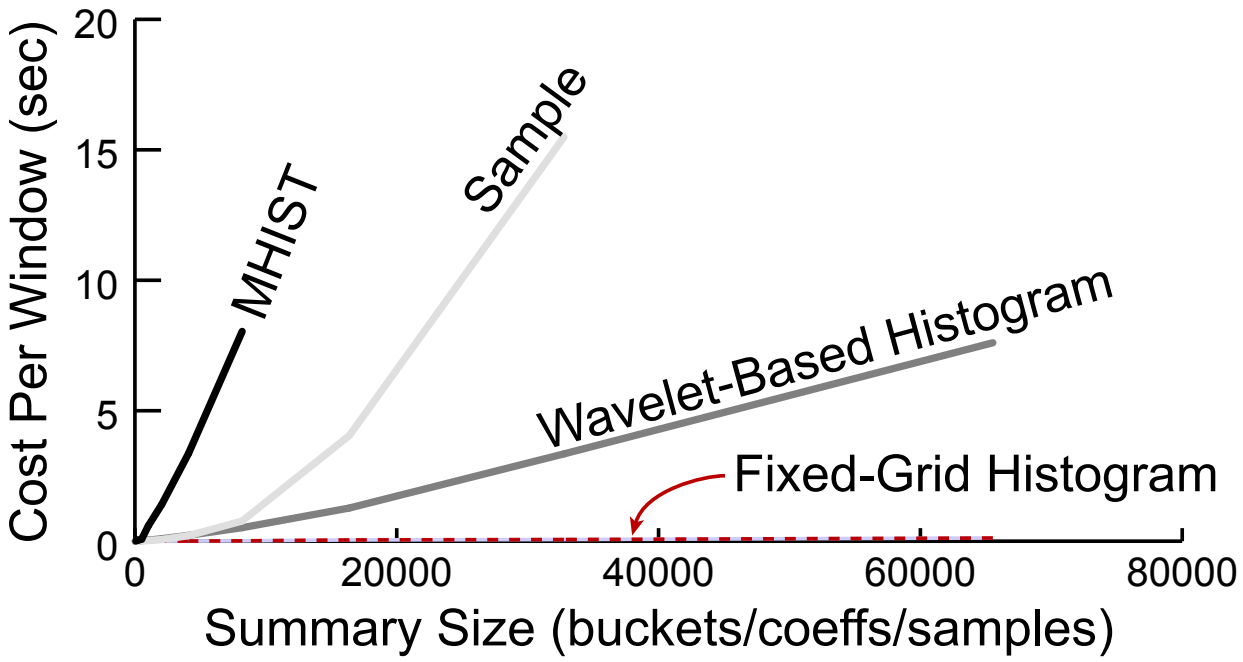

Figure 4.5: The time required to compute a single window of a shadow query using four kinds of summary data structure. The X axis represents the granularity of the summaries; the Y axis represents execution time.

4.7.2 Measuring C_{shadow}

My second experiment measured the value of the C_{shadow} constant as a function of summary granularity. The experiment measured the cost of performing the shadow query for a query involving a stream-table equijoin.

Figure 4.5 shows the results of this experiment. The cost of the join was sensitive to summary size for all summaries studied. The join costs of the four summary types were separated by significant constant factors, with MHISTs taking the longest, followed by reservoir samples, wavelet-based histograms, and fixed-grid histograms.

Again, MHISTs were significantly slower than the other histogram-based summaries. In this case, the discrepancy was due to MHIST buckets not being aligned with each other along the join dimension. This misalignment meant that each bucket joined with several other buckets and produced a large number of result buckets.

I also conducted a version of this experiment in which I varied the number of tuples inserted into the summaries. Beyond 100 tuples, the cost of the shadow

query was insensitive to the number of tuples.

4.7.2.1 Discussion

My evaluation of the four approximation schemes I have implemented shows that three of them can summarize tuples fast enough to be useful for Data Triage. On the machine used in this experiment, C_{full}, the time to process a tuple in a conventional query, typically ranges from 1×10^{-4} to 1×10^{-3} seconds, depending on query complexity. The compression functions for the three summary types can consume tuples considerably faster, with C_{sum} values of approximately 1×10^{-5} for fixed-grid or wavelet-based histograms and 1×10^{-6} for samples. I expect these times would drop significantly on a production implementation.

My shadow query microbenchmark shows that simple fixed-grid histograms have very small values of C_{shadow}, even at very fine summary granularities. Even accounting for their relatively inefficient partitioning function, these simple histograms should work better than the other summary types studied for queries with short time windows or tight delay constraints. For window sizes of 10 seconds or more, the sampling and wavelet-based histogram techniques should provide acceptable performance.

4.8 End-To-End Experimental Evaluation

The microbenchmark study in the preceding section examined the performance of the summarization and approximate query processing components of my Data Triage implementation. The results of this study showed that Data Triage has the potential to provide significant improvements in system capacity while meeting tight delay constraints. In this section, I present an end-to-end experimental study that verifies that my full Data Triage implementation can live up to this potential in a realistic environment.

I used a 105-MB trace of the traffic to and from the HTTP server `www.lbl.gov` as the input to my experiments. The query used in the experiments was a variant of the example query from the previous chapter. The current implementation of band joins in TelegraphCQ is inefficient, so I modified the query to be an equijoin on the most significant 16 bits of the IP address and created multiple lookup table entries for each subnet:

```
select count(*), ts
from Packets P [range '10 seconds' slide '10 seconds'],
     Whois W
where P.src_addr_pfx = W.addr_pfx
limit delay to ...
```

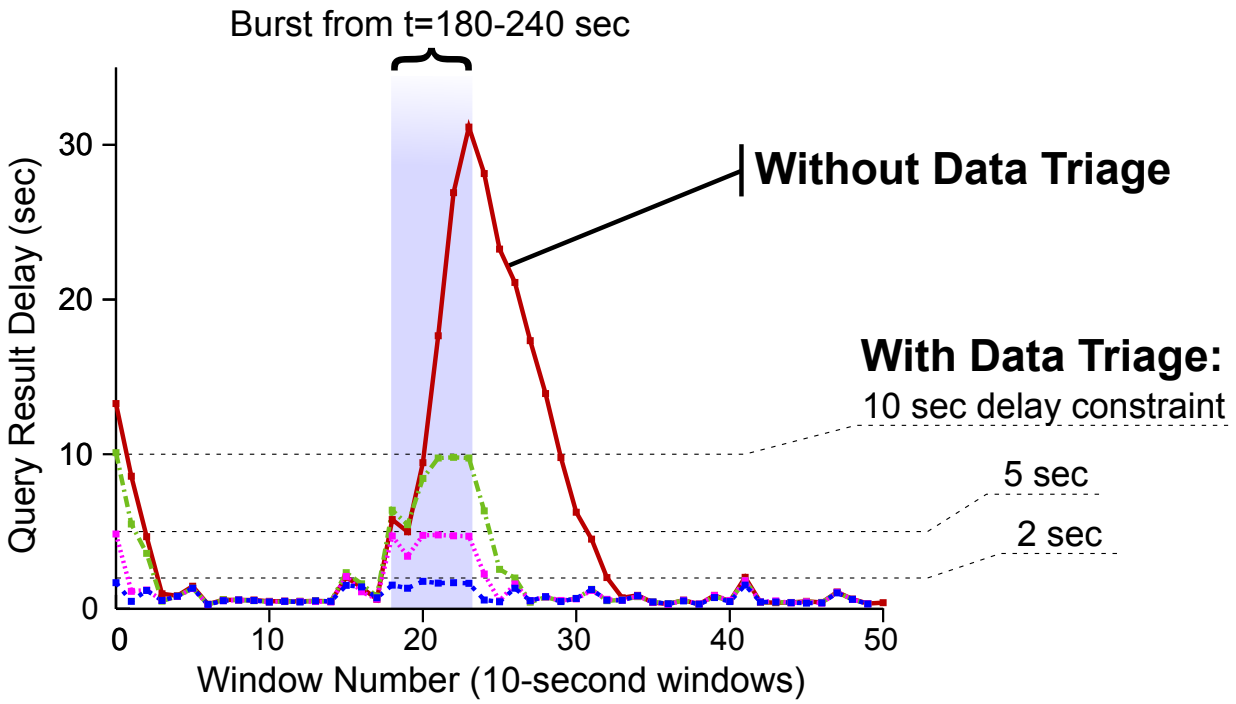

Figure 4.6: A comparison of query result latency with and without Data Triage on with the system provisioned for the 90th percentile of load. The data stream was a timing-accurate trace of a web server. Each line is the average of 10 runs of the experiment.

I ran my experiments on a server with two 1.4 GHz Pentium III CPUs and 1.5 GB of main memory. To simulate using a less powerful embedded CPU, I wrote a program that would "play back" the trace at a multiple of its original speed and decreased the delay constraint and window size of the query accordingly. I used reservoir samples as the approximation method for this experiment. I adjusted the trace playback rate to 10 times the original rate. At this data rate, my system was provisioned for the 90th percentile of packet arrival rates in my trace.

4.8.1 Query Result Latency

For my first experiment, I ran the query both with and without Data Triage and measured the latency of query results. I computed latency by measuring the time at which the system output the result for each window and subtracting the window's last timestamp from this figure. I repeated the experiment 10 times and recorded the average latency for each time window.

Figure 4.6 shows a graph of query result latency during the first 500 seconds of the trace. The the line marked "Without Data Triage" shows the latency of the

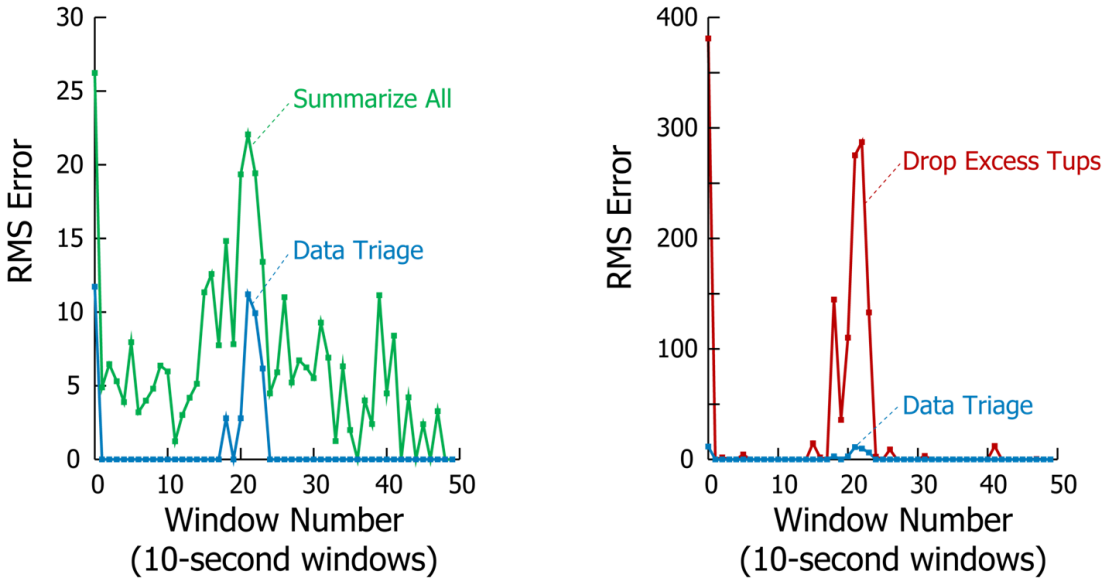

Figure 4.7: A comparison of query result accuracy using the same experimental setup as in Figure 4.6 and a 2-second delay constraint. Data Triage outperformed the other two load-shedding methods tested. Each line is the average of 10 runs of the experiment.

query on an unmodified version of TelegraphCQ. The other lines show the latency of TelegraphCQ with Data Triage and delay constraints of 10, 5, and 2 seconds, respectively.

Approximately 180 seconds into the trace, a 50-second burst exceeds the query processor's capacity. Without Data Triage, the unmodified version of TelegraphCQ falls steadily behind the trace and does not catch up until 90 seconds after the end of the burst.

With Data Triage enabled, the Triage Scheduler shunts excess tuples to the shadow query as needed to satisfy the delay constraint. As the graph shows, the system triages just enough tuples to avoid violating the constraint, performing full processing on as much of the input data as possible.

4.8.2 Query Result Accuracy

My second experiment examined how well Data Triage does at keeping approximation error in check in the presence of bursty data rates. The experiment compared the Data Triage approach against two alternate methods of handling bursty streams. The metric that I used for comparison was the amount of approximation error in the query results. I measured result error within each time window using a root-mean-squared error metric. That is, I defined the error for time win-

dow w as:

$$E_w = \sqrt{\frac{\sum_{g \in \text{groups}} (\text{actual}(g) - \text{reported}(g))^2}{|\text{groups}|}} \tag{4.8}$$

where groups is the set of groups in the query's output for a given time window.

Using this error metric and the same query and experimental setup as the previous experiment, I measured the result error of three load-shedding methods:

- **Data Triage** as described in this chapter
- **Drop Excess Tuples:** When the delay constraint is about to be violated, drop the remaining tuples in the window.
- **Summarize All:** Generate summaries of all tuples and perform approximate query processing on the summaries.

I used a reservoir sample as the summary type for both Data Triage and the Summarize All technique. I tuned the reservoir size to the delay constraint and the maximum data rate in the trace.

Figure 4.7 shows the results for the first 500 seconds of this experiment. Throughout the trace, Data Triage provides more accurate results than either of the other methods. During the bursts in windows 0 and 18-21, Data Triage processes as many tuples as possible before resorting to approximation. The Drop Excess Tuples method, on the other hand, generates query results that are missing significant chunks of the data. Likewise, the Summarize All method drops tuples that could have been processed fully.

During the periods in between bursts, both Data Triage and the Drop Excess Tuples method processed all tuples in each window, producing no error. The error for Summarize All also decreased somewhat during these lulls, as the reservoir sample covered a larger portion of the data in the window.

Chapter 5

Histograms for IP Address Data

5.1 Introduction

Data Triage relies on extensive previous work in query approximation to provide a fast but approximate way of generating query answers. My architecture treats the approximation method as a "black box" consisting of a summarizer and a set of operators. This approach allows the system to use different approximation schemes for different queries, which is important because different schemes work better for different queries.

In the process of implementing and testing Data Triage, I discovered an important class of queries on which existing approximation techniques provide poor approximations. These queries involve joining streams of *unique identifiers* like IP addresses with tables of metadata about the objects to which those identifiers refer, then aggregating according to the metadata. The general form of such a query is:

```
select   G.GroupId, AGG(...)
from     UIDStream U [range ... slide   ...],
         GroupTable G
where    G.uid = U.uid
group by G.GroupId;
```

where AGG is an aggregate.

This class of query has many important uses in the network monitoring domain, such as:

- Producing a breakdown of network traffic by administrative domain.
- Comparing present traffic patterns against clusters identified in an offline analysis.
- Geocoding IP addresses to produce a graph of physical location.

In my early experiments, I tried several existing techniques for approximating this class of queries, and the results were disappointing. Techniques based on sampling resulted in high approximation error for groups represented by few tuples. Histogram-based techniques tended to group together ranges of the IP address space corresponding to different entries in the lookup table, again resulting in large errors.

Since Data Triage needs an effective approximation technique to be able to cover this class of queries, I have developed novel histogram-based techniques for approximating these queries. My histograms exploit the inherent hierarchical structure of the IP address space and and other unique identifier spaces to produce

a compact representation of a set of identifiers that can be joined with a lookup table.

Although I developed them for network monitoring in a system with Data Triage, these histograms have additional uses in several other contexts. The class of query that these histograms target arises in a number of different monitoring scenarios, ranging from supply chain monitoring to military intelligence. In these applications, a monitoring station receives a stream of unique identifiers, such as UPC symbols or credit card numbers. The monitor uses one or more lookup tables to map the identifiers to information about the objects they identify, then produces periodic reports that break down the stream in terms of this information.

Like the IP address space, many of these unique identifiers have an inherent hierarchical structure, for reasons similar to those outlined in Section 5.2 of this chapter. Many types of identifier are assigned in batches, and there are often additional technical reasons (Efficient postal routing with zip codes and validation of credit card numbers are two examples.) for enforcing a hierarchy. Unlike the IP address hierarchy, these other hierarchies are generally not binary hierarchies. However, as I explain in Section 5.5.1, my algorithms extend to arbitrary hierarchies in a straightforward manner.

My histograms can be especially helpful in cases where the stream of unique identifiers and the lookup tables are at different locations, separated by low-bandwidth links. A histogram of a time window of the stream serves as a compressed representation of the information in that time window.

This chapter starts by explaining the origins and basic structure of the IP address hierarchy. I then explain how my histograms work and give a formal problem definition. Then I present efficient algorithms for precomputing a histogram partitioning function given a lookup table and an estimate of the stream's data distribution. Finally, I present an experimental evaluation of my techniques on real network data.

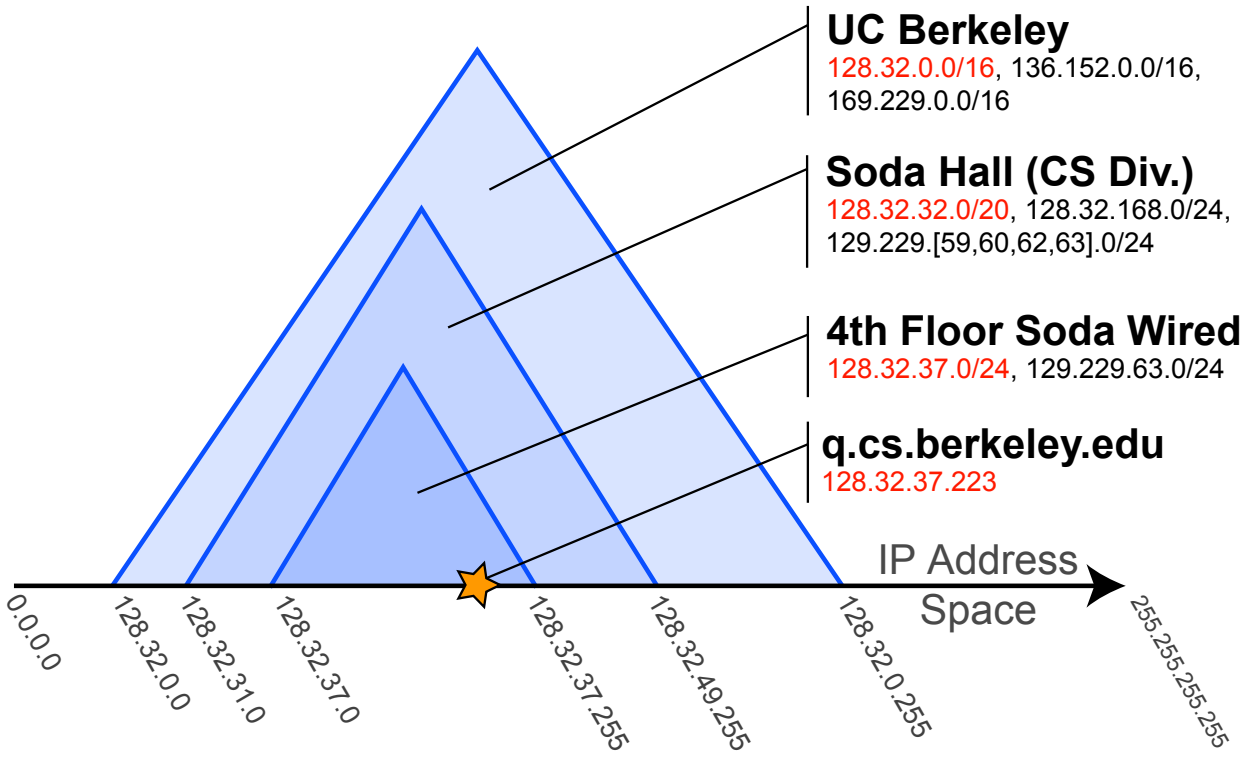

Figure 5.1: Diagram of the IP address hierarchy in the vicinity of my workstation. Address range sizes are not to scale.

5.2 The IP Address Hierarchy

At first glance, an IP address appears to be a random string of bits. On closer examination, however, there turns out to be a strong hierarchical structure in most of the IP address space.

Figure 5.1 shows the IP address space in the vicinity of my workstation, `q.cs.berkeley.edu`. I gleaned the information in the diagram from tables on the web site of U.C. Berkeley's Communication and Network Services division. The IP addresses in the diagram are divided into nested ranges that show a strong correlation with physical locations (e.g. fourth floor of Soda Hall), administrative domains (e.g. Computer Science Division), and logical location (e.g. inside the U.C. Berkeley campus network).

This hierarchy and its correlation with physical, administrative, and logical location arise due to several administrative and technical factors:

- **Administrative Factors:** The Internet Assigned Numbers Authority (IANA) [7]

controls the allocation of IP addresses throughout the Internet. IANA allocates *IP address prefixes* according to the Classless Interdomain Routing (CIDR) standard [35]. An address prefix consists of all the IP addresses that share a given number of their most significant bits. For example, the prefix `128.32.0.0/16` represents all addresses that share their most significant 16 bits with the address `128.32.0.0`; that is, the range from `128.32.0.0` to `128.32.255.255`, inclusive.

IANA generally assigns large IP address prefixes to Internet Service Providers and other large organizations. These organizations in turn allocate sub-prefixes to customers or suborganizations, who may further subdivide their address blocks, and so on. As a result, most of the IP address space can be divided into nested blocks that correspond closely to customer-provider relationships.

- **Technical Factors:** The CIDR prefixes themselves exist because they greatly simplify the design of Internet routers. The Border Gateway Protocol (BGP) routers that route most Internet backbone traffic use routing tables that map prefixes of the IP address space to routing destinations. The destination address in a packet is routed to the destination that shares the longest possible address prefix [88]. This design allows routers to truncate routing tables by removing longer prefixes, as well as allowing for efficient hardware acceleration [73, 55, 105].

  As a side effect, longest-prefix-match routing creates a strong incentive for network administrators to make the physical and logical structure of their networks correlate closely with a tree of nested IP address prefixes. If all the computers on a local network share the same IP address prefix, then routers will need only one routing table entry to find the most efficient route to those computers. This correlation keeps routing tables compact and helps ensure that routers will choose the most efficient route for a given packet.

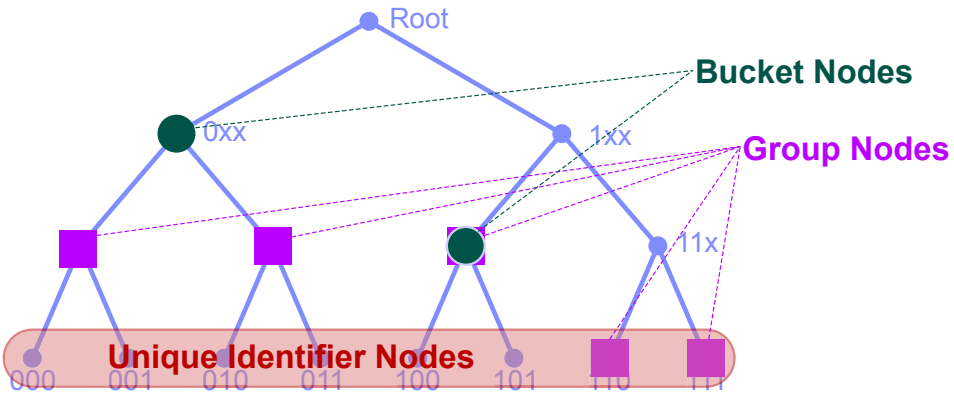

Figure 5.2: A 3-level binary hierarchy of unique identifiers.

5.3 Problem Definition

The remainder of this chapter describes algorithms for choosing optimal histogram partitioning functions over a general hierarchical unique identifier space. I start with a description of the theoretical problem that I solve in the rest of the chapter. First, I specify the classes of partitioning function that my algorithms generate. Then I describe the criteria that I use to rank partitioning functions.

My partitioning functions operate over streams of unique identifiers (UIDs). These unique identifiers form the leaves of a hierarchy, which I call the *UID hierarchy*. Figure 5.2 illustrates a simple binary UID hierarchy. My work handles arbitrary hierarchies, as I show in Section 5.5.1, but I limit my discussion here to binary hierarchies for ease of exposition.

As Figure 5.2 shows, certain nodes within the UID hierarchy will have special significance in my discussion:

- *Group nodes* (shown as squares in Figure 5.2) define the *groups* within the user's GROUP BY query. In particular, each group node resides at the top of a subtree of the hierarchy. The UIDs at the leaves of this subtree are the members of the group. In my problem definition, these subtrees cannot overlap.
- *Bucket nodes* (large circles in Figure 5.2) define the *partitions* of my partitioning functions. During query execution, each of these partitions defines a *bucket* of a histogram. The semantics of the bucket nodes vary for different classes of partitioning functions, as I discuss in the next section.

In a nutshell, my goal is to approximate many squares using just a few circles; that is, to estimate aggregates at the group nodes by instead computing aggregates for

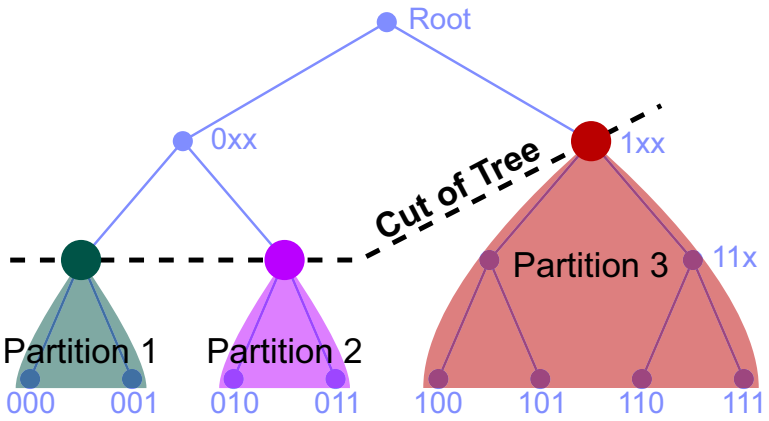

Figure 5.3: A partitioning function consisting of nonoverlapping subtrees. The roots of the subtrees form a cut of the main tree. In this example, the UID 010 is in Partition 2.

a carefully-chosen (and much smaller) collection of bucket nodes.

5.3.1 Classes of Partitioning Functions

The goal of my algorithms is to choose optimal histogram partitioning functions. I represent my partitioning functions with sets of bucket nodes within the hierarchy. I have studied three different ways of interpreting a set of bucket nodes: *Nonoverlapping*, *Overlapping*, and *Longest-Prefix-Match*. The sections that follow define the specifics of each of these interpretations.

5.3.1.1 Nonoverlapping Partitioning Functions

My simplest class of partitioning functions is for *nonoverlapping* partitionings. A nonoverlapping partitioning function divides the UID hierarchy into disjoint subtrees, as illustrated by Figure 5.3. I call the hierarchy nodes at the roots of these subtrees the *bucket nodes*. Note that the bucket nodes form a cut of the hierarchy. Each unique identifier maps to the bucket of its ancestor bucket node. For example, in Figure 5.3, the UID 010 maps to Partition 2.

Nonoverlapping partitioning functions have the advantage that they are easy to compute. In Section 5.4.2.2, I will present a very efficient algorithm to compute the optimal nonoverlapping partitioning function for a variety of error metrics. Compared with my other types of partitioning functions, nonoverlapping partitioning functions produce somewhat inferior histograms in my experiments. However, the speed with which these functions can be chosen makes them an attractive choice for lookup tables that change frequently.

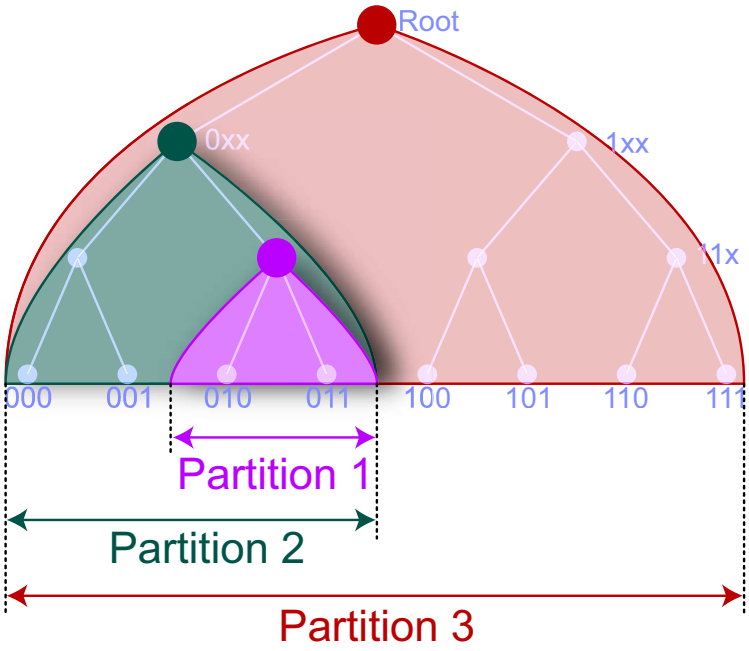

Figure 5.4: An overlapping partitioning function. Each unique identifier maps to the buckets of all bucket nodes above it in the hierarchy. In this example, the UID 010 is in Partitions 1, 2, and 3.

5.3.1.2 Overlapping Partitioning Functions

The second class of functions I consider is the class of *overlapping* partitioning functions. Figure 5.4 shows an example of this kind of function. Like a nonoverlapping function, an overlapping partitioning function divides the UID hierarchy into subtrees. However, the subtrees in an overlapping partitioning function may overlap. As before, the root of each subtree is called a bucket node. In this case, "partitioning function" is something of a misnomer, since a unique identifier maps to the "partitions" of *all* the bucket nodes between it and the root. In the example illustrated in the diagram, the UID 010 maps to Partitions 1, 2, and 3.

Overlapping partitioning functions provide a strictly larger solution space than nonoverlapping functions. I have adapted my dynamic programming algorithm for nonoverlapping partitioning functions to the space of overlapping functions. The worst-case running time of the new algorithm is larger by a logarithmic factor, due to the larger size of the solution space. This increase in running time is offset by a decrease in error. In my experiments, overlapping partitioning functions produce histograms that more efficiently represent network traffic data, compared with existing techniques.

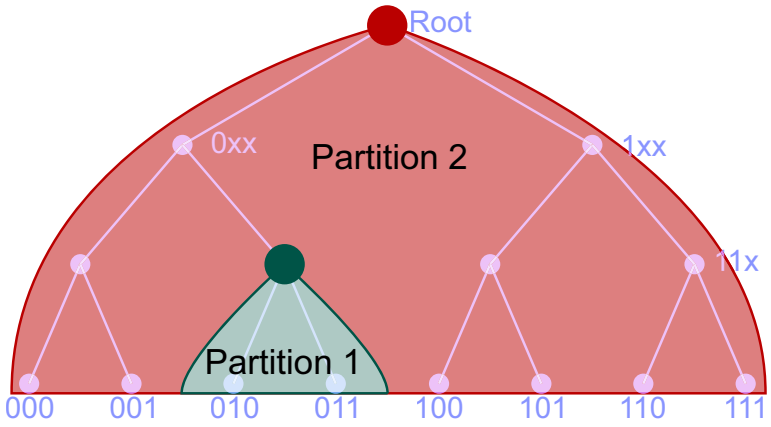

Figure 5.5: A longest-prefix-match partitioning function over a 3-level hierarchy. The highlighted nodes are called *bucket nodes*. Each leaf node maps to its closest ancestor's bucket. In this example, node 010 is in Partition 1.

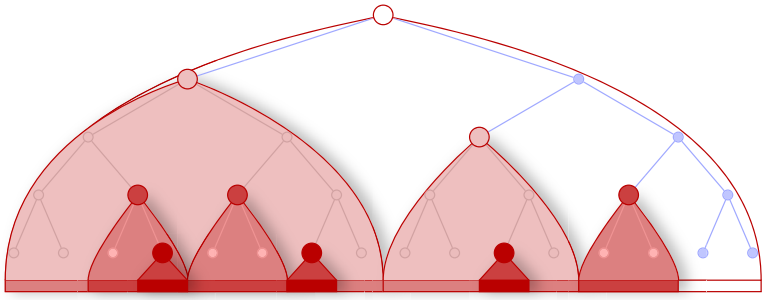

Figure 5.6: A more complex longest-prefix-match partitioning function, showing some of the ways that partitions can nest.

5.3.1.3 Longest-Prefix-Match Partitioning Functions

My final class of partitioning functions is called the *longest-prefix-match* partitioning functions. A longest-prefix-match partitioning function uses bucket nodes to define partition subtrees, as with an overlapping partitioning function. However, in the longest-prefix-match case, each UID maps only to the partition of its *closest* ancestor bucket node (selected in the histogram). Figure 5.5 illustrates a simple longest-prefix-match function. In this example, UID 010 maps to Partition 1 only. Figure 5.6 illustrates a more complex longest-prefix-match partitioning function. As the figure shows, partitions can be arbitrarily nested, and a given partition can have multiple "holes".

Longest-prefix-match functions are inspired by the routing tables for

inter-domain routers on the Internet. As noted earlier in this chapter, these routing tables map prefixes of the IP address space to destinations, and each address is routed to the destination of the longest prefix that matches it. This routing algorithm not only reflects the inherent structure of Internet addresses, it reinforces this structure by making it efficient for an administrator to group similar hosts under a single prefix.

Longest-prefix-match partitioning has the potential to produce histograms that give very compact and accurate representations of network traffic. However, choosing an optimal longest-prefix-match partitioning function turns out to be a difficult problem. I propose an algorithm that explores a limited subset of longest-prefix-match partitionings and requires at least cubic time (while offering certain approximation guarantees for the resulting histogram), as well as two sub-quadratic heuristics that can scale to large data sets. In my experiments, longest-prefix-match partitioning functions created with these heuristics produce better histograms in practice than optimal partitioning functions from the other classes.

5.3.2 Measuring Optimality

Having described the classes of partitioning functions that my algorithms produce, I can now present the metric I use to measure the relative "goodness" of different partitioning functions.

5.3.2.1 The Query

The introduction to this chapter gave a general description the type of query that can be approximated with my histograms. To simplify my formal problem definitions, the theoretical analysis in this chapter uses a slightly constrained version of this query:

```
select  G.gid, count(*)
from    UIDStream U [sliding window],
        GroupHierarchy G
where   G.uid = U.uid
        -- GroupHierarchy places all UIDs below
        -- a group node in the same group.
group by G.node;
```

As in the introduction, this query joins UIDStream, a stream of unique identifiers, with a lookup table, Group−Hierarchy, and then performs aggregation. However, the table Group−Hierarchy in the above query explicitly refers to the nodes of the hierarchy. In this chapter, I focus on groups that consist of non-overlapping subtrees of the UID hierarchy. I call the root of each such subtree a *group node*. Note that,

since the subtrees cannot overlap, no group node can be an ancestor of another group node. Also for ease of exposition, the aggregate in this query is a COUNT aggregate; the extension of my work to other SQL aggregates is straightforward.

5.3.2.2 The Query Approximation

My algorithms generate partitioning functions for the purposes of approximating a query like the one in the previous section. The input of this approximation scheme is a window's worth of tuples from the UIDStream stream. The Summarizer component of Data Triage (See Section 6.3) uses the partitioning function to partition the UIDs in the window into histogram buckets, and the system keeps a count for each bucket. Within each bucket, the approximation algorithm assumes that the counts are uniformly distributed among the groups that map to the bucket. This uniformity assumption is a common feature of most histogram-based query approximations [81] and leads to an estimated count for each group. For overlapping partitioning functions, only the closest enclosing bucket is used to estimate the count for each group.

5.3.2.3 The Error Metric

The query approximation in the previous section produces an estimated count for each group in the original query. There are many ways to quantify the effectiveness of such an approximate answer, and different metrics are appropriate to different applications. My algorithms work for a general class of error metrics that I call *distributive error metrics*.

A distributive error metric is a distributive aggregate [41] $\left\langle \text{start}, \oplus, \text{finalize} \right\rangle$, where:

- start is a function on groups that converts the actual and estimated counts for a group into a "partial state record" (PSR);
- $\oplus$ is a commutative and associative operator that merges the two PSRs; and,
- finalize is a function that converts a PSR into a numeric error.

In addition to being distributive, the aggregate that defines a distributive error metric must also satisfy the following "monotonicity" properties for any PSRs A,

B, and C[1]:

$$\text{finalize}(B) > \text{finalize}(C) \rightarrow \text{finalize}(A \oplus B) \geq \text{finalize}(A \oplus C) \tag{5.1}$$

$$\text{finalize}(B) = \text{finalize}(C) \rightarrow \text{finalize}(A \oplus B) = \text{finalize}(A \oplus C) \tag{5.2}$$

As an example, consider the common *average error* metric:

$$\text{Error} = \frac{\sum_{g \in G} |g.\text{actual} - g.\text{approx}|}{|G|} \tag{5.3}$$

where G is the set of groups in the query result. Average error can be defined as:

$$\text{start}(g) = \langle |g.\text{actual} - g.\text{approx}|, 1 \rangle \tag{5.4}$$

$$\langle s_1, c_1 \rangle \oplus \langle s_2, c_2 \rangle = \langle s_1 + s_2, c_1 + c_2 \rangle \tag{5.5}$$

$$\text{finalize}(\langle s, c \rangle) = \frac{s}{c} \tag{5.6}$$

Note that this metric uses an intermediate representation of $\langle \text{sum}, \text{count} \rangle$ while summing across buckets. A distributive error metric can use any fixed number of counters in a PSR.

In addition to the average error metric defined above, many other useful measures of approximation error can be expressed as distributive error metrics. Some examples include:

- RMS error:

$$\text{Error} = \sqrt{\frac{\sum_{g \in G} (g.\text{actual} - g.\text{approx})^2}{|G|}} \tag{5.7}$$

- Average relative error:

$$\text{Error} = \frac{\sum_{g \in G} \frac{|g.\text{actual} - g.\text{approx}|}{\max(g.\text{actual}, b)}}{|G|} \tag{5.8}$$

  where b is a constant to prevent division by zero (typically chosen as a low-percentile actual value from historical data [37]).

- Maximum relative error:

$$\text{Error} = \max_{G \in G} \left(\frac{|g.\text{actual} - g.\text{approx}|}{\max(g.\text{actual}, b)} \right) \tag{5.9}$$

I examine all four of these error metrics in my experiments.

[1] These properties ensure that the principle of local optimality needed by my dynamic programs holds. Intuitively, the properties mean that error increases monotonically as a solution is constructed from subtrees of the hierarchy; hence the name "monotonicity".

5.4 Algorithms

Having defined the histogram construction problems I solve in this chapter, I now present dynamic programming algorithms for solving them. Section 5.4.1 gives a high-level description of my general dynamic programming approach. Then, Section 5.4.2 gives specific recurrences for choosing partitioning functions.

5.4.1 High-Level Description

My algorithms perform dynamic programming over the UID hierarchy. In the context of the Data Triage architecture, these algorithms would be run periodically to update the histogram partitioning function used by the Summarizer component in response to changes in the lookup table or the distribution of IP addresses in the stream.

I expect that the number of groups, $|G|$, will be very large. To keep the running time for each batch tractable, I focus on making my algorithms efficient in terms of $|G|$.

For ease of exposition, I will assume for the time being that the hierarchy is a binary tree; later on, I will relax this assumption. For convenience, I number the nodes of the hierarchy 1 through n, such that the children of the node with index i are nodes $2i$ and $2i + 1$. Node 1 is the root.

The general structure of all my algorithms is to traverse the hierarchy bottom-up, building a dynamic programming table E. Each entry in E will hold the smallest error for the subtree rooted at node i, given that B nodes in that subtree are bucket nodes. (In some of my algorithms, there will be additional parameters beyond i and B, increasing the complexity of the dynamic program.) I also annotate each entry E with the set of bucket nodes that produce the chosen solution. In the end, my algorithms will look for the solution that produces the least error at the root (for any number of buckets $\leq b$, the specified space budget for the histogram).

5.4.2 Recurrences

For each type of partitioning function, I will introduce a *recurrence relation* (or "recurrence") that defines the relationship between entries of the table E. In this section, I present the recurrence relations that allow us to find optimal partitioning functions using the algorithm in the previous section. I start by describing the notation I use in my equations.

5.4.2.1 Notation

Table 5.1 summarizes the variable names I use to define my recurrences. For ease of exposition, I also use the following shorthand in my equations:

- If A and B are PSRs, I say that $A < B$ if $\mathsf{finalize}(A) < \mathsf{finalize}(B)$.

Variable	Description
U	The universe of unique identifiers.
H	The UID hierarchy, a set of nodes $h_1, h_2, \ldots, h_n$. I order nodes such that the children of h_i are h_{2i} and h_{2i+1}.
G	The group nodes; a subset of H.
b	The given budget of histogram buckets.
start	The starting function of the error aggregate (see Section 5.3.2.3).
$\oplus$	The function that merges error PSRs (Section 5.3.2.3).
$\mathsf{finalize}$	The function that converts the intermediate error PSRs to a numeric error value (Section 5.3.2.3).
$\mathsf{grperr}(i)$	The result of applying start and $\oplus$ to the groups below h_i (see Section 5.4.2).

Table 5.1: Variable names used in the equations in this chapter.

- For any set of group nodes $G = \{g_1, \cdots, g_k\}$, $\mathsf{grperr}(G)$ denotes the result of applying the starting and transition functions of the error aggregate to G:

$$\mathsf{grperr}(G) = \mathsf{start}(g_1) \oplus \mathsf{start}(g_2) \oplus \cdots \oplus \mathsf{start}(g_k) \tag{5.10}$$

5.4.2.2 Nonoverlapping Partitioning Functions

Recall from Figure 5.3 that a nonoverlapping partitioning function consists of a set of nodes that form a cut of the UID hierarchy. Each node in the cut maps the UIDs in its child subtrees to a single histogram bucket.

Let $E[i, B]$ denote the minimum total error possible using B nodes to bucketize the subtree rooted at h_i (the i^{th} node in the hierarchy). Then:

$$E[i, B] = \begin{cases} \mathsf{grperr}(i) & \text{if } B = 1, \\ \min_{1 \le c \le B} \left(E[2i, c] \oplus E[2i+1, B-c] \right) & \text{otherwise} \end{cases} \tag{5.11}$$

where $\oplus$ represents the appropriate operation for merging errors for the error measure and $\mathsf{grperr}(i)$ denotes the result of applying the start and $\oplus$ components of the error metric to the groups below h_i.

Intuitively, this recurrence consists of a base case ($B = 1$) and a recursive case ($B > 1$). In the base case, the only possible solution is to make node node i a bucket node. For the recursive case, the algorithm considers all possible ways of dividing the current bucket budget B among the left and right subtrees of h_i, and simply selects the one resulting in the smallest error.

Observe that the algorithm does not need to consider making any node below a group node into a bucket node. So the algorithm only needs to compute entries

of E for nodes that are either group nodes or their ancestors. The number of such nodes is $O(|G|)$, where G is the set of group nodes. Not counting the computation of grperr, the algorithm does at most $O(b^2)$ work for each node it touches ($O(b)$ work for each of $O(b)$ table entries), where b is the number of buckets. A binary-search optimization is possible for certain error metrics (e.g., maximum relative error), resulting in a smaller per-node cost of $O(b \log b)$.

For RMS error, one can compute all the values of $\mathsf{grperr}(i)$ in $O(|G|)$ amortized time by taking advantage of the fact that the approximate value for a group is simply the average of the actual values within, which can be computed by carrying sums and counts of actual values up the tree. So my algorithm runs in $O(|G|b^2)$ time overall for RMS error. For other error metrics, computing the values of grperr takes $O(|G| \log |U|)$ amortized time (where U is the set of unique identifiers), so the algorithm requires $O(|G|(b^2 + \log |U|))$ time.

5.4.2.3 Overlapping Partitioning Functions

In this section, I extend the recurrence of the previous section to generate overlapping partitioning functions, as illustrated in Figure 5.5. As the name suggests, overlapping partitioning functions allow configurations of bucket nodes in which one bucket node's subtree overlaps another's. To cover these cases of overlap, I add a third parameter, j, to the table E from the previous section to create a table $E[i, B, j]$. Parameter j represents the index of the closest ancestor of node i that has been selected as a bucket node. I add the j parameter because the algorithm needs to know about the enclosing partition to decide whether to make node i a bucket node. In particular, if node i is *not* a bucket node, then the groups below node i in the hierarchy will map to node j's partition.

Similarly, I augment grperr with a second argument: $\mathsf{grperr}(i, j)$ computes the error for the groups below node i when node j is the closest enclosing bucket node. The new dynamic programming recurrence can be expressed as:

$$E[i, B, j] = \begin{cases} \mathsf{grperr}(i, j) & \text{if } B = 0, \\ \min_{0 \le c \le B} \left(E[2i, c, i] \oplus E[2i+1, B-c-1, i] \right) & \\ \quad \text{if } B \ge 1 \text{ and } i = j, & (i \text{ is a bucket node}) \\ \min_{0 \le c \le B-1} \left(E[2i, c, j] \oplus E[2i+1, B-c, j] \right) & \\ \quad \text{otherwise} & (i \text{ is not a bucket node}) \end{cases} \tag{5.12}$$

Intuitively, the recurrence considers all the ways to divide a budget of B buckets among node i and its left and right subtrees, given that the next bucket node up the hierarchy is node j. For the cases in which node i is a bucket node, the recurrence

conditions on node i being its children's closest bucket node.

This algorithm computes $O(|G|bh)$ table entries, where h is the height of the tree, and each entry takes (at most) $O(b)$ time to compute. Assuming that the UID hierarchy forms a balanced tree, my algorithm will run in $O(|G|b^2 \log |U|)$ time.

5.4.2.4 Longest-Prefix-Match Partitioning Functions

Longest-prefix-match partitioning functions are similar to the overlapping partitioning functions that were discussed in the previous section. Both classes of functions consist of a set of bucket nodes that define nested partitions. A longest-prefix-match partitioning function, however, maps a given unique identifier to the partition of its closest ancestor bucket node, as opposed to mapping the UID to the partitions of all bucket nodes between it and the root. This difference in the semantics of the bucket node set renders the optimal histogram construction problem significantly harder.

An algorithm that finds a longest-prefix-match partitioning function must decide whether each node in the hierarchy is a bucket node. Intuitively, this choice is hard to make because it must be made for every node at once. A given partition can have several (possibly nested) subpartitions that act as "holes", removing chunks of the UID space from the parent partition. Each combination of holes produces a different amount of error both within the holes themselves and also in the parent partition.

For example, consider the example in Figure 5.7. Assume for the sake of argument that node A is a bucket node. Should node B also be a bucket node? This decision depends on what other nodes below A are also bucket nodes. For example, making node C a bucket node will remove C's subtree from A's partition. This choice could change the error for the groups below B, making B a more or less attractive candidate to also be a bucket node. At the same time, the decision whether to make node C a bucket node depends on whether node B is a bucket node. Indeed, the decision for each node in the subtree could depend on decisions made at every other subtree node.

In the sections that follow, I describe an exact algorithm that explores a limited subset of longest-prefix-match partitionings, by essentially restricting the number of holes in each bucket to a small constant. The resulting algorithm can offer certain approximation guarantees, but requires at least $\Omega(n^3)$ time. Since cubic running times are essentially prohibitive for the scale of data sets I consider, I also develop two sub-quadratic heuristics.

5.4.2.5 k-Holes Technique

One can reduce the longest-prefix-match problem's search space by limiting the number of holes per bucket to a constant k. This reduction yields a polynomial-

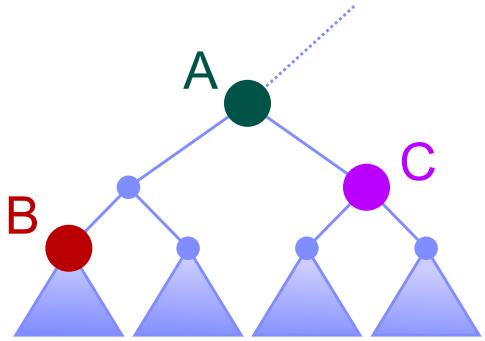

Figure 5.7: Illustration of the interdependence that makes choosing a longest-prefix-match partitioning function difficult. The benefit of making node B a bucket node depends on whether node A is a bucket node — and also on whether node C is a bucket node.

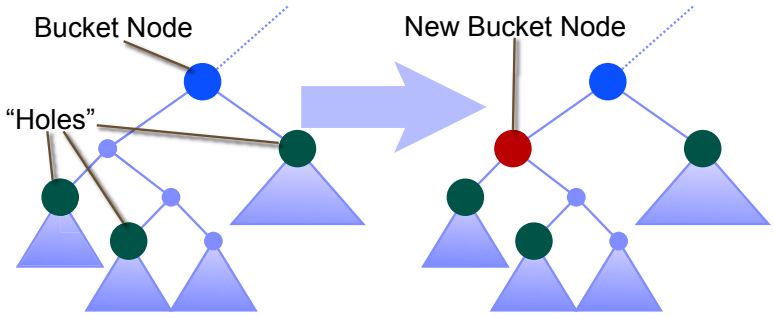

Figure 5.8: Illustration of the process of splitting a partition with n "holes" into smaller partitions, each of which has at most k holes, where $k < n$. In this example, a partition with 3 holes is converted into two partitions, each with two holes.

time algorithm for finding longest-prefix-match partitioning functions.

Observe that, if $k \geq 2$, one can convert any longest-prefix-match partition with m holes into the union of several k-hole partitions. Figure 5.9 illustrates how this conversion process works for an example configuration. In the example, adding a bucket node converts a partition with 3 holes into two partitions, each with 2 holes. Given any set of b bucket nodes, one can apply this process recursively to all the original partitions to produce a new set of partitions, each of which has at most k holes. In general, this conversion adds at most $\lfloor \frac{b}{k-1} \rfloor$ additional bucket nodes to the original solution.

Consider what happens if one applies this conversion to the optimal set of b

bucket nodes. If the error metric satisfies the "super-additivity" property [70]:

$$\text{Error}(P_1) + \text{Error}(P_2) \leq \text{Error}(P_1 \cup P_2) \tag{5.13}$$

for any partitions P_1 and P_2, the conversion will not increase the overall error. (Note that several common error metrics, e.g., RMS error, are indeed super-additive [70].) So, if the optimal b-partition solution has error E, there must exist a k-hole solution with at most $b(1 + \lfloor \frac{b}{k-1} \rfloor)$ partitions and an error of at most E.

I now give a polynomial-time dynamic programming algorithm that finds the best longest-prefix-match partitioning function with k holes in each bucket. The dynamic programming table for this algorithm is in the form:

$$E\left[i, B, j, H\right]$$

where i is the current hierarchy node, B is the number of partitions at or below node i, j is the closest ancestor bucket node, and $H = \{h_1, \ldots, h_l\}, l \leq k$ are the holes in the node j's partition.

To simplify the notation and avoid repeated computation, I use a second table $F\left[i, B\right]$ to tabulate the best error for the subtree rooted at i, *given that node i is a bucket node*.

To handle base cases, I extend grperr with an a third parameter. grperr(i, j, H) computes the error for the zero-bucket solution to the subtree rooted at i, given that node j is a bucket node with the holes in H.

The recurrence for the k-holes case is similar to that of my overlapping-partitions algorithm, with the addition of the second table F. I define this recurrence in Figure 5.9. Intuitively, the first two cases of the recurrence for E are base cases, and the remaining ones are recursive cases. The first base case prunes solutions that consider impossible sets of holes. The second base case computes the error when there are no bucket nodes (and, by extension, no elements of H) below node i.

The first recursive case looks at all the ways that the bucket budget B could be divided among the left and right subtrees of node i, given that node i is *not* a bucket node. The second recursive case finds the best solution for i's subtree in which node i is a bucket node with $B - 1$ bucket nodes below it. Keeping the table F avoids needing to recompute the second recursive case of E for every combination of j and H.

The table E has $O(b|G|^{k+1} \log |U|)$ entries, and each entry takes $O(b)$ time to compute. Table F has $O(b|G|)$ entries, and each entry takes $O(b|G|^k)$ time to compute. The overall running time of the algorithm is $O(b^2|G|^{k+1} \log |G|)$.

Although the above algorithm runs in polynomial time for a given value of

$$E[i, B, j, H] =$$

$$\begin{cases} \infty & \text{if } |H| > k \\ & \quad \text{or } |H \cap \text{subtree}(i)| > B \\ & \quad \text{or } \exists h_1, h_2 \in H . h_1 \in \text{subtree}(h_2), \\ \text{grperr}(i, j, H) & \text{if } B = 0, \\ \min \begin{cases} \min_{0 \le c \le B} \left(E[2i, c, j, H] \oplus E[2i+1, B-c, j, H] \right) \\ \qquad (i \text{ is not a bucket node}) \\ F[i, B] \text{ (only if } i \in H) \\ \qquad (i \text{ is a bucket node}) \end{cases} & \text{if } B \ge 1 \end{cases}$$

$$F[i, B] = \min_{\substack{H \subseteq \text{subtree}(i) \\ 0 \le c \le B-1}} E[2i, c, i, H] + E[2i+1, B-c-1, i, H]$$

Figure 5.9: The recurrence for the k-holes algorithm.

k, its running time (for $k \geq 2$) is at least cubic in the number of groups, making it impractical for monitoring applications with thousands of groups. In the sections that follow, we describe two heuristics for finding good longest-prefix-match partitioning functions in sub-quadratic time.

5.4.2.6 Greedy Heuristic

As noted earlier, choosing a longest-prefix-match partitioning function is hard because the choice must be made for every node at once. One way around this problem is to choose each bucket node independently of the effects of other bucket nodes. Intuitively, making a node into a bucket node creates a hole in the partition of the closest bucket node above it in the hierarchy. The best such holes tend to contain groups whose counts are very different from the counts of the rest of the groups in the parent bucket. So, if a node makes a good hole for a partition, it is likely to still be a good hole after the contents of other good holes have been removed from the partition.

My overlapping partitioning functions are defined such that adding a hole to a partition has no effect on error for groups outside the hole. Consider the example in Figure 5.7. For an overlapping partitioning function, the error for B's subtree only depends on what is the closest ancestor bucket node; making C a bucket node does not change the contents of A's overlapping partition. In other words, overlapping partitioning functions explicitly codify the independence assumption

in the previous paragraph. Assuming that this independence assumption holds, the overlapping partitioning function algorithm in Section 5.4.2.3 will find bucket nodes that are also good longest-prefix-match bucket nodes. Thus, my greedy algorithm simply runs the overlapping algorithm and then selects the best b buckets (in terms of bucket approximation error) from the overlapping solution. As my experiments demonstrate, this turns out to be an effective heuristic for longest-prefix-match partitionings.

5.4.2.7 Quantized Heuristic

My second heuristic for the longest-prefix-match case is a quantized version of a pseudopolynomial algorithm. In this section, I start by describing a pseudopolynomial dynamic programming algorithm for finding longest-prefix-match partitioning functions. Then, I explain how to quantize the table entries in the algorithm to make it run in polynomial time.

My pseudopolynomial algorithm uses a dynamic programming table $E[i, B, g, t, d]$ where:

- i is the current node of the UID hierarchy;
- B is the current bucket node budget;
- g is the number of group nodes in the subtree rooted at node i;
- t is the number of tuples whose UIDs are in the subtree rooted at node i; and,
- d, the *bucket density*, is the ratio of tuples to groups in the smallest selected ancestor bucket containing node i.

The algorithm also requires a version of grperr that takes a subtree of groups and a bucket density as arguments. This aggregate uses the density to estimate the count of each group, then compares each of these estimated counts against the group's actual count.

One can compute E by using the recurrence in Figure 5.10. Intuitively, the density of the enclosing partition determines the benefit of making node i into a bucket node. My recurrence chooses the best solution for each possible density value. In this way, the recurrence accounts for every possible configuration of bucket nodes in the rest of the hierarchy. The algorithm is polynomial with regard to the number of values taken by each of the table indices i, B, g, t, and d.

More precisely, the recurrence will find the optimal partitioning if the values of g and t range from 0 to the total number of groups and tuples, respectively; with d taking on every possible value of $\frac{t}{g}$. The number of entries in the table will be $O(|G|^3 T^2 b)$, where T is the number of possible values for t. All the base

$$E[i,B,g,t,d] =$$

$$\begin{cases} \text{grperr}(i,d) & \text{if } B = 0 \\ & \quad \text{and } g = \text{number of group nodes below } i \\ & \quad \text{and } t = \text{number of tuples below } i \\ \infty & \text{if } B = 0 \\ & \quad \text{and } (t \neq \text{number of tuples below } i \\ & \qquad \text{or } g \neq \text{number of group nodes below } i) \\ \min_{b,g',t'} & \begin{cases} \begin{cases} E[2i,b,g',t',d] \\ +E[2i+1,B-b,g-g',t-t',d] \end{cases} \\ \qquad \text{(Node } i \text{ is not a bucket node)} \end{cases} \\ & \begin{cases} \begin{cases} E[2i,b,g',t',d] \\ +E[2i+1,B-b-1,g-g',t-t',d] \\ \text{if } d = \frac{t}{g} \end{cases} \\ \qquad \text{(Node } i \text{ is a bucket node)} \end{cases} \\ & \text{if } B \geq 1 \end{cases}$$

Figure 5.10: The recurrence for a pseudopolynomial algorithm for finding longest-prefix-match partitioning.

cases can be computed in $O(|G|^2T)$ amortized time, but the recursive cases each take $O(|G|Tb)$ time. So, the overall running time of this algorithm is $O(|G|^4T^3b^2)$. However, each combination of bucket nodes could potentially lead to distinct values of t T (the number of values of t) is therefore $O(|G|^b)$. Alternately, T is also bounded by the total number of tuples in all the groups. Both of these bounds on T make the pseudopolynomial algorithm intractable in practice; real-world lookup tables can have thousands or millions of groups, and network links can receive thousands of packets per second.

One can approximate the above algorithm by considering only *quantized* values of the counters g, t and d. That is, round the values of each counter to the closest of a set of k exponentially-distributed values $(1+\Theta)^i$. (Of course, k is logarithmic in T, the number of values of t.) Recall that each entry of the dynamic programming table is in the form $E[i,B,g,t,d]$. Since the quantized algorithm chooses from among k values for each of g,t, and d, the algorithm generates $O(k^3b)$ table entries for each node of the hierarchy. For each table entry, the quantized algorithm considers each of k values for g' and t' and therefore does $O(k^2b)$ work. The overall running time for the quantized algorithm is $O(k^5|G|b^2)$.

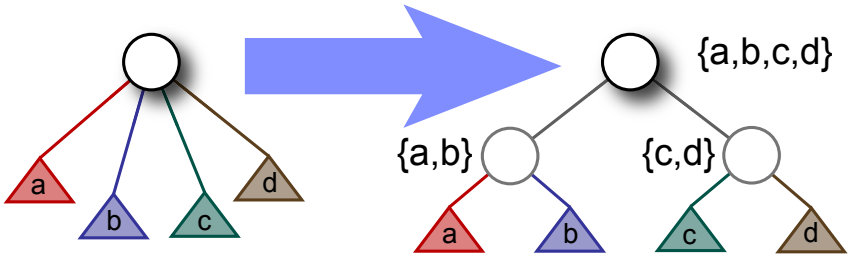

Figure 5.11: Diagram of the technique to extend my algorithms to arbitrary hierarchies by converting them to binary hierarchies. Each node of the binary hierarchy is labeled with its children from the old hierarchy.

5.5 Refinements

Having defined my core algorithms for finding my three classes of partitioning functions, I now present useful refinements to my techniques. The first of these refinements extends my algorithms to hierarchies with arbitrary fanout. The second of these refinements focuses on choosing partitioning functions for approximating *multidimensional* GROUP BY queries. The third makes my algorithms efficient when most groups have a count of zero. My final refinement greatly reduces the space requirements of my algorithms. All of these techniques apply to all of the algorithms presented thus far.

5.5.1 Extension to Arbitrary Hierarchies

Extending my algorithms to arbitrary hierarchies is straightforward. Conceptually, one can convert any hierarchy to a binary tree, using the technique illustrated in Figure 5.11. As the diagram shows, the conversion labels each node in the binary hierarchy with the set of child nodes from the original hierarchy that are below it. One can then rewrite the dynamic programming formulations in terms of these lists of nodes. For nonoverlapping buckets, the recurrence becomes:

$$E[\{i\},B] = E[\{j_1,\ldots,j_n\},B] \text{ if } j_1,\ldots,j_n \text{ were } i\text{'s children}$$

$$E[\{j_1,\ldots,j_n\},B] = \begin{cases} \mathsf{grperr}\left(\{j_1,\ldots,j_n\}\right) \text{ if } B=1, \\ \min_{1\le c\le B}\left(\begin{array}{c} E[\{j_1,\ldots,j_{n/2}\},c] \\ \oplus\, E[\{j_{n/2+1},\ldots,j_n\},B-c] \end{array}\right) \\ \text{otherwise} \end{cases}$$

A similar transformation converts $\mathsf{grperr}(i)$ to $\mathsf{grperr}\left(\{j_1,\ldots,j_n\}\right)$. The same transformation also applies to the dynamic programming tables for the other algo-

rithms.

The number of interior nodes in the graph is still $O(|G|)$ after the transformation, so the transformation does not increase the order-of-magnitude running time of the nonoverlapping buckets algorithm. For the overlapping and longest-prefix-match algorithms, the longest path from the root to the leaves increases by a multiplicative factor of $O(\log(\text{fanout}))$, increasing "big-O" running times by a factor of $\log^2(\text{fanout})$.

5.5.2 Extension to Multiple Dimensions

This chapter so far has dealt with queries over single-dimensional streams of unique identifiers:

```
select  G.GroupId, AGG(...)
from    UIDStream U [range ... slide  ...],
        GroupTable G
where   G.uid = U.uid
group by G.GroupId;
```

Some network monitoring applications require a variant of this query that breaks down traffic according to multiple unique identifiers:

```
select  G1.gid, G2.gid, AGG(...)
from    UIDStream U [sliding window],
        GroupTable G1,
        GroupTable G2
where   G1.uid = U.uid1
        and G2.uid = U.uid2
group by G.GroupId;
```

The most common application of such multidimensional queries is breaking down traffic by source and destination subnet. Other examples include tracking the relationship between subnet and DNS name or between country and web URL. In this section, I extend my algorithms to support building histograms that approximate these multidimensional queries.

My histograms extend naturally to multiple dimensions while still finding optimal histogram partitioning functions in polynomial time for a given dimensionality. In d dimensions, I define a bucket as an d-tuple of hierarchy nodes. I assume that there is a separate hierarchy for each of the d dimensions. Each bucket covers the rectangular region of space defined by the ranges of its constituent hierarchy nodes. Figure 5.12 illustrates a single bucket of a two-dimensional histogram built using this method. I denote the rectangular bucket region for nodes i_1 through i_d as $r(i_1, \ldots, i_d)$.

The extension of the non-overlapping buckets algorithm to d dimensions uses a dynamic programming table with entries in the form $E[(i_1, \ldots, i_d), B]$, where i_1 through i_d are nodes of the d UID hierarchies. Each entry holds the best possible error for $r(i_1, \ldots, i_d)$ using a total of B buckets. I also define a version of grperr that

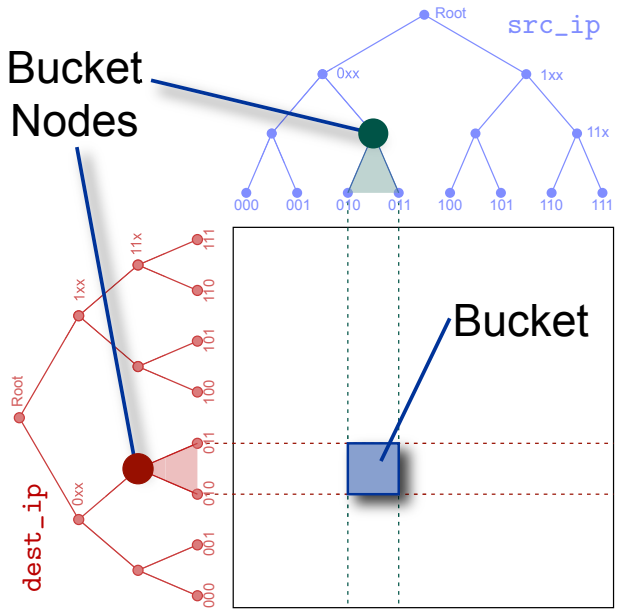

Figure 5.12: Diagram of a single bucket in a two-dimensional hierarchical histogram. The bucket occupies the rectangular region at the intersection of the ranges of its bucket nodes.

aggregates over the region $r(i_1, \cdots, i_d)$: grperr $(i_1, \ldots, i_d)$

$E[(i_1, \ldots, i_d), B]$ is computed based on the entries for all subregions of $r(i_1, \ldots, i_d)$, in all combinations that add up to B buckets. For a two-dimensional binary hierarchy, the dynamic programming recurrence is shown below. Intuitively, the algorithm considers each way to split the region (i, j) in half along one dimension. For each split dimension, the algorithm considers every possible allocation of the B bucket nodes between the two halves of the region.

$$E[(i_1, i_2), B] = \begin{cases} \text{grperr}\,(i_1, i_2) & \text{if } B = 1, \\ min \begin{cases} min_{1 \le c \le B} E[(i_1, 2i_2), c] \oplus E[(i_1, 2i_2 + 1), B - c] \\ min_{1 \le c \le B} E[(2i_1, i_2), c] \oplus E[(2i_1 + 1, i_2), B - c] \end{cases} \\ \text{otherwise} \end{cases}$$

The extension of the overlapping buckets algorithm to multiple dimensions is similar to the extension of the nonoverlapping algorithm. I make explicit the constraint, implicit in the one-dimensional case, that every bucket region in a given solution be strictly contained inside its parent region, with no partial overlap. For

$$
E[(i_1,i_2),B,(j_1,j_2)] =
\begin{cases}
\text{grperr}\,((i_1,i_2),(j_1,j_2)) & \text{if } B=0,\\
\min\begin{cases}
\min_{0\le c\le B-1}\left(E[(2i_1,i_2),c,(i_1,i_2)]\oplus E[(2i_1+1,i_2),B-c-1,(i_1,i_2)]\right)\\
\min_{0\le c\le B-1}\left(E[(i_1,2i_2),c,(i_1,i_2)]\oplus E[(i_1,2i_2+1),B-c-1,(i_1,i_2)]\right)\\
\qquad((i_1,i_2)\text{ is a bucket region})\\
\min_{0\le c\le B}\left(E[(2i_1,i_2),c,(j_1,j_2)]\oplus E[(2i_1+1,i_2),B-c,(j_1,j_2)]\right)\\
\min_{0\le c\le B}\left(E[(i_1,2i_2),c,(j_1,j_2)]\oplus E[(i_1,2i_2+1),B-c,(j_1,j_2)]\right)\\
\qquad((i_1,i_2)\text{ is not a bucket region})
\end{cases}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{otherwise}
\end{cases}
$$

Figure 5.13: Recurrence for finding overlapping partitioning functions in two dimensions.

the two-dimensional case, the recurrence is given in Figure 5.13. My algorithms for finding longest-prefix-match buckets can be extended to multiple dimensions by applying the same transformation.

Computing a V-Optimal histogram partitioning has been shown to be NP-hard [71]. In contrast, the multidimensional extensions of my algorithms run in polynomial time for a given dimensionality. The running time of the extended nonoverlapping algorithm is $O(|G|^d db^2)$ for RMS error; and the running time of the extended overlapping buckets algorithm is $O(db^2|G|^d \log^d |U|)$, where d is the number of dimensions. Similarly, the multidimensional version of my quantized heuristic runs in $O(db^2|G|^d)$ time.

5.5.3 Sparse Group Counts

For the network monitoring queries that I target in this chapter, it is often the case that the counts of most groups are zero. There is a very large universe of IP addresses, and most points on the Internet can theoretically reach a significant partition of the IP address space. At a given point in time, however, a typical network link transmits traffic to and from only small portions of this space.

With some straightforward optimizations, my algorithms can take advantage of cases when the group counts are sparse (e.g. mostly zero). These optimizations make the running time of my algorithms depend only on the height of the hierarchy and the number of nonzero groups.

To improve the performance of the nonoverlapping buckets algorithm in Section 5.4.2.2, I observe that the error for a subtree whose groups have zero count will always be zero. This observation means that the algorithm can ignore any subtree

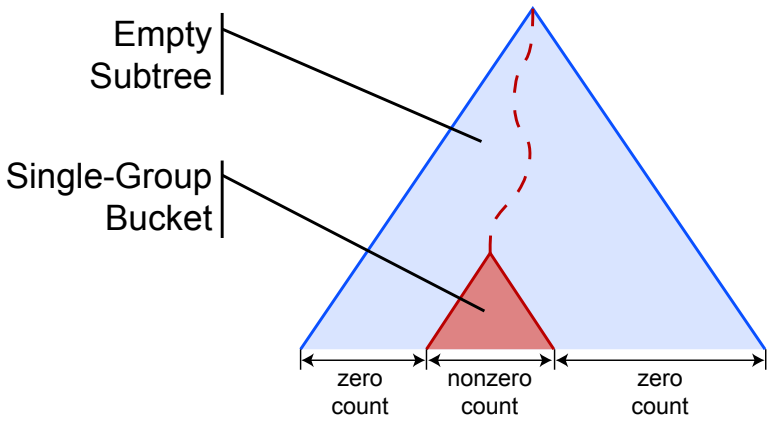

Figure 5.14: One of the *sparse buckets* that allow my overlapping histograms to represent sparse group counts efficiently. Such a bucket produces zero error and can be represented in $O(\log \log |U|)$ more bits than a conventional bucket.

whose leaf nodes all have a count of zero. Furthermore, the system does not need to store any information about buckets with counts of zero, as these buckets can be easily inferred from the non-empty buckets on the fly.

For overlapping and longest-prefix-match buckets, I introduce a new class of bucket, the *sparse bucket*. A sparse bucket consists of a single-group *sub-bucket* and an empty subtree that contains it, as shown in Figure 5.14. As a result, the approximation error within a sparse bucket is always zero. Since the empty subtree has zero count and can be encoded as a distance up the tree from the sub-bucket, a sparse bucket takes up only $O(\log \log |U|)$ more space than a single normal bucket.

A sparse bucket dominates any other solution that places bucket nodes in its subtree. As a result, my overlapping buckets algorithm does not need to consider any such solutions when it can create a sparse bucket. Dynamic programming can start at the upper node of each candidate sparse bucket. Since there is one candidate sparse bucket for each nonzero group, the algorithm runs in $O(gb^2 \log |U|)$ time, where g is the number of nonzero groups.

For my target monitoring applications, it is important to note that the time required to produce an approximate query answer from one of my histograms is proportional to the number of groups the histogram predicts will have nonzero count. Because of this relationship, the end-to-end running time of the system can be sensitive to how aggressively the histogram marks empty ranges of the UID space as empty. Error metrics that penalize giving a zero-count group a nonzero count will make the approximate group-by query run much more quickly.

5.5.4 Space Requirements

A naive implementation of my algorithms would require large in-memory tables. However, a simple technique developed by Guha [43] reduces the memory overhead of the algorithms to very manageable sizes. The basic strategy is to compute only the error and number of buckets on the left and right children at the root of the tree. Once entry $E[i, \cdots]$ has been used to compute all the entries for node $\lfloor \frac{i}{2} \rfloor$, the entry can be garbage-collected.

To reconstruct the entire bucket set, the technique uses a divide-and-conquer strategy: First, run the algorithm over the entire hierarchy to determine how many bucket nodes are in the root's left and right subtrees, then recurse on those subtrees. This technique divides the original problem into two subproblems, each of which is half the size of the original problem. To further reduce memory requirements, the nodes themselves could be stored on disk in this order and read into memory as needed. In the worst case, it takes $O(\log |U|)$ passes (one pass for each level of the hierarchy) of divide-and-conquer to reconstruct the entire solution.

When Guha's technique is applied to my algorithm for finding nonoverlapping partitioning functions for RMS error, each round of subproblems takes a total of $O(|G|(b^2 + \log |U|))$ time. So the overall running time of the nonoverlapping algorithm increases to $O(|G| \log |U|(b^2 + \log |U|))$. Likewise, the running time of my algorithm for finding overlapping partitioning functions also increases by a factor of $\log |U|$ to $O(|G|b^2 \log^2 |U|)$ time. As Guha notes, algorithms that require $O(|G|^2)$ or more time will see no increase in running time when his technique is applied. My algorithms for finding longest-prefix-match partitioning functions, as well as my multidimensional algorithms, all fall into this category.

In my actual implementation, I use a less aggressive form of pruning that avoid the $O(\log |U|)$ increase in running time of Guha's technique. My implementation prunes all elements of the table *except* those that are part of the optimal solution to a subproblem at the current level of the hierarchy under consideration. When dynamic programming reaches the root of the hierarchy, the optimal set of bucket nodes can be reconstructed as if the entire table were in memory. Theoretically, this technique may not prune any table entries at all. In my experiments, however, the technique results in a space reduction comparable to that of Guha's technique, without requiring multiple passes to recover the bucket nodes. Intuitively, the partial solutions for a given subtree tend to overlap; the best b-bucket solution for a subtree will share most of its bucket nodes (and hence most of its optimal sub-problems) with the best $b - 1$-bucket solution, and so on.

The number of table entries that must be kept in memory at a given time is also a function of the order in which the algorithm processes the nodes of the UID hierarchy. The maximum working set size of my algorithms can be minimized by

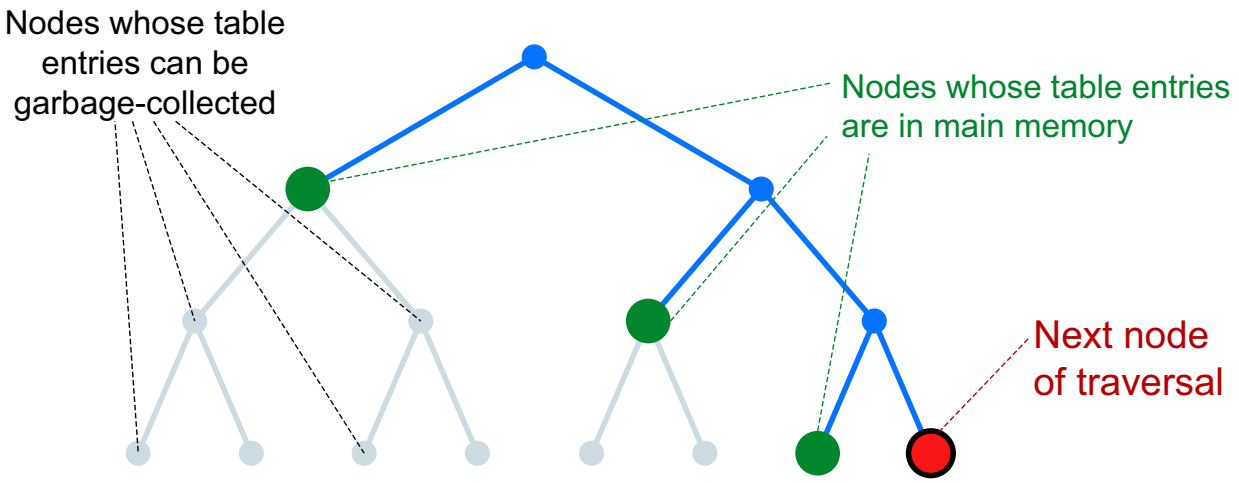

Figure 5.15: Illustration of garbage-collecting unneeded table entries during a preorder traversal of the hierarchy. At most, the algorithms in this chapter need to keep entries for one node from each level of the hierarchy at a given time.

processing nodes in the order of a preorder traversal, keeping the memory footprint to a minimum. At a given point in the preorder traversal, a pass of dynamic programming will be considering table entries associated with the nodes along a cut in the hierarchy that contains at most one node from each level, as illustrated in Figure 5.15. There are $O(\log |U|)$ such nodes at any time (assuming a balanced UID hierarchy), and the only table entries that need to be in memory are the ones for the nodes along the cut. Since my nonoverlapping algorithm has b table entries for each node, the algorithm needs to keep only $O(b \log |U|)$ table entries in memory at a time. Similarly, my overlapping partitions algorithm requires $O(b \log^2 |U|)$ space. My quantized heuristic requires $O(k^3 b \log^2 |U|)$ space, where k is the number of quanta for each counter.

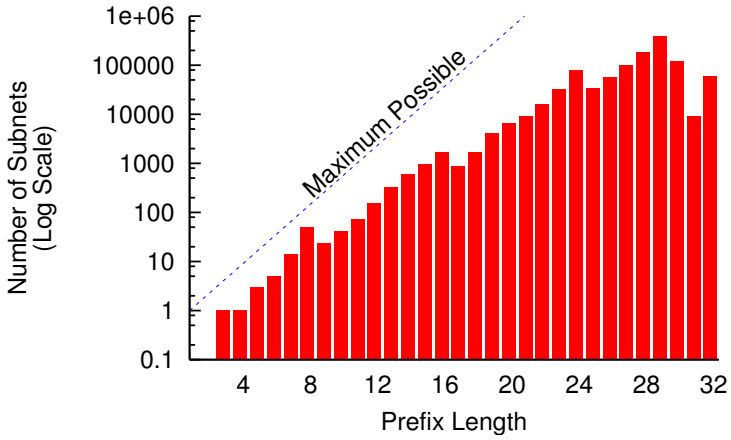

Figure 5.16: The distribution of IP prefix lengths in my experimental set of subnets. The dotted line indicates the number of possible IP prefixes of a given length (2^{length}). Jumps at 8, 16, and 24 bits are artifacts of an older system of subnets that used only three prefix lengths.

5.6 Experimental Evaluation

To measure the effectiveness of my techniques, I conducted a series of evaluations on real network monitoring data and metadata.

The WHOIS databases store ownership information on publicly accessible subnets of the Internet. Each database serves a different set of addresses, though WHOIS providers often mirror each others' entries. I downloaded publicly-available dumps of the RIPE and APNIC WHOIS databases [90, 5] and merged them, removing duplicate entries. I then used this table of subnets to generate a table of 1.1 million nonoverlapping IP address prefixes that completely cover the IP address space. Each prefix corresponds to a different subnet. The prefixes ranged in length from 3 bits (536 million addresses) to 32 bits (1 address), with the larger address ranges denoting unused portions of the IP address space. Figure 5.16 shows the distribution of prefix lengths.

I obtained a large trace of "dark address" traffic on a slice of the global Internet. The destinations of packets in this trace are IP addresses that are not assigned to any active subnet. The trace contains 7 million packets from 187866 unique source addresses. Figure 5.17 gives a breakdown of this traffic according to the subnets in my subnet table.

I chose a query that counts the number of packets in each subnet:

```
select S.id, count(*)
```

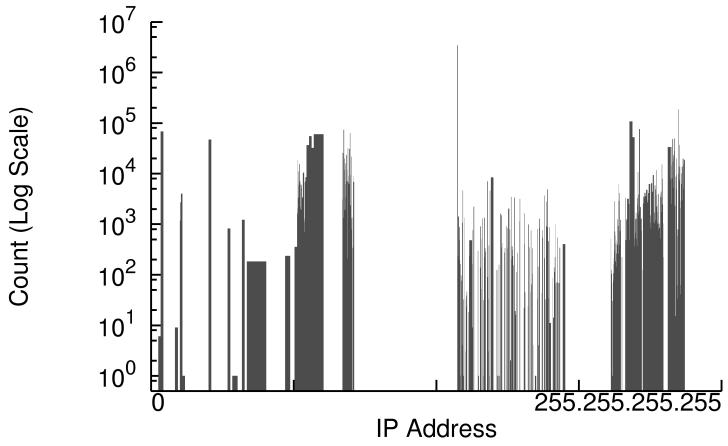

Figure 5.17: The distribution of network traffic in my trace by source subnet. Due to quantization effects, most ranges appear wider than they actually are. Note the logarithmic scale on the Y axis.

```
from
    Packet P,
    Subnet S
where
    -- Adjacent table entries with the same subnet
    -- are merged into a single table entry
    P.src_ip >= I.id and P.src_ip <= I.id
group by S.id
```

I used six kinds of histogram to approximate the results of this query:

- Hierarchical histograms with nonoverlapping buckets
- Hierarchical histograms with overlapping buckets
- Hierarchical histograms with longest-prefix-match buckets, generated with the greedy heuristic
- Hierarchical histograms with longest-prefix-match buckets, generated with the quantized heuristic
- End-biased histograms [50]
- V-Optimal histograms [52]

An end-biased histogram consists of a set of single-group buckets for the $b-1$ groups with the highest counts and a single multi-group bucket containing the count for all remaining groups. I chose to compare against this type of histogram for several reasons. End-biased histograms are widely used in practice. Also, construction of these histograms is tractable for millions of groups, and my data set contained 1.1 million groups. Additionally, end-biased histograms model skewed distributions well, and the traffic in my data set was concentrated in a relatively small number of groups.

A V-Optimal histogram is an optimal histogram where each bucket corresponds to an arbitrary contiguous range of values. For RMS error, the V-Optimal algorithm of Jagadish *et al.*[52] can be adapted to run in $O(|G|^2)$ time, where G is the set of nonzero groups. For an arbitrary distributive error metric, the algorithm takes $O(|G|^3)$ time, making it unsuitable for the sizes of data set I considered. I therefore used RMS error to construct all the V-Optimal histograms in my study.

I studied the four different error metrics discussed in Section 5.3.2.3:

- Root Mean Square (RMS) error
- Average error
- Average relative error
- Maximum relative error

Note that these errors are computed across vectors of groups in the result of the grouped aggregation query, not across vectors of histogram buckets.

For each error metric, I constructed hierarchical histograms that minimize the error metric. I compared the error of the six histogram types, repeating the experiment at histogram sizes ranging from 10 to 20 buckets in increments of 1 and from 20 to 1000 buckets in increments of 10.

5.6.1 Experimental Results

I divide my experiment results according to the type of error metric used. For each error metric, I give a graph of query result estimation error as a function of the number of histogram buckets. The dynamic range of this error can be as much as two orders of magnitude, so the y axes of my graphs have logarithmic scales.

5.6.1.1 RMS Error

My first experiment measured RMS error. The RMS error formula emphasizes larger deviations, making it sensitive to the accuracy of the groups with the highest counts. Longest-prefix-match histograms produced with the greedy

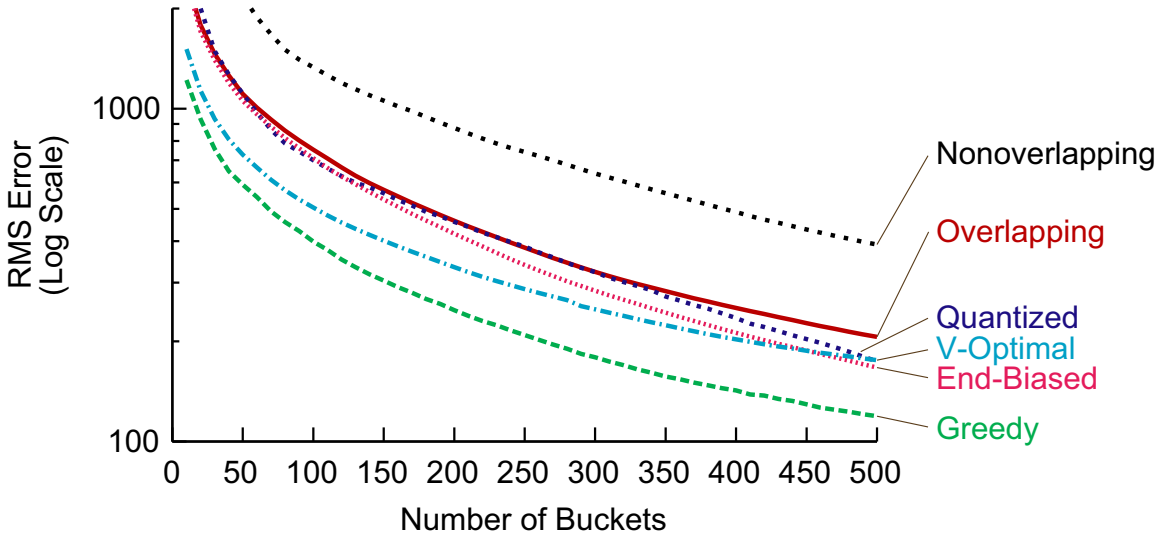

Figure 5.18: RMS error in estimating the results of my query with the different histogram types.

heuristic were the clear winner, by virtue of their ability to isolate these "outlier" groups inside nested partitions. Interestingly, the quantized heuristic fared relatively poorly in this experiment, finishing at the middle of the pack. The heuristic's logarithmically-distributed counters were unable to capture sufficiently fine-grained information to produce more accurate results than the greedy heuristic.

5.6.1.2 Average Error

My second experiment used average error as an error metric. Figure 5.19 shows the results of this experiment. As with RMS error, the greedy heuristic produced the lowest error, but the V-Optimal histograms and the quantized heuristic produced results that were almost as good. Average error puts less emphasis on groups with very high counts, and these heuristics emphasize such outliers in their partition choices. The other types of histogram produced significantly higher error. As before, I believe this performance difference is mainly due to the ability of longest-prefix-match and V-Optimal histograms to isolate outliers by putting them into separate buckets.

5.6.1.3 Average Relative Error

My third experiment compared the three histogram types using average relative error as an error metric. Compared with the previous two metrics, relative error emphasizes errors on the groups with smaller counts. Figure 5.20 shows the results of this experiment. The quantized heuristic produced the best histograms

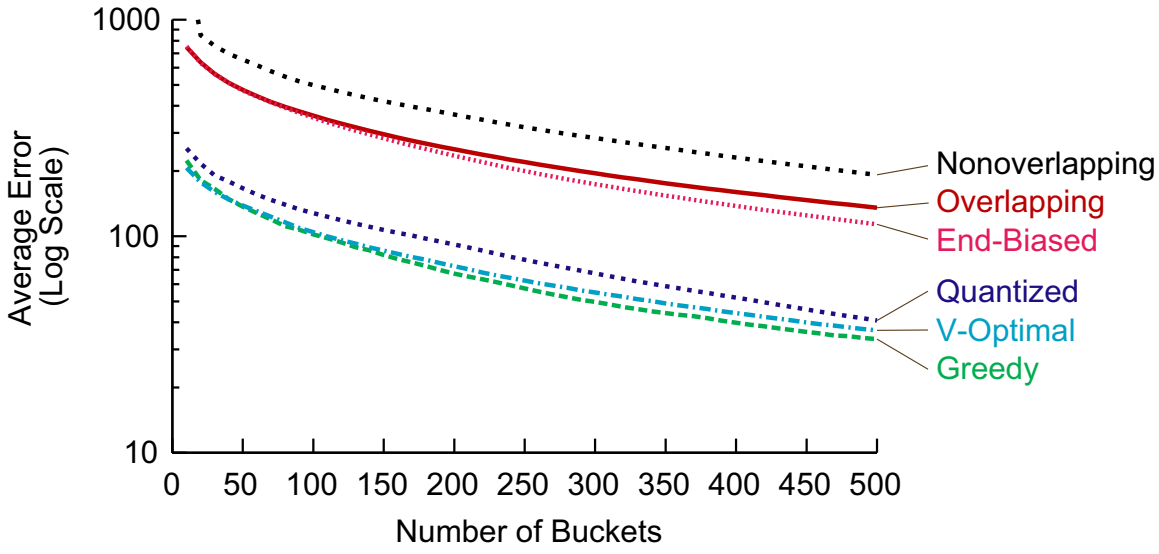

Figure 5.19: Average error in estimating the results of my query with the different histogram types.

for this error metric. The heuristic's quantized counters were better at tracking low-count groups than they were at tracking the larger groups that dominated the other experiments. V-Optimal histograms produced low error at smaller bucket counts, but fell behind as the number of buckets increased.

5.6.1.4 Maximum Relative Error

My final experiment used maximum relative error. This error metric measures the ability of a histogram to produce low error for every group at once. Results are shown in Figure 5.21. Histograms with overlapping partitioning functions produced the lowest result error for this error measure. Interestingly, the greedy heuristic was unable to find good longest-prefix-match partitioning functions for the maximum relative error measure. Intuitively, the heuristic assumes that removing a hole from a partition has no effect on the mean count of the partition. Most of the time, this assumption is true; however, when it is false, the resulting histogram can have a large error in estimating the counts of certain groups. Since the maximum relative error metric finds the maximum error over the entire set of groups, a bad choice anywhere in the UID hierarchy will corrupt the entire partitioning function.

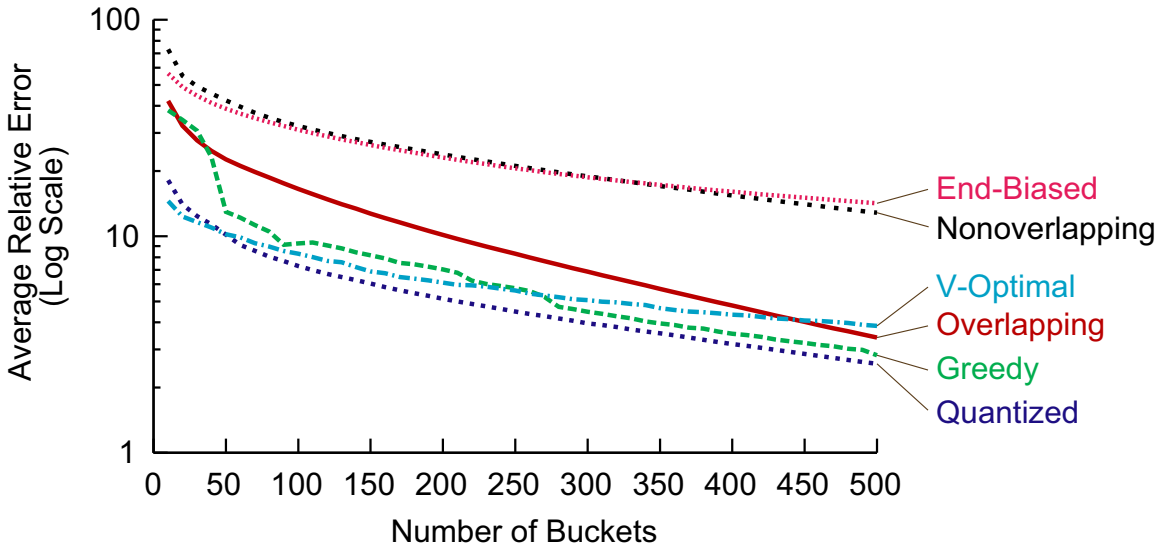

Figure 5.20: Average relative error in estimating the results of my query with the different histogram types. Longest-prefix-match histograms significantly outperformed the other two histogram types.

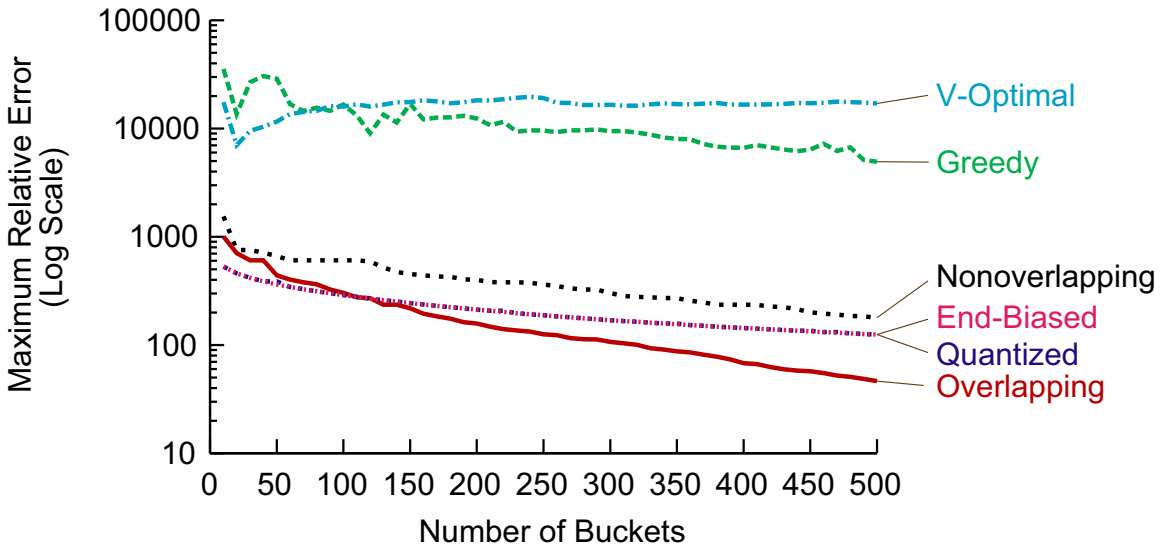

Figure 5.21: Maximum relative error in estimating the results of my query with the different histogram types.

Chapter 6

Deployment Study

Good judgment comes from experience, and experience comes from bad judgment.

— Barry LePatner

6.1 Introduction

This chapter presents a detailed performance study of the Data Triage architecture in the context of an end-to-end network monitoring system. The system in question arose from a collaboration between myself and researchers at Lawrence Berkeley National Laboratory. The high-level aim of this collaboration was to build a software platform for monitoring the network traffic of the Department of Engineering national laboratories.

From the perspective of this dissertation, the study had several goals:

- Demonstrate the feasibility of using a general-purpose streaming query processor to monitor high-speed networks
- Show that Data Triage is necessary to maintain low query result latency in such a context
- Demonstrate that my prototype implementation of Data Triage can scale to support a "real-world" workload
- Show that the extension of Data Triage for archived streams as described in Section 3.4 also works in such a setting

6.2 Background

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 change. 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.

Network availability and performance are another important priority at DOE. Large data transfers are an integral part of many important projects at the labs. Multimillion-dollar particle accelerators cannot run unless enough network bandwidth is available to absorb their data streams. DOE scientists running simulations routinely ship large data sets across the world, and increased use of grid comput-

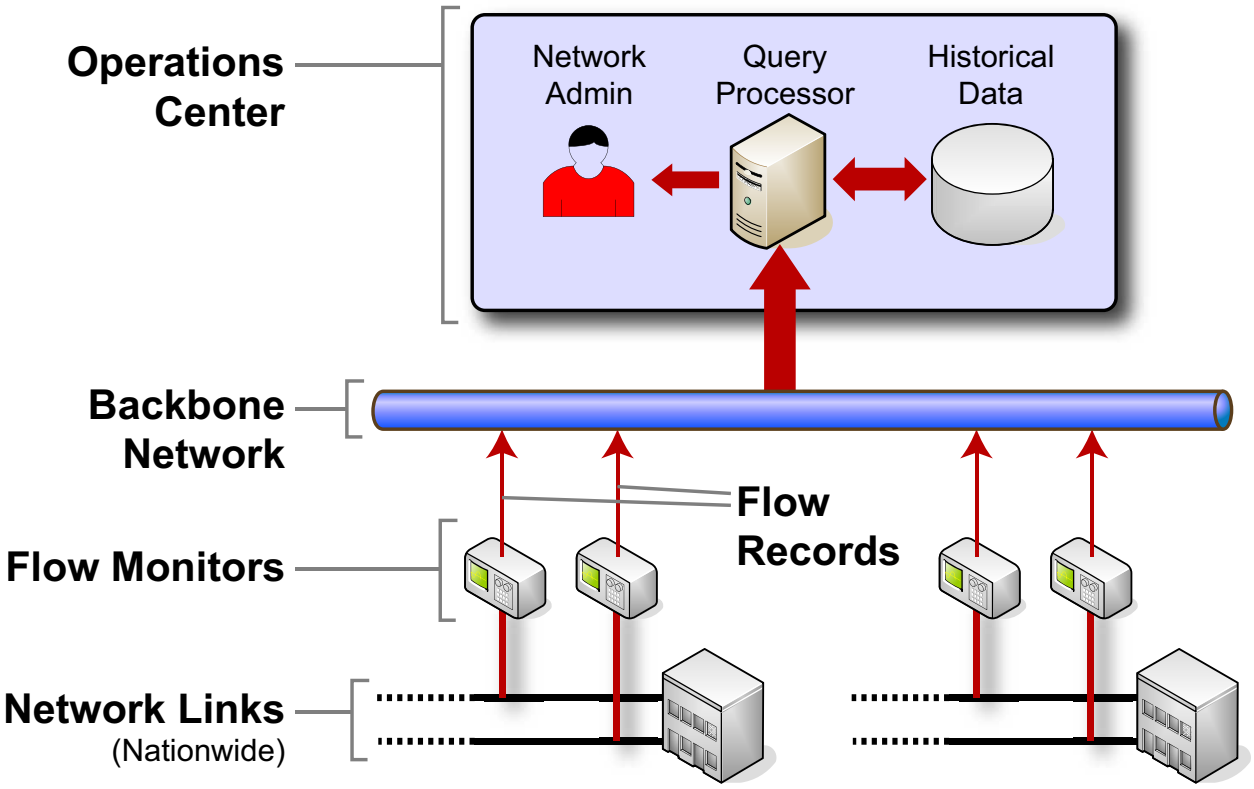

Figure 6.1: High-level block diagram of the proposed nationwide network monitoring infrastructure for the DOE labs.

ing is making network uptime even more essential.

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 6.1 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.

This chapter describes my experiences working with researchers at Lawrence Berkeley National Laboratories to build 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, pat-

terns, 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 each generate tens of 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.

My collaborators have been working on FastBit [11, 101], a high-performance bitmap index with a declarative SQL interface. Working with the FastBit team, I designed an end-to-end system that uses TelegraphCQ for streaming query processing and FastBit to provide access to historical data. This design easily meets our first two requirements, since TelegraphCQ and FastBit both use flexible declarative query languages and FastBit provides access to historical data. Our main question going forward was whether the combination of these two general-purpose systems can handle sufficiently high data rates to meet the performance requirement.

6.2.1 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 6.2 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 6.3. This figure shows the histogram of flow record rates. We observed that this distribution is heavy-tailed.

The NERSC traffic is one of three wide-area network links currently deployed at Berkeley Lab, and the lab's employees represent 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

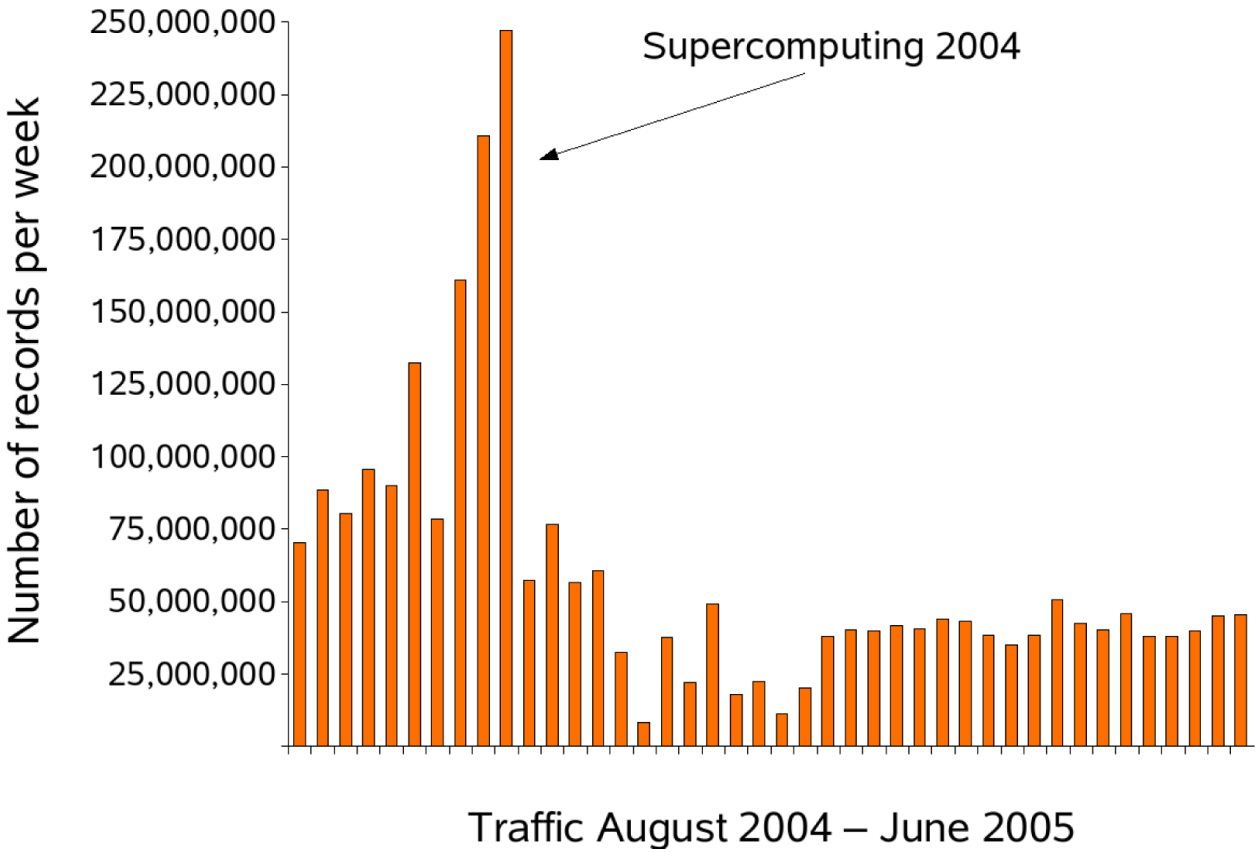

Figure 6.2: Flow records per week in our 42-week snapshot of Berkeley Lab's connection to the NERSC backbone.

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.

6.2.2 Query Complexity

Based on a literature survey and my own conversations with networking researchers, I developed a sample workload of queries for monitoring networks. I describe this workload in detail in Section 6.5. Each of my 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. A real deployment would run a mix of 10 to 100 such queries, monitoring various

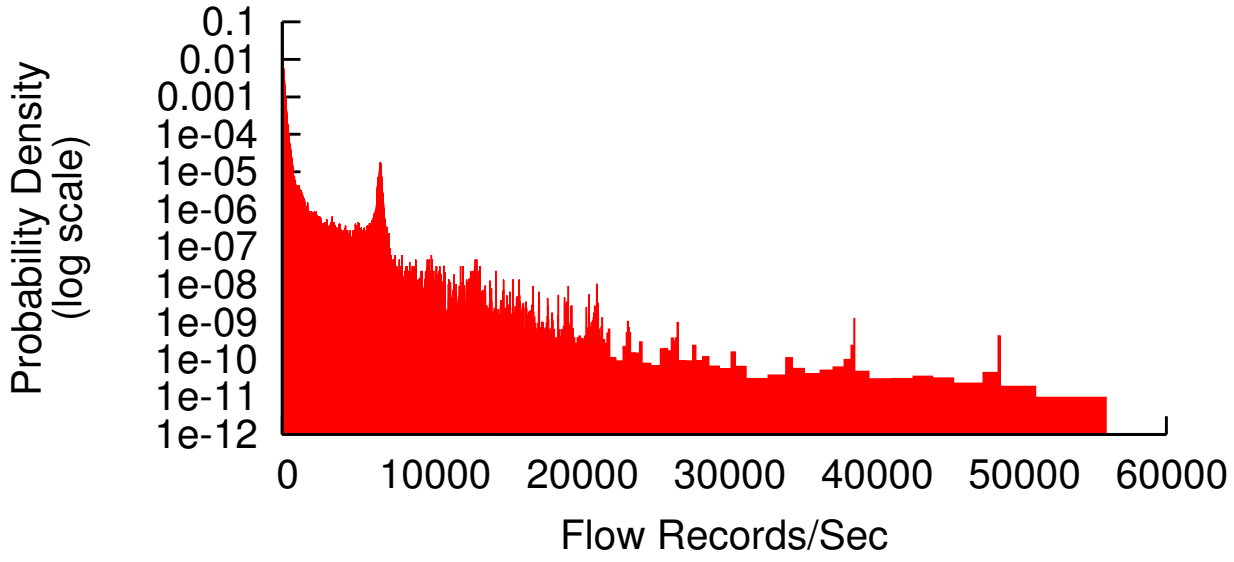

Figure 6.3: Histogram of the number of flow records per second in the data set in Figure 6.2. The distribution is heavy-tailed, with a peak observed rate of 55,000 records per second.

aspects of the network in real time.

6.2.3 Analyzing The Performance Problem

Based on my earlier experience with smaller prototypes, I hypothesized that TelegraphCQ would be the major bottleneck in our combined system and that the system would need Data Triage to provide reasonable response times in the face of bursty network traffic. I worked with the FastBit team to design a performance study that would break down the performance of the system in terms of both data throughput and query result latency. The Berkeley Lab researchers optimized the FastBit engine and tweaked the historical components of my query workload for better performance. I debugged and performance tuned TelegraphCQ, as well as adding query language support for my query workload. I also created a robust high-performance implementation of Data Triage to match the performance-tuned TelegraphCQ. Then I wrote and optimized the code that ties these three components together. I designed a set of experiments that break down system performance in terms of the individual components of the system, as well as end-to-end experiments for evaluating the entire system. The FastBit team measured the performance of the FastBit component of our system. I measured the performance of TelegraphCQ by itself, both with and without my Data Triage implementation. Finally, I ran experiments that measured the performance of the entire system. The

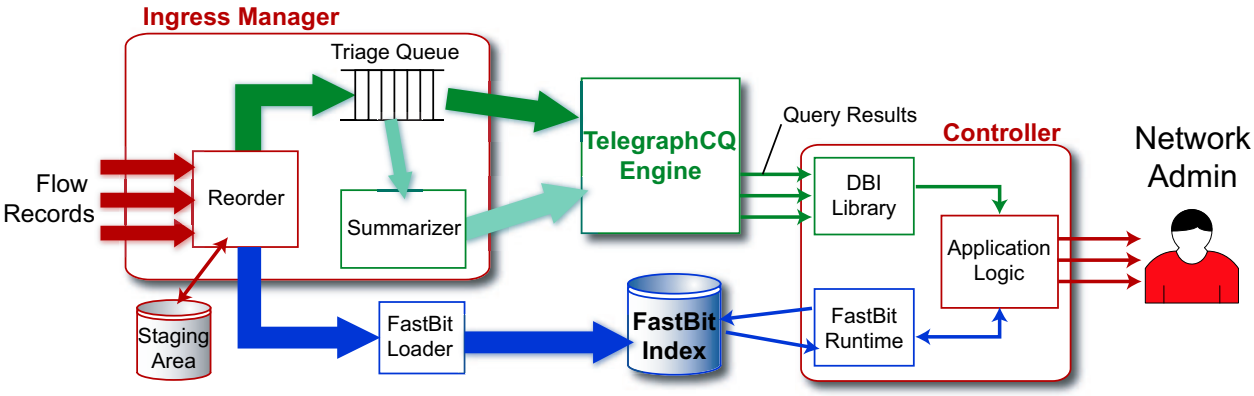

Figure 6.4: Block diagram of our network monitoring prototype. Our system combines the data retrieval capabilities of FastBit with the stream query processing of TelegraphCQ.

remainder of this chapter describes the steps of our performance study in detail, starting with my overall design and data/query workload and then describing our experimental results.

6.3 Architecture

This section gives a high-level overview of the architecture I designed for our prototype monitoring system and provides references for further reading on individual components of our system. Figure 6.4 shows how the pieces of our architecture fit together. The major 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. I give an overview of TelegraphCQ in Section 1.6 of this dissertation.

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. My design uses FastBit to provide fast associative access to archived network monitoring data. The keys to FastBit's efficiency are its vertical data organization and efficient bitmap compression algorithms [96, 114, 115].

The **Ingress Manager** component corresponds to the Triage Process in my design for Data Triage with archival data (See Section 3.4). The Ingress Manager process contains a Triage Queue, Triage Scheduler, and Summarizer, as well as code that stages data to disk for loading into FastBit. This component also merges incoming streams of flow records and converts data into formats that TelegraphCQ

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

and FastBit understand.

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. I 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.

6.4 Data

All our performance benchmarks are based on real network connection data that researchers at Berkeley Lab collected on the Lab's NERSC supercomputing backbone connection. The data represents a time period of 5 weeks and consists of 121 million TCP/IP flow 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 6.1 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.

I mapped the attributes in the flow records 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;
```

Recall from Section 1.6 that 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; my system design 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, my LBNL collaborators 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 6.2 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 6.2: Relationship between columns of our TelegraphCQ and FastBit schemas

6.5 Queries

Based on a literature search and discussions with networking researchers, I 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. These queries represent an important contribution: To my knowledge, no other collection of similarly complex live/archive queries for network monitoring has been published in the data management, streaming query processing, or network measurement literature.

In the sections that follow, I present these queries in the native query languages of TelegraphCQ and FastBit, respectively, using the schemas from Section 6.4.

Each query has one or more parameters that are bound at runtime. Following the convention of most SQL databases, I denote these variable parameters with a preceding colon. For example, in the first "elephants" query below, the parameter :windowsz is variable. At runtime, the Controller component of the system substitutes the appropriate values for for the variable parameters in the templates.

6.5.1 Elephants

Goal: This analysis finds 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. Reports significant traffic sources

[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".

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 output 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])
);
```

6.5.2 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 the query looks 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: Produces a breakdown of traffic that is not from the hosts in the FastBit query's results.

```
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 ... and src_ip != :X100
group by src_ip , dst_ip
order by sum(bytes_sent) desc
limit 100 per window;
```

6.5.3 Portscans

Goal: This analysis finds behavior that suggests port scanning activity[2]. 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. The ORDER BY and LIMIT clause at the end of the query indicates that TelegraphCQ should return the top 100 hosts according to "fanout".

```
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: Given the IP addresses returned by the TelegraphCQ query (:X1 through :X100), this query determine which hosts those addresses have historically communicated with. Application logic can use this information to differentiate between suspicious behavior and normal traffic patterns.

Note that FastBit queries with no aggregates have an implicit count(*) aggre-

[2] A *port* is a numeric identifier within a single host that is associated with a single process running on the host. The term *portscan* refers to the common practice of making blind connection attempts to a remote host to determine which network ports are associated with running programs [112].

gate.

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

6.5.4 Anomaly detection

Goal: This analysis compares 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 a change metric such as the L_1 norm or Euclidean distance. More sophisticated analyses involving multiple historical traffic matrices are also possible [99].

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
```

6.5.5 Dispersion

Goal: This analysis finds subnets at a parametrizable IP prefix length that appear in two time windows within a given period, and reports the time lag be-

tween these two windows. Certain patterns of time lag can indicate malicious behavior[22]. 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 [22].

TelegraphCQ Query: Produces a breakdown of traffic by subnet. The parameter : prefixlen determines the length of the IP address prefix that defines a subnet. WTIME(∗) is a TelegraphCQ built-in function that returns the latest timestamp in the current time window.

```
with
WindowResults1 as
(select
    network(set_masklen( src_ip , : prefixlen )) 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.

Ideally, this query would group results by network subnet as in the TelegraphCQ query above (e.g. select network(set_masklen(IPS, : prefixlen))). However, 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
```

6.6 Experiments

In this section, I evaluate the performance of our system in a series of experiments. I start by analyzing the major components of the system separately to determine the peak data throughput of each component. I then benchmark the throughput of the entire system and compare this end-to-end throughput against that of the individual components. Finally, I evaluate the performance of the system with realistic packet arrival rates derived from the timestamps in Lawrence Berkeley National Lab's 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.

6.6.1 TelegraphCQ Without Data Triage

My first experiment examined the performance tradeoffs of the TelegraphCQ component of the system without Data Triage. I focused on two parameters: query type and window size. Recall that each of the five TelegraphCQ queries in my workload produces results for discrete windows of time, and the size of these windows is given as a parameter in the query.

Bypassing the Data Triage components of the Ingress Manager, I sent week 5 of the trace directly through TelegraphCQ and measured the total running time for each of my five TelegraphCQ queries. From this running time, the window size, and the number of tuples in the trace, I computed an average throughput figure for the query processor. I 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 6.5 shows the measured throughput of TelegraphCQ in my experiments. Each query showed an initial increase in throughput as window size increased. As window size continued to increase, throughput eventually reached a peak and declined somewhat. The "dispersion" query showed a particularly pronounced instance of this pattern, with throughput increasing from 6,000 to 25,000 tuples

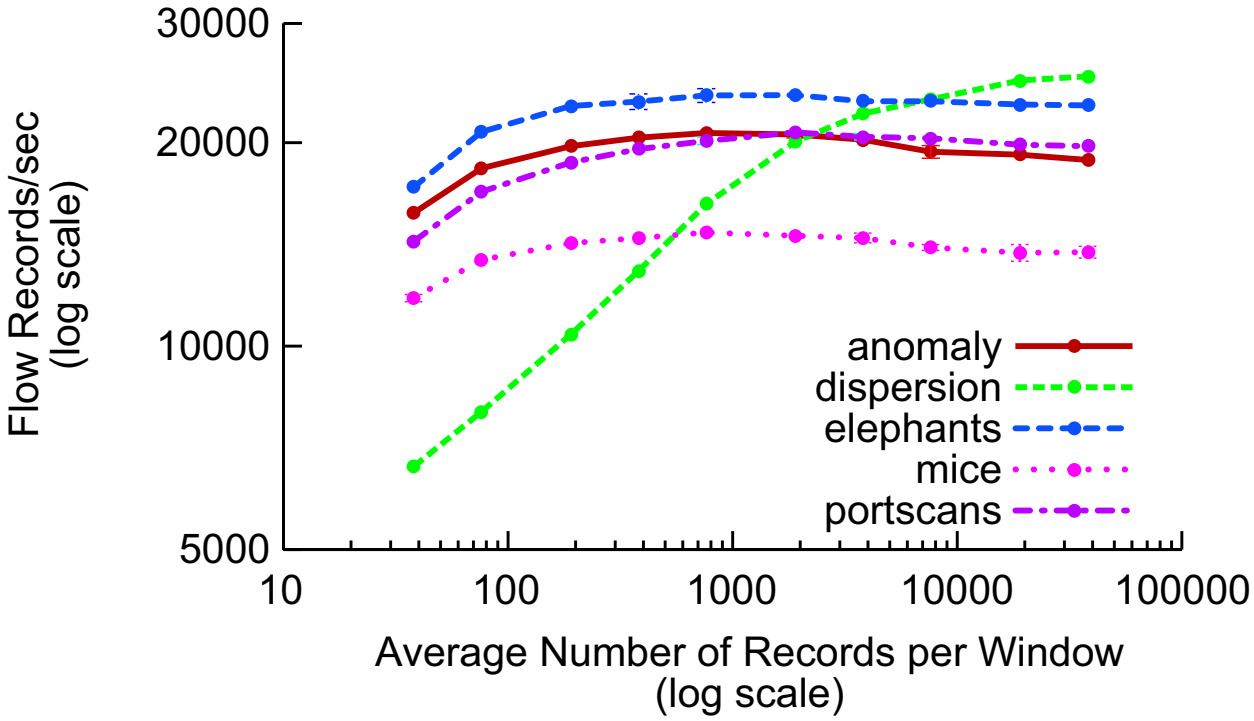

Figure 6.5: The throughput of TelegraphCQ running my queries with varying window sizes over the 5th week of the NERSC trace. Note the logarithmic scale.

across the range of window sizes targeted. The remaining queries had much flatter throughput curves.

I profiled my TelegraphCQ testbed to determine the reason for the changes in throughput I observed. I found that the relatively low throughputs at smaller window sizes were due to the large number of result tuples the queries produce at those 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. I traced the slight falloff in performance at larger window sizes to increased memory footprint, as TelegraphCQ buffers the tuples in the current window in memory.

With the appropriate choice of window size, a single instance of TelegraphCQ can handle the average aggregate flow record traffic of all the DOE labs. A small cluster of machines should be able to run a 10-20 query workload at this average data rate. It is important to note, however, that such a setup would buffer excess data and return delayed query results whenever data rates exceeded the average. The experiments in Section 6.6.7 quantify this delay and explore methods for re-

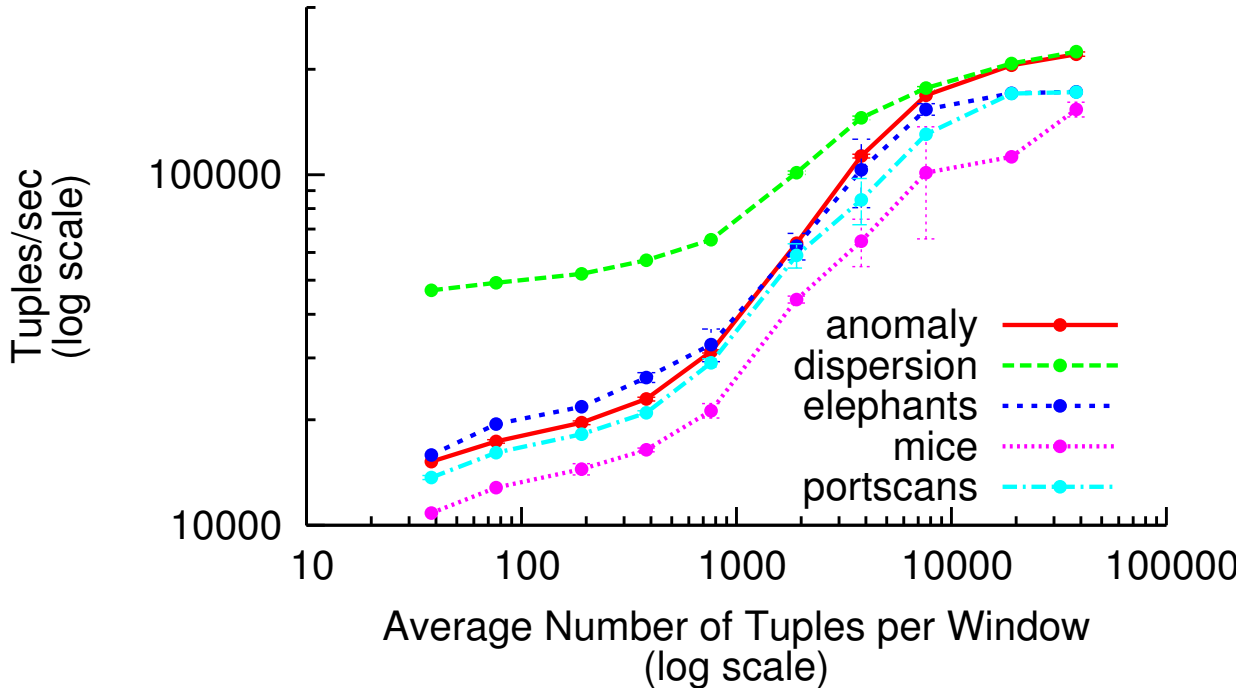

Figure 6.6: The throughput of TelegraphCQ with Data Triage, running my queries with varying window sizes over the 5th week of the NERSC trace. Note the logarithmic scale.

ducing it.

6.6.2 TelegraphCQ With Data Triage

My next experiment measured the peak throughput of the streaming component of the system when Data Triage is used. I used the same setup as the experiment in the previous section, except that the Data Triage components in the Ingress Manager were enabled. I used a 1000-tuple reservoir sample as the summary type in this experiment. To push the system to its peak data rate, I replaced the normal Triage Scheduler with a scheduler that triages every input tuple. For each run of the experiment, I measured both peak throughput and result error. To measure result error, I logged the results of the main and shadow queries to disk during each run of the experiment. After each run of the experiment, a postprocessing step measured query result error by comparing the results from the Data Triage runs against the exact answer computed in a separate run with Data Triage disabled.

My postprocessing script used *weighted percent error* to measure result error. Intuitively, this error metric measures percent error across the groups, weighted by the actual aggregate values. More formally, weighed percent error is defined as

follows:

$$\text{Error} = \frac{\sum_{g \in G} |g.\text{actual} - g.\text{approx}|}{\sum_{g \in G} \text{actual}} \tag{6.1}$$

where G is the set of groups. The error was averaged over all the time windows in the trace.

Figure 6.6 shows the throughput results of this experiment. In general, throughput at small average window sizes was comparable to that of TelegraphCQ without Data Triage. The 1000-tuple reservoir kept most or all of the data in most window, so the system was doing about as much work with Data Triage as without. As the number of tuples per window increased, the peak throughput supported by my Data Triage implementation increased far beyond that which TelegraphCQ can handle by itself. Data rates peaked at between 155000 and 225000 flow records per second, depending on query.

The system's relative throughput performance on the different queries was similar to that of the experiment in Section 6.6.1, with the exception of the "dispersion" query. This query did not exhibit the same slowdown as before at smaller window sizes. This difference was due to the query's number of output tuples being $O(n^2)$ in the number of flow records per window. Execution time in the previous experiment was dominated by time windows with large amounts of traffic. With almost every tuple going through a reservoir sample, these high-traffic windows were "clipped" in the Data Triage experiment.

Figure 6.7 shows the measured result error of the queries. As window size increased, the reservoir sample discarded larger portions of the data in each window, resulting in higher error. The amount of error induced varied significantly across queries. In particular, the "elephants," "mice," and "anomaly" queries produced higher error than the "dispersion" and "portscan" queries.

The "dispersion" and "portscan" queries performed well largely because they count the number of *distinct* values that meet a certain constraint. For example, the first stage of the dispersion query finds all the distinct network address prefixes that occur in each time window:

```
with
WindowResults1 as
(select
    network(set_masklen( src_ip ,  : prefixlen )) as  prefix ,
    wtime(*) as tcqtime
from Flow [range by :windowsz slide by :windowsz]
group by prefix)
```

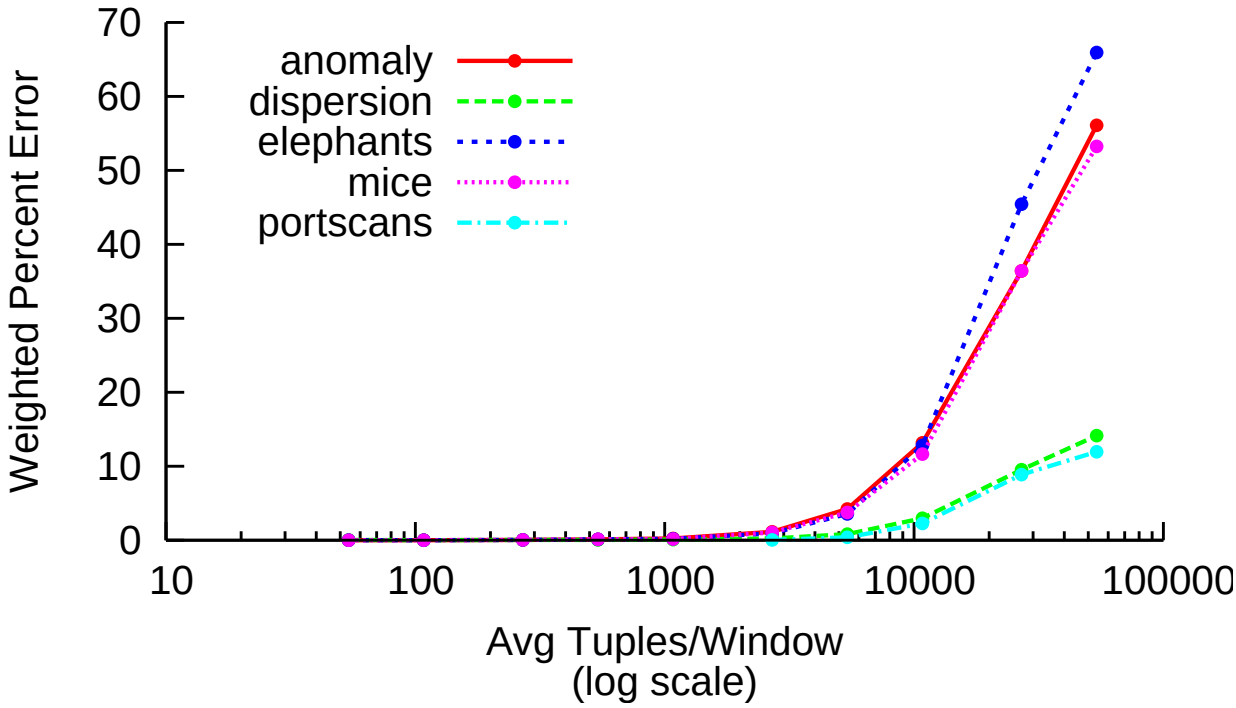

Figure 6.7: Result error of TelegraphCQ with Data Triage, running my queries with varying window sizes over the 5th week of the NERSC trace.

...

Due to the projection and duplicate elimination operations performed and the heavy-tailed distribution of our network traffic across prefixes, such queries can be approximated from a small random sample of the input stream with relatively high accuracy [47].

The remaining three queries produced higher result error. Analyzing the approximate results of these queries in detail, I determined that these larger errors were due to a combination of query and data characteristics. All three high-error queries computed SUM aggregates over the bytes_sent field of the flow records. When using a random sample to approximate such an aggregate, the expected relative error for the sum of a group is given by:

$$\frac{\sqrt{\frac{(1-f)\sigma^2 N}{f}}}{\text{actual sum}} \tag{6.2}$$

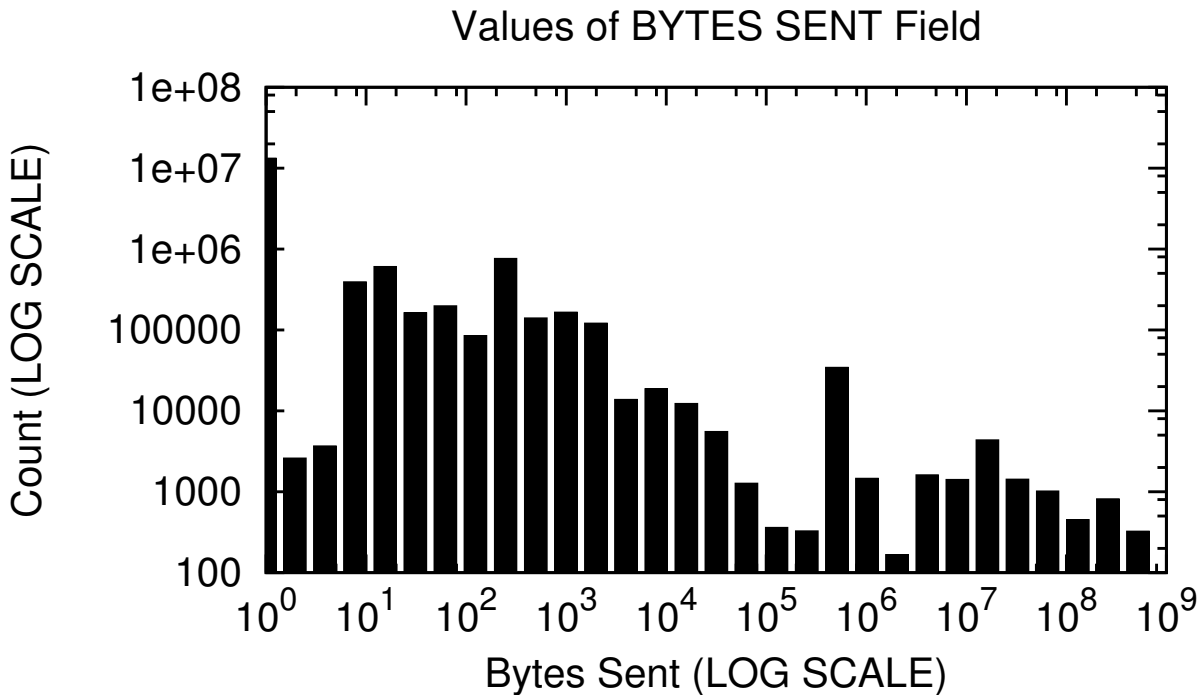

Figure 6.8: A histogram of observed values in the bytes_sent field of the flow records in our trace. Note the logarithmic scale on both axes. The high variance of these values makes it difficult to approximate queries that aggregate over this field.

where N is the number of tuples in the group, f is the fraction of tuples that are included in the sample, and and σ is the variance of the values in the group [39].

As Figure 6.8 shows, the values of the bytes_sent field have extremely high variance. Since the approximation error for a SUM aggregate is proportional to the variance of the aggregate column, random sampling produced significant errors on the queries that computed SUM aggregates over the bytes_sent column.

To demonstrate that the error I observed was in fact largely due to computing sums of a high-variance column, I replaced the SUM aggregates in the elephants, mice, and anomaly queries with COUNT aggregates and reran the experiment. As Figure 6.9 shows, this change decreased the error of the elephants and mice queries significantly.

Error for the anomaly detection query also decreased with the switch to a COUNT aggregate, but by a smaller factor. Since it computes a complete traffic matrix, the anomaly detection query produces results with a large number of groups. The random sample is spread across the groups, so the effective sample

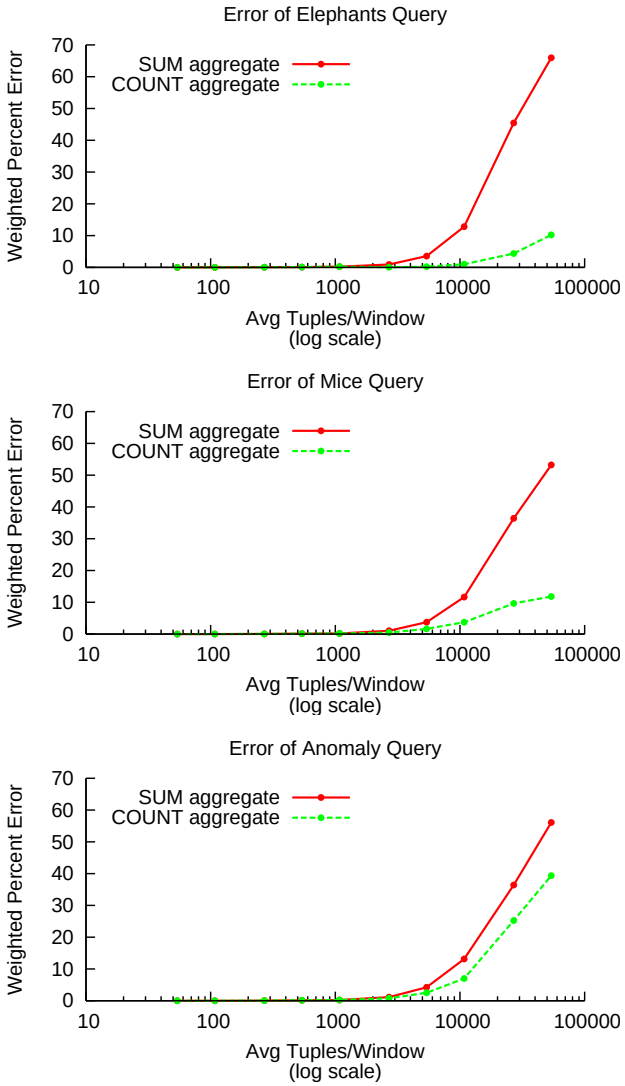

Figure 6.9: An illustration of the decrease in result error that occurs when SUM aggregates over a high-variance field are replaced with COUNT aggregates.

size within each group is small. Indeed, as Figure 6.10 shows, the vast majority of ⟨source, destination⟩ pairs occurred only once in a given time window. If the random sample happens not to contain the tuple from one of these singleton groups, then the error of that group will be 100 percent.

Overall, my implementation of Data Triage gives TelegraphCQ an additional order of magnitude of throughput capacity for this data and workload. This additional capacity comes at the cost of producing approximate answers. The amount

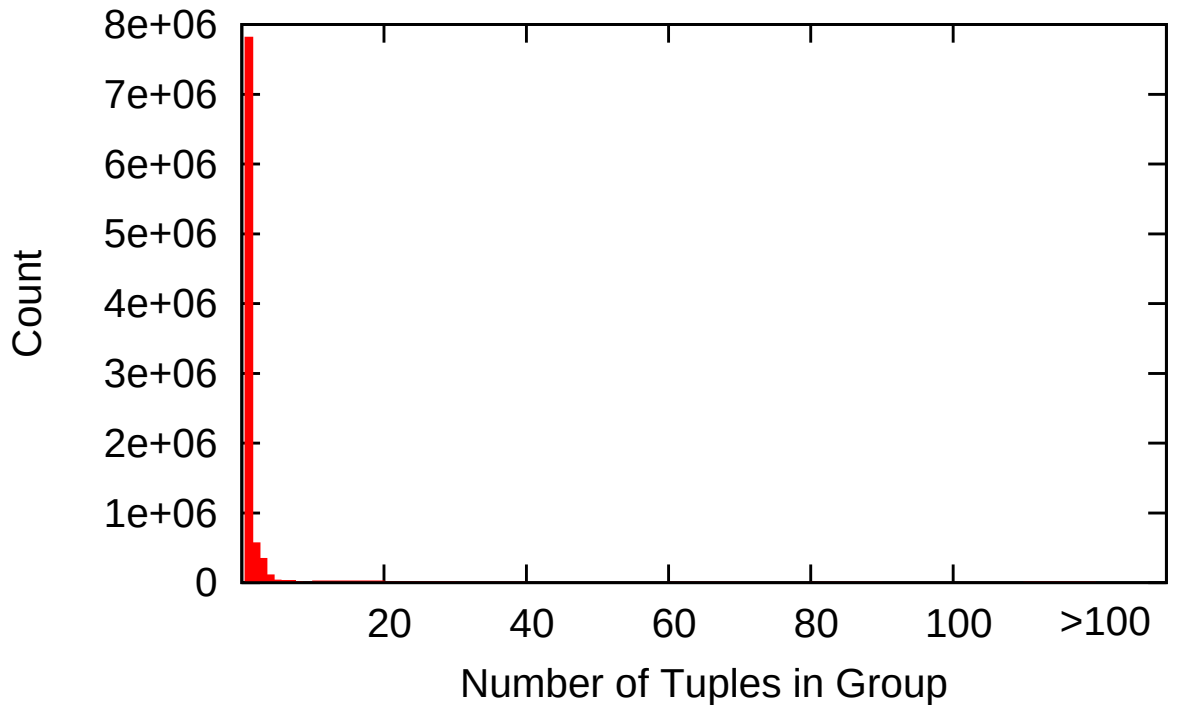

Figure 6.10: Histogram of the number of tuples in a group in the output of the "anomaly detection" query. A count of k means that a particular $\langle$source, destination$\rangle$ address pair appeared k times in a time window. The bins of the histogram for more than 10 tuples are hidden below the border of the graph.

of approximation error observed varies depending on the query, the data distribution, and the strengths of the particular approximation technique in use. Some parts of the query and data workload pushed the reservoir sampling technique beyond its useful range. Other approximation techniques may be able to correct for this deficiency. In general, when using query approximation, it is important to match the approximation technique to the expected query and data characteristics. Early observations of this fact motivated my design choice to modularize Data Triage's approximation code.

6.6.3 FastBit

The next set of experiments measured the throughput of the FastBit component of the system. I designed these experiments and worked with the FastBit team to implement them. The experiments analyze FastBit's performance in two separate components: loading the index and querying the index. To measure index load performance, we conducted an experiment to determine the effect of batch size on

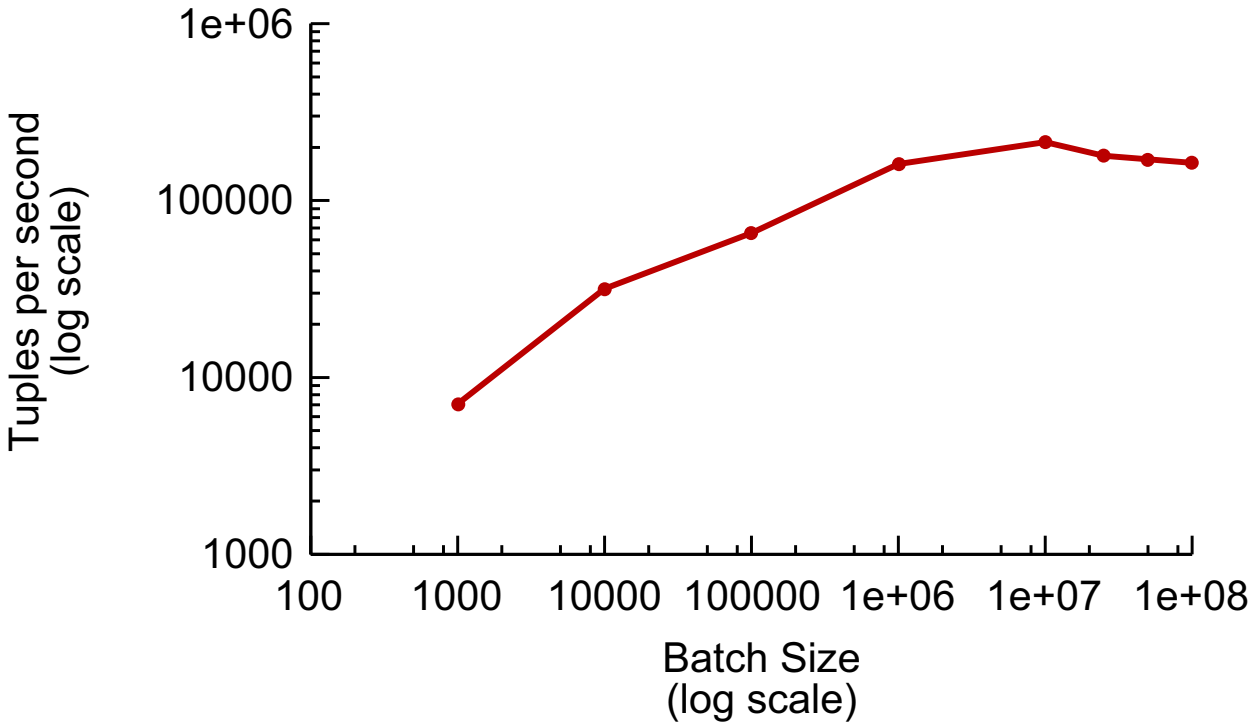

Figure 6.11: Speed for appending data and building the bitmap index in batches of sizes ranging from 1,000 to 100,000,000. Each tuple contains 11 attributes (48 bytes).

load throughput. For querying performance, we measured average response time while varying the amount of historical data incorporated into the analysis.

6.6.3.1 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 of various sizes to identify the most efficient batch size. In particular, the batch sizes ranged from 1,000 to 100,000,000 tuples. Each tuple contains 11 attributes.

Figure 6.11 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.

When the batch size is small, say, 1000 tuples, fixed overheads such as opening index scans and parsing schema information dominate the overall running time. When the batch size is in the millions, the total time required to load the index is dominated by the overhead of constructing index records and writing them to disk.

Once the batch size increases beyond 10 million records, throughput decreases somewhat due to caching effects. To enable query processing during index build operations, FastBit keeps two copies of the index. These copies are stored in index directories called the *active directory* and *backup directory*. During a bulk load operation, the system first loads new tuples into the backup directory. Once the backup directory is up to date, FastBit swaps the active and backup directories and copies the new index records to the former active directory. When the batch size surpasses 10 million records, the newly-created records no longer fit in the operating system's buffer cache, so the final bulk copy operation must read data from disk instead of memory.

6.6.3.2 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 6.12 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. One 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

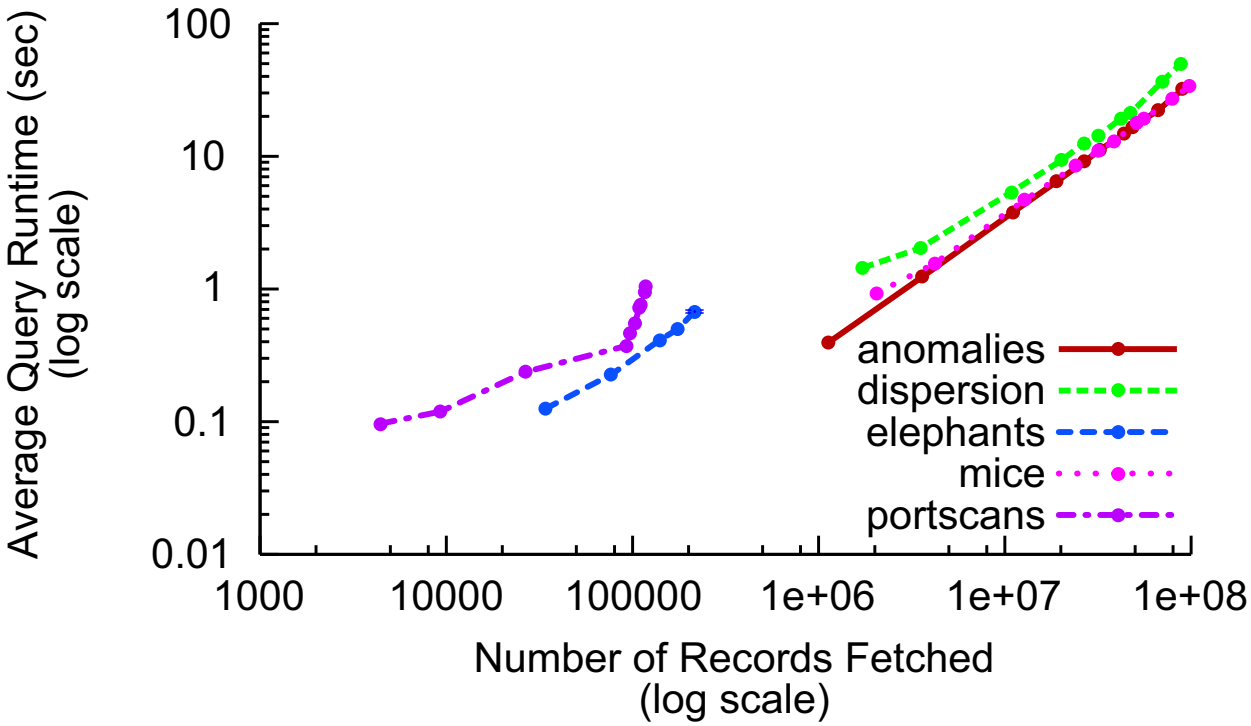

Figure 6.12: FastBit index lookup time for 5 types of historical queries with various lengths of history denoted by variable : history .

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.

6.6.4 Controller

Overall, the Controller component was not a bottleneck. I measured Controller overhead by running the TelegraphCQ query select ∗ from Flow on TelegraphCQ via the Controller, while sending flow data directly to TelegraphCQ. This query produces an output tuple for every input tuple and therefore puts stress on the Controller. I found that the TelegraphCQ Front End process dominated CPU usage. The rate at which the Controller can consume tuples was not a bottleneck. For the workload I studied in this chapter, the analysis and report generation functions

of the Controller consumed far less CPU time than the process of receiving and parsing result tuples from TelegraphCQ. This distribution of work may change in the future if our system were to incorporate complex predictive logic such as that proposed by Soule *et al.*[99]. Currently, however, the Controller component is not a bottleneck.

6.6.5 End-to-end Throughput Without Data Triage

My next two sets of experiments combined TelegraphCQ, the FastBit loader, the FastBit query engine, the Controller, and the Ingress Manager to create an end-to-end system. For the first round of experiments, I disabled the Data Triage components of the Ingress Manager. I fed our trace through this system and measured the time until the Controller produced its last report. I 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 without Data Triage.

I benchmarked each pair of TelegraphCQ/FastBit queries separately. The ranges of time selected in the FastBit queries were as depicted in Section 6.5. I varied the window size of the TelegraphCQ queries over the range of time intervals used in Section 6.6.1. Flow records were appended to the FastBit index in batches of 1 million.

Figure 6.13 shows the results of this experiment. Each thick line shows the performance 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 dominated 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 the system 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) caused a temporary lag in the query response time. However, the decrease in overall throughput due to loading the index was negligible.

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 three factors. As I observed in the previous section, the TelegraphCQ component of the queries runs slowly at smaller window sizes. Also, the FastBit query for this analysis is also slower

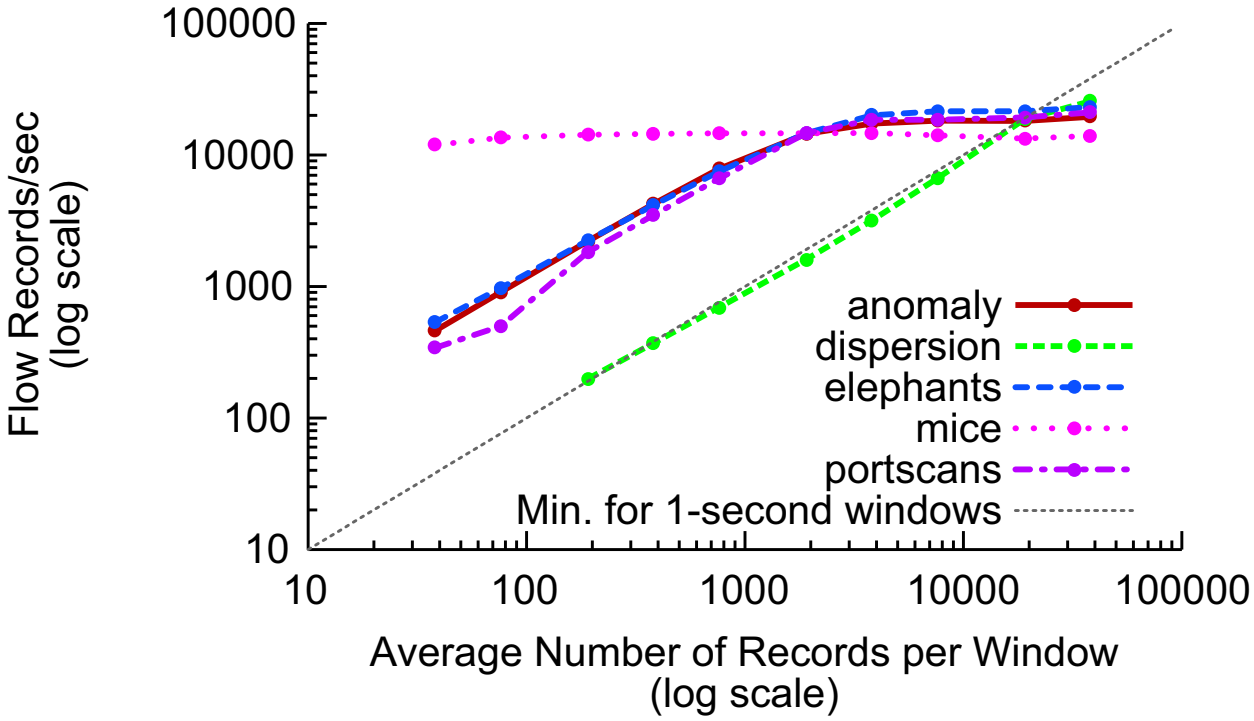

Figure 6.13: End-to-end throughput with varying window sizes over the 5th week of the NERSC trace. Note the logarithmic scale. The thin dotted line indicates the minimum throughput necessary to deliver results at 1-second intervals.

since the amount of data fetched is very large. Finally, as I noted in Section 6.5, 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. The FastBit team is working to remove this bottleneck by adding support for arithmetic expressions in FastBit's `select` lists.

In Figure 6.13 I 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, the 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 my estimates (See Section 6.2.1) of the average combined data rates for all DOE labs. However, the system tested in this experiment cannot handle my 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 [87] to trade off query result accuracy for response time, or add additional CPU capacity (possibly via parallelism [95]) to the system. The

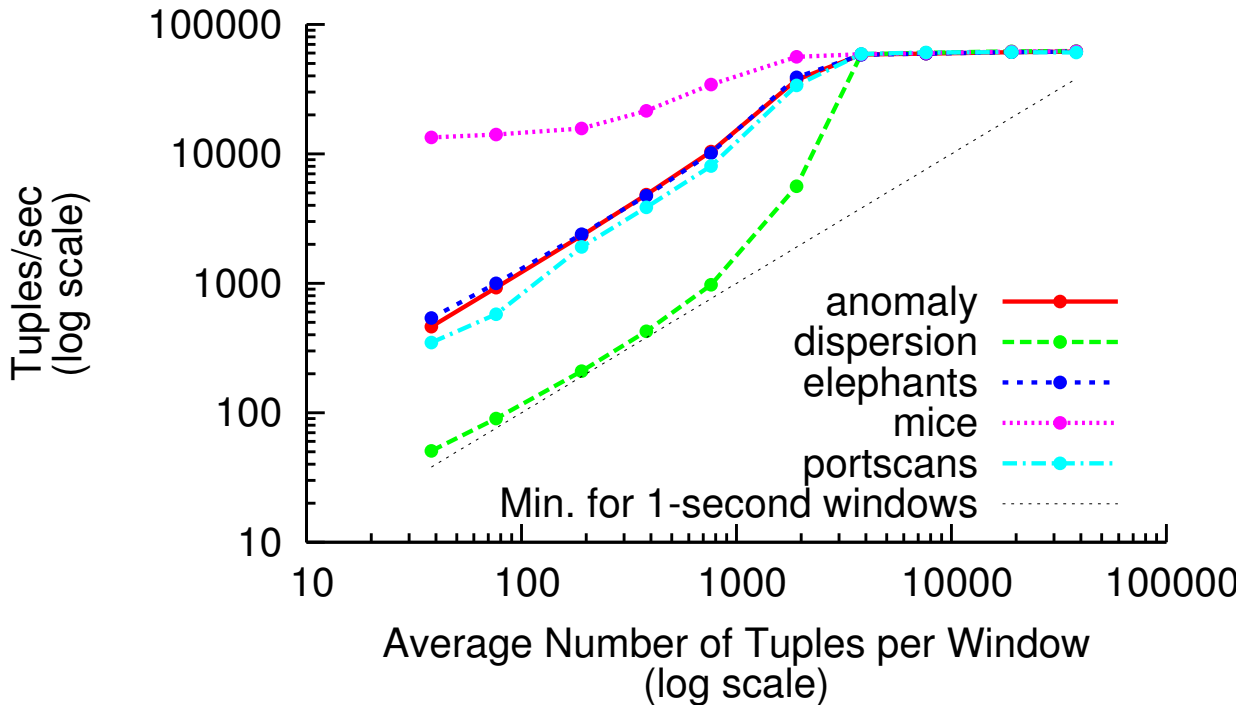

Figure 6.14: End-to-end throughput at varying window sizes with Data Triage enabled. Note the logarithmic scale. The thin dotted line indicates the minimum throughput necessary to deliver results at 1-second intervals.

experiments in Section 6.6.7 explore the tradeoffs between these three approaches.

6.6.6 End-to-end Throughput With Data Triage

My next experiment was similar in overall structure to the experiment in the previous section, except that it measured throughput with Data Triage. As before, the experiment used the end-to-end system, including index loading and both TelegraphCQ and FastBit queries. For this second experiment, I also enabled the Data Triage components in the Ingress Manager. Figure 6.14 shows the results of this experiment. Throughput is limited by FastBit query performance when the number of tuples per window is small. As the number of tuples per window increases, throughput goes up, eventually reaching a plateau of roughly 60000 tuples per second for all of the queries. This plateau is due primarily to the speed with which the system could stage tuples to disk and load them into FastBit. The flow records start as a single stream of comma-delimited text, and the FastBit loader expects a separate binary file for each column. Conversion of text to binary and the writing of multiple binary files consumes approximately as much CPU time as

the actual index load operation, limiting overall throughput.

Since the TelegraphCQ components of the system had not changed from those used in Section 6.6.2, result accuracy in estimating the results of the TelegraphCQ queries was the same as before. Approximation was not used for the FastBit portion of the queries.

In practice, index loading should not have a serious effect on the system's ability to absorb bursts without delaying query results. In the query workload I used in this study (See Section 6.5, the historical data that is used for comparison with current trends is always at least 24 hours in the past. Strong diurnal and weekly patterns in network traffic make it unnatural to compare current trends against traffic from a different time of day or day of the week. Because my queries do not need data from the most recent 24 hours of history, the system is free to defer index loading for up to 24 hours if a sustained burst of traffic exceeds its indexing capacity. This deferred-loading strategy is similar to that used by Chandrasekaran and Franklin in their work on indexing archived streams [20].

Deferring indexing gives the system significant additional "headroom" to consume data during such bursts, as Figure 6.15 shows. The graph in Figure 6.15 illustrates query throughput when indexing operations are (temporarily) disabled. As the graph illustrates, the end-to-end system can sustain bursts in excess of 200000 flow records per second as long as the system can defer index loading until a relatively quiet period in the future.

6.6.7 Timing-Accurate Playback and Latency

My next set of experiments measured the end-to-end latency of query results over time. I wrote a program that plays back the trace at a multiple of real time, using the timestamps embedded in the trace to determine when to send flow records to the system. I ran this program on the trace used in the previous sections' experiments, selecting speed multiples such that the average data rate was from 50 to 200 percent of the system's measured throughput capability from the experiments in Section 6.6.5. I measured the delay of query results by comparing the time at the end of the window with the time at which the system produced the window's report for the user. I repeated the experiment both with the Data Triage components of the system both enabled and disabled.

Figures 6.16 through 6.20 show the results of the experiments. Overall, when the average data rate was 50% of the system's capacity, the system was generally able to return results with a delay of less than 10-15 seconds, with or without Data Triage. However, when the average rate of data arrival was equal to the system's capacity, the no-triage runs of the experiment built up a persistent delay that only gradually decreased during periods of relatively light traffic. In contrast, with Data

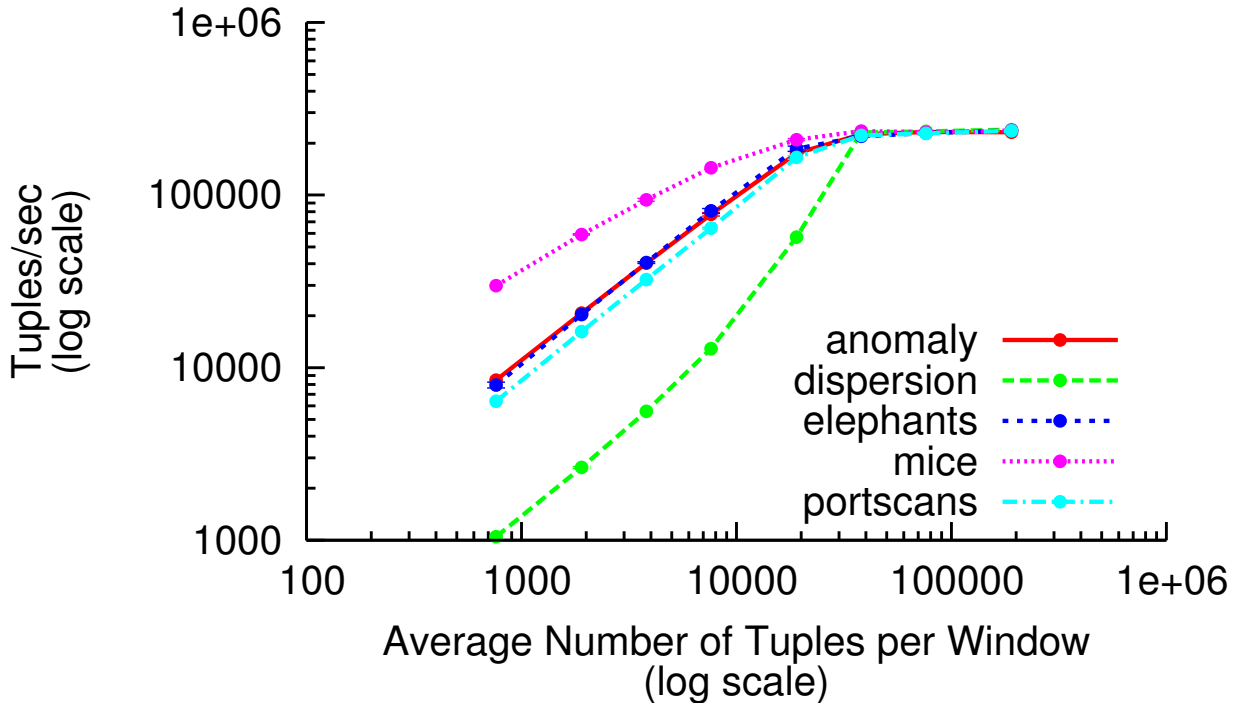

Figure 6.15: End-to-end throughput at varying window sizes with Data Triage enabled and index loading disabled. Note the logarithmic scale.

Triage enabled, delay remained consistently below 10 seconds.

At 200% of capacity, query result delay increases continually without Data Triage. After ten minutes of traffic at this rate, the no-triage runs of the experiment were 5-10 minutes behind real time. Such a delay would be highly detrimental to the efforts of administrators to find the cause of the surge in bandwidth usage. With Data Triage enabled, however, delay remained below 10 seconds as before.

Delay when running the "dispersion" query followed the same general trend as that of the other runs. However, the result delay with Data Triage enabled occasionally reached as high as 20 seconds before dropping back below the 10-second delay constraint. The current version of TelegraphCQ delays most of the processing for each time window until the end of the window. Also, the query's innermost subquery specified a 100-second time window that advances at 10-second intervals. The current version of TelegraphCQ recomputes query results for the entire window each time it advances, which in this case increases the amount of work for this subquery by a factor of 10. These factors create a significant delay between the time a tuple enters TelegraphCQ and the time that processing triggered by the

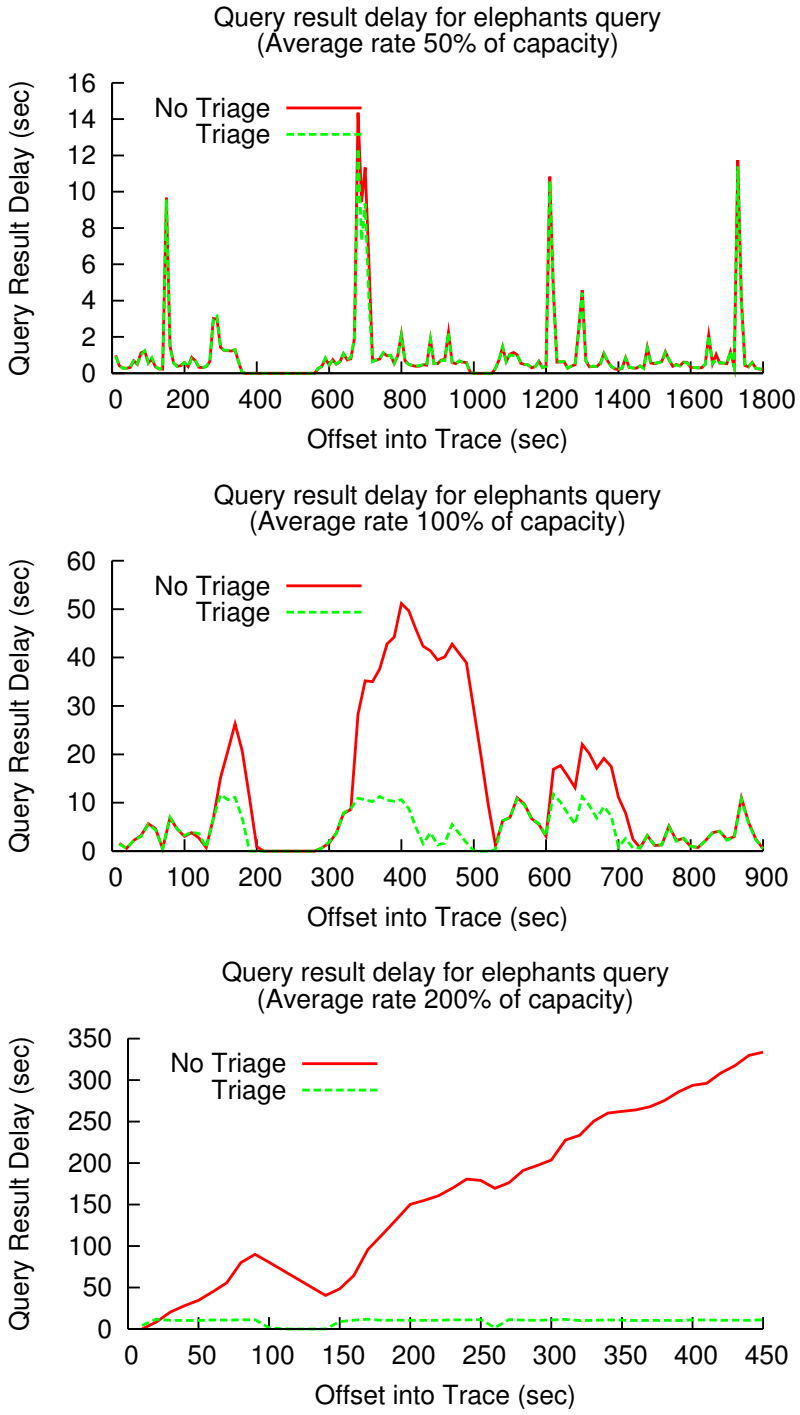

Figure 6.16: Measured query result latency over time for the "elephants" query. In the first graph, the system was running at 50% of average capacity. The second graph shows 100% capacity, and the third graph shows 200% capacity.

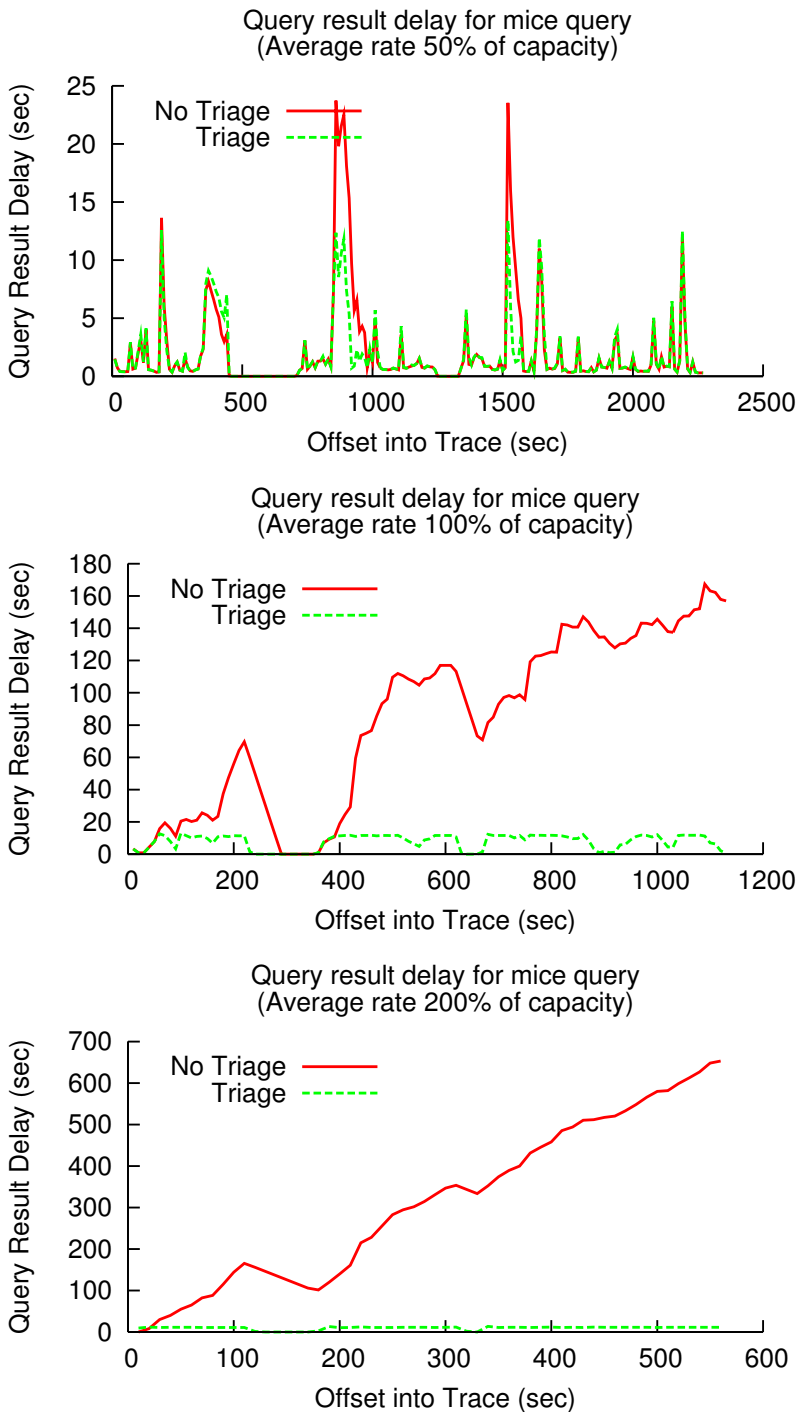

Figure 6.17: Measured query result latency over time for the "mice" query. In the first graph, the system was running at 50% of average capacity. The second graph shows 100% capacity, and the third graph shows 200% capacity.

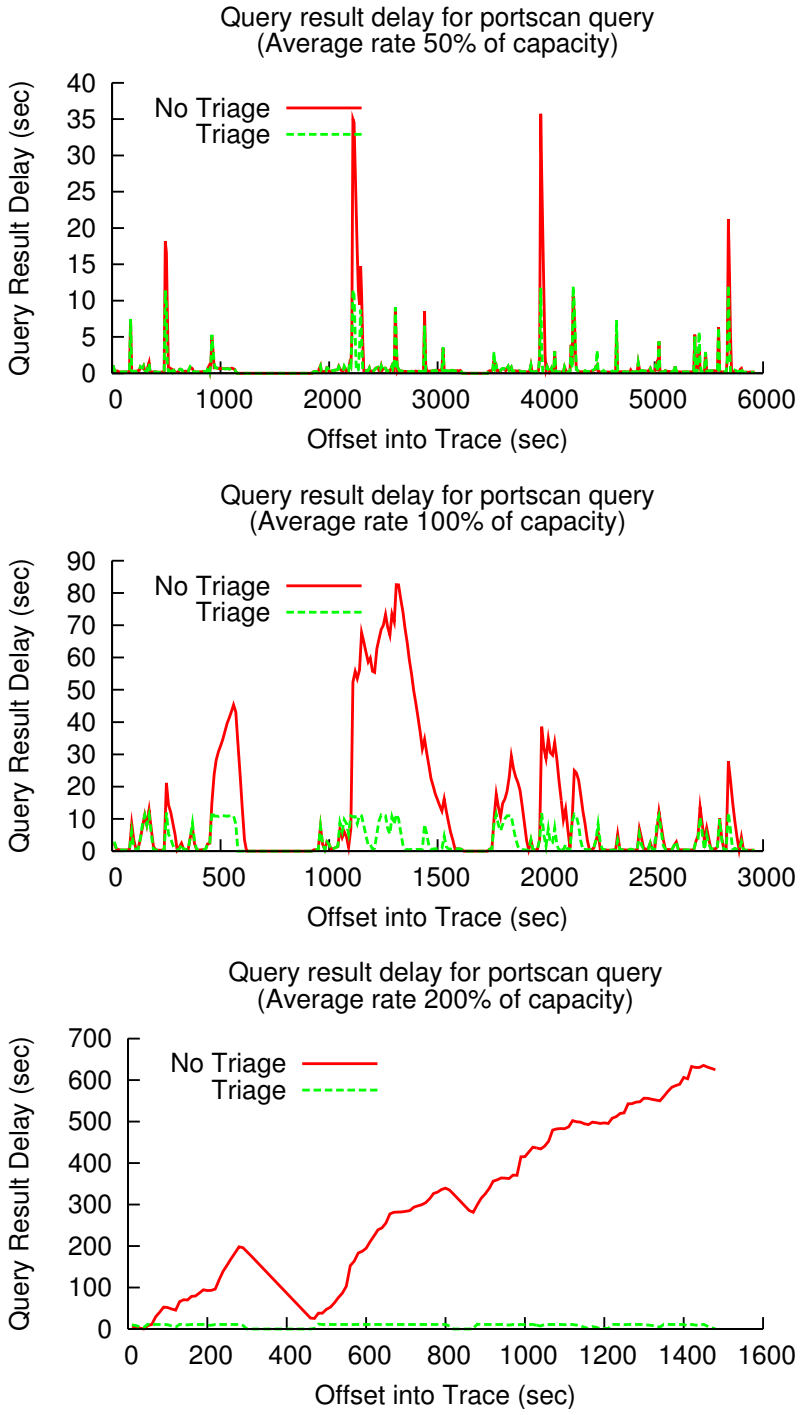

Figure 6.18: Measured query result latency over time for the "portscans" query. In the first graph, the system was running at 50% of average capacity. The second graph shows 100% capacity, and the third graph shows 200% capacity.

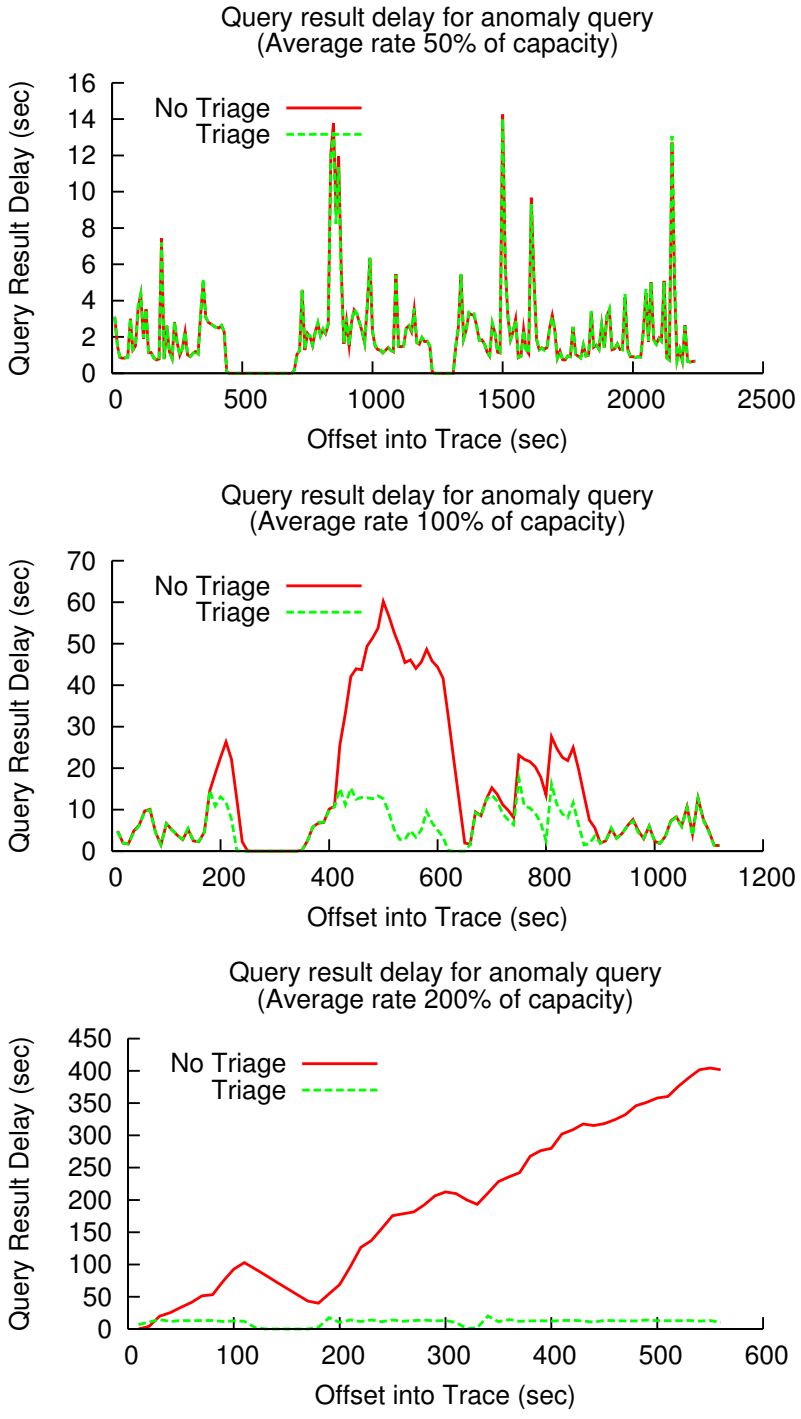

Figure 6.19: Measured query result latency over time for the "anomaly detection" query. In the first graph, the system was running at 50% of average capacity. The second graph shows 100% capacity, and the third graph shows 200% capacity.

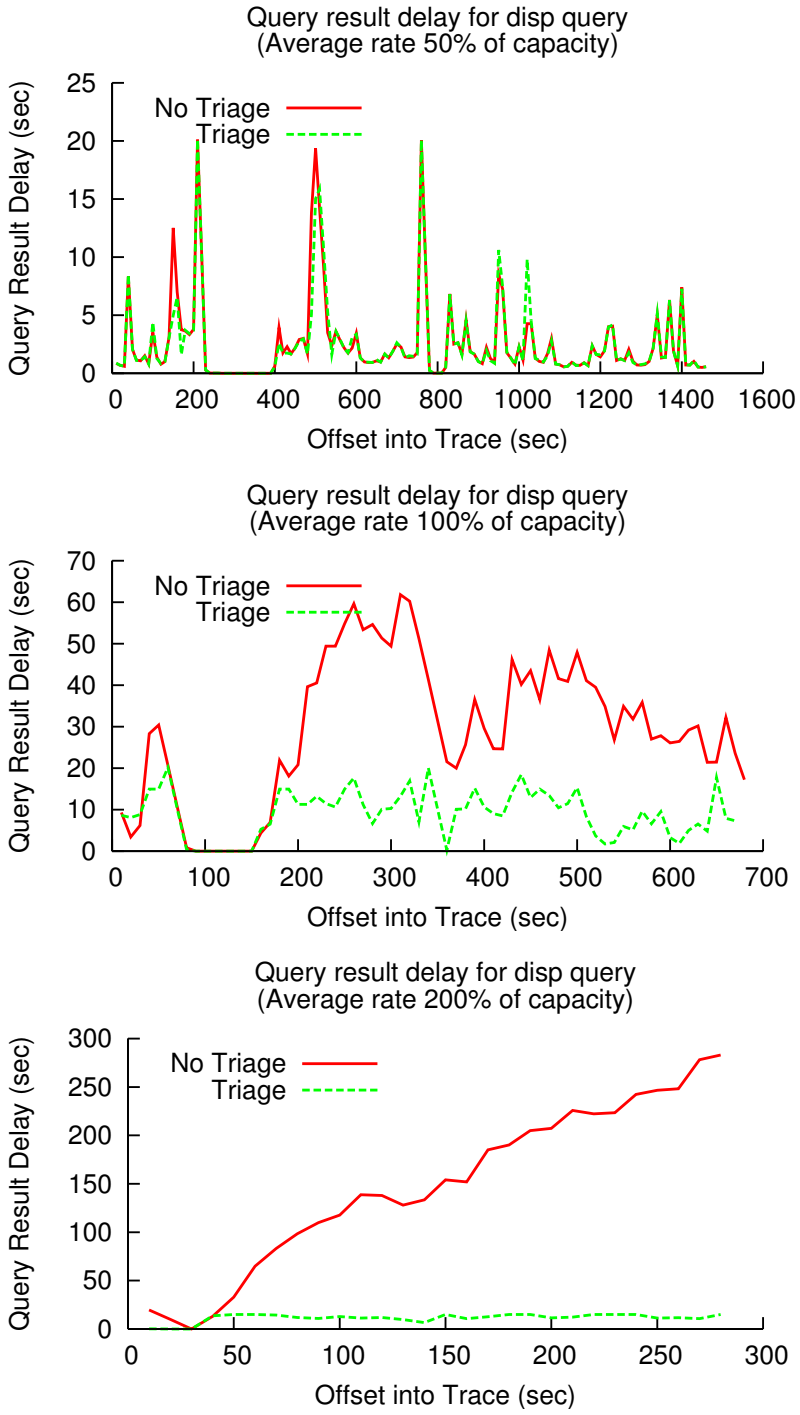

Figure 6.20: Measured query result latency over time for the "dispersion" query. In the first graph, the system was running at 50% of average capacity. The second graph shows 100% capacity, and the third graph shows 200% capacity.

tuple is completed. As a result, when the Triage Scheduler begins triaging tuples, TelegraphCQ has already buffered a large number of tuples internally, leading to a delay before response time begins to improve.

Developers on the commercial version of TelegraphCQ are currently making improvements that should greatly reduce this internal delay. In particular, these developers are implementing a hash-based GROUP BY to avoid buffering an entire window's results, as well as efficient methods of incrementally computing query results when the RANGE of a window clause is greater than its SLIDE.

6.6.8 Measuring the Latency-Accuracy Tradeoff

My final experiment measured the effect of the user's choice of delay constraint on query result accuracy. For this experiment, I used the setup from the previous section's experiments. First, I ran the TelegraphCQ components of the queries without Data Triage and logged the full query answers to disk. Then I enabled all components of the system, including the Data Triage and indexing components, and reran the trace at a 100% average load, logging the results of the main and shadow queries to disk. I repeated the experiment while varying the delay constraint from 2 seconds to 100 seconds. Afterwards, a postprocessing step measured query result error using the same technique used in Section 6.6.2.

Figure 6.21 shows the results of this experiment. All five queries behaved similarly as the delay constraint varied, with error decreasing as the number of tuples the system needed to triage went down. Interestingly, the relative amounts of error across the queries differed compared with the results in Section 6.6.2.

Whereas the earlier experiment forced the system to triage every tuple, the experiment in this section "played back" the trace at a multiple of real time, allowing the Triage Scheduler to choose which tuples to triage. In this real-time experiment, the queries consumed data at different rates, and these rates varied depending on the characteristics of the data itself. For example, when there were many combinations of address and port in a given time window, the "portscans" query, which computes the set of unique $\langle$address, port$\rangle$ combinations, slowed down by a larger degree than the other queries. Overall, the errors across queries were closer together in the real-time experiment, indicating that the timing of triage decisions can be as important to query result error as the quality of the query approximation itself.

6.7 Conclusions

The results in this chapter demonstrate that an end-to-end system that uses Data Triage for monitoring high-speed networking data streams is feasible. They also show that my extension Data Triage to support archival is a viable architecture. I demonstrated that, without Data Triage, a network monitoring system can bottleneck on the streaming query processor, causing significant delays in query result generation. I I found that overprovisioning the monitoring hardware by a factor of 2 to 4 would keep query result latency below a 10-second threshold. With Data Triage enabled, query result latency would stay under control, using significantly less hardware than would be necessary to achieve the same latency without Data Triage.

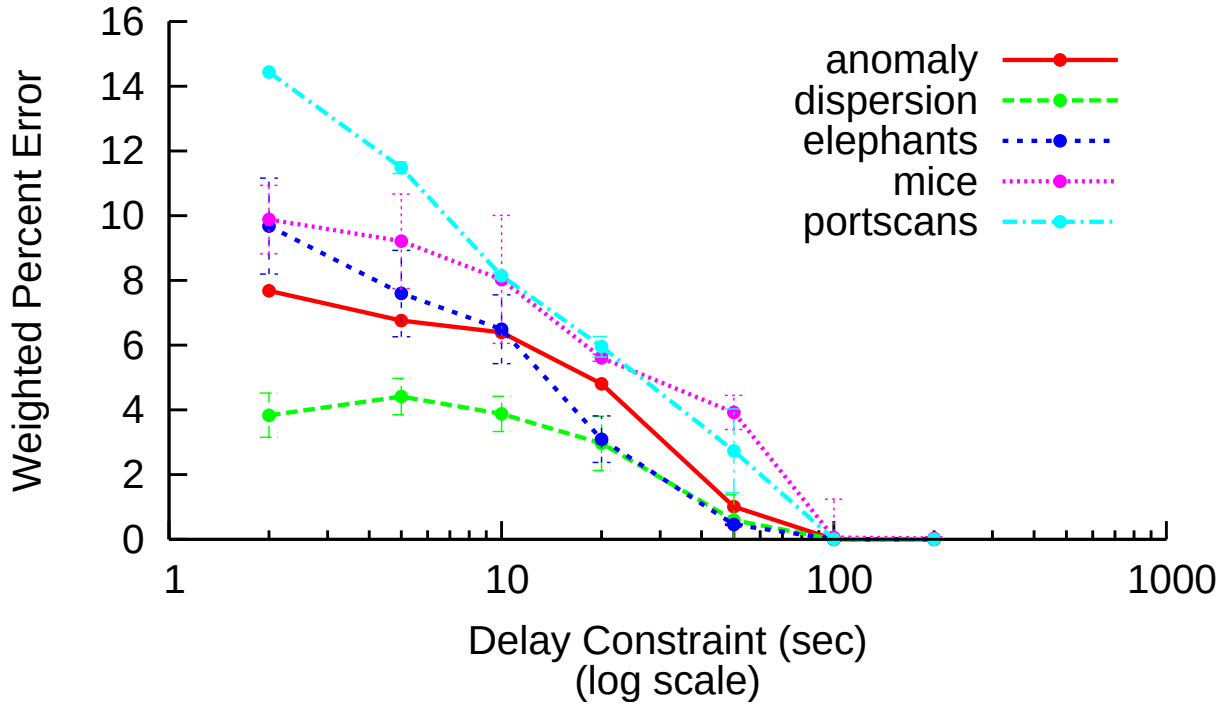

Figure 6.21: Error of query results with Data Triage as the delay constraint varies. The average data rate was 100% of TelegraphCQ's capacity. Each point represents the mean of 5 runs of the experiment; error bars indicate standard deviation.

Chapter 7

Conclusion

People do not like to think. If one thinks, one must reach conclusions. Conclusions are not always pleasant.

— Helen Keller (1880 - 1968)

Enterprise networks have been increasing in complexity while at the same time becoming more vital to the operations of global organizations. These trends create a need for network monitoring systems that run declarative queries and can observe the low level traffic on each network link. Unfortunately, passive network monitoring currently requires a significant amount of processing power to keep up with the maximum data rate of a network link, and hardware costs represent a significant barrier to the adoption of such an infrastructure.

My approach to solving this hardware cost problem takes advantage of the bursty nature of network traffic by using hardware provisioned for the average data rate on the network link, which is generally at least an order of magnitude less than the maximum rate. Of course, such a strategy requires a comprehensive strategy for handling the overload that will inevitably occur when a burst of high-speed network traffic exceeds the system's capacity.

In this dissertation, I have described the principle components of such a strategy:

- The Data Triage architecture, which adds a fallback mechanism based on approximate query processing to a streaming query processor.
- The Delay Constraints API and associated scheduling algorithm, which control the tradeoffs between query result delay and accuracy in the Data Triage architecture.
- New histogram-based query approximations that allow Data Triage's approximate data path to cover an important class of network monitoring query beyond those that existing techniques already support.

Finally, I described a deployment study that demonstrates that my Data Triage implementation functions properly in the context of an end-to-end monitoring system being developed for the U.S. Department of Energy's laboratory networks.

Bibliography

[1] T.F. Abdelzaher, J.A. Stankovic, Chenyang Lu, Ronghua Zhang, and Ying Lu. Feedback performance control in software services. *Control Systems Magazine*, June 2003. 1.3

[2] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, pages 487–498, 2000. 2.2.1

[3] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286, 1999. 2.2.1, 3.1.1

[4] AmberPoint. Closed-loop governance: Amberpoint solutions, 2006. http://www.amberpoint.com/solutions/governance.shtml. 1.3

[5] APNIC. Whois database, October 2005. ftp://ftp.apnic.net/apnic/whois-data/APNIC/apnic.RPSL.db.gz. 5.6

[6] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford University, 2003. 1.6.1.1

[7] Internet Assigned Numbers Authority. http://www.iana.org. 5.2

[8] Raian Babcock, Surajit Chaurhuri, and Gautam Das. Dynamic sample selection for approximate query processing. In *SIGMOD*, 2003. 2.2.1

[9] J.C.R. Bennett and Hui Zhang. WF2Q: Worst-case fair weighted fair queueing. In *INFOCOM*, volume 1, pages 120–128, 1996. 2.3

[10] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, pages 509–517, 1975. 4.7.1

[11] E. Wes Bethel, Scott Campbell, Eli Dart, Kurt Stockinger, and Kesheng Wu. Accelerating Network Traffic Analysis Using Query-Driven Visualization. In *2006 IEEE Symposium on Visual Analytics Science and Technology (to appear)*, 2006. 2.5, 6.2

[12] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: a multidimensional workload-aware histogram. *SIGMOD Record*, 30(2):211–222, 2001. 2.4

[13] Shaofeng Bu, Laks V.S. Lakshmanan, and Raymond T. Ng. Mdl summarization with holes. In *VLDB*, 2005. 2.4

[14] A. Buchsbaum, G. Fowler, B. Krishnamurthy, K. Vo, and J. Wang. Fast prefix matching of bounded strings. In *ALENEX*, 2003. 2.4

[15] Ahmet Bulut and Ambuj K. Singh. Swat: Hierarchical stream summarization in large networks. In *ICDE*, 2003. 2.2.1

[16] W. Burlesona, J. Ko, D. Niehaus, K. Ramamritham, J.A. Stankovic, G. Wallace, and C. Weems. The spring scheduling coprocessor: a scheduling accelerator. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 1999. 2.3

[17] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *VLDB Journal*, 10(2–3):199–223, 2001. 2.2.1, 3.1.1

[18] C. Y. Chan and Y. E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. In *SIGMOD*, 1999. 2.5

[19] S. Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003. 1.6.1.1, 1.6.1.2

[20] Sirish Chandrasekaran and Michael Franklin. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *VLDB*, 2004. 2.5, 3.4, 6.6.6

[21] S. Chaudhuri, R. Motwani, and V.R. Narasayya. On random sampling over joins. In *SIGMOD*, 1999. 2.2.1

[22] B-C. Chen, V. Yegneswaran, P. Barford, and R. Ramakrishnan. Toward a Query Language for Network Attack Data. In *NetDB Workshop*, 2006. 2.5, 6.5.5

[23] AT&T Corp. Making the case for enterprise mobility. Technical Report AB-0421-01, AT&T, May 2006. 1.2

[24] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, 2003. 2.2

[25] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: high performance network monitoring with an sql interface. In *SIGMOD*, 2002. 1.4, 2.2

[26] Roger O. Crockett. Ip's killer apps are coming. *BusinessWeek*, March 2006. 1.2

[27] Abhinandan Das, J. E. Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD*. ACM Press, 2003. 2.2.1

[28] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM Press. 2.3

[29] Amol Deshpande and Joseph Hellerstein. On using correlation-based synopses during query optimization. Technical Report CSD-02-1173, U.C. Berkeley EECS Department, May 2002. 2.4, 3.1.1

[30] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational experiences with high-volume network intrusion detection. In *CCS*, pages 2–11. ACM Press, 2004. 2.2

[31] eMarketer. Online ad spending still growing in the us and uk. *eMarketer.com*, June 2006. 1.2

[32] Pl Erik Eng and Morten Haug. Automatic response to intrusion detection. Master's thesis, Agder University College, Grimstad, June 2004. 1.3

[33] Rajeev Motwani *et al*. Query processing, resource management, and approximation in a data stream management system. In *CIDR*. ACM Press, 2003. 2.2.1

[34] Information Technology for European Advancement. The explosive growth of service-oriented architecture adoption. Technical report, ITEA, 2005. 1.2

[35] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, September 1993. ftp://ftp.internic.net/rfc/rfc1519.txt. 2.4, 5.2

[36] Minos Garofalakis and Phillip Gibbons. Approximate query processing: Taming the terabytes!, 2001. 2.2.1

[37] Minos Garofalakis and Amit Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *PODS*, 2004. 2.4, 5.3.2.3

[38] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *SIGMOD Conference*, 2001. 2.2.1

[39] Larry Gonick and Woollcott Smith. *The Cartoon Guide to Statistics*. HarperPerennial, 1993. 6.6.2

[40] Goetz Graefe. Goetz graefe. *SIGMOD Record*, 18(9):509–516, 2006. 2.5

[41] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997. 5.3.2.3

[42] Aberdeen Group. Hosted crm gains momentum as salesforce.com paces the market. Technical report, Aberdeen Group, October 2003. 1.2

[43] Sudipto Guha. Space efficiency in synopsis construction algorithms. In *VLDB*, 2005. 5.5.4

[44] Sudipto Guha, Nick Koudas, and Divesh Srivastava. Fast algorithms for hierarchical range histogram construction. In *PODS*, 2002. 2.4

[45] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995. 3.3.2

[46] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD*, pages 287–298. ACM Press, 1999. (document), 3.3.3, A.1

[47] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997. 6.6.2

[48] Gianluca Iannaccone, Christophe Diot, and Derek McAuley. The CoMo white paper. Technical Report IRC-TR-04-017, Intel Research, September 2004. 2.2

[49] Yannis E. Ioannidis. Universality of serial histograms. In *VLDB*, pages 256–267, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. 2.4

[50] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD*, pages 233–244, New York, NY, USA, 1995. ACM Press. 2.4, 5.6

[51] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 24–27 1998. 2.4, 3.1.1

[52] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998. 5.6

[53] Jaewoo Kang, Jeffrey F. Naughton, and Stratis D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003. 2.2.1

[54] Panagiotis Karras and Nikos Mamoulis. One-pass wavelet synopses for maximum-error metrics. In *VLDB*, 2005. 2.4

[55] Masayoshi Kobayashi, Tutomu Murase, and Atsushi Kuriyama. A longest prefix match search engine for multi-gigabit ip processing. In *IEEE International Conference on Communications*, 2000. 5.2

[56] Stefan Kornexl, Vern Paxson, Holger Dreger, Anja Feldmann, and Robin Sommer. Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In *Internet Measurement Conference*, 2005. 2.5

[57] Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Optimal histograms for hierarchical range queries (extended abstract). In *PODS*, pages 196–204, 2000. 2.4

[58] Sailesh Krishnamurthy and Michael J. Franklin. Shared hierarchical aggregation over receptor streams. Technical Report UCB-CSD-05-1381, UC Berkeley, 2005. 4.5, 4.5.1

[59] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006. 4.5.2

[60] Iosif Lazaridis and Sharad Mehrotra. Capturing sensor-generated time series with quality guarantees. In *ICDE*, 2003. 2.2.1

[61] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In Deepinder P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, 1993. 1.4

[62] L. Lim, M. Wang, and J. S. Vitter. Sash: A self-adaptive histogram set for dynamically changing workloads. In *VLDB*, 2003. 2.2.1

[63] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. 2.3

[64] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 56, Washington, DC, USA, 1999. IEEE Computer Society. 2.3

[65] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, 2002. 1.6.1.2

[66] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *VLDB*, 2000. 2.2.1, 2.4, 4.7

[67] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD*, pages 448–459, 1998. 2.2.1, 2.4, 3.1.1

[68] Research In Motion. Blackberry for sales professionals. Technical report, RIM, 2006. 1.2

[69] M. Muralikrishna and D. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *SIGMOD*, pages 28–36, 1998. 2.4

[70] S. Muthukrishnan, Viswanath Poosala, and Torsten Suel. "On Rectangular Partitionings in Two Dimensions: Algorithms, Complexity, and Applications". In *ICDT*, Jerusalem, Israel, January 1999. 5.4.2.5, 5.4.2.5

[71] S. Muthukrishnan, Viswanath Poosala, and Torsten Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. *Lecture Notes in Computer Science*, 1540:236–256, 1999. 5.5.2

[72] J. Nagle. On packet switches with infinite storage. *Communications, IEEE Transactions on*, 35(4):435–438, April 1987. 2.3

[73] Peter Newman, Greg Minshall, Tom Lyon, Larry Huston, and Ipsilon Networks Inc. Ip switching and gigabit routers. *IEEE Communications Magazine*, 1997. 2.4, 5.2

[74] F. Olken and D. Rotem. Random sampling from databases - a survey. In *Statistics and Computing (invited paper)*, pages 25–42, 1995. 2.2.1

[75] P. O'Neil and D. Quass. Improved Query Performance with Variant Indices. In *SIGMOD*, 1997. 2.5

[76] Dmitry Pavlov and Heikki Mannila. Beyond independence: Probilistic models for query approximation on binary transaction data. 2.2.1

[77] Dmitry Pavlov and Padhraic Smyth. Probabilistic query models for transaction data. In *KDD*, pages 164–173, 2001. 2.2.1

[78] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *USENIX Security Symposium*, January 1998. 2.5

[79] Vern Paxson and Sally Floyd. Wide-area traffic: the failure of Poisson modeling. *IEEE Trans. Networking*, 3(3):226 – 224, 1995. 1.4

[80] 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. 2.5

[81] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305, 1996. 2.4, 5.3.2.2

[82] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *The VLDB Journal*, pages 486–495, 1997. 2.4, 3.1.1, 3.2.1, 4.7

[83] Postgresql database system. `http://www.postgresql.org`. 1.6.1

[84] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2002. 1.6.1.1

[85] K. Ramamritham and J. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1, 1984. 2.3

[86] Ramaswamy Ramaswamy, Ning Weng, and Tilman Wolf. A network processor based passive measurement node. In *Proc. of Passive and Active Measurement Workshop (PAM)*, pages 337–340, Boston, MA, March 2005. (Extended Abstract). 2.2

[87] Frederick Reiss and Joseph M. Hellerstein. Declarative Network Monitoring with an Underprovisioned Query Processor. In *ICDE*, 2006. 2.5, 6.6.5

[88] Y. Rekhter and T. Li. RFC 1771: A Border Gateway Protocol 4 (BGP-4), March 1995. http://www.ietf.org/rfc/rfc1771.txt. 5.2

[89] R.H. Riedi and W. Willinger. *Self-similar Network Traffic and Performance Evaluation*. Wiley, 2000. 1.4

[90] RIPE. Whois database, September 2005. ftp://ftp.ripe.net/ripe/dbase/ripe.db.gz. 5.6

[91] Martin Roesch. Snort-Lightweight Intrusion Detection for Networks. In *USENIX LISA*, 1999. 2.5

[92] Martin Roesch. Snort - lightweight intrusion detection for networks. Technical report, snort.org, 2006. 2.2

[93] Doron Rotem, Kurt Stockinger, and Kesheng Wu. Optimizing Candidate Check Costs for Bitmap Indices. In *CIKM*, 2005. 2.5

[94] David V. Schuehler and John Lockwood. TCP-Splitter: A TCP/IP flow monitor in reconfigurable hardware. In *Hot Interconnects*, pages 127–131, Stanford, CA, August 2002. 1.4

[95] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly Available, Fault-Tolerant, Parallel Dataflows. In *SIGMOD*, 2004. 6.6.5

[96] Arie Shoshani, Luis Bernardo, Henrik Nordberg, Doron Rotem, and Alex Sim. Multidimensional Indexing and Query Coordination for Tertiary Storage Management. In *SSDBM*, July 1999. 6.3

[97] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, 1996. 2.3

[98] 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. 2.5

[99] A. Soule, K. Salamatian, and N. Taft. Combining filtering and statistical methods for anomaly detection. In *ACM/Sigcomm Internet Measurement Conference (IMC)*, 2005. 6.5.4, 6.6.4

[100] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins, 2004. 2.2.1

[101] K. Stockinger and K.Wu et al. Network Traffic Analysis With Query Driven Visualization - SC 2005 HPC Analytics Results. In *Super Computing*, 2005. 2.5, 6.2

[102] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. *SIGCOMM Comput. Commun. Rev.*, 28(4):118–130, 1998. 2.3

[103] Yufei Tao, Jimeng Sun, and Dimitris Papadias. Selectivity estimation for predictive spatio-temporal queries. In *ICDE*, 2003. 2.4, 3.1.1

[104] Nesime Tatbul, Ugur Centintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003. 2.2.1

[105] Ravikumar V.C. and Rabi N. Mahapatra. Tcam architecture for ip lookup using prefix properties. *IEEE Micro*, 24(2):60–69, 2004. 5.2

[106] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985. 3.1.1, 4.7

[107] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994. 2.3

[108] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical Report MIT/LCS/TM-528, MIT, 1995. 2.3

[109] M. Wang, J. S. Vitter, L. Lim, and S. Padmanabhan. Wavelet-based cost estimation for spatial queries. In *Proceedings of the SSTD conference*, 2001. 2.2.1

[110] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *USENIX Security Symposium*, 2004. 1.3, 4.2

[111] Richard West and Karsten Schwan. Dynamic window-constrained scheduling for multimedia applications. In *ICMCS, Vol. 2*, pages 87–91, 1999. 2.3

[112] Wikipedia. Port scanner — wikipedia, the free encyclopedia, 2006. [Online; accessed 21-October-2006]. 2

[113] Kesheng Wu, Ekow Otoo, and Arie Shoshani. A Performance Comparison of Bitmap Indices. In *CIKM*, 2001. 2.5

[114] Kesheng Wu, Ekow Otoo, and Arie Shoshani. On the Performance of Bitmap Indices for High Cardinality Attributes. In *VLDB*, 2004. 2.5, 6.3

[115] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. An Efficient Compression Scheme For Bitmap Indices. *ACM Transactions on Database Systems*, 31:1–38, 2006. 2.5, 6.3

Appendix A

Differential Relational Algebra

This appendix defines a set of differential operators that correspond to the relational algebra operators $\langle \sigma, \pi, \times, - \rangle$. The crux of these derivations is to capture the effects on relational algebra expressions when tuples disappear from their base relations.

A.1 Selection

I define the differential selection operator $\widehat{\sigma}(S_{noisy}, S_+, S_-) \equiv (R_{noisy}, R_+, R_-)$ based on the standard selection operator σ as follows:

$$\widehat{\sigma}(S_{noisy}, S_+, S_-) \equiv (\sigma(S_{noisy}), \sigma(S_+), \sigma(S_-)). \tag{A.1}$$

Intuitively, the differential selection operator simply applies the conventional selection operator to all three parts of the stream.

A.2 Projection

The definition of the differential projection operator is similar to that of the differential selection operator:

$$\widehat{\pi}(S_{noisy}, S_+, S_-) \equiv (\pi(S_{noisy}), \pi(S_+), \pi(S_-)). \tag{A.2}$$

A.2.1 Duplicate Elimination

It is important to note that the differential projection operator only works properly for multisets. This is not a problem for most queries, as multiset semantics are the default for both SQL and CQL. However, SELECT DISTINCT queries require an additional layer of post-processing to remove duplicates. A query in the form:

```
select distinct <columns> from <streams> where <predicates>
```

must be converted into the following before the rewrites in the remainder of this chapter can be applied:

```
select distinct
from (select <columns> from <streams> where <predicates>) as subquery
```

Then the inner subquery is rewritten, leaving the outer query to evaluate the DISTINCT clause At runtime, each copy of a tuple in the *additive noise* or *noisy* parts of the result "cancels out" a copy of the tuple in the *subtractive noise* component.

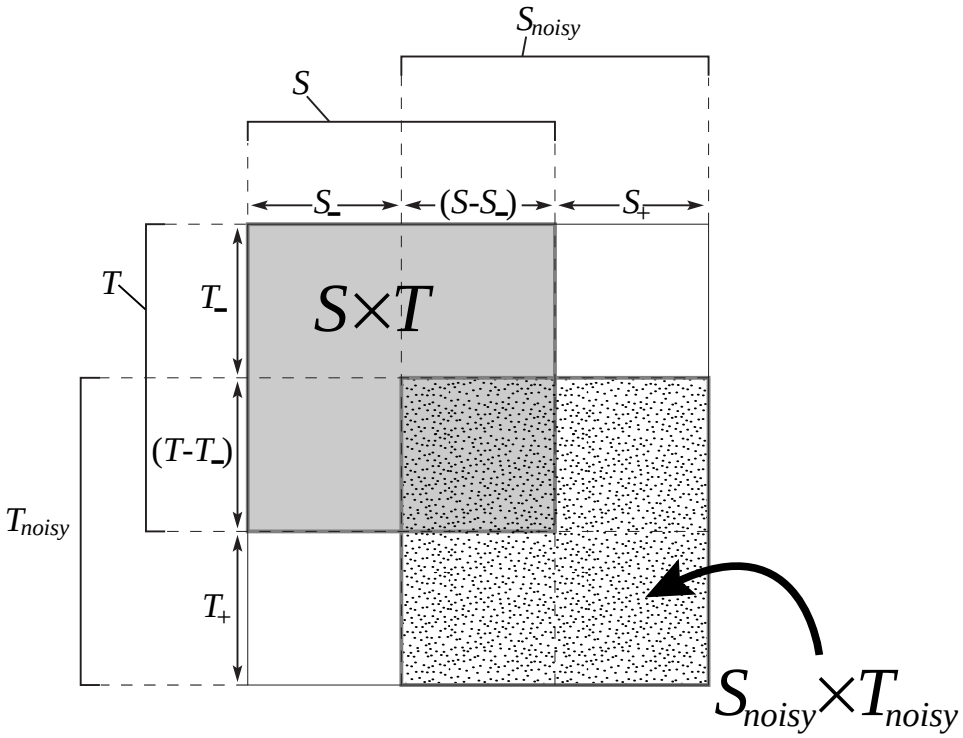

Figure A.1: Intuitive version of the differential cross product definition. The large square represents the tuples in the cross-product of $(S + S_+)$ with $(T + T_+)$, borrowing a visual metaphor from Ripple Join [46]. The smaller shaded square inside the large square represents the cross product of the original relations S and T. The speckled square represents the cross product of these two relations after the tuples in S_- and T_- are removed and the tuples in S_+ and T_- are added to S and T, respectively. The differential cross product computes the difference between these two cross products. Note that $S_{noisy} - S_+ = S - S_-$.

A.3 Cross Product

The definition of the differential cross-product operator is more complicated than the previous two operators.

Since

$$S_{noisy} \equiv S + S_+ - S_- \tag{A.3}$$

$$T_{noisy} \equiv T + T_+ - T_-, \tag{A.4}$$

for any relations S and T:

$$\begin{aligned} S \times T &= S_{noisy} \times T_{noisy} \\ &\quad - \left(S_+ \times T_+ + S_+ \times (T_{noisy} - T_+) + (S_{noisy} - S_+) \times T_+\right) \\ &\quad + \left(S_- \times T_- + S_- \times (T_{noisy} - T_+) + (S_{noisy} - S_+) \times T_-)\right). \end{aligned}$$

Figure A.1 shows the intuition behind the above formula.

Accordingly, I define the differential cross product operator $\widehat{\times}$ as

$$(S_{noisy}, S_+, S_-) \widehat{\times} (T_{noisy}, T_+, T_-) \equiv (R_{noisy}, R_+, R_-) \tag{A.5}$$

where

$$R_{noisy} = S_{noisy} \times T_{noisy}$$

and

$$R_+ = S_+ \times T_+ + S_+ \times (T_{noisy} - T_+) + (S_{noisy} - S_+) \times T_+$$

and

$$R_- = S_- \times T_- + S_- \times (T_{noisy} - T_+) + (S_{noisy} - S_+) \times T_-).$$

A.4 Join

The relational join operator is the composition of cross-product and selection. Similarly, the differential join operator can be derived from the differential selection and cross-product operators:

$$(S_{noisy}, S_+, S_-)\widehat{\bowtie}(T_{noisy}, T_+, T_-) \equiv \widehat{\sigma}\left((S_{noisy}, S_+, S_-)\widehat{\times}(T_{noisy}, T_+, T_-)\right) \tag{A.6}$$

Substituting in the definitions of the differential selection and cross-product operators produces the following definition:

$$(S_{noisy}, S_+, S_-)\widehat{\bowtie}(T_{noisy}, T_+, T_-) \equiv (R_{noisy}, R_+, R_-) \tag{A.7}$$

where

$$R_{noisy} = S_{noisy} \bowtie T_{noisy}$$

and

$$R_+ = S_+ \bowtie T_+ + S_+ \bowtie (T_{noisy} - T_+) + (S_{noisy} - S_+) \bowtie T_+$$

and

$$R_- = S_- \bowtie T_- + S_- \bowtie (T_{noisy} - T_+) + (S_{noisy} - S_+) \bowtie T_-).$$

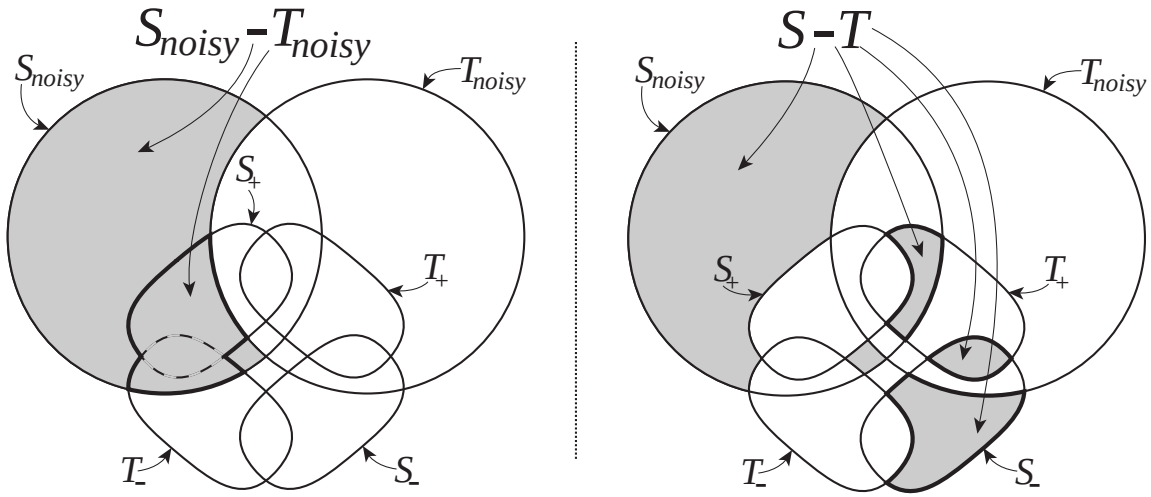

Figure A.2: Intuitive version of the definition of the differential set difference operator, $(S_{noisy}, S_+, S_-) \widehat{-} (T_{noisy}, T_+, T_-)$. This operator computes a delta between $(S_{noisy} - T_{noisy})$ and $(S - T)$, where S_{noisy} represents the tuples that remain in S after the query processor drops some of its input tuples. The shaded area in the Venn Diagram on the left contains the tuples in $(S_{noisy} - T_{noisy})$, and the shaded area in the diagram on the right contains the tuples in $(S - T)$. The shaded regions outlined in bold in one Venn diagram are not shaded in the other diagram.

A.5 Set Difference

The set difference operator has the interesting property that removing tuples from one of its inputs can *add* tuples to its output. I model this behavior by defining the differential set difference operator $\widehat{-}$ as

$$(S_{noisy}, S_+, S_-) \widehat{-} (T_{noisy}, T_+, T_-) \equiv (R_{noisy}, R_+, R_-), \tag{A.8}$$

where

$$R_{noisy} = S_{noisy} - T_{noisy}$$

and

$$R_+ = (S_+ - T_{noisy}) + \big((T_- - S_+) \cap S_{noisy}\big)$$

and

$$R_- = (S_+ \cap T_-) + \big((S_{noisy} \cap T_+) - S_+\big) + (S_- - T_- - T_{noisy})$$

Figure A.2 gives an intuition for this definition.