Adaptive Filters for Continuous Queries over Distributed Data Streams*

Chris Olston, Jing Jiang, and Jennifer Widom
Stanford University
{olston, jjiang, widom}@cs.stanford.edu

Abstract

We consider an environment where distributed data sources continuously stream updates to a centralized processor that monitors continuous queries over the distributed data. Significant communication overhead is incurred in the presence of rapid update streams, and we propose a new technique for reducing the overhead. Users register continuous queries with precision requirements at the central stream processor, which installs filters at remote data sources. The filters adapt to changing conditions to minimize stream rates while guaranteeing that all continuous queries still receive the updates necessary to provide answers of adequate precision at all times. Our approach enables applications to trade precision for communication overhead at a fine granularity by individually adjusting the precision constraints of continuous queries over streams in a multi-query workload. Through experiments performed on synthetic data simulations and a real network monitoring implementation, we demonstrate the effectiveness of our approach in achieving low communication overhead compared with alternate approaches.

1 Introduction

Query processing over *continuous data streams* has received considerable attention recently, *e.g.*, [CCC+02, MF02, MWA+03]. We consider distributed environments in which remote data sources continuously push updates to a central *stream processor*, whose job is to evaluate multiple *continuous queries* over the streamed data [BW01, CDTW00, LPT99, MSHR02]. In these environments, significant communication overhead is incurred in the presence of rapid update streams. We offer an effective method for reducing communication cost, taking advantage of the fact that many applications do not require exact precision for their continuous queries—examples are discussed shortly. When applications do not require exact precision and data values do not fluctuate wildly, approximate answers of sufficient precision usually can be computed from a small fraction of the input streams. In our approach, users submit quantitative *precision constraints* along with continuous queries to the stream processor, and the stream processor installs *filters* at the remote data sources. The filters adapt to changing conditions to minimize communication cost while guaranteeing that all continuous queries still receive the updates necessary to provide answers of adequate precision at all times. In this way, users are offered fine-grained control over the tradeoff between query answer precision and communication cost. Imprecision of query results is bounded numerically so applications need not deal with any uncertainty.

Many stream-oriented applications do not need exact answers, yet require quantitative guarantees regarding the precision of approximate answers [YV00]. For example, consider wireless sensor networks, *e.g.*, [EGPS01, KKP99, MF02, PK00], which enable continuous monitoring of environmental conditions such as light, temperature, sound, vibration, structural strain, etc. [MSHR02]. Since the battery life of miniature sensors is severely limited, and radio usage is the dominant factor determining battery life [MBC+01, PK00], it is crucial to reduce the amount of data transmitted, even if a small increase in local processing by the sensor is required [MF02]. Many applications that rely on sensor data can tolerate approximate answers having a controlled degree of imprecision [MBC+01], making our approach ideal for reducing data transmission. Other examples with continuous queries over distributed data that can tolerate a bounded amount of imprecision include industrial process monitoring, stock quote services, online auctions, wide-area resource accounting, and load balancing for replicated servers [SBS+02, YV00].

*This work was supported by the National Science Foundation under grants IIS-0118173 and IIS-9817799 and by a National Science Foundation graduate research fellowship.

Next we focus on one particular application, network monitoring, and give examples of continuous queries that arise in the context of that application to motivate our work. Then in the remainder of Section 1 we provide an overview of our approach.

1.1 Example Application: Network Monitoring

Managing complex computer networks requires tools that, among other things, continually report the status of network elements in real time, for applications such as traffic engineering, reliability, billing, and security, *e.g.*, [DR01, vRB01]. Network monitoring applications do not typically require absolute precision [vRB01]. Thus, our approach can be used to reduce monitoring communication overhead between distributed network elements and a central monitoring station, while still providing quantitative precision guarantees for the approximate answers reported.

Real-time network monitoring workloads often consist of a set of queries that perform aggregation across distributed network elements [DR01, vRB01]. The data to be aggregated is most commonly selected or grouped by identifiers such as *source-address* and *destination-address*, or by attributes such as packet type. We now give two concrete examples of continuous query workloads for network monitoring applications.

Example 1: Network path latencies are of interest for infrastructure applications such as manual or automated traffic engineering, *e.g.*, [VGLA00], or quality of service (QoS) monitoring. Path latencies are computed by monitoring the queuing latency of each router along the path, and summing the current queue latencies together with known, static transmission latencies. Since the queue latency at each router generally changes every time a packet enters or leaves the router, a naive approach could generate monitoring traffic whose volume far exceeds the volume of normal traffic, a situation that is clearly unacceptable. Fortunately, path latency applications can generally tolerate approximate answers with bounded absolute numerical error (such as latency within 5 ms of accuracy), so using our approach obtrusive exact monitoring is avoided.

Example 2: Network traffic volumes are of interest to organizations such as internet service providers (ISP's), corporations, or universities, for a number of applications including security, billing, and infrastructure planning. Since it is often inconvenient or infeasible for individual organizations to configure routers to perform monitoring, a simple alternative is to instead monitor end hosts within the organization. We list several traffic monitoring queries that can be performed in this manner, and then motivate their usefulness. These queries form the basis of performance experiments on a real network monitoring system we have implemented; see Section 5.

- Q_1 Monitor the volume of remote login (telnet, ssh, ftp, etc.) requests received by hosts within the organization that originate from external hosts.
- Q_2 Monitor the volume of incoming traffic received by all hosts within the organization.
- Q_3 Monitor the volume of incoming SYN packets received by all hosts within the organization.
- Q_4 Monitor the volume of outgoing DNS lookup requests originating from within the organization.
- Q_5 Monitor the volume of traffic between hosts within the organization and external hosts.

Queries Q_1 through Q_4 are motivated by security considerations. One concern is illegitimate remote login attempts, which often occur in bursts that can be detected using query Q_1. Another concern is denial-of-service (DoS) attacks. To detect the early onset of one form of incoming DoS attacks, organizations can monitor the total volume of incoming traffic received by all hosts using query Q_2. Another form of DoS attack is characterized by a large volume of incoming "SYN" packets that can consume local resources on hosts within the organization, which can be monitored using query Q_3. Organizations also may wish to detect suspicious behavior originating from inside the organization, such as users launching DoS attacks, which may entail sending an unusually large number of DNS lookup requests detectable using query Q_4. In all of these examples, current results of the continuous query can be compared against data previously monitored at similar times of day or calendar periods that represents "typical" behavior, and the detection of atypical or unexpected behavior can be followed by more detailed and costly investigation of the data. Finally, organizations can monitor the overall traffic volume in and out of the organization using query Q_5, to help plan infrastructure upgrades or track the cost of network usage billed by a service provider.

If traffic monitoring is not performed carefully, many of these queries may be disruptive to the communication infrastructure of the organization [HMPW01]. Fortunately, these applications also do not require exact precision in query answers as long as the precision is bounded by a prespecified amount. Note that precision requirements may change over time. For example, during periods of heightened suspicion about DoS attacks, the organization may wish to obtain higher precision for queries Q_2 and Q_3 even at the cost of increased communication overhead.

1.2 Overview of Approach

We focus on continuous queries such as the network monitoring examples above. All of these queries compute aggregate values over streams of updates to numeric (real) data objects, which may originate from many remote data sources. The conventional answer to an aggregation query is a single real value. We define a *bounded approximate* *answer* (hereafter *bounded answer*) to be a pair of real values L and H that define an interval $[L, H]$ in which the precise answer is guaranteed to lie. Precision is quantified as the width of the range $(H - L)$, with 0 corresponding to exact precision and ∞ representing unbounded imprecision. A *precision constraint* for a continuous query is a user-specified constant $\delta \geq 0$ denoting a maximum acceptable interval width for the answer, *i.e.*, $0 \leq H - L \leq \delta$ at all times.

Our goal is to provide guaranteed bounds $[L, H]$ as answers to continuous queries at all times, while filtering update streams at the sources as much as possible. For each remote data object O whose updates are sent to the central stream processor for continuous query evaluation, a *stream filter* is installed at O's source. Each filter maintains a numeric bound $[L_O, H_O]$ of width W_O centered around the most recent numeric update V (where V is the new value for O) that passed the filter, *i.e.*, $L_O = V - \frac{W_O}{2}$ and $H_O = V + \frac{W_O}{2}$. The filter eliminates from the stream all updates V that lie inside O's bound, *i.e.*, that satisfy $L_O \leq V \leq H_O$. Each time an update V passes the filter and is transmitted to the central processor the filter recenters the bound around V by setting $L_O := V - \frac{W_O}{2}$ and $H_O := V + \frac{W_O}{2}$. The central stream processor knows each object O's bound width W_O, and uses it to maintain a cached copy of its bound $[L_O, H_O]$ based on filtered updates received from O's source. The stream processor can be assured that the source (master) value of O remains within the bound until the next update of O is received. (Message latency is addressed in Section 4.)

Continuous queries are registered at the stream processor and whenever a relevant update is received on an input stream query results are updated accordingly. Each continuous query (CQ) Q has an associated precision constraint δ_Q. We assume any number of arbitrary CQ's with arbitrary individual precision constraints. The challenge is to ensure that at all times the bounded answer to every continuous query Q is of adequate precision, *i.e.*, has width at most δ_Q, while filtering streams as much as possible to minimize total communication cost. As a simple example, consider a single CQ requesting the current average of n data values whose update streams are transmitted from different sources, with a precision constraint δ. We can show arithmetically that the width of the answer bound is the average of the widths of the n individual bounds. Thus, one obvious way to guarantee the precision constraint is to use filters with a bound of width δ for each of the n objects. Although this simple policy, which we call *uniform allocation*, is correct (the answer bound is guaranteed to satisfy the precision constraint at all times), it is not generally the best policy. To see why, it is important to understand the effects of update filter bound width [OLW01].

1.2.1 Effects of Filter Bound Width

A filter bound that is narrow, *i.e.*, $H - L$ is small, enables continuous queries to maintain more precise answers, but will fail to filter out a significant portion of the update stream, leading to high communication cost. Conversely, a bound that is wide, *i.e.*, $H - L$ is large, can reduce the stream rate substantially due to a more restrictive filter, but consequently results in more imprecision in query answers. Uniform allocation can perform poorly for the following two reasons:

1. If multiple continuous queries are issued on overlapping sets of objects, different bound widths may be assigned to the same object. While we could simply choose to use the smallest bound width for the filter, the higher update stream rate may be wasted on all but a few queries.

2. Uniform bound allocation does not account for data values that change at different rates due to different rates and magnitudes of updates. In this case, we prefer to allocate wider bounds to data values that change rapidly, and narrower bounds to the rest.

Our performance experiments (Section 5) that compare uniform against nonuniform bound allocation policies provide strong empirical confirmation of these observations.

Reason 2 above indicates that a good nonuniform bound width allocation policy depends heavily on the data update rates and magnitudes, which are likely to vary over time, especially during the long lifespan of continuous queries [MSHR02]. In Example 1 from Section 1.1, a router may alternate between periods of rapidly fluctuating queue sizes (and therefore queue latencies) and steady state behavior, depending on packet arrival characteristics. Therefore, in addition to nonuniformity, we propose an *adaptive* policy, in which bound widths are adjusted continually to match current conditions.

Determining the best bound width allocation at each point in time without incurring excessive communication overhead is challenging, since it would seem to require a single site to have continual knowledge of data update rates and magnitudes across potentially hundreds of distributed sources. Moreover, the problem is complicated by Reason 1 above: we may have many continuous queries with different precision constraints involving overlapping sets of data objects. In Example 1 from Section 1.1, multiple paths whose latencies are monitored will not generally be disjoint, *i.e.*, they may share routers, and precision constraints may differ due to differences in path lengths (number of routers) as well as discrepancies in user precision requirements for different paths.

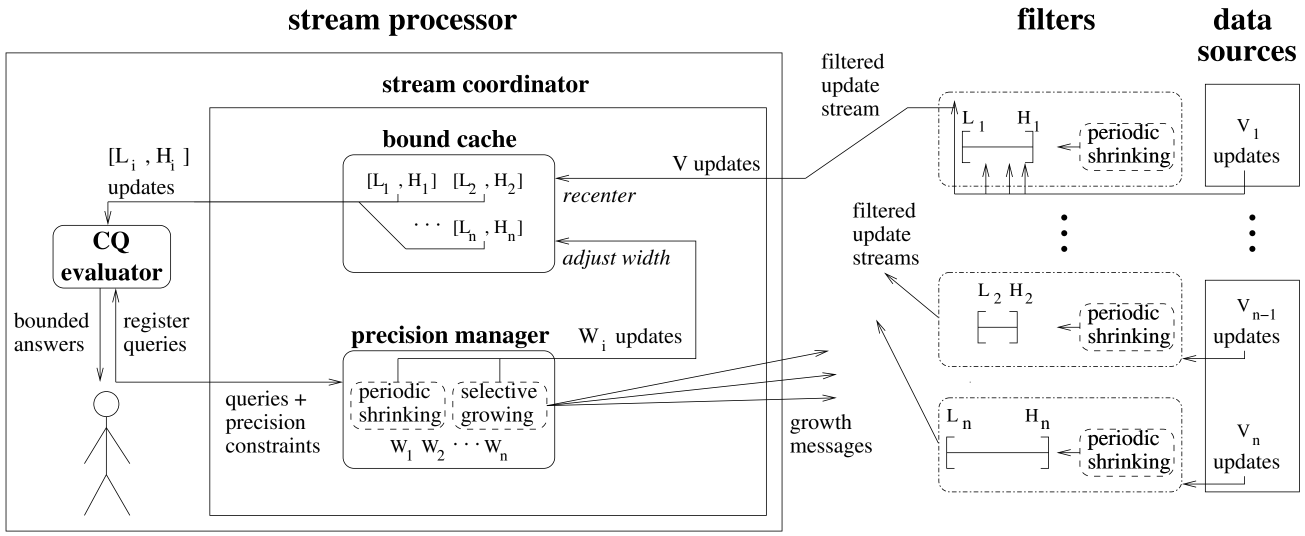

Figure 1: Our approach to stream filtering for continuous queries.

1.2.2 Adaptive Bound Width Adjustment

We have developed a low-overhead algorithm for setting bound widths for stream filters adaptively to reduce communication costs while always guaranteeing to meet the precision constraints of an arbitrary set of registered CQ's. The basic idea is as follows. Each source filter for an object's update stream shrinks the bound width periodically, at a predefined rate. Assuming the bounds begin in a state where all CQ precision constraints are satisfied (we will guarantee this to be the case), shrinking bounds only improves precision, so no precision constraint can become violated due to shrinking. The central stream processor maintains a mirrored copy of the periodically shrinking bound width of each object. Each time the bound width of each object shrinks, the stream processor reallocates the "left-over" width to the objects at the central processor it benefits the most, ensuring all precision constraints will remain satisfied.

1.2.3 Overall Approach

Our approach is illustrated in Figure 1:

- *Data sources*, on the right, each store master values for one or more objects, and they generate streams of updates to those values.
- *Filters* intercept update streams from sources and maintain periodically-shrinking bounds for the objects. Each filter forwards updates that fall outside its bound to the central stream processor, shown on the left, and recenters that bound.
- A *stream coordinator* in the central stream processor receives all streamed updates from the filters.
- A *precision manager* inside the stream coordinator maintains a copy of the periodically shrinking bound width for each object. It reallocates width as described earlier and notifies the corresponding sources via growth messages.
- A *bound cache* inside the stream coordinator receives all bound width changes (growth and shrinks) from the precision manager, along with all value updates streamed from the data sources via the filters. The bound cache maintains a copy of the bound for each object.
- A *CQ evaluator* in the central stream processor receives updates to bounds from the bound cache and provides updated continuous query answer bounds to the user.

Two aspects of our approach are key to achieving low communication cost. First, width shrinking is performed simultaneously at both the stream processor and the sources without explicit coordination. Second, the precision manager uses selective growth to tune the width allocation adaptively. Informally, we minimize the overall cost to guarantee individual precision constraints over arbitrary overlapping CQ's by assigning the widest bounds to the data values that currently are updated most rapidly and are involved in the fewest queries with the largest precision constraints.

In addition to the overall approach, specific contributions of this paper are as follows:

- In Section 3 we specify the core of our approach: a low-overhead, adaptive algorithm for assigning filters to data sources to reduce update stream rates. To guarantee adequate precision for multiple overlapping CQ's while minimizing communication, the precision manager (Figure 1) uses an optimization technique based on systems of linear equations.

- In Section 4 we describe mechanisms in the bound cache (Figure 1) to handle replica consistency issues that arise due to nonnegligible stream message latencies.
- In Section 5 we describe our implementation of a real network traffic monitoring system based on Example 2 in Section 1.1. We provide experimental evidence that our approach significantly reduces overall communication cost compared to a uniform allocation policy for a workload of multiple continuous queries with precision constraints.

2 Related Work

The idea of using numeric bounds and queries with precision constraints to offer a smooth tradeoff between precision and performance in distributed data processing environments was introduced originally in [OLW01, OW00]. However, that work addressed *one-time* rather than continuous queries, resulting in a very different approach. Specifically, [OW00] developed algorithms for optimally combining approximate cached data with exact source data to meet the precision requirement for a single query at a single time. Follow-on work [OLW01] proposed a technique for adjusting cached approximations to minimize the overall communication cost under a workload of one-time queries like those in [OW00]. The precision level of each data object is adjusted in isolation, independent of the precision levels of other objects. Both [OLW01, OW00] exploit the property that for many one-time queries, answer precision can be improved to acceptable levels by accessing remote sources at query-time. In contrast, we focus on applications that require continuous answers to queries. For these applications, an answer of adequate precision for each continuous query must be maintained by the central stream processor at all times.

In *quasi copies* [ABGMA88], centrally maintained approximations are permitted to deviate from exact source values by constrained amounts, thereby providing precision guarantees. Recent work [SBS$^+$02] extends the ideas of [ABGMA88], proposing an architecture in which a network of repositories cooperate to deliver data with precision guarantees to a large population of remote users. In an environment with multiple cooperating repositories, data may be propagated through several nodes before ultimately reaching the end-user application, so latency can be a significant concern. The work in [SBS$^+$02] focuses on selecting topologies and policies for cooperating repositories to minimize the degree to which latency causes precision guarantees to be violated. However, the work in [ABGMA88] and [SBS$^+$02] does not address queries over multiple data values, whose answer precision is a function of the precision of the input values, so they do not need to deal with the optimization problem we address for minimizing communication.

Recent work on reactive network monitoring [DR01] addresses scenarios where users wish to be notified whenever the sum of a set of values from distributed sources exceeds a prespecified critical value. In their solution each source notifies a central processor whenever its value exceeds a certain threshold, which can be either a fixed constant or a value that increases linearly over time. The local thresholds are set to guarantee that in the absence of notifications, the central processor knows that the sum of the source values is less than the critical value. The thresholds in [DR01], which are related to the bounds in our algorithm, are set uniformly across all sources. Similarly, in [YV00], which focuses on bounded approximate values under symmetric replication, error bounds are allocated uniformly across all sites that can perform updates. In contrast to these approaches, we propose a technique in which bound widths are allocated nonuniformly and adjusted adaptively based on stream transmission costs and data change rates.

Maintaining numeric bounds on aggregated values from multiple sources can be thought of as ensuring the continual validity of distributed constraints. Most work on distributed constraint checking, *e.g.*, [BBC80, GW93, Huy97] only considers insertions and deletions from sets, not updates to data values. We are aware of three proposals in which sources communicate among themselves to verify numerical consistency constraints across sources containing changing values: *data-value partitioning* [SS90], the *demarcation protocol* [BGM92], and recent work by Yamashita [Yam02]. The approach of these proposals could in principle be applied to our setting: sources could renegotiate bound widths in a peer-to-peer manner with the goal of reducing stream rates. However, in many scenarios it may be impractical for sources to keep track of the other sources involved in a continuous query (or many continuous queries) and communicate with them directly, and even if practical it may be necessary to contact multiple peers before finding one with adequate "spare" bound width to share. It seems unlikely that the overhead of inter-source communication is warranted to potentially save a single stream message. Furthermore, the algorithms in [BGM92, SS90, Yam02] are not designed for the purpose of minimizing communication cost, and they do not accommodate multiple queries with overlapping query sets.

Some work on real-time databases, *e.g.*, [LLSY91], focuses on scheduling multiple complex, time-consuming computation tasks that yield imprecise results that improve over time. In contrast, our work does not focus on how best to schedule computations, but rather on how to filter data update streams in a distributed environment while bounding the resulting imprecision.

Finally, [OW02] addresses the problem of maximizing data precision subject to constraints on the availability of communication resources. That work considers the inverse of the problem we address in this paper of minimizing communication while meeting constraints on data precision. The choice of which of these complementary approaches is more appropriate in a given scenario depends on characteristics of the environment and application. An interesting topic for future work is to consider policies for automatically choosing or switching between the two approaches, possibly at a per-source granularity.

3 Algorithm Description

In this section we provide more details of our approach, then we describe our algorithm for adjusting stream filter bound widths adaptively, *i.e.*, we focus on the *precision manager* in Figure 1 which is the core of our approach. Recall that our goal is to minimize communication cost while satisfying the precision constraints of all queries at all times. We consider continuous queries that operate over any fixed subset of the remote data values. (We do not consider selection predicates over remote values [OW00], and we assume that all insertions and deletions of new objects into the data set are propagated immediately to the central stream processor via special streams.)

Queries can perform any of the five standard relational aggregation functions: COUNT, MIN, MAX, SUM, and AVG. Of these, COUNT can always be computed exactly in our setting, SUM can be computed from AVG and COUNT, and MIN and MAX are symmetric. It also turns out, as we show in Appendix A, that for the purposes of bound width setting MIN queries can be treated as a collection of AVG queries. Therefore, from this point forward we discuss primarily the AVG function. Note that queries can request the value of an individual data object by posing an AVG query over a single object. For flexibility we also allow objects to be weighted in SUM and AVG queries, formalized in Appendix B.

Each registered continuous query Q_j specifies a *query set* $\mathcal{S}_j$ of objects and a *precision constraint* δ_j. (Users may later alter the precision constraint δ_j of any currently registered continuous query Q_j; see Section 3.4.) The query set $\mathcal{S}_j$ is a subset of a set of n data objects $O_1, O_2, \ldots, O_n$. Each data object O_i has an exact value V_i stored at a remote source that streams updates after filtering to the central stream processor. Say there are m registered continuous AVG queries $Q_1, Q_2, \ldots, Q_m$, with query sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$, respectively. Then the exact answer to AVG query Q_j is $\frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \le i \le n, O_i \in \mathcal{S}_j} V_i$. Our goal is to be able to compute an approximate answer continuously that is within Q_j's precision constraint δ_j, using cached bounds maintained by the central stream processor.

Note that this goal handles *sliding window* queries [MWA$^+$03] as well as queries over the most recent data values only. For the aggregation functions we consider, if an aggregate value is continuously computed to meet a certain precision constraint, then the result of further aggregating over time using any type of window also meets that same precision constraint. However, our algorithm does not necessarily minimize cost for sliding window queries, because sliding windows offer some leniency in the way precision bounds are set: bounds wider than δ are acceptable as long as they are compensated for by bounds narrower than δ within the same time-averaged window. Our algorithm would need to be modified to take advantage of this additional leniency in precision, which is a topic of future work.

Symbol	*Meaning*
n	number of data objects across all sources
O_i	data object ($i = 1 \ldots n$)
V_i	exact source value of object O_i
$[L_i, H_i]$	bound for object O_i
W_i	width of bound for object O_i ($W_i = H_i - L_i$)
C_i	update/growth msg. communication cost for O_i
$\mathcal{C}$	overall communication cost
λ	stream message latency tolerance
m	number of registered continuous queries
Q_j	registered query ($j = 1 \ldots m$)
$\mathcal{S}_j$	set of objects queried by Q_j
δ_j	precision constraint of query Q_j
$\mathcal{T}$	adjustment period (algorithm parameter)
S	shrink percentage (algorithm parameter)
P_i	streamed update period of O_i (last $\mathcal{T}$ time units)
B_i	burden score of O_i (computed every $\mathcal{T}$ time units)
T_j	burden target of Q_j (computed every $\mathcal{T}$ time units)
D_i	deviation of O_i (determines growth priority)

Table 1: Model and algorithm symbols.

Let us assume that all messages (including update streams) are transmitted instantaneously and all computation is instantaneous, for now. In Section 4 we discuss how we handle realistic, non-negligible latencies. When the precision manager sends a bound growth message for object O_i to its source, or an update is transmitted along the data stream from the source to the stream processor (recall Figure 1), we model the cost as a known numerical constant C_i. (Considering the possibility of batching stream updates from the same source is a topic of future work.) For convenience, the symbols we have introduced and others we will introduce later in this section are summarized in Table 1.

Before presenting our general adaptive algorithm for adjusting bound widths, we describe two simple cases in which the bound width of certain objects should remain fixed. First, consider an object O that is involved only in queries that request a bound on the value of O alone (AVG queries over one value). Then it suffices to fix the bound width of O to be the smallest of the precision constraints: $W_O = \min(\delta_j)$ for queries Q_j with $\mathcal{S}_j = \{O\}$. Second, for objects that are not included in any currently registered query, the bound width should be fixed at ∞ so that all updates are filtered from its update stream and none are transmitted to the central stream processor. The remainder of the objects, namely those that are involved in at least one query over multiple objects, pose our real challenge.

To guarantee that all precision constraints are met, the following constraint must hold for each query Q_j:

$$\sum_{1 \leq i \leq n, O_i \in \mathcal{S}_j} W_i \quad \leq \quad \delta_j \cdot |\mathcal{S}_j|$$

In other words, the sum of bound widths for each query must not exceed the product of the precision constraint and the number of objects queried. Initially, the bounds can be set in any way that meets the precision constraint of every query, *e.g.*, by performing uniform allocation for each query, and for objects assigned multiple bound widths, taking the minimum. Then, as discussed in Section 1.2.3, our general strategy is to reallocate bound width adaptively among the objects participating in each query. Reallocation is accomplished with low communication overhead by having bounds shrink periodically over time and having the stream processor's precision manager periodically select one or more bounds to grow based on current conditions. In Section 3.1 we describe the exact way in which bounds are shrunk in our algorithm, and then in Section 3.2 we describe when and how bounds are grown. In Section 3.3 we provide empirical validation that our algorithm converges on good bounds.

3.1 Bound Shrinking

Every object O_i has a corresponding bound width W_i that is maintained simultaneously at both the central stream coordinator and at the source filter. Periodically, every $\mathcal{T}$ time units (seconds, for example), O_i's bound width is decreased symmetrically at both the source filter and the stream coordinator by setting $W_i := W_i \cdot (1 - S)$. The constant $\mathcal{T}$ is a global parameter called the *adjustment period*, and S is a global parameter called the *shrink percentage*. The effect is to decrease the bound width by the fraction S every time unit, rendering the update stream filter less *restrictive* over time. (A filter is more restrictive when it blocks more updates from being streamed.) All adjustments to the bound width—decreases as well as increases—occur at intervals of $\mathcal{T}$ time units. Note that updates may be streamed to the central stream processor at any time but they simply reposition bounds without altering the width. We will discuss good settings for algorithm parameters $\mathcal{T}$ and S in Section 5.

To ensure correctness, each time the bound width for object O_i shrinks, changing the filter condition, the source must re-apply the filter to the current data value V_i. If V_i passes the new filter and has not already been streamed as an update, the source must generate V_i on the update stream.

3.2 Bound Growing

Every $\mathcal{T}$ time units, when all the bound widths shrink automatically as described in the previous section, the precision manager selects certain bound widths to grow instead, making the corresponding stream filters more restrictive. Selecting bounds to increase (and how much) is one of the most intricate parts of our approach.

The first step is to assign a numerical *burden score* B_i to each queried object O_i. Conceptually, the burden score embodies the degree to which an object is contributing to the overall communication cost due to streamed updates. (We use *streamed updates* to refer to those data updates that pass their filter and are sent to the stream processor.) The burden score is computed as $B_i = \frac{C_i}{P_i \cdot W_i}$ where recall that C_i is the cost to send a streamed update of object O_i, and W_i is the current bound width. P_i is O_i's estimated *streamed update period* since the previous width adjustment action, computed as $P_i = \frac{\mathcal{T}}{N_i}$ where N_i is the number of updates of O_i received by the stream coordinator in the last $\mathcal{T}$ time units. (If $N_i = 0$ then $P_i = \infty$ so $B_i = 0$.) The burden formula is fairly intuitive since, *e.g.*, a wide bound or long streamed update period reduces B_i. The exact mathematical derivation is given in Appendix C.

Once each object's streamed update period and burden score have been computed, the second step is to assign a value T_j, called the *burden target*, to each AVG query Q_j. Conceptually, the burden target of a query represents the lowest overall burden required of the objects in the query in order to meet the precision constraint at all times. Since understanding the way we compute burden targets is rather involved, we present our method later in Section 3.2.1, and summarize the process here. For queries over objects involved in no other queries, the burden target is set equal to the average of the burden scores of objects participating in that query. For queries that overlap it turns out that assigning burden targets requires solving a system of m equations with $T_1, T_2, \ldots, T_m$ as m unknown quantities. Because solving this system of equations exactly at run-time is likely to be expensive, we find an approximate solution by running an iterative linear equation solver until it converges within a small error ϵ. (Performance is evaluated in Section 3.2.2.)

Once a burden target has been assigned to each query, the third step is to compute for each object O_i its *deviation* D_i:

$$D_i = \max \left\{ B_i - \sum_{1 \leq j \leq m, O_i \in \mathcal{S}_j} T_j \ , \ 0 \right\}$$

Deviation indicates the degree to which an object is "overburdened" with respect to the burden targets of the queries that access it. To achieve low overall stream rates, it is desirable to equally distribute the burden across all objects involved in a given query. We justify this claim mathematically in Appendix C, and we verify it empirically in Sections 3.3 and 5.

To see how we can even out burden, recall that the burden score of object O_i is $B_i = \frac{C_i}{P_i \cdot W_i}$, so if the bound $[L_i, H_i]$ were to increase in size, B_i would decrease.[1] Therefore, the burden score of an overburdened object can be reduced by growing its bound. Growth is allocated to bounds using the following greedy strategy. Queried objects are considered in decreasing order of deviation, so that the most overburdened objects are considered first. (It is important that ties be resolved randomly to prevent objects having the same deviation—most notably 0—from repeatedly being considered in the same order.) When object O_i is considered, the maximum possible amount by which the bound can be grown without violating the precision constraint of any query is computed as:

$$\Delta W_i = \min_{1 \le j \le m, O_i \in \mathcal{S}_j} \left(\delta_j \cdot |\mathcal{S}_j| - \sum_{1 \le k \le n, O_k \in \mathcal{S}_j} W_k \right)$$

If $\Delta W_i = 0$, then no action is taken. For each nonzero growth value, the precision manager increases the width of the bound for O_i symmetrically by setting $L_i := L_i - \frac{\Delta W_i}{2}$ and $H_i := H_i + \frac{\Delta W_i}{2}$. After all growth has been allocated the precision manager sends a message to each source having objects whose bound width was selected for growth.

In summary, the procedure for determining bound width growth is as follows:

1. Each object is assigned a *burden score* based on its stream transmission cost, estimated streamed update period, and current bound width.

2. Each query is assigned a *burden target* by either averaging burden scores or invoking an iterative linear solver (described next in Section 3.2.1).

3. Each object is assigned a *deviation* value based on the difference between its burden score and the burden targets of the queries that access it.

4. The objects are considered in order of decreasing deviation, and each object O_i is assigned the maximum possible bound growth ΔW_i when it is considered.

Complexity and scalability of this approach are discussed in Section 3.2.2.

[1] This reasoning relies on P_i not decreasing when W_i increases, a fact that holds intuitively and is discussed further in Appendix C.

3.2.1 Burden Target Computation

We now describe how to compute the burden target T_j for each query Q_j, given the burden score B_i of each object O_i (Step 2 above). Recall that conceptually the burden target for a query represents the lowest overall burden required of the objects in the query in order to meet the precision constraint at all times. For motivation consider first the special case involving a single AVG query Q_k over every object $O_1, \ldots, O_n$. In this scenario, the goal for adjusting the burden scores simplifies to that of equalizing them (as shown mathematically in Appendix C) so that $B_1 = B_2 = \cdots = B_n = T_k$. Therefore, given a set of burden scores that may not be equal, a simple way to guess at an appropriate burden target T_k is to take the average of the current burden scores, *i.e.*, $T_k = \frac{1}{|\mathcal{S}_k|} \cdot \sum_{1 \le i \le n, O_i \in \mathcal{S}_k} B_i$. In this way, objects having higher than average burden scores will be given high priority for growth to lower their burden scores, and those having lower than average burden scores will shrink by default, thereby raising their burden scores. On subsequent iterations, the burden target T_k will be adjusted to be the new average burden score. This overall process results in convergence of the burden scores.

We now generalize to the case of multiple queries over different sets of objects. It is useful to think of the burden score of each object involved in multiple queries as divided into components corresponding to each query over the object. Let $\theta_{i,j}$ represent the portion of object O_i's burden score corresponding to query Q_j so that $\sum_{1 \le j \le m, O_i \in \mathcal{S}_j} \theta_{i,j} = B_i$. The goal for adjusting burden scores in the presence of overlapping queries is to have the burden score B_i of each object O_i equal the sum of the burden targets of the queries over O_i (as shown in Appendix C). This goal is achieved if for each query Q_j over O_i, $\theta_{i,j} = T_j$. Therefore, our overall goal can be restated in terms of θ values as requiring that for every query Q_j, $\theta_{1,j} = \theta_{2,j} = \cdots = \theta_{n,j} = T_j$ (the $\theta_{i,j}$ values for objects $O_i \notin \mathcal{S}_j$ are irrelevant). Therefore, given a set of θ values, a simple way to guess at an appropriate burden target T_j for each query Q_j is by taking the average of the θ values of objects involved in Q_j, *i.e.*, $T_j = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \le i \le n, O_i \in \mathcal{S}_j} \theta_{i,j}$. For each object/query pair O_i/Q_j, we can express $\theta_{i,j}$ in terms of B_i, which is known, and the θ values for the other queries over O_i, which are unknown: $\theta_{i,j} = B_i - \sum_{1 \le k \le m, k \ne j, O_i \in \mathcal{S}_k} \theta_{i,k}$. If we replace each occurrence of $\theta_{i,k}$ by T_k for all $k \ne j$ (because we want each $\theta_{*,k}$ to converge to T_k), we have $\theta_{i,j} = B_i - \sum_{1 \le k \le m, k \ne j, O_i \in \mathcal{S}_k} T_k$. Substituting this expression in our formula for guessing at burden targets based on θ values, we arrive at the following expression:

$$T_j = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \le i \le n, O_i \in \mathcal{S}_j} \left(B_i - \sum_{1 \le k \le m, k \ne j, O_i \in \mathcal{S}_k} T_k \right)$$

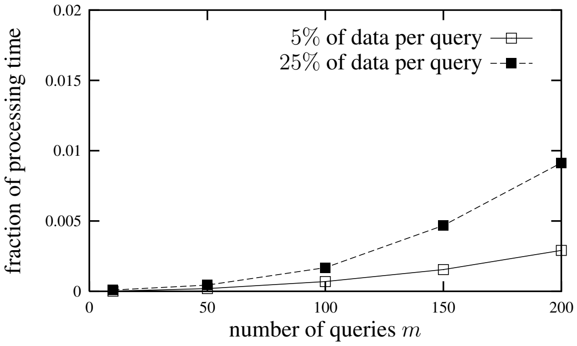

Figure 2: Scalability of linear system solver.

This result is a system of m equations with $T_1, T_2, \ldots, T_m$ as m unknown quantities, which can be solved using a linear solver package.

3.2.2 Algorithm Complexity and Scalability

Let us consider the complexity of our overall bound growth algorithm, which is executed once every $\mathcal{T}$ time units. Most of the steps involve a simple computation per object, and the objects must be sorted once. In the last step, to compute ΔW_i efficiently, the precision manager can continually track the difference ("leftover width") between each query's precision constraint and the current answer's bound width. Then for each object we use the precomputed leftover width value for each query over that object. When queries are over overlapping sets of objects, an iterative linear solver is required to compute the burden targets, which we expect to dominate the computation. The solver represents the system of m equations having $T_1, T_2, \ldots, T_m$ as m unknown quantities as an m by $(m+1)$ matrix, where entries correspond to pairs of queries. Fortunately, the matrix tends to be quite sparse: whenever the query sets $\mathcal{S}_x$ and $\mathcal{S}_y$ of two queries Q_x and Q_y are disjoint, the corresponding matrix entry is 0. For this reason, along with the fact that we can tune the number of iterations, burden target computation using an iterative linear system solver should scale well. We use a publicly-available iterative solver package called *LASPack* [Ska96], although many alternatives exist. Convergence was generally achieved in very few iterations, and the average running time on a modest workstation was only 2.73 milliseconds in our traffic monitoring implementation using multiple overlapping queries (Section 5).

To test the scalability of our algorithm to a larger number of queries and data objects than we used in our implementation, we generated two sets of synthetic workloads consisting of AVG queries over a real-world 200-host network traffic data set (details on this data set are provided in Section 5). We treated each host as a simulated data source with one traffic level object. In one set of workloads, each query is over a randomly-selected 5% (10) of the data sources. In the second set of workloads, each query is over 25% (50) of the data sources, resulting in a much higher degree of overlap among queries. (The degree of overlap determines the density of the linear equation matrix, which is a major factor in the solver running time.) Varying the number of queries m, we measured the average running time on a Linux workstation with a 933 MHz Pentium III processor. We set the error tolerance for the LASPack iterative solver small enough that no change in the effectiveness of our overall algorithm could be detected. Figure 2 shows the fraction of available processing time used by the linear solver when it is invoked once every 10 seconds (when time units are in seconds and $\mathcal{T} = 10$, which turns out to be a good setting as we explain later in Section 5). Allocating bound growth to handle 200 queries over 25% of 200 data sources requires only around 1% of the CPU time at the stream processor.

3.3 Validation Against Optimized Strategy

We performed an initial validation of our bound width allocation strategy based on periodic shrinking and selective growing using a discrete event simulator with synthetic data. The goal of our simulation experiments is to show that our algorithm converges on the best possible bound widths, given a steady-state data set. For this purpose, we generated data for one object per simulated source following a random walk pattern, each with a randomly-assigned step size, and compared two unrealistic algorithms. In the "idealized" version of our algorithm, messages sent by the stream coordinator to sources instructing them to grow their bounds incur no communication cost. Instead, only stream transmission costs were measured, to focus on the bound width choices only. We compared the overall stream transmission cost against the stream transmission cost when bound widths are set statically using an optimization problem solver, described next.

The nature of random walk data makes it possible to simplify the problem of setting bound widths statically to a nonlinear optimization problem, described in Appendix F. While nonlinear optimization problems with inequality constraints are difficult to solve exactly, an approximate solution can be obtained with methods that use iterative refinement. We used a package called FSQP [LZT97], iterating 1000 times with tight convergence requirements to find static bound width settings as close as possible to optimal.

Figure 3 shows the results of comparing the idealized version of our adaptive algorithm against the optimized static allocation, using a continuous AVG query over ten data sources under uniform costs. The x-axis shows the precision constraint δ, and the y-axis shows the overall cost per time unit. In a second experiment we used a workload of five AVG queries whose query sets were chosen randomly from the 10 objects. Figure 4 shows the result of this experiment, for which the size of the query sets was

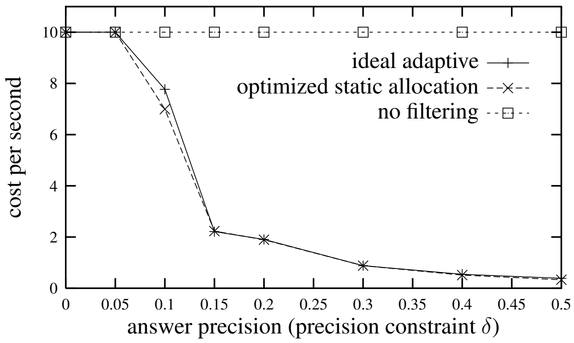

Figure 3: Ideal adaptive algorithm vs. optimized static allocation, random walk data.

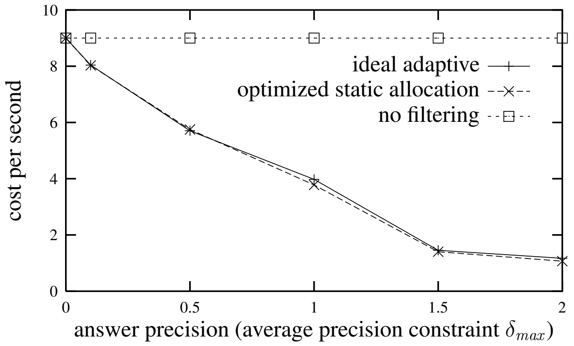

Figure 4: Ideal adaptive algorithm vs. optimized static allocation, multiple queries.

assigned randomly between 2 and 5, and the precision constraint of each query was randomly assigned a value between 0 and δ_{max}, plotted on the x-axis. (For both workloads we also simulated nonuniform costs, and since the results were similar in both cases we omit them.) These results demonstrate that our adaptive bound width setting algorithm converges on bounds that are on par with those selected by an optimizer based on knowledge of the random walk step sizes.

3.4 Handling Precision Constraint Adjustments

Users may at any time choose to alter the precision constraint δ_j of any currently running continuous query Q_j. If the user increases δ_j (weaker precision), then additional bound width is allocated automatically by the bound growth algorithm at the central precision manager at the end of the current adjustment period. If the user decreases δ_j (stronger precision), bound growth is suppressed, and the automatic bound shrinking process will reduce the overall answer bound width over time until the requested precision level is reached. If an immediate improvement in answer precision is required, the central precision manager must proactively send messages to sources requesting explicitly that bounds be shrunk.

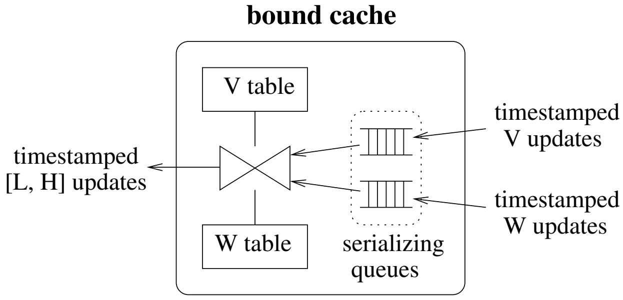

Figure 5: Bound cache for consistent and ordered bound updates.

4 Coping with Latency

In a real implementation of our approach we must cope with message and computation latency. Suppose that each message, including streamed update messages and bound growth messages, has an associated transmission latency as well as a processing delay by both the sender and receiver. We first note that due to such latencies, bound growth will be applied at sources after it is applied at the central stream coordinator, and in the interim period the source filter is less restrictive than it could be. This phenomenon leads to a chance that some unnecessary updates are transmitted to the stream processor, but correctness is not jeopardized. To reduce the delay for growth messages and lessen the chance of unnecessary streamed updates, the stream coordinator can begin the growth allocation process prior to the end of each adjustment period, and base the computations on preliminary streamed update rate estimates.

Communication and computation latency for update streams is of more concern because, if handled naively, continuous queries may not access consistent data across all sources, leading to incorrect answers. To ensure continuous query answers based on consistent data, source filters timestamp all updates transmitted to the stream processor. (We assume closely synchronized clocks, as in [Lam78, Mil91].) Similarly, the precision manager timestamps all bound width updates with an adjustment period boundary. Value and width updates are converted into bound updates via the bound cache (recall Figure 1). Bound updates also have associated timestamps (we will discuss how they are assigned shortly), and our CQ evaluator (Figure 1) treats bound update timestamps as logical update times for the purposes of query processing. Correctness can only be guaranteed if the CQ evaluator receives bound updates monotonically in timestamp order, in which case it produces a new output value for every unique timestamp it receives as part of any update. When multiple updates have the same timestamp, the query evaluator treats them as a single atomic transaction and only produces a new output value for the last update with the same timestamp.

To ensure that the CQ evaluator receives bound updates that represent a consistent state and arrive in timestamp order, the bound cache in the central stream coordinator is implemented using a combination of two *serializing queues* (described shortly) and a *symmetric hash join* [WA91] (or other non-blocking join operator), as illustrated in Figure 5. The join operator combines value updates with width updates to produce bound updates that mirror the bounds maintained by source filters, using object identifier equality as the join condition. Each hash table stores only the most recent value or width update for each object, based on timestamp, and each join result is assigned a timestamp equal to the timestamp of the input that generated the result.

Join inputs must arrive in timestamp order to ensure correct behavior. One way to guarantee global timestamp ordering across all V and W update streams is to delay processing of each update received on a particular stream until at least one update with a greater timestamp has been received on each of the other streams [Lam78]. This approach is impractical in our setting, however, because it can result in unbounded delays unless additional communication is performed, and delays tend to be longer when the number of update streams is large. Instead, we take an approach similar to one taken in the field of streaming media to handle unordered packets with variable latency (see, *e.g.*, [MKT98]), which relies on a reasonable latency upper bound. In our approach, serializing queues are positioned between the value and width update streams and the join. The effect of each serializing queue is to order updates by timestamp, and release each update U as soon as the current time t_{now} reaches $t_U + \lambda$, where t_U is U's timestamp and λ is the *latency tolerance*: an upper bound on the latency for any streamed update message that holds with high probability and is determined empirically based on the networking environment. As long as all update messages obey this latency tolerance and appropriate queue scheduling is used, we can be assured that the serializing queues together output to the join a monotonic stream of updates ordered by timestamp. Of course in practice occasional messages may be delayed by more than λ, resulting in temporary violations of precision guarantees, an unavoidable effect in any distributed environment with unbounded delays. Larger values of λ reduce the likelihood that update messages arrive late, but also increase the delay before results are released to the user. In Section 5.3 we show that using a reasonable choice of λ, late update messages are very rare.

4.1 Exploiting Constrained Change Rates

In some applications, certain data objects may have known maximum change rates, or at least bounds on change rate that hold with very high probability. If each data object O_i participating in a continuous query Q has maximum change rate R_i, then an approximate answer to Q that bounds the answer at time $t_{now}-\epsilon$ (for some local processing delay ϵ at the central stream processor), rather than time $t_{now} - \lambda - \epsilon$, can be provided by having the stream coordinator "pad" the bounds to account for recent changes rather than using serializing queues with a built-in delay as discussed above. Padding is performed by adding $\phi_i = 2 \cdot R_i \cdot \lambda$ symmetrically to the width of each updated bound $[L_i, H_i]$ after it is produced by the join. If this technique is employed, a reduced precision constraint $\delta'_Q \leq \delta_Q$ should be used for the purposes of bound width allocation and adjustment to ensure that padded answer bounds meet the original precision constraint δ_Q. The value of δ'_Q depends on the amount of padding and the type of query. For example, for AVG queries, we can set $\delta'_Q = \delta_Q - \frac{1}{|\mathcal{S}_Q|} \cdot \sum_{1 \leq i \leq n, O_i \in \mathcal{S}_Q} \phi_i$.

5 Implementation and Experimental Validation

We evaluated the performance of our technique and its practical applicability by building a real network traffic monitoring system. The system currently runs continuous queries over 10 hosts in our research group's network, following Example 2 from Section 1.1. In our implementation, a special monitoring program executes on each host. It captures network traffic activity using the *TCPdump* utility and computes packet rate measurements as needed by the queries in the workload, representing them as time-varying numerical data objects. We use time units of one second, which matches the granularity at which our TCPdump monitor is able to capture data. Each host acts as a data source, and in all cases objects and their updates correspond to a one-minute moving window over packet rate measurements. We use queries $Q_1 - Q_5$ from Example 2 of Section 1.1, so different experiments use different objects. For example, query Q_5 uses one object for each of the 10 sources (hosts) for the overall windowed traffic volume between that host and external hosts. Each data object is assigned a bound width for update filtering at the source. Bounds are cached at a central monitoring station, which updates the aggregated answers to continuous queries as bound widths shrink and grow and as data updates stream in. The communication cost (streamed update or growth message) for each object is modeled as a uniform unit cost.

The first step in our experimentation was to determine good settings for the two algorithm parameters $\mathcal{T}$ (adjustment period) and S (shrink percentage). We experimented with a real-world network traffic data set in our simulator, with both uniform and nonuniform costs, and also with live data in our network monitoring implementation, and found that the following settings worked well in general: $\mathcal{T} = 10$ time units to achieve low growth message overhead relative to the timescale at which the data changes, and $S = 0.05$ (5%) to allow adaptivity while avoiding erratic bound width adjustments that tend to degrade performance. We also determined that our algorithm is not highly sensitive to the exact parameter settings. Setting or adjusting these param-

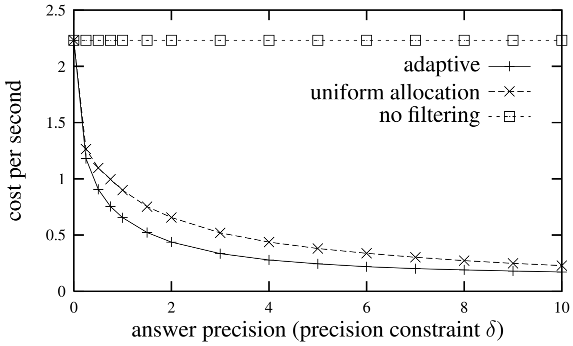

Figure 6: Adaptive algorithm vs. uniform static bound setting, query Q_5 using network monitoring implementation.

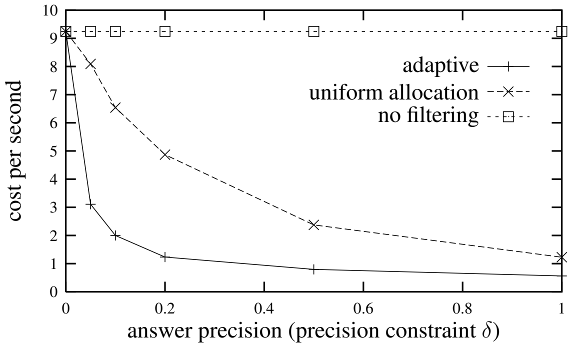

Figure 7: Adaptive algorithm vs. uniform static bound setting, single query over large-scale network data using simulator.

eters automatically is a topic of future work.

5.1 Single Query

We now present our first experimental results showing the effectiveness of our algorithm. We begin by considering a simple case involving a single continuous AVG query. We used query Q_5 from Example 2 of Section 1.1 applied over the 10 sources. Q_5 monitors the average rate of traffic to and from our organization, which ranged from about 100 to 800 packets per second.

Since the optimized static bound width allocation described in Section 3.3 relies on knowing the random walk step size, it is not applicable to real-world data so cannot be used for comparison. Assuming data update patterns are not known in advance, the only obvious method of static allocation is to set all bound widths uniformly. Thus, we compare our algorithm against this setting.

Figure 6 compares the overall communication cost incurred in our real-world implementation by our adaptive algorithm compared to uniform static allocation, measuring cost for 21 hours after an initial warm-up period. The continuous query monitors the average traffic level with precision constraint δ ranging from 0 to 10 packets per second. Our algorithm offers a mild improvement over uni-

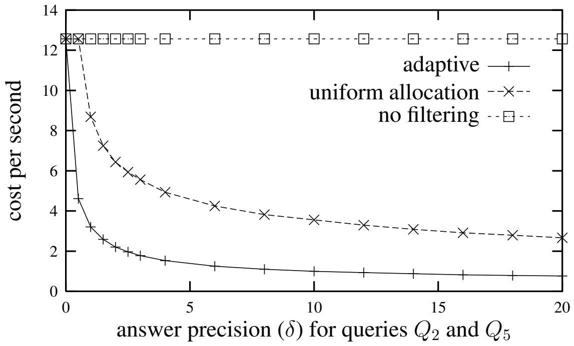

Figure 8: Adaptive algorithm vs. uniform static bound setting, queries $Q_1 - Q_5$ using network monitoring implementation.

form bound allocation for a single query, bearing in mind that the experiment was over small-scale network monitoring data available for monitoring on a few hosts within our organization.

To test our algorithm on large-scale network data with many hosts, we ran a simulation on publicly available traces of network traffic levels between hosts distributed over a wide area during a two hour period [PF95]. For each host, average packet rates ranged from 0 to about 150 packets per second, and we randomly selected 200 hosts as our simulated data sources. Figure 7 shows the results using our simulator over this large-scale data set, accounting for all communication costs. With this data set our algorithm significantly outperforms uniform static allocation for queries that can tolerate a moderate level of imprecision (small to medium precision constraints). For queries with very weak precision requirements (large precision constraints), even naive allocation schemes achieve low cost, and the slight additional overhead of our algorithm causes it to perform about on par with uniform static allocation.

5.2 Multiple Queries

We now describe our experiments with multiple continuous queries having overlapping query sets. We used a workload of the five continuous AVG queries $Q_1 - Q_5$ from Example 2 in Section 1.1. $2^5 - 1$ "measurement groups" are defined at each source based on which subsets of the five query predicates a packet satisfies. Each measurement group is aggregated and acts as a data object whose updates are filtered with a bound and streamed to the central monitoring station. (It may seem more natural for sources to further aggregate data objects into one object per query; we discuss this option shortly in Section 5.2.1.)

Figure 8 shows the results of our experiments measuring cost for 23 hours after an initial warm-up period. The x-axis shows the precision constraints used for queries Q_2 and Q_5. The other queries monitored a much lower volume of data (by a factor of roughly 100) so for each run we set their precision constraints to $1/100$th that shown on the x-axis. As discussed in Section 1.2.1, uniform static bound width allocation can be performed for multiple overlapping queries if for each data object involved in more than one query we maintain the narrowest bound assigned. Our algorithm significantly outperforms uniform static allocation for queries that can tolerate a moderate level of imprecision (small to medium precision constraints). For example, using reasonable precision constraints of $\delta = 4$ for queries Q_2 and Q_5 and $\delta = 0.04$ for queries Q_1, Q_3, and Q_4, our algorithm achieves a cost of only 1.6 messages per second, compared with a cost of 5.4 with uniform static bound width allocation. Furthermore, as with all previous results reported, the overall cost decreases rapidly as the precision constraint is relaxed, offering significant reductions in communication cost compared with not filtering.

5.2.1 Source Aggregation

In the multiple-query workload it may appear advantageous for sources to further aggregate data objects to form one object per query whose updates are streamed to the monitoring station, instead of one per query subset. Interestingly, doing so (a process we call *source aggregation*) does not always result in lower overall cost, and whether it is cheaper to perform source aggregation depends on the data, query workload, and user-specified precision constraints. In Appendix G we show mathematically that there are reasonable conditions under which source aggregation is expected to achieve lower cost, and other reasonable conditions under which cost is lower without source aggregation. Note that the choice of whether to perform source aggregation can be made independently for each source and for each independent set of overlapping queries, and the best overall configuration may be to perform source aggregation selectively.

In general, if there is a large disparity between the precision constraints of overlapping queries, source aggregation achieves lower overall communication cost for update stream transmission because queries with large precision constraints can use separate wide bounds not constrained by other queries with small precision constraints. On the other hand, if most updates are to objects involved in multiple queries, it is preferable in terms of overall communication cost not to apply source aggregation, to avoid redundantly applying those updates to one object per relevant query. As an extreme case, consider Example 1 from Section 1.1 in which each source (router) maintains a single queue latency value accessed by multiple path latency queries, and all updates at each source apply to objects involved multiple queries. Source aggregation would have each router maintain one copy of its queue size measurement for each path latency query, each with a bound having a potentially different width. Updates would fall outside the bounds at different times causing unnecessary updates to be transmitted to the central stream processor.

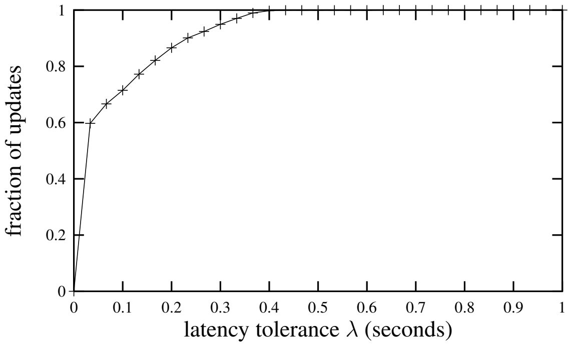

Figure 9: Fraction of updates arriving after the maximum latency tolerance λ for query Q_5.

As future work we plan to design and experiment with an algorithm that monitors the expected cost of using versus not using source aggregation and switches adaptively between them.

5.3 Impact of Message Latency

Our last experiment measures update message latency. In Figure 9 we vary the maximum latency tolerance λ (recall Section 4) and measure the fraction of updates arriving within λ for query Q_5 during a 21-hour period. In our implementation filtered update streams are transmitted over a local area network. A value of $\lambda = 0.4$ seconds, which is reasonable since data changes are meaningful on a scale of about 1 second in our case, ensures that 99.8% of updates are received on time. When a moderate precision constraint for this query of $\delta = 5$ is used, updates exceeding the latency allowance occur only about once every 65.7 minutes. When an update does arrive late, the resulting inconsistency in the output is brief, and based on our measurements plotted in Figure 9 the overall fraction of time the answer is consistent (*fidelity* in the terminology of [SBS$^+$02]) is at least 99.997%. By adjusting λ, higher fidelity can be achieved at the expensive of delayed output, or vice-versa. ([MKT98] proposes an algorithm for adjusting the latency tolerance adaptively in a similar context based on observed latency distributions.)

6 Summary and Future Work

We specified a new approach for reducing communication cost in an environment of centralized continuous query processing over distributed data streams. Our approach hinges on specifying precision constraints for continuous queries, which are used to generate adaptive filters at remote data sources that significantly reduce update stream rates while still guaranteeing sufficient precision of query results at all times. Our approach enables users or applications to trade precision for lower communication cost at a fine granularity by individually adjusting precision constraints of continuous queries. Imprecision of query results is bounded numerically so applications need not deal with any uncertainty.

To validate our approach we performed a number of experiments using simulations and a real network monitoring implementation. Our experiments demonstrated:

- For a steady-state scenario our algorithm converges on bound widths that perform on par with those selected statically using an optimization problem solver with complete knowledge of data update behavior.
- In the case of a single continuous query, our algorithm significantly outperforms uniform bound width allocation in some cases, and in other cases our algorithm is only somewhat better than uniform allocation. As future work we plan to characterize those cases for which our algorithm achieves a significant improvement over uniform static allocation, and those cases for which uniform allocation suffices.
- In the case of multiple overlapping continuous queries, our algorithm significantly outperforms uniform bound width allocation.

While our optimization techniques are specialized to aggregation queries over numeric values, general continuous query processing can in theory be performed over bounded values to produce bounded answers with precision guarantees. Further work in this area includes understanding how imprecision propagates through more complex query plans, and developing appropriate optimization techniques for adapting remote filter predicates in these more complex environments.

Acknowledgments

We thank David Cheriton, Hector Garcia-Molina, Ion Stoica, Cheng Yang, and Dapeng Zhu for their helpful discussions and feedback.

References

[ABGMA88] R. Alonso, D. Barbara, H. Garcia-Molina, and S. Abad. Quasi-copies: Efficient data sharing for information retrieval systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 443–468, Venice, Italy, March 1988.

[BBC80] P. A. Bernstein, B. T. Blaustein, and E. M. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 126–136, Montreal, Canada, October 1980.

[BGM92] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 373–387, Vienna, Austria, March 1992.

[BO03] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2003.

[BW01] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3):109–120, September 2001.

[CCC+02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, August 2002.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–390, Dallas, Texas, May 2000.

[DR01] M. Dilman and D. Raz. Efficient reactive monitoring. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, Anchorage, Alaska, April 2001.

[EGPS01] D. Estrin, L. Girod, G. Pottie, and M. Srivastava. Instrumenting the world with wireless sensor networks. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Salt Lake City, Utah, May 2001.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, Massachusetts, 1989.

[GW93] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, Washington, D.C., May 1993.

[HMPW01] A. Householder, A. Manion, L. Pesante, and G. Weaver. Managing the threat of denial-of-service attacks. Technical report, CMU Software Engineering Institute CERT Coordination Center, October 2001. http://www.cert.org/archive/pdf/Managing_DoS.pdf.

[Huy97] N. Huyn. Maintaining global integrity constraints in distributed databases. *Constraints*, 2(3/4):377–399, 1997.

[KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Network Monitoring (MobiCom 99)*, pages 271–278, Seattle, Washington, August 1999.

[Lam78] L. A. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LLSY91] J. W. S. Liu, K. Lin, W. Shih, and A. C. Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering*, 11(4):610–628, 1999.

[LZT97] C. T. Lawrence, J. L. Zhou, and A. L. Tits. User's guide for CFSQP version 2.5: A C code for solving (large scale) constrained nonlinear (minimax) optimization problems, generating iterates satisfying all inequality constraints. Technical report TR-94-16r1, Institute for Systems Research, University of Maryland, 1997.

[MBC+01] R. Min, M. Bhardwaj, S. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan. Low-power wireless sensor networks. In *Proceedings of the Fourteenth International Conference on VLSI Design*, Bangalore, India, January 2001.

[MF02] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, California, February 2002.

[Mil91] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10), October 1991.

[MKT98] S. B. Moon, J. Kurose, and D. Towsley. Packet audio playout delay adjustment: Performance bounds and algorithms. *ACM/Springer Multimedia Systems*, 6:17–28, January 1998.

[MSHR02] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, June 2002.

[MWA+03] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[OLW01] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 355–366, Santa Barbara, California, May 2001.

[OW00] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 144–155, Cairo, Egypt, September 2000. (Extended version available at http://www-db.stanford.edu/pub/papers/trapp-ag.ps.).

[OW02] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 73–84, Madison, Wisconsin, June 2002. (Extended version available at http://www.db.stanford.edu/ olston/publications/bes.pdf.).

[PF95] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[PK00] G.J. Pottie and W.J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):551–558, May 2000.

[SBS+02] S. Shah, A. Bernard, V. Sharma, K. Ramamritham, and P. Shenoy. Maintaining temporal coherency of cooperating dynamic data repositories. In *Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases*, Hong Kong, China, August 2002.

[Ska96] T. Skalicky. Laspack reference manual, 1996. http://www.tu-dresden.de/mwism/skalicky/ laspack/laspack.html.

[SS90] N. Soparkar and A. Silberschatz. Data-value partitioning and virtual messages. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 357–367, Nashville, Tennessee, April 1990.

[Ste91] J. Stewart. *Calculus: Early Transcendentals, Second Edition*. Brooks/Cole, 1991.

[VGLA00] S. Vutukury and J.J. Garcia-Luna-Aceves. A traffic engineering approach based on minimum-delay routing. In *Proceedings of the IEEE International Conference on Computer Communications and Networks*, Las Vegas, Nevada, October 2000.

[vRB01] R. van Renesse and K. Birman. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. Technical report, Cornell University, 2001.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, 1991.

[Yam02] T. Yamashita. Dynamic replica control based on fairly assigned variation of data with weak consistency for loosely coupled distributed systems. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems*, pages 280–289, Vienna, Austria, July 2002.

[YV00] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 123–133, Cairo, Egypt, September 2000.

Appendix

A MIN Queries

We consider MIN queries, and show that for the purposes of bound width setting they can be treated as a collection of AVG queries. Consider a MIN query Q_j over query set $\mathcal{S}_j$ with precision constraint δ_j. First, we show that if the bound for each object $O_i \in \mathcal{S}_j$ has width at most δ_j, the precision constraint is always met. To see this fact, observe that the answer $[L, H] = [\min(L_i), \min(H_i)]$, and it has width $H - L = \min(H_i) - \min(L_i) \leq H_k - \min(L_i)$, for the upper bound H_k of any object $O_k \in \mathcal{S}_j$. If we choose O_k to be the object with the lowest lower bound, *i.e.*, $L_k = \min(L_i)$, we obtain $H - L \leq H_k - L_k$. Thus, if all bounds have width at most δ_j, then the answer bound has width $H - L \leq \delta_j$.

We now show the converse: if the bound for some $O_i \in \mathcal{S}_j$ has width greater than δ_j, then the precision constraint cannot be guaranteed. To see this fact, consider an object $O_i \in \mathcal{S}_j$ whose value is far greater than the minimum value of the queried objects. It may seem safe to assign a bound $[L_i, H_i]$ of width exceeding δ_j to O_i, as long as L_i is greater than the lowest lower bound, since $[L_i, H_i]$ will not contribute to the answer bound. However, if the stream processor receives an update of one or more objects, causing L_i to suddenly become the lowest lower bound and H_i the lowest upper bound, then the new answer bound has width greater than δ_j. Although this situation could be remedied by requesting a tighter bound for O_i from its source, this procedure would incur a delay during which Q_j's answer bound violates its precision constraint, breaking our requirement of continuous precision for continuous queries. Therefore, for a MIN query Q_j, the bound for each queried object must have width at most δ_j at all times, and those widths are guaranteed to uphold the precision constraint. (In circumstances where occasional precision constraint violations for short periods of time are tolerable, the technique of [BO03], which avoids communication altogether for objects far from the current minimum, can be used instead.)

Based on the above observations, for the purposes of bound width setting, a MIN query Q_j with precision constraint δ_j over a set of objects $\mathcal{S}_j$ is equivalent to a set of single-object queries over each $O_i \in \mathcal{S}_j$ with precision constraint δ_j for each. Since single-object queries are AVG queries over one object, the techniques in Section 3 can be applied to MIN queries without modification.

Overall, since SUM can be computed from AVG, and MAX is symmetric to MIN, the Section 3 techniques can be used for any workload consisting of a combination of SUM, AVG, MIN, and MAX queries.

B Modifications to Handle Weighted AVG and SUM Queries

We introduce modifications to handle continuous queries that monitor the weighted average (or weighted sum by extension) of a set of data values. Say there are m such registered continuous queries $Q_1, Q_2, \ldots, Q_m$. For each query Q_j, each object O_i in Q_j's query set $\mathcal{S}_j$ may have an associated (positive or negative) weight $K_{i,j}$ for the query. We let $K_{i,j} = 1$ in the absence of a specified weight. For notational convenience let us say $K_{k,j} = 0$ for all objects O_k in the stream processor but not in Q_j's query set $\mathcal{S}_j$. Then the exact answer to AVG query Q_j is $\frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \leq i \leq n} K_{i,j} \cdot V_i$, and our goal is to be able to continuously compute an approximate answer from bounds that is within Q_j's precision constraint δ_j. Note that weighted AVG and SUM queries are quite flexible, *e.g.*, we can monitor the difference between two values by using weights 1 and -1.

To guarantee that all precision constraints are met, the following constraint must hold for each query Q_j (see Appendix D for a derivation):

$$\sum_{1 \leq i \leq n} |K_{i,j}| \cdot W_i \leq \delta_j \cdot |\mathcal{S}_j|$$

In other words, the weighted sum (in absolute value) of bound widths for each query must not exceed the product of the precision constraint with the number of objects queried. Initially, the bounds can be set in any way that meets the precision constraint of every query, *e.g.*, allocating bound widths uniformly for each query and for each object using the minimum width assigned as described in Section 1.2.1. Then, our adaptive algorithm described in Section 3 can be used to adjust bound widths, with the following three modifications to the bound growth process (Section 3.2 and Section 3.2.1) to incorporate weights.

First, we must modify the burden target computation described in Section 3.2.1. Recall that $\theta_{i,j}$ represents the portion of object O_i's burden score corresponding to query Q_j. Appropriately weighting the average $\theta_{*,j}$ value for query Q_j across all queried objects $O_i \in \mathcal{S}_j$, we have $T_j = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \leq i \leq n, O_i \in \mathcal{S}_j} \frac{\theta_{i,j}}{|K_{i,j}|}$. If we substitute our expression for $\theta_{i,j}$ and replace each occurrence of $\theta_{i,k}$ by $|K_{i,k}| \cdot T_k$ for all $k \neq j$, we arrive at the following expression:

$$T_j = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \le i \le n, O_i \in \mathcal{S}_j} \left(\frac{B_i}{|K_{i,j}|} - \sum_{1 \le k \le m, k \ne j} \frac{|K_{i,k}|}{|K_{i,j}|} \cdot T_k \right)$$

Second, when computing the deviation D_i of each object O_i from the burden targets of the queries that access it, we must weight the burden targets appropriately:

$$D_i = \max\{B_i - \sum_{1 \le j \le m} |K_{i,j}| \cdot T_j, 0\}$$

Finally, during greedy allocation of growth, ΔW_i, the maximum possible amount by which object O_i's bound can be grown without violating the precision constraint of any weighted query is computed as:

$$\Delta W_i = \min_{1 \le j \le m, O_i \in \mathcal{S}_j} \frac{\delta_j \cdot |\mathcal{S}_j| - \sum_{1 \le k \le n} |K_{k,j}| \cdot W_k}{|K_{i,j}|}$$

C Mathematical Justification for Bound Growth Strategy

We give a mathematical model for the behavior of objects whose update streams are filtered using bounds, and we use it to justify the bound growth allocation strategy presented in Section 3.2 and extended in Appendix B to handle weighted queries. We model the behavior of objects with filtered update streams as follows. For an object O_i whose exact value V_i varies with time, we assume that the streamed update period P_i is a function of the bound width $W_i = H_i - L_i$, and signify this relationship by writing $P_i(W_i)$ instead of P_i. Intuitively, when a bound is narrow, the actual value is likely to exceed it more often and therefore the streamed update period will be short. Conversely, when a bound is wide we expect the streamed update period to be longer. The precise relationship between W_i and P_i depends on the behavior of V_i.

Since each streamed update of object O_i incurs a cost C_i, we can express the communication cost of the entire system as:

$$\mathcal{C} = \sum_{1 \le i \le n} \frac{C_i}{P_i(W_i)}$$

If no continuous queries are registered, then zero cost can be achieved by setting all bounds to $[-\infty, \infty]$. However, as we will derive in Appendix D, each query Q_j with weights $K_{1,j}, K_{2,j}, \ldots, K_{n,j}$ and precision constraint δ_j imposes the following constraint on the bound widths:

$$\sum_{1 \le i \le n} |K_{i,j}| \cdot W_i \le \delta_j \cdot |\mathcal{S}_j|$$

(Recall that we set $K_{i,j} = 0$ for objects $O_i \notin \mathcal{S}_j$, and that all queries can be treated as weighted AVG queries.) We are now faced with the optimization problem of minimizing the overall cost $\mathcal{C}$ while satisfying the above constraint for each of m queries $Q_1, Q_2, \ldots, Q_m$.

Unless the function $P_i(W_i)$ is inversely proportional to W_i, which is unlikely as discussed above, we are faced with a nonlinear optimization problem with inequality constraints. Since such problems are very difficult to solve, we decided to try treating the inequality constraints as equality constraints to get an idea of the form of the solution. (We later verified the success of this approach by comparing results obtained using the algorithm which we derive from it with results obtained by executing a nonlinear optimization problem solver that operates over synthetic data; see Section 3.3.) We can apply the method of Lagrange Multipliers [Ste91] to minimize $\mathcal{C}$ under a set of m equality constraints of the form:

$$\sum_{1 \le i \le n} |K_{i,j}| \cdot W_i = \delta_j \cdot |\mathcal{S}_j|$$

The solution [Ste91] has the property that there are a set of m constants $\lambda_1, \lambda_2, \ldots, \lambda_m$ such that for all i:

$$C_i \cdot \frac{\partial}{\partial W_i} \left(\frac{1}{P_i(W_i)} \right) = \sum_{1 \le j \le m} |K_{i,j}| \cdot \lambda_j$$

To evaluate the derivative we make the assumption that the function $P_i(W_i)$ has roughly the form $P_i(W_i) = Z_i \cdot (W_i)^p$, where each Z_i is an arbitrary constant and p can be any positive real number. For example, this model with $p = 2$ applies to data that follows a random walk pattern, as shown in Appendix E. Assuming $P_i(W_i)$ roughly follows this form, we can evaluate the partial derivative to obtain the following expression:

$$\frac{C_i}{P_i \cdot W_i} = M \cdot \sum_{1 \le j \le m} |K_{i,j}| \cdot \lambda_j$$

where M is a constant. Finally, let the burden target $T_j = M \cdot \lambda_j$ and recall that the burden score $B_i = \frac{C_i}{P_i \cdot W_i}$, giving:

$$B_i = \frac{C_i}{P_i \cdot W_i} = \sum_{1 \le j \le m} |K_{i,j}| \cdot T_j$$

According to this formula, we want P_i and W_i to be set such that the burden score B_i of each object O_i roughly equals the weighted sum of the burden targets of all queries over O_i. Our algorithm described in Section 3 converges to this state by monitoring the burden scores and increasing W_i, and consequently P_i as well, to decrease B_i when it becomes significantly higher than the weighted sum of estimated targets.

D Derivation of Constraint Formula

We now derive the constraint formula referenced in Appendices B and C. The bound $[L, H]$ on the answer to a weighted average query Q_j is computed from the weight $K_{i,j}$ associated with each object O_i along with its bound $[L_i, H_i]$. The lowest possible weighted average L occurs when the values of objects with a positive weight are as small as possible and the values of objects with a negative weight are as large as possible. In other words, whenever $K_{i,j} \ge 0$, the answer lower bound has $V_i = L_i$, and whenever $K_{i,j} < 0$, the answer lower bound has $V_i = H_i$. The converse holds for the upper answer bound H. Therefore, a tight bound $[L, H]$ on the weighted average answer is:

$$L = \frac{1}{|\mathcal{S}_j|} \cdot \left(\sum_{1 \le i \le n, K_{i,j} \ge 0} K_{i,j} \cdot L_i + \sum_{1 \le i \le n, K_{i,j} < 0} K_{i,j} \cdot H_i \right)$$

$$H = \frac{1}{|\mathcal{S}_j|} \cdot \left(\sum_{1 \le i \le n, K_{i,j} \ge 0} K_{i,j} \cdot H_i + \sum_{1 \le i \le n, K_{i,j} < 0} K_{i,j} \cdot L_i \right)$$

which can be rewritten as:

$$L = \frac{1}{|\mathcal{S}_j|} \cdot \left(\sum_{1 \le i \le n, K_{i,j} \ge 0} |K_{i,j}| \cdot L_i - \sum_{1 \le i \le n, K_{i,j} < 0} |K_{i,j}| \cdot H_i \right)$$

$$H = \frac{1}{|\mathcal{S}_j|} \cdot \left(\sum_{1 \le i \le n, K_{i,j} \ge 0} |K_{i,j}| \cdot H_i - \sum_{1 \le i \le n, K_{i,j} < 0} |K_{i,j}| \cdot L_i \right)$$

The answer bound width is $H - L$, which using the second pair of formulae above simplifies to $\frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \le i \le n} |K_{i,j}| \cdot (H_i - L_i)$, making our constraint:

$$\sum_{1 \le i \le n} |K_{i,j}| \cdot W_i \le \delta_j \cdot |\mathcal{S}_j|$$

E Streamed Update Period for Random Walk Data

We derive an expression for the expected streamed update period $P_i(W_i)$ as a function of the bound width W_i for an object O_i that changes according to a random walk pattern. In the random walk model, after t steps of size s_i, the probability distribution of the value is a binomial distribution with variance $\sigma_i^2 = (s_i)^2 \cdot t$ [GKP89]. Chebyshev's Inequality [GKP89] gives an upper bound on the probability $\mathcal{P}$ that the value is beyond any distance k from the starting point: $\mathcal{P} \le \frac{\sigma_i^2}{k^2}$. If we let $k = \frac{W_i}{2}$, treat the upper bound as a rough approximation, and solve for t when $\mathcal{P} = 1$, we obtain $t \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$, which is roughly the expected streamed update period, so $P_i(W_i) \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$. This model for the streamed update period of random walk data was verified empirically in [OLW01]. In relation to our general expression in Appendix C, for random walks $Z_i = \frac{1}{(2 \cdot s_i)^2}$ and $p = 2$.

F Optimized Static Bound Width Allocation

Recall from Appendix C that the overall cost $\mathcal{C} = \sum_{1 \le i \le n} \frac{C_i}{P_i(W_i)}$, where $P_i(W_i)$ is the streamed update period as a function of the bound width of O_i. In Appendix E, we derived an approximate formula for this function in the random walk case: $P_i(W_i) \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$, which depends on the step size s_i. If the step sizes of all the objects are known, then a good static bound width allocation can be found by solving the following nonlinear optimization problem: minimize $\sum_{1 \le i \le n} \frac{C_i \cdot (2 \cdot s_i)^2}{(W_i)^2}$ in the presence of m constraints of the form $\sum_{1 \le i \le n, O_i \in \mathcal{S}_j} W_i \le \delta_j \cdot |\mathcal{S}_j|$.

G Mathematical Analysis of Source Aggregation

We analyze the effect of source aggregation mathematically. We show that cases exist where source aggregation is advantageous in terms of minimizing stream transmission cost, and cases also exist where it is not. Suppose there are three data values V_0, V_1, and V_2 at a single data source, and two continuous SUM queries Q_1 and Q_2 at the stream processor. Q_1 computes $V_0 + V_1$ with a precision constraint of δ_1, and Q_2 computes $V_0 + V_2$ with precision constraint δ_2. If source aggregation is not applied, then three bounds are maintained by the stream processor: $[L_0, H_0]$, $[L_1, H_1]$, and $[L_2, H_2]$, corresponding to the three source values V_0, V_1, and V_2. Let width $W_i = H_i - L_i$. The query precision constraints require that $W_0 + W_1 \le \delta_1$ and $W_0 + W_2 \le \delta_2$. On the other hand, if source aggregation is applied, then two bounds are maintained by the stream processor: a bound $[L_{Q1}, H_{Q1}]$ on $V_{Q1} = V_0 + V_1$ and a bound $[L_{Q2}, H_{Q2}]$ on $V_{Q2} = V_0 + V_2$, where $W_{Q1} \le \delta_1$ and $W_{Q2} \le \delta_2$.

If all stream transmission costs are equal, then cost is determined by the sum of the streamed update frequencies of all bounds, which can be estimated using our random walk model from Appendix E. The streamed update frequency for a bound of width W_i on the value V_i is $F_i \approx \frac{(2 \cdot s_i)^2}{(W_i)^2}$, where s_i is the random walk step size of V_i. Applying the law that variances are additive for sums of independent random variables [GKP89], we can compute the streamed update frequency for the source-aggregated bounds $F_{Q1} \approx \frac{((2 \cdot s_0)^2 + (s \cdot s_1)^2)}{(W_{Q1})^2}$ and $F_{Q2} \approx \frac{((2 \cdot s_0)^2 + (s \cdot s_2)^2)}{(W_{Q2})^2}$. Therefore, the overall stream transmission cost if source aggregation is not applied is $\mathcal{C}_n = F_0 + F_1 + F_2$, and the overall cost if source aggregation is applied is $\mathcal{C}_p = F_{Q1} + F_{Q2}$.

Suppose $\delta_1 \gg \delta_2$, so there is a large disparity in the precision constraints of the two queries. For this purpose we can treat δ_1 as ∞, and it is easy to verify mathematically that for any step sizes and any bound widths that meet the constraints, $\mathcal{C}_p \le \mathcal{C}_n$ so source aggregation achieves lower overall stream transmission cost. On the other hand, suppose $s_0 \gg s_1$ and $s_0 \gg s_2$, so value V_0 involved in both queries changes much more rapidly than V_1 and V_2, which are each only involved in a single query. If we take the extreme case where $s_1 = s_2 = 0$, then if source aggregation is not performed any reasonable width allocation strategy will assign $W_1 = 0$, $W_2 = 0$, and $W_0 = \min\{\delta_1, \delta_2\}$, and we can derive that $\mathcal{C}_n \le \mathcal{C}_p$.