Declarative Network Monitoring with an Underprovisioned Query Processor (Extended Version)

Frederick Ralph Reiss
Joseph M. Hellerstein

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-38
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-38.html

April 10, 2006

Acknowledgement

This work was supported by an NDSEG fellowship, as well as by a grant from the National Science Foundation, award number IIS-0205647.

Declarative Network Monitoring with an Underprovisioned Query Processor (Extended Version)

Frederick Reiss
U.C. Berkeley

Joseph M. Hellerstein
U.C. Berkeley

Abstract

Many of the data sources used in stream query processing are known to exhibit bursty behavior. We focus here on passive network monitoring, an application in which the data rates typically exhibit a large peak-to-average ratio. Provisioning a stream query processor to handle peak rates in such a setting can be prohibitively expensive.

In this paper, we propose to solve this problem by provisioning the query processor for typical data rates instead of much higher peak data rates. To enable this strategy, we present mechanisms and policies for managing the tradeoffs between the latency and accuracy of query results when bursts exceed the steady-state capacity of the query processor.

We describe the current status of our implementation and present experimental results on a testbed network monitoring application to demonstrate the utility of our approach.

1 Introduction

Many of the emerging applications in stream query processing are known to exhibit high-speed, bursty data rates. The behavior of data streams in such applications is characterized by relatively long periods of calm, punctuated by "bursts" of high-speed data. The peak data rate exhibited in a burst is typically many times the average data rate.

In this paper, we focus on an application that is particularly prone to bursty data: passive network monitoring. A passive network monitor (See Figure 1) is a device attached to a high-traffic network link that monitors and analyzes the packets on the link.

There has been significant interest in bringing the power of declarative queries to passive network monitors [8, 16]. Declarative queries are easy to change in response to the evolution of networked applications, protocols and attacks, and they free network operators from the drudgery of hand-optimizing their monitoring code.

However, high-speed bursts and tight time constraints make implementing declarative query processing for network monitoring a difficult problem. The bursty nature of network traffic is well-documented in the literature [21, 25]. Situations like SYN floods can multiply the effects of bursts by increasing bandwidth usage and decreasing packet size simultaneously. Under such situations, even keeping simple counters during bursts is considered difficult [11]. Bursts often produce not only *more* data, but also *different* data than usual. This will often be the case, for example, in crisis scenarios, such as a denial of service attack or a flash crowd. Because network operators need real-time data on the status of their networks, network monitors need to provide timely answers under bursty load without dropping packets.

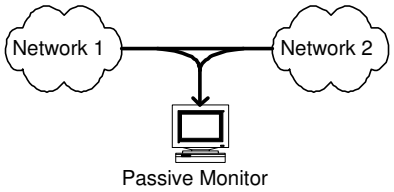

Figure 1: A typical passive network monitoring setup.

Many researchers have attempted to bring declarative queries to network monitoring by *scaling up the query engine* to the network's peak rate through various approaches. Some systems accelerate query processing with custom, reconfigurable, or parallel hardware [29]. Other systems construct queries by connecting blocks of highly-optimized special-purpose code [8]. Additionally, researchers have developed algorithms and data structures for computing approximate answers in small amounts of time and space [13, 5, 7].

Realistically, making query processing scale to "line speed" involves compromises. Custom hardware is expensive; specialized code limits the flexibility of the query language; approximate query answers are inferior to exact answers.

In this paper, we navigate these compromises with a different approach. We do *not* scale our query processor to the network's maximum speed. Instead, we use a query processor that can provide timely, exact answers under the *typical* load on the network connection. This normal load can be an order of magnitude lower than the maximum load.

Of course, bursts will exceed the steady-state capacity of a system that is configured in such a way. This overload can lead to increased latency of query results, if the system

buffers excess data; or decreased query result accuracy, if the system discards excess data. The key to provisioning for typical data rates is to manage the overload behavior of the system.

To meet this goal, we allow the user to specify *delay constraints* that bound the latency of query results. Our system manages its buffering to ensure that it satisfies the user's delay constraints. When it is impossible to meet a delay constraint while processing all input data, we apply approximate query processing techniques to minimize the impact on query result accuracy. To enable this tight integration of buffer management and approximation without extensive modifications to our query engine, we have developed an architecture that we call Data Triage.

2 Technical Contributions

The remaining sections of this paper describe our solutions to the technical challenges of provisioning for typical network load. From a query processing standpoint, the main contributions of this paper are as follows:

- Taking advantage of the bursty nature of network traffic to reduce the cost of provisioning a declarative network monitor (Section 5)
- Using windowed delay constraints to drive adaptive load shedding (Section 6)
- The Data Triage architecture (Section 8)
- Implementation of approximation and Data Triage without modifications to the core query engine (Sections 8 and 9)
- Stream query processing experiments with timing-accurate network traces (Section 11)

3 Related Work

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 [32]. Our work differs from this approach in two ways: First of all, we use fixed end-to-end delay constraints, whereas Aurora's *drop* operators minimize a local cost function given the resources available. Secondly, our 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 [9, 18, 30]. Our work is complementary to these approaches. In this paper, we do not focus on choosing "victim" tuples in the event of overflow; rather, we 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 [4, 20]. Load shedding systems that use this approach lossily compress sets of tuples and perform query processing on the compressed sets. The STREAM data manager [12] 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; we focus here on provisioning a system to handle the steady state well and to degrade gracefully when bursts lead to temporary overload.

Real-time databases focus on providing answers to static queries within tight time bounds. CASE-DB [24] is a real-time database system that meets its real-time constraints by giving approximate answers to queries. Our windowed delay constraints differ from this previous work in that we maintain timing constraints continuously over unpredictable streaming data.

An overview of ongoing work on Data Triage appeared as a short paper in ICDE 2005 [27].

4 Query Model

The work in this paper uses the query model of the current development version of TelegraphCQ [6]. Queries in TelegraphCQ are expressed in CQL [2], 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 in Figure 2.

```
-- 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);
```

Figure 2: Schema for the queries in Figure 3.

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 paper, we focus on 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.

TelegraphCQ can combine multiple `SELECT` statements by using a variant of the SQL99 `WITH` construct. The im-

plementation of the `WITH` clause in TelegraphCQ supports recursive queries, but we do not consider recursion in this paper.

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. For ease of exposition, we limit ourselves in this paper to the case where `RANGE` and `SLIDE` are equal; that is, to hopping windows. We briefly discuss the extension of our work to general sliding windows in Section 8.3.

Figure 3 gives several example network monitoring queries that demonstrate the utility of this query model. We will use the first of these queries as a running example throughout the paper.

```
-- 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(*)
        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);
```

Figure 3: Sample queries in TelegraphCQ CQL

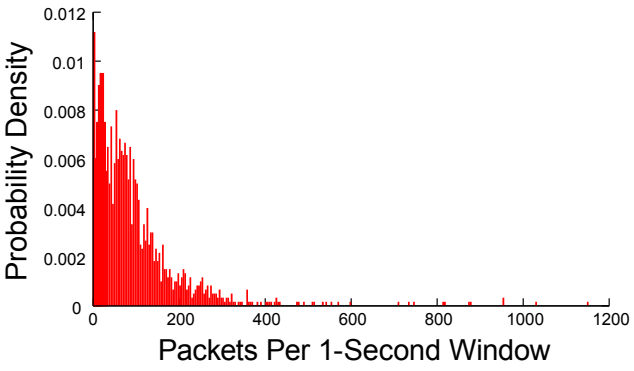

Figure 4: Distribution of packet rates in a trace of HTTP traffic from `www.lbl.gov`.

5 Data Rates in Passive Network Monitoring

Having provided a brief overview of the necessary background material, we now turn to the main topic of this paper: Taking advantage of bursty network traffic to reduce the cost of monitoring networks with a stream query processor.

Analyses of network traffic from a wide variety of sources have shown them to exhibit self-similar, bursty behavior along a number of dimensions [21, 25]. In this paper, we focus on one of these parameters, the *rate of packet arrival*. Packet arrival rates tend to follow heavy-tailed distributions, with high-speed bursts occurring over a broad range of time scales [28]. For example, Figure 4 shows the distribution of packet arrival rates in a trace from the web server `www.lbl.gov`.

Network monitoring systems are typically provisioned with sufficient CPU capacity to handle the maximum packet arrival rate on the network connection. Because of the heavy-tailed distribution of packet arrival rates, this maximum rate tends to be significantly higher than the vast majority of network traffic.

In this paper, we advocate provisioning a network monitor to handle the *90th to 95th percentile* of traffic. This approach allows significant reductions in the CPU requirements of the monitor, while enabling the monitor to process all data most of the time.

For example, Figure 5 shows the ratio between the 80th through 100th percentiles of the packet arrival rate distribution in Figure 4. This analysis is conservative, in that we assume that the maximum rate observed in our trace is the maximum rate the connection can sustain. Assuming that the network monitor's CPU requirements are proportional to packet arrival rate, provisioning for the 95th percentile of packet arrival rate would reduce CPU requirements for monitoring this network connection by 76 percent.

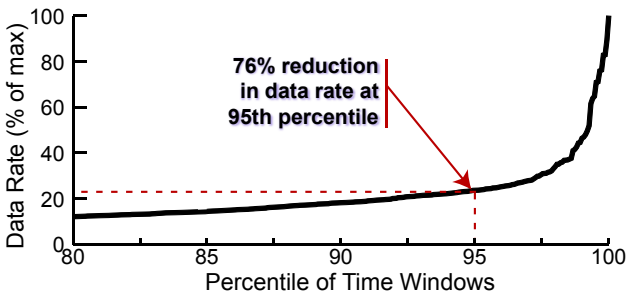

Figure 5: The ratio between the maximum data rate in the trace in 4 and the 80th to 100th percentile packet arrival rates. Provisioning a network monitor for the 95th percentile of this distribution reduces CPU requirements by 76 percent.

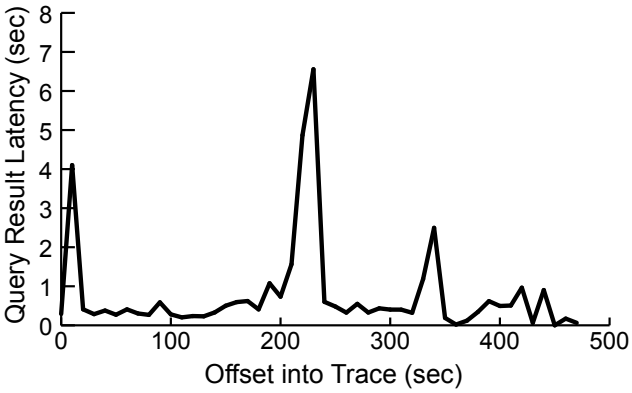

Figure 6: Query result latency for a simple aggregation query over a 10-second time window. The query processor is provisioned for the 90th percentile of packet arrival rates. Data is a trace of a web server's network traffic.

5.1 Microbenchmark Analysis

The analysis in the previous section is encouraging, but its naive application is not very effective. Figure 6 shows the results of an experiment to measure the actual latency of query results.

The data source for this experiment was the packet trace used in the previous section. We played back this trace through TelegraphCQ, using a schema similar to that in Figure 2. We configured TelegraphCQ to run the query:

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

To simulate a less-powerful machine, we 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, our query processor was provisioned for the 90th percentile of packet arrival rates. The graph shows the observed latency between query result generation and the end of each time window.

The results of this experiment demonstrate that naively provisioning the query processor for "typical" data rate can cause unacceptable increases in latency. The largest delays occur because the query processor does not follow the second assumption we made in the previous section: that processing the data from a given window does not interfere with processing of subsequent windows. Longer bursts cause cumulative increases in latency across time windows.

In the sections that follow, we propose the use of *delay constraints* to codify acceptable delay and the Data Triage architecture to ensure that a query processor meets its delay constraints.

6 Delay Constraints

While it has the potential to reduce significantly the cost of network monitoring hardware, the strategy of provisioning for typical load leads to tradeoffs with query result latency. Navigating these tradeoffs in a principled way requires a way for the user to specify what constitutes acceptable latency for a given query.

A *delay constraint* is a user-defined bound on the latency between data arrival and query result generation. Figure 7 shows an example our proposed syntax for delay constraints. Our modification to CQL adds the optional clause

```
LIMIT DELAY TO [interval]
```

to the end of the `SELECT` statement.

```
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 7: Sample query with a delay constraint

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 we 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 we have studied contain windowed `GROUP BY` and aggregation, so we concentrate here on delay constraints for windowed queries.

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 1: Variables used in Sections 6 through 8

Table 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; we 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

$$\text{deadline}(t) = \text{end}(w_t) + D - C_{tup} \quad (1)$$

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.

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

7 Satisfying Delay Constraints

In the previous section, we introduced the concept of delay constraints as a way for the application to specify its tolerance for query result latency.

The overall goal of our work is to satisfy delay constraints with a query processor that is provisioned for the 90th to 95th percentile of its data rates. To handle bursts, such a query processor needs a mechanism for trading off the accuracy of query results against increased data processing speed. To provide such a mechanism, we leverage the extensive previous work in approximate query processing.

7.1 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 [33, 1], multidimensional histograms [26, 17, 10, 31], and wavelet-based histograms [5, 22].

The focus of this paper is not to develop new methods of approximate query processing. Instead, we leverage previous work to manage latency and accuracy with bursty data streams. Because no single summarization method has been shown to dominate all others, we have developed and implemented a framework that allows us to employ a broad spectrum of techniques.

Our framework divides a given summarization 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 computes aggregate values or generates a representative set of tuples from a summary.

These primitives allow one to approximate continuous queries of the type described in Section 4. 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.

With the current state-of-the-art in summarization techniques, the tuning parameters need to be set prior to the creation of the summary. Of course, a query processor that employs approximation to meet delay constraints cannot predict whether a burst will occur in the current time window. Such a system must tune its summaries to the maximum data rate that could possibly occur.

Unfortunately, the values of the tuning parameters that enable such a high data rate will result in relatively high approximation errors. The summarization technique will overcompress the present data, because it cannot predict the future behavior of the stream.

Our solution to this problem is to use approximation as a "safety valve" rather than a primary data path. Our system

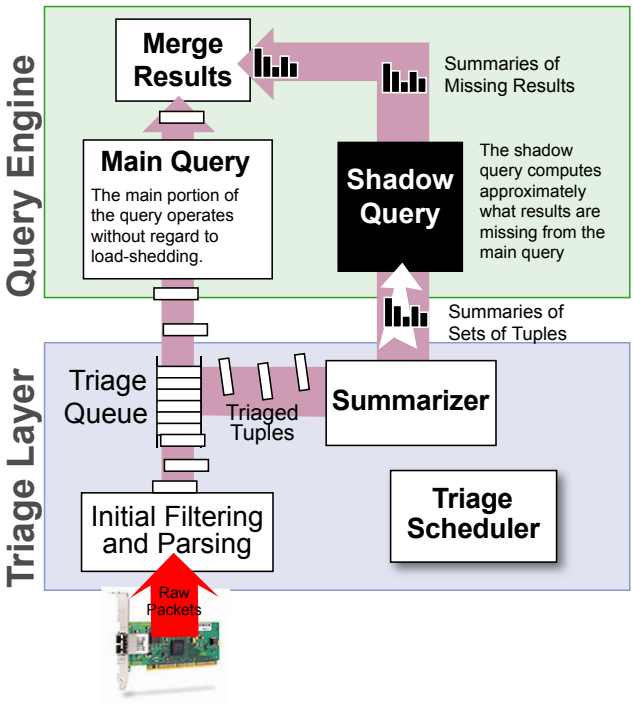

Figure 8: An overview of the Data Triage architecture. Data Triage acts as a middleware layer within the network monitor, isolating the real-time components from the best-effort query processor to maintain end-to-end responsiveness.

maintains two data paths: one that uses normal query processng and one that uses approximation. The system only sends tuples through the approximate data path when processing those tuples fully would violate the delay constraint.

8 Data Triage

In the previous section, we motivated our approach of using approximation as a fallback mechanism to satisfy delay constraints. Implementing this approach without major modifications to the query engine leads to an architecture that we call Data Triage.

Figure 8 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 and produces streams of relational tuples containing detailed information about the packet stream. Our current implementation of this layer is based on the low-level packet-processing routines of the Bro intrusion detection system [11].
- The *Main Query* operates inside a stream query processor, in our case TelegraphCQ. The user-specified query consumes tuples from the initial parsing and filtering layer. Architecturally, the key characteristic of the main 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 some data to the approximate data path 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 a fast but approximate alternate datapath. We give a detailed description of our scheduling algorithms and delay constraints in Section 8.1.
- 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.

We have implemented Data Triage in the TelegraphCQ stream query processor. In the process of doing so, we overcame several challenges. The sections that follow describe the approaches we used to construct simple and efficient solutions to these problems:

- Choosing which tuples to *triage* and when to triage them (Section 8.1)
- Constructing approximate shadow queries efficiently (Sections 8.2 and 9)
- Determining the maximum data rate that Data Triage can support with a given approximation technique (Section 8.4).

8.1 The Triage Scheduler

The Triage Scheduler is the control component of Data Triage, with control over the Triage Queue and the Summarizer. The scheduler's primary purpose is to ensure that the system meets the 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
- When to transfer the current summary from the Summarizer to the shadow query.

Sending tuples to the summarizer supports significantly higher data rates, but compression operations do not have

nonzero cost. As we will show in our 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 we have: $C_{tup} = \begin{cases} C_{full} & \text{(for main query)} \\ C_{sum} & \text{(for shadow query)} \end{cases}$

The Triage Scheduler also needs to account for the cost of sending summaries through the shadow query. We let C_{shadow} denote the cost per time window of the shadow query, including the cost of merging query results. We assume that C_{shadow} is constant regardless of the number of tuples triaged. We revisit this assumption in Section **??**.

C_{shadow} is a *per-window* cost. We 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 6, we obtain the new equation:

$$\text{deadline}(w) = \text{end}(w) + D - C_{shadow} - \begin{cases} C_{full} & \text{(for main query)} \\ C_{sum} & \text{(for shadow query)} \end{cases} \quad (2)$$

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.

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

$$\text{Time to process remaining tuples in window} \leq \text{Time before delay constraint violated}. \quad (3)$$

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:

$$nC_{full} \leq W + D - C_{shadow} - O, \quad (4)$$

or equivalently

$$n \leq \left(\frac{W + D - C_{shadow}}{C_{full}} \right) - \frac{O}{C_{full}}. \quad (5)$$

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. We note that the Scheduler must maintain the invariant simultaneously for *all* windows whose tuples could be in the Triage Queue.

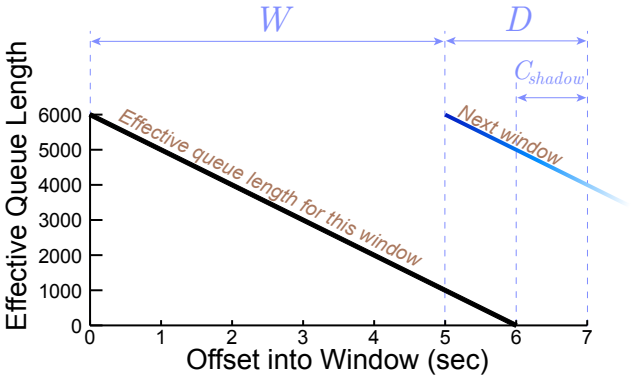

Figure 9: The effective length of the Triage Queue for tuples belonging to a 5-second time window, as a function of offset into the window. The delay constraint is 2 seconds, and C_{shadow} is 1 second.

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 our philosophy of using approximation as a fallback mechanism, our 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 9 illustrates this variation in effective queue length.

8.2 Approximate Query Processing Framework

Another challenge of implementing Data Triage is adding the approximate query processing components to the query engine without rearchitecting the system. We have met this challenge by developing a common framework for different types of approximate query processing and mapping this framework 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 a summary scheme is the summarization data structure, which provides a compact, lossy representation of a set of relational tuples. We used the user-defined datatype functionality of TelegraphCQ to implement several types of summary data structure, 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. One way to implement these is as functions that take a set of tuples as an argument and return a summary. To avoid the space overhead of storing large sets of tuples, our implementation takes in a stream of tuples and incrementally adds them to the summary.

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. For example, some of the operators for the MHIST multidimensional histogram type [26] are as follows:

```
-- 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 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. We 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 ...
```

Having implemented the components of a given summarization scheme, we can construct Data Triage's shadow queries and merging logic using query rewriting. Section 9 demonstrates this process on a simple example query.

8.3 General Sliding Windows

The previous sections have described our implementation of Data Triage 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 paper can accomodate arbitrary combinations of TelegraphCQ window clauses with the following changes:

- Use the method described in [19] 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 this section, we explain the first and last of these steps in greater detail.

8.3.1 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, we would like 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. 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.

8.3.2 Using aggregates to merge summaries

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

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

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

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

8.4 Provisioning Data Triage

Data Triage uses approximate query processing as a fallback mechanism to ensure that an underprovisioned query processor can meet a delay constraint. The effectiveness

of this approach of course depends on the summary implementation being faster than the general-purpose query processor.

In particular, we would like to know:

- If the approximation method in Data Triage's shadow query performs at a given level, what degree of underprovisioning will Data Triage permit?
- How quickly can different query approximation methods process data?

In this section, we address both of these questions. We start with a theoretical analysis of approximate query processing performance as it applies to Data Triage, then we apply our theory to an experimental analysis of several approximation methodologies.

As in previous sections, our analysis assumes that the time windows in the user's query are hopping windows.

8.4.1 Data Triage and System Capacity

An important design goal of Data Triage is what we call the *Do-no-harm principle*:

> *If the system has time to process all the tuples in a time window fully, it should do so.*

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

We characterize the CPU cost of a summarization/approximation scheme by two parameters, C_{shadow} and C_{sum}. We 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 paragraphs that follow, we derive the relationship between the summarization parameters, C_{shadow} and C_{sum}, and the system capacity parameters, R_{exact} and R_{peak}.

8.4.2 C_{shadow}

Recall that 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

$$\frac{W - C_{shadow}}{C_{full}}. \tag{6}$$

Effectively, R_{exact} is reduced 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.

8.4.3 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, we have

$$R_{peak} = \left(1 - \frac{C_{shadow}}{W}\right)\left(\frac{1}{C_{sum}} - \left(\left\lfloor \frac{W}{D} \right\rfloor + 1\right)\frac{1}{C_{full}}\right) \tag{7}$$

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

In summary, the ratio between C_{sum} and C_{full} (the time to process a tuple in the main query) acts as a bound on the cost savings through underprovisioning.

8.5 Performance Analysis of Approximation Techniques

We have implemented several summary types within the framework we described earlier in this paper:

- Multidimensional histograms with a fixed grid of buckets
- MHIST multidimensional histograms [26]
- Wavelet-based histograms [23]
- Reservoir sampling [33]

All of the approximation schemes we 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.

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

8.5.1 Measuring C_{sum}

Our 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. We measured the insertion cost across a range of summary granularities.

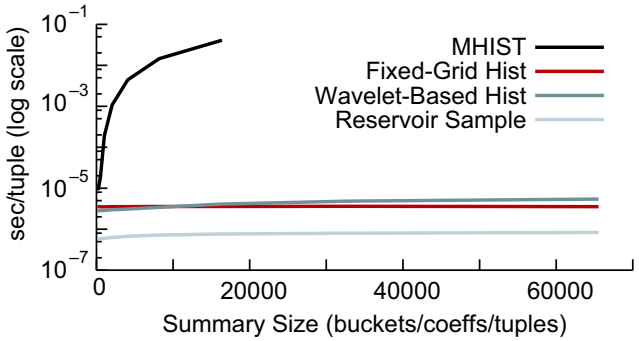

Figure 10: The CPU cost of inserting a tuple into the four types of summary we implemented. The X axis represents the granularity of the summary data structure.

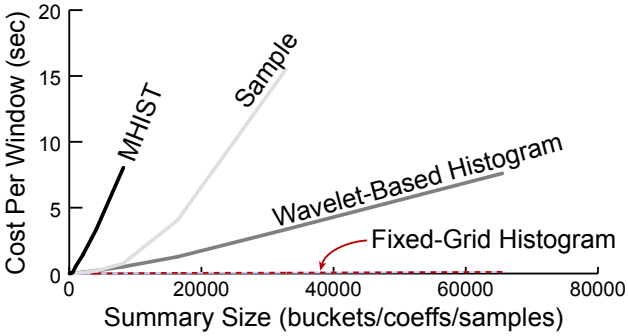

Figure 11: The time required to compute a single window of a shadow query using four kinds of summary data structure. The X axis represents the granularity of the summaries; the Y axis represents execution time.

Figure 10 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 extremely low, though it did increase somewhat at larger sample sizes, probably due to caching effects.

Fixed-grid histograms provided low insertion times across a wide 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 we 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, insertion into our MHIST implementation would slower than normal tuple processing. The high insertion cost stems mostly from the lack of an index to the MHIST buckets. Using a kd-tree [3] to map tuples to buckets would rectify this problem. Even with these optimizations, our MHIST implementation would still have a higher insertion cost than the other summaries, as evidenced by the leftmost point on the curve.

8.6 Measuring C_{shadow}

Our 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 stream-table join query.

Figure 11 shows our results. 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.

We also conducted a version of this experiment in which we 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. We omit detailed results due to space constraints.

8.6.1 Discussion

Our evaluation of the four approximation schemes we have implemented shows that three of them can summarize tuples fast enough to be useful for Data Triage. In the current version of TelegraphCQ, 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. We expect these times to drop significantly as we optimize our code.

Our 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.

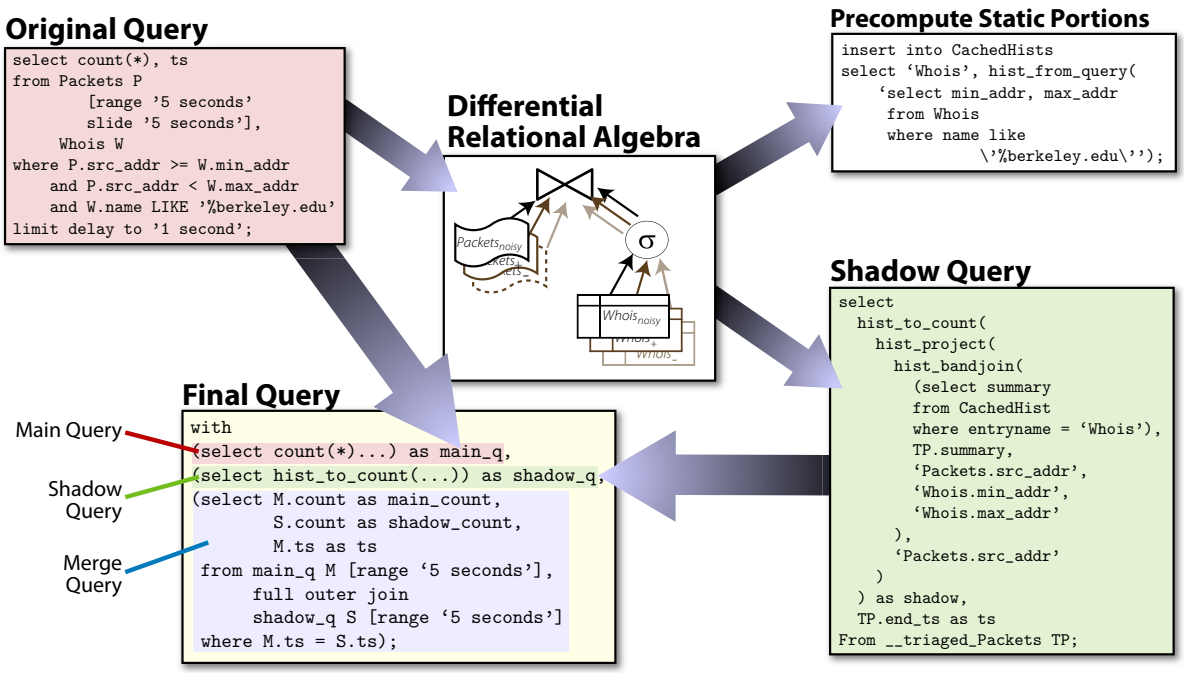

Figure 12: An illustration of the Data Triage query rewrite algorithm as applied to an example query. The algorithm produces a main query, a shadow query, and auxiliary glue queries to merge their results. This example uses multidimensional histograms as a summary datatype.

9 Lifetime of a Query

To illustrate the methods we use to construct shadow queries and how these methods interact with our implementation of Data Triage, we will now describe the query rewrite and execution process as it applies to the query in Figure 7. The query reports the number of packets coming from Berkeley domains every 5 seconds.

9.1 Summary Streams

Recall the sample CQL schema from Section 4. 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. We plan to allow the user to choose the summary type at query execution time in a future version of TelegraphCQ.

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. [1]

9.2 Query Rewrite

Figure 12 shows the query rewrite process as applied to our sample query. Our query rewriting methodology is based on an algebraic formalism that allows us to correctly rewrite queries into shadow queries. We use relational algebra to build a set of *differential relational algebra operators*. Our approach here resembles past work in maintaining materialized views [14], though our setting is different.

Briefly, each differential operator propagates changes from the inputs to the outputs of the corresponding relational algebra operator. The differential relational algebra divides each relation S into *noisy,* additive noise, and *sub-*

[1] Our system also creates a second summary stream that summarizes *non*-triaged tuples. This second stream is only necessary for queries with stream-stream joins and is only activated for such queries.

tractive noise components[2] S_{noisy}, S_+ and S_-, such that:

$$S_{noisy} \equiv S + S_+ - S_- \tag{8}$$

where $+$ and $-$ are the multiset union and multiset difference operators, respectively.

Our 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 we described in Section 8.2 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 user submits this rewritten query to the query engine, which begins executing the main and shadow queries.

9.3 Query Execution

When the rewritten query enters the system, the TelegraphCQ engine passes the delay constraint and window size to the Triage Scheduler. 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 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 user can add these two counts to obtain an estimate of the true query results. Keeping the results of the main and shadow queries separate provides feedback as to how much approximation went into the overall result.

10 Differential Relational Algebra

In this section, we define 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.

10.1 Selection

We create the differential selection operator $\widehat{\sigma}(S_{noisy}, S_+, S_-) \equiv (R_{noisy}, R_+, R_-)$ from the standard selection operator σ as follows:

$$\widehat{\sigma}(S_{noisy}, S_+, S_-) \equiv (\sigma(S_{noisy}), \sigma(S_+), \sigma(S_-)). \tag{9}$$

Intuitively, the differential selection operator simply applies traditional selection to all three channels.

10.2 Projection

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

$$\widehat{\pi}(S_{noisy}, S_+, S_-) \equiv (\pi(S_{noisy}), \pi(S_+), \pi(S_-)). \tag{10}$$

It is important to note that the differential projection operator only works properly for multisets. The problem of handling these `SELECT DISTINCT` queries is one that we are currently investigating.

10.3 Cross Product

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

Since

$$S_{noisy} \equiv S + S_+ - S_- \tag{11}$$
$$T_{noisy} \equiv T + T_+ - T_-, \tag{12}$$

for any relations S and T, we have

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

Figure 13 shows the intuition behind the above formula.

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

$$(S_{noisy}, S_+, S_-) \widehat{\times} (T_{noisy}, T_+, T_-) \equiv (R_{noisy}, R_+, R_-) \tag{13}$$

where

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

and

[2] The *additive noise* component is necessary because operations like negation and set difference can cause additional tuples to appear in an expression's output when tuples are removed from its inputs.

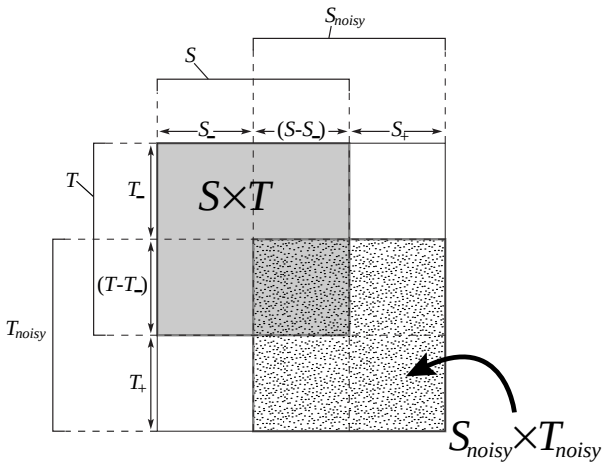

Figure 13: Intuitive version of the differential cross product definition. The large square represents the tuples in the cross-product of $(S+S_+)$ with $(T+T_+)$, borrowing a visual metaphor from Ripple Join [15]. The smaller shaded square inside the large square represents the cross product of the 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 $S_{noisy} - S_+ = S - S_-$.

$$\begin{aligned} R_+ &= S_+ \times T_+ \\ &\quad + S_+ \times (T_{noisy} - T_+) \\ &\quad + (S_{noisy} - S_+) \times T_+ \end{aligned}$$

and

$$\begin{aligned} R_- &= S_- \times T_- \\ &\quad + S_- \times (T_{noisy} - T_+) \\ &\quad + (S_{noisy} - S_+) \times T_-). \end{aligned}$$

10.4 Join

The derivation of the differential join operator follows the same line of reasoning as that of the differential cross product operator and produces essentially the same definition. For brevity, we omit this derivation.

10.5 Set Difference

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

$$(S_{noisy}, S_+, S_-) \widehat{-} (T_{noisy}, T_+, T_-) \equiv (R_{noisy}, R_+, R_-), \quad (14)$$

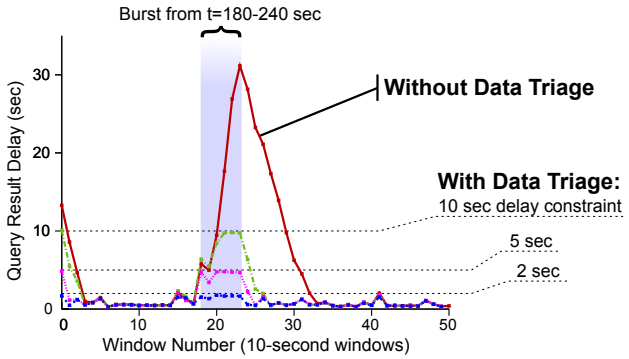

Figure 15: A comparison of query result latency with and without Data Triage on with the system provisioned for the 90th percentile of load. The data stream was a timing-accurate trace of a web server. Each line is the average of 10 runs fo the experiment.

where

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

and

$$\begin{aligned} R_+ &= (S_+ - T_{noisy}) \\ &\quad + ((T_- - S_+) \cap S_{noisy}) \end{aligned}$$

and

$$\begin{aligned} R_- &= (S_+ \cap T_-) \\ &\quad + ((S_{noisy} \cap T_+) - S_+) \\ &\quad + (S_- - T_- - T_{noisy}) \end{aligned}$$

Figure 14 gives an intuition for this defintion.

11 Experimental Validation

We conducted experiments on our prototype implementation of Data Triage to measure how well it satisfied delay constraints in a realistic environment. We used a 105-MB trace of the traffic to and from the HTTP server `www.lbl.gov` as the input to our experiments.

The query used in the experiments was a variant of the example query from earlier in the paper. The current implementation of band joins in TelegraphCQ is inefficient, so we modified the query to be an equijoin on the most significant 16 bits of the IP address instead.

We ran our 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, we 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. We used reservoir samples as the approximation method for this experiment. We adjusted the trace playback rate to 10 times the original rate. At this data rate, our system was provisioned for the 90th percentile of packet arrival rates in our trace.

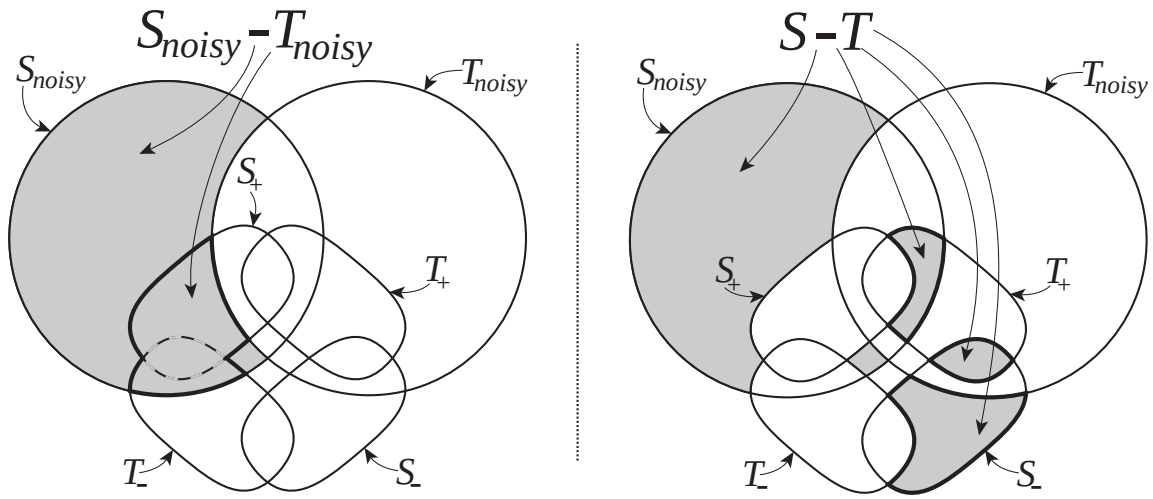

Figure 14: Intuitive version of the definition of the differential set difference operator, $(S_{noisy}, S_+, S_-) \widehat{-} (T_{noisy}, T_+, T_-)$. This operator computes a delta between $(S_{noisy} - T_{noisy})$ and $(S - T)$, where S_{noisy} represents the tuples that remain in S after the query 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.

11.1 Latency

For our first experiment, we ran the query both with and without Data Triage and measured the latency of query results. We determined 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. We repeated the experiment 10 times and recorded the average latency for each time window.

Figure 15 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 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.

11.2 Result Accuracy

Our second experiment measured the accuracy of query results with three methods of satisfying a 2-second delay constraint. We measured result error using a root-mean-squared error metric. That is, we defined the error for time window w as:

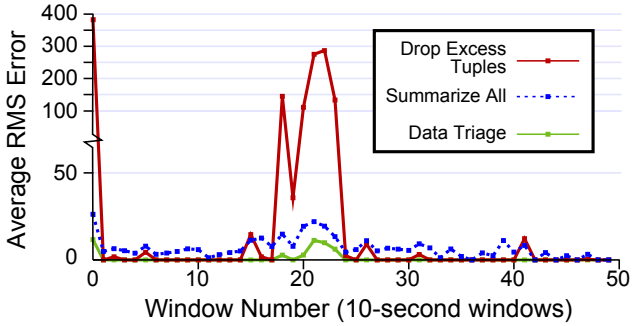

Figure 16: A comparison of query result accuracy using the same experimental setup as in Figure 15 and a 2-second delay constraint. Data Triage outperformed the other two load-shedding methods tested. Each line is the average of 10 runs of the experiment.

$$E_w = \sqrt{\frac{\sum_{g \in \text{groups}} (\text{actual}(g) - \text{reported}(g))^2}{|\text{groups}|}} \qquad (15)$$

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

- **Data Triage** as described in this paper

- **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.

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

Figure 16 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.

12 Conclusion

In this paper, we observed that there are significant potential cost savings to provisioning a network monitor for typical data rates as opposed to the maximum load. We found that controlling the tradeoff between provisioning and latency is key to enabling these cost savings. We described windowed delay constraints, a way of specifying tolerances for query result latency, and the Data Triage architecture that we use to implement delay constraints. We presented a theoretical analysis of scheduling and provisioning for Data Triage, as well as a formally-based, practical query rewrite scheme that allows us to implement Data Triage without modifying the core of the query engine. Finally, we used our implementation to perform experiments that demonstrate the Data Triage can satisfy windowed delay constraints on an underprovisioned stream query processor.

Acknowledgments

This work was supported by an NDSEG fellowship, as well as by a grant from the National Science Foundation, award number IIS-0205647.

We would like to thank Yang Zhang for his help in implementing wavelet-based histograms in TelegraphCQ. We would also like to thank Boon Thao Loo, Ryan Huebsch, Yanlei Diao, and Shariq Rizvi for helpful comments on early drafts of this paper.

References

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286, 1999.

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

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

[4] A. Bulut and A. K. Singh. Swat: Hierarchical stream summarization in large networks. In *ICDE*, 2003.

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

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[7] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[8] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, 2003.

[9] A. Das, J. E. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*. ACM Press, 2003.

[10] A. Deshpande and J. Hellerstein. On using correlation-based synopses during query optimization. Technical Report CSD-02-1173, U.C. Berkeley EECS Department, May 2002.

[11] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *CCS*, pages 2–11. ACM Press, 2004.

[12] R. M. *et al*. Query processing, resource management, and approximation in a data stream management system. In *CIDR*. ACM Press, 2003.

[13] M. Garofalakis and P. Gibbons. Approximate query processing: Taming the terabytes!, 2001.

[14] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[15] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD*, pages 287–298. ACM Press, 1999.

[16] G. Iannaccone, C. Diot, and D. McAuley. The CoMo white paper. Technical Report IRC-TR-04-017, Intel Research, September 2004.

[17] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 24–27 1998.

[18] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

[19] S. Krishnamurthy and M. J. Franklin. Shared hierarchical aggregation over receptor streams. Technical Report UCB-CSD-05-1381, UC Berkeley, 2005.

[20] I. Lazaridis and S. Mehrotra. Capturing sensor-generated time series with quality guarantees. In *ICDE*, 2003.

[21] W. E. Leland, M. S. Taqq, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic. In D. P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, 1993.

[22] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD*, pages 448–459, 1998.

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

[24] G. Ozsoyoglu, S. Guruswamy, K. Du, and W.-C. Hou. Time-constrained query processing in CASE-DB. *Knowledge and Data Engineering*, 7(6):865–884, 1995.

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

[26] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *CIDR*. ACM Press, 2003.

[27] F. Reiss and J. M. Hellerstein. Data triage: An adaptive architecture for load shedding in telegraphcq (short paper). In *ICDE*, 2005.

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

[29] D. V. Schuehler and J. Lockwood. TCP-Splitter: A TCP/IP flow monitor in reconfigurable hardware. In *Hot Interconnects*, pages 127–131, Stanford, CA, Aug. 2002.

[30] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins, 2004.

[31] Y. Tao, J. Sun, and D. Papadias. Selectivity estimation for predictive spatio-temporal queries. In *ICDE*, 2003.

[32] N. Tatbul, U. Centintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.

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