Self-Tuning, Bandwidth-Aware Monitoring for Dynamic Data Streams

Navendu Jain†, Praveen Yalagandula‡, Mike Dahlin⋆, Yin Zhang⋆
†*Microsoft Research* ‡*HP Labs* ⋆*The University of Texas at Austin*

Abstract*— We present SMART, a self-tuning, bandwidth-aware monitoring system that maximizes result precision of continuous aggregate queries over dynamic data streams. While prior approaches *minimize bandwidth cost* under fixed precision constraints, they may still overload a monitoring system during traffic bursts. To facilitate practical deployment of monitoring systems, SMART therefore *bounds the worst-case bandwidth cost* for overload resilience. The primary challenge for SMART is how to dynamically select updates at each node to maximize query precision while keeping per-node monitoring bandwidth below a specified budget. To address this challenge, SMART's hierarchical algorithm (1) allocates bandwidth budgets in a near-optimal manner to maximize global precision and (2) self-tunes bandwidth settings to improve precision under dynamic workloads. Our prototype implementation of SMART provides key solutions to (a) prioritize pending updates for multi-attribute queries, (b) build bounded fan-in, load-aware aggregation trees to improve accuracy, and (c) combine temporal batching with arithmetic filtering to reduce load and to quantify result staleness. Our evaluation using simulations and a network monitoring application shows that SMART incurs low overheads, improves accuracy by up to an order of magnitude compared to uniform bandwidth allocation, and performs close to the optimal algorithm under modest bandwidth budgets.

I. INTRODUCTION

Distributed stream processing systems [1]–[3] must provide high performance and high fidelity for query processing as they grow in scale and complexity. In these systems, data streams are often bursty where input rates may unexpectedly increase [4]–[6]. Examples include network traffic monitoring, identifying distributed denial-of-service (DDoS) attacks on the Internet, financial stocks monitoring, web click stream analysis, and event-driven monitoring in sensor networks. Therefore, it is desirable for these systems to bound the monitoring load while still providing useful accuracy guarantees on the query results. Existing techniques [7]–[12] aim to address this problem by minimizing the monitoring cost while promising an *a priori* error bound (e.g., $\pm$ 10%) on query precision.

Unfortunately, although these techniques effectively reduce load, they are unsuitable in dynamic, high-volume data stream environments for three reasons.

(1) **Setting precision a priori requires workload knowledge:** Choosing error bounds *a priori* is unintuitive when workloads are not known in advance or may change unpredictably over time e.g., should the error be 10% or 30%? Conversely, it may be easier to set the monitoring budget e.g., a system administrator is willing to pay 0.1% of network bandwidth for monitoring.

(2) **Bad precision setting hurts performance:** A bad choice of the error bound may significantly degrade the quality of a query result when the error bound is too large [13] or incur a high communication and processing cost when the error bound is too small.

(3) **Bursty traffic imposes unacceptable overheads:** Even with reasonable error bounds, monitoring systems may still risk overload under bursty and often unpredictable traffic conditions as we illustrate with an example below.

To address these challenges, we propose that for many monitoring systems, the right approach is to fix the bandwidth cost while optimizing query precision. By bounding the worst case bandwidth cost, monitoring systems achieve three key benefits. First, they avoid need for a priori workload knowledge or keeping up with workload changes to optimally set precision bounds. Second, they can maximize precision during periods of low workload and provide a graceful degradation under high workloads. Third, they avoid risk of overload during traffic bursts and still be able to deliver results with useful accuracy. We motivate these benefits using an example.

Motivating Example: We present a simple example to illustrate the challenges for scalable monitoring under bursty workloads. We simulate a set of 10 data sources connected to a centralized monitor with an incoming bandwidth limit of 5 messages per second. The input workload distribution is modeled based on the standard exponential distribution with a parameter λ, and upon each arrival, the value of attribute a_i at data source i is updated according to random walk model in which the value either increases or decreases by an amount sampled uniformally from [0.5, 1.5]. Figure 1 shows the load-error tradeoff for a single data source. As expected, on increasing the error budget, arithmetic filtering [9], [10] quickly decreases the load as a majority updates get filtered.

To quantify the monitoring cost, we use λ=10 and compute a SUM aggregate under a baseline error budget of 2 and its corresponding expected load of 0.5 (Figure 1), effectively setting the total *expected* cost for monitoring 10 data sources at 5 messages per second. Figure 2 shows the *induced* message load at the central monitor under a fixed error budget of 2. We observe that under peak data arrivals, the system sometimes incurs message costs 4x higher than the expected cost of 5 messages per second. This deviation is due to bursty

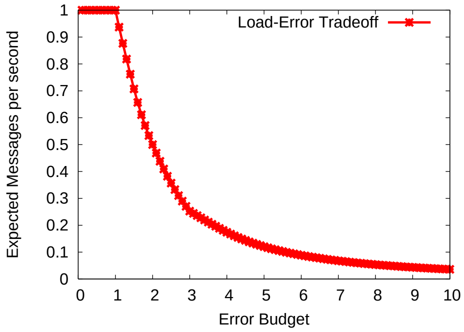

Fig. 1. Expected outgoing message load vs. error budget for a single attribute at a data source under a random walk workload.

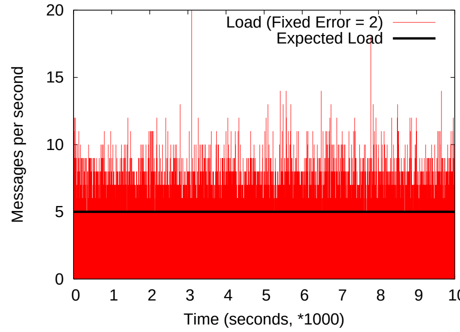

Fig. 2. Monitoring system induces overload under bursty workloads to bound result error. The system overload is up to 4x over expected load.

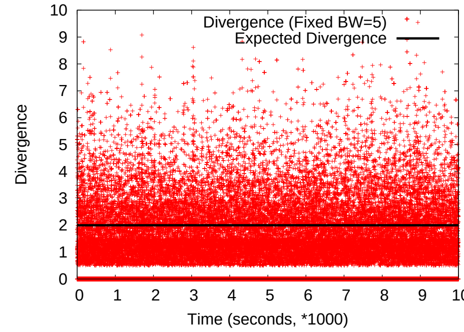

Fig. 3. Monitoring system causes high inaccuracy under bursty workloads to bound load. The error is up to 5x higher over expected divergence.

poisson arrivals—the probability of 2x, 4x, and 8x overload is 0.18, 0.015, and 9e-6, respectively, and worse for many other distributions. Conversely, Figure 3 shows the divergence between data source attributes and their cached values at the central monitor under a fixed load of 5. In this case, the system may provide highly inaccurate results having up to 5x more error than the expected error of 2. As a result, existing monitoring techniques are ineffective under bursty workloads—they risk overload, loss of accuracy, or both.

Our Contributions: Our goal is to bound the bandwidth cost while still delivering query results with useful accuracy. To achieve this goal, SMART provides three key techniques:

(1) **Maximize precision under fixed bandwidth budget:** SMART formulates a bandwidth-aware optimization problem whose goal is to maximize the result precision of an aggregate query in a hierarchical aggregation tree subject to given bandwidth constraints. This model provides a closed-form, near-optimal solution to self-tune bandwidth settings for achieving high accuracy using only local and aggregated information at each node in the tree. Further, SMART provides a "graceful degradation" in query precision as the available bandwidth decreases. Our experimental results show that self-tuning bandwidth settings improves accuracy by an order of magnitude over uniform bandwidth allocation and performs close to the optimal algorithm under modest bandwidth budgets.

(2) **Scalability via multiple aggregation trees:** SMART builds on recent work that uses distributed hash tables (DHTs) to construct scalable, load-balanced forests of self-organizing aggregation trees [14]–[16]. Scalability to tens of thousands of nodes and millions of attributes is achieved by mapping different attributes to different trees. For each tree in this forest of aggregation trees, SMART's self-tuning algorithm adjusts bandwidth settings to achieve high precision under dynamic workloads where the estimate of bandwidth vs. precision trade-offs, and hence the optimal distribution, can change over time.

(3) **High performance by combining arithmetic filtering and temporal batching:** SMART integrates temporal batching with arithmetic filtering to reduce the monitoring load and to quantify staleness of query results. For high-volume workloads, it is advantageous for nodes to batch multiple updates that arrive close in time and send a single combined update. In an aggregation tree, this temporal batching allows leaf sensors to condense a series of updates into a periodic report and allows internal nodes to combine updates from different subtrees before transmitting them further. Further, it bounds the delay (e.g., up to 30 seconds) from when an update occurs at a leaf until it is reported at the root.

We have implemented a prototype of SMART on PRISM [17], a scalable aggregation system built on top of FreePastry [18]. Our prototype provides two key optimizations. First, it constructs bounded fan-in, bandwidth-aware aggregation trees to improve accuracy and to quickly detect anomalies in heterogeneous environments. Recent studies [7], [10], [19]–[21] suggest that only a few attributes (e.g., network flows) generate a significant fraction of the total traffic in many monitoring applications. Thus, for fast anomaly detection, an aggregation tree should be able to quickly route important updates towards the root such that no internal node becomes a bottleneck due to either high in-degree or low bandwidth. Second, for multi-attribute queries, SMART provides a refresh schedule that selects and prioritizes attributes for refreshing in order to minimize the error in query results.

We evaluate SMART through extensive simulations and a real network monitoring application of detecting distributed heavy hitters. Experience with this application illustrates the improved precision and scalability benefits: for a given monitoring budget, SMART's adaptivity can significantly improve the query precision while monitoring a large number of attributes. Compared to uniform bandwidth allocation, SMART improves accuracy by up to an order of magnitude and provides accuracy within 27% of the optimal algorithm under modest bandwidth budgets.

Example Queries: For concreteness, we list several real-world application queries to illustrate the types of queries SMART was designed to support.

Q_1 Identify the *heavy hitter* network flows that send the highest traffic in aggregate across all network endpoints.

Q_2 Find the top-k ports across all nodes that have been heavily scanned in the recent past, possibly indicating worm activity.

Q_3 Monitor anomaly conditions e.g., SUM(nodes sensing fire) $\geq$ threshold, MAX(chemical concentration) in a physical sensor network.

Q_4 Monitor the top-k popular web objects in a wide-area content distribution network such as Akamai.

All these aggregate queries require processing a large number of rapid update streams in limited bandwidth/battery-life environments, and all can benefit from SMART. In Section VI, we show results for Q_1 using a network monitoring application we have implemented.

In summary, this paper makes the following contributions.

- We identify the key limitations of previous "fix error, minimize load" techniques, and we address them by bounding the worst-case load while still providing query results with useful accuracy.
- We describe a practical, self-tuning, bandwidth-aware monitoring system that adapts bandwidth budgets to maximize precision of continuous aggregate queries under high-volume, dynamic workloads.
- Our implementation provides key optimizations to improve accuracy and to quickly detect anomalies in heterogeneous environments.
- Our evaluation demonstrates that SMART provides a key substrate for scalable monitoring: it provides high accuracy in dynamic environments and performs close to the optimal algorithm under modest bandwidth budgets.

The rest of this paper is organized as follows. Section II discusses related work. Section III describes PRISM [17], a scalable DHT-based aggregation system, and precision-performance tradeoffs that underlie SMART. Section IV describes the SMART adaptive algorithm that self-tunes bandwidth settings to improve result accuracy. Section V and VI present the implementation and experimental evaluation of SMART. Finally, Section VII highlights our conclusions.

II. Related Work

SMART builds upon prior "fix error, minimize load" techniques [7]–[12], but it departs in three significant ways driven by our focus on achieving overload resilience for practical, scalable monitoring. First, SMART reformulates the key optimization problem, which we believe is an important contribution. While prior approaches minimize cost under fixed precision constraints, bursty, high-volume workloads may still overload a monitoring system as discussed in Section I. In fact, it is precisely during these abnormal events that a monitoring system needs to avoid overload but still deliver results with useful accuracy in real-time. SMART therefore bounds the worst-case system cost to provide overload resilience. Second, the technical advances to solve this new problem are significant. While SMART uses Chebyshev inequality [10] to capture the load vs. error trade-off, it faces new constraints of limited bandwidth budgets both at a parent and each of its children in a hierarchical aggregation tree. Given these constraints, SMART self-tunes bandwidth settings in a near-optimal manner to achieve high accuracy under dynamic workloads. Third, SMART's prototype provides key solutions to prioritize updates, build bandwidth-aware trees, and combine temporal batching with arithmetic filtering.

Recently, load shedding techniques [5], [22] have been proposed to handle overload conditions. The key idea is to carefully drop some tuples to reduce processing load but at the expense of reducing query result accuracy. These approaches handle load spikes assuming either the CPU [5], [22] or main memory [23] to be the key resource bottleneck. In comparison, bandwidth is the primary resource constraint in SMART.

Best-effort cache synchronization techniques [24], [25] aim to minimize the divergence between source data objects and cached copies in a *one-level* tree. In these techniques, each object is refreshed individually. In comparison, SMART performs hierarchical query processing for scalability and correlates updates to the same attribute at different data sources to maximize the accuracy of an aggregate query result. To our knowledge, there has not been prior work on maximizing precision of aggregate queries in hierarchical trees under limited bandwidth.

Babcock and Olston [26] focus on efficiently computing top-k aggregate values given fixed error in a one-level tree but do not consider how to maximize precision of top-k results under fixed bandwidth in hierarchical trees. Silberstein et al. propose a sampling-based approach at randomly chosen time steps for computing top-k queries in sensor networks [27]. Their approach focuses on returning the k nodes with the highest sensor readings. In comparison, SMART focuses on large-scale aggregate queries such as distributed heavy hitters which require computing a global aggregate value for each of the tens of thousands to millions of attribute across all the nodes in the network.

Sketches are small-space data structures that provide approximate answers to aggregate queries [28]. These techniques, however, require error bounds to be set a priori to provide the approximation guarantees.

III. Point of Departure

SMART extends PRISM [17] which embodies two key abstractions for scalable monitoring: aggregation and DHT-based aggregation. SMART then introduces controlled tradeoffs between precision bounds and monitoring bandwidth.

A. DHT-based Hierarchical Aggregation

Aggregation is a fundamental abstraction for scalable monitoring [12], [14]–[16], [20] because it allows applications to access summary views of global information and detailed views of rare events and nearby information.

SMART's aggregation abstraction defines a tree spanning all nodes in the system. As Figure 4 illustrates, in PRISM each physical node is a leaf, and each subtree represents a logical group of nodes. An internal non-leaf node, which we call a *virtual node*, is simulated by a physical leaf node of

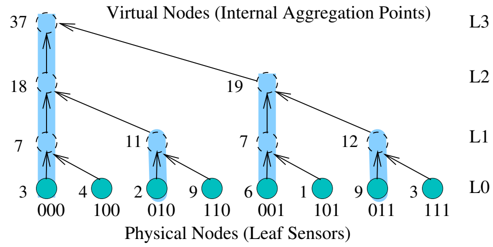

Fig. 4. The aggregation tree for key 000 in an eight node system. Also shown are the aggregate values for a simple SUM() aggregation function.

the subtree rooted at the virtual node. Figure 4 illustrates the computation of a simple SUM aggregate.

SMART leverages DHTs [16], [29], [30] to construct a forest of aggregation trees and maps different attributes to different trees for scalability. DHT systems assign a long (e.g., 160 bits), random ID to each node and define a routing algorithm to send a request for ID i to a node $root_i$ such that the union of paths from all nodes forms a tree *DHTtree*$_i$ rooted at the node *root*$_i$. By aggregating an attribute with ID i = hash(attribute) along the aggregation tree corresponding to *DHTtree*$_i$, different attributes are load balanced across different trees. This approach can provide aggregation that scales to a large number of nodes and attributes [14], [15].

B. Query Result Approximation

SMART quantifies the precision of query results in terms of numeric error between the reported result and the actual value. We define the numeric approximation of a query result using Arithmetic Imprecision [9]–[11], [31]. *Arithmetic imprecision* (AI) deterministically bounds the numeric difference between a reported value of an attribute and its true value. For example, an AI of 10% ensures that the reported value either underestimates or overestimates the true value by at most 10%.

When applications do not need exact answers [9]–[12], arithmetic imprecision reduces load by filtering updates that lie within the AI error range of the last reported value. Next, we describe how SMART uses the AI mechanism to enforce the numeric error bounds while reducing bandwidth load.

Mechanism: We first describe the basic mechanism for enforcing AI for each aggregation subtree in the system.

To enforce AI, each aggregation subtree T for an attribute has an error budget δ_T that defines the maximum inaccuracy of any result the subtree will report to its parent for that attribute. The root of each subtree divides this error budget among itself δ_{self} and its children δ_c (with $\delta_T \geq \delta_{self} + \sum_{c \in children} \delta_c$), and the children recursively do the same. Note that δ_c determines the child's error filter to cull as many updates as possible before sending them to the parent, and δ_{self} is useful for applying additional filtering after combining all updates received from the children. Here we present the AI mechanism for the SUM aggregate since it is likely to be common in network monitoring [9], [10] and financial applications [32]; other standard aggregation functions (e.g., MAX, MIN, AVG, etc.) are similar and defined precisely in a technical report [17].

This arrangement reduces system load by filtering small updates that fall within the range of values cached by a subtree's parent. In particular, after a node A with error budget δ_T reports a range $[V_{min}, V_{max}]$ for an attribute value to its parent (where $V_{max} \leq V_{min} + \delta_T$), if the node A receives an update from a child c, the node A can skip updating its parent as long as it can ensure that the true value of the attribute for the subtree lies between V_{min} and V_{max}, i.e., if

$$\begin{array}{lcl} V_{min} & \leq & \sum_{c \in children} V^c_{min} \\ V_{max} & \geq & \sum_{c \in children} V^c_{max} \end{array} \qquad (1)$$

where $[V^c_{min}, V^c_{max}]$ denote the latest update received from c.

SMART maintains per-attribute δ_T values so that different attributes with different error requirements and different update patterns can use different δ budgets in different subtrees.

C. Case-study Application

To guide the system development of SMART and to drive our performance evaluation, we have built a case-study application using the PRISM aggregation framework: a distributed heavy hitter detection service. Distributed Heavy Hitters (DHH) detection is important for both detecting network traffic anomalies such as DDoS attacks, botnet attacks, and flash crowds as well as for accounting and bandwidth provisioning [19].

We define heavy hitters as entities that account for a significant proportion of the total activity measured in terms of number of packets, bytes, connections, etc. [19] in a distributed system—for example, the 100 IPs that account for the most incoming traffic in the last 10 minutes [19]. To answer this distributed query, the key challenge is scalability for aggregating per-flow statistics for tens of thousands to millions of concurrent flows across all the network endpoints in real-time. For example, a subset of the Abilene [33] traces used in our experiments includes 80 thousand flows that send about 25 million updates per hour.

To scalably compute the global heavy hitters list, we chain two aggregations where the results from the first feed into the second. First, PRISM calculates the total incoming traffic for each destination address from all nodes in the system using SUM as the aggregation function and hash(HH-Step1, destIP) as the key. For example, tuple (H = hash(HH-Step1, 72.179.58.7), 900 KB) at the root of the aggregation tree T_H indicates that a total of 900 KB of data was received for 72.179.58.7 across all vantage points in the network during the last time window. In the second step, we feed these aggregated total bandwidths for each destination IP into a SELECT-TOP-100 aggregation with key hash(HH-Step2, TOP-100) to identify the TOP-100 heavy hitters among all flows.

In Section VI we show how SMART's self-tuning algorithm adapts bandwidth settings to monitor a large number of attributes and provides high result accuracy by filtering majority of mice flows [19] (attributes with low frequency) while prioritizing updates for the heavy-hitter flows; we expect typical monitoring applications to have a large number of mice flows but only a few heavy hitters [7], [10], [19]–[21].

IV. SMART DESIGN

In this section we present the SMART design and describe our self-tuning algorithm that adapts bandwidth settings at each node to maximize query precision.

A. System Model

We focus on distributed stream processing environments with a large number of data sources that perform in-network aggregation to compute continuous aggregate queries over incoming data streams. The bandwidth resources for query processing may be limited at a number of points in the network. In particular, a node j's *outgoing bandwidth* may be constrained (B_j^O), a node j's *incoming bandwidth* may be constrained (B_j^I), or both. Note that constraining the incoming bandwidth bounds (a) the control traffic overhead for monitoring and (b) the CPU processing load for computing the aggregation function across incoming data inputs. Finally, these bandwidth capacities may vary among nodes in heterogeneous environments and with time if bandwidth is shared with other applications.

At any time, each attribute's numeric value is bounded by an AI error δ. If δ is small, then updates may frequently drive an attribute's value out of its last reported range $[V_{min}, V_{max}]$, forcing the system to send messages to update the range. A system can, however, reduce its bandwidth requirements by increasing δ. Thus, if a hierarchical aggregation system has bandwidth constraints, it should determine a δ value at each aggregation point that meets the bandwidth constraints into and out of that point, and it should select these δ values so as to minimize the total AI for the attribute. In particular, rather than splitting each node's incoming bandwidth evenly among its children, the system should attempt to assign bandwidth to where it will do the most good by reducing the resulting imprecision.

In the rest of this section, we describe the SMART algorithm for minimizing imprecision while meeting bandwidth constraints in four steps.

First, we describe a simplified algorithm for a one-level aggregation tree and static workloads. For each leaf node i in the system, this algorithm calculates an ideal error setting δ_i and corresponding expected bandwidth consumption b_i such that (1) each leaf node's outbound bandwidth (i.e., rate of updates sent to the root) is at most its outgoing bandwidth budget, (2) the root node's incoming bandwidth (i.e., sum of update rates inbound from children) is at most its incoming bandwidth budget, and (3) the sum of the δ_i's is minimized given the first two constraints.

Second, we describe how to handle dynamic workloads where the estimate of AI error δ vs. bandwidth trade-offs, and hence the optimal distribution, can change over time. A key challenge here is throttling the rate at which the system redistributes bandwidth budgets across nodes since such redistribution also incurs bandwidth costs.

Third, we generalize the algorithm to handle multi-level aggregation trees.

Symbol	Meaning
B_i^O	outgoing bandwidth constraint for node i
B_i^I	incoming bandwidth constraint for node i
u_i	input update rate at node i
σ_i	standard deviation of node i's input workload
$child(i)$	all children of node i
δ_i	node i's AI error setting
b_i	node i's outgoing bandwidth load
δ_i^{opt}	node i's optimal AI error setting
b_i^{opt}	node i's optimal outgoing bandwidth load

TABLE I
SUMMARY OF KEY NOTATIONS.

Finally, we discuss how our implementation copes with variability. In particular, SMART sets the per-node δs so that the *average* bandwidth meets a target. However, spikes of update load for an attribute or coincident updates for multiple attributes could cause *instantaneous* bandwidth to exceed the target. To avoid such instantaneous overload, SMART therefore prioritizes pending updates based on the impact they will have on their aggregate values and drains them to the network at the target rate, similar to broadcast scheduling [34].

B. One-Level Tree

Quantify Bandwidth vs. AI Precision Tradeoff: To estimate the optimal distribution of load budgets among different nodes, we use a simple load vs. error tradeoff model based on Chebyshev inequality [10]. This model quantifies query precision that can be achieved under a given bandwidth budget.

Let X be a random variable with finite expectation μ and variance σ^2. Chebyshev's inequality states that for any $k \geq 0$,

$$Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{2}$$

For AI filtering, the term $k\sigma$ represents the error budget δ_i for node i. Substituting for k in Equation 2 gives:

$$Pr(|X - \mu| \geq \delta_i) \leq \frac{\sigma_i^2}{\delta_i^2} \tag{3}$$

This equation implies that if the error budget is smaller than the standard deviation (i.e., $\delta_i \leq \sigma_i$ and $k \leq 1$), then δ_i is unlikely to filter many data updates. In this case, Equation 3 provides only a weak bound on the message cost: the probability that each incoming update will trigger an outgoing message is bounded by 1. However, if $\delta_i \geq k\sigma_i$ for any $k \geq 1$, the fraction of unfiltered updates is probabilistically bounded by $\frac{\sigma_i^2}{\delta_i^2}$. In general, given the input update rate u_i for node i with error budget δ_i, the expected message cost for node i per unit time is:

$$M_i = \text{MIN}\left\{u_i,\ \frac{\sigma_i^2}{\delta_i^2} * u_i\right\} \tag{4}$$

We use the expected message cost M_i to estimate node i's optimal outgoing bandwidth b_i^{opt}.

Estimate Optimal Bandwidth Settings under Fixed Load: To estimate the optimal load distribution and settings of AI error δs at each node in a one-level tree rooted at node r, we formulate an optimization problem of minimizing the total error for a SUM aggregate at root r under given bandwidth budgets B_r^I (at the root) and B_i^O (at child i):

$$\begin{aligned} &\text{MIN} \sum_{i \in child(r)} \delta_i^{opt} \\ &\text{s.t.} \sum_{i \in child(r)} \frac{\sigma_i^2 * u_i}{(\delta_i^{opt})^2} \leq B_r^I \\ &\forall i \in child(r), \quad b_i^{opt} = \frac{\sigma_i^2 * u_i}{(\delta_i^{opt})^2} \leq \min\{B_i^O, u_i\} \end{aligned} \tag{5}$$

where b_i^{opt} denotes the estimated optimal setting of outgoing bandwidth of node i to meet the global objective of minimizing the total error $\sum_{i \in child(r)} \delta_i^{opt}$ subject to two constraints: (1) node i's outgoing bandwidth budget $\left(b_i^{opt} \leq \min\{B_i^O, u_i\}\right)$ and (2) incoming bandwidth budget at root r $\left(\sum_{i \in child(r)} b_i^{opt} \leq B_r^I\right)$.

Enforcing Incoming Bandwidth Constraint: To solve Equation (5), we first relax the outgoing bandwidth constraints (i.e., $\forall i, \ b_i^{opt} \leq \min\{B_i^O, u_i\}$) but enforce the incoming bandwidth constraint. Later, we provide a solution when the outgoing bandwidth constraints are also enforced. Using Lagrangian Multipliers, the above formulation yields a closed-form and computationally inexpensive solution [17]:

$$\delta_i^{opt} = \sqrt{\frac{\sum_{c \in child(r)} \sqrt[3]{\sigma_c^2 * u_c}}{B_r^I} * \sqrt[3]{\sigma_i^2 * u_i}} \tag{6}$$

which provides a closed-form formula for setting b_i^{opt}:

$$b_i^{opt} = \frac{\sigma_i^2 * u_i}{(\delta_i^{opt})^2} = B_r^I * \frac{\sqrt[3]{\sigma_i^2 * u_i}}{\sum_{c \in child(r)} \sqrt[3]{\sigma_c^2 * u_c}} \tag{7}$$

As a simple example, if $u_i = \sigma_i = 1$, then each node sets the same δ_i for the attribute as $\delta_i = \sqrt{\frac{N}{B_r^I}}$, and the bandwidth budget for node i will be $\frac{1}{\delta_i^2} = \frac{B_r^I}{N}$ i.e., given a total bandwidth budget and uniform workload distribution across nodes, each node gets a uniform share of the bandwidth to update each attribute.

Note that to set b_i^{opt} (Equation (7)), each node needs to know $\sum_{c \in child(r)} \sqrt[3]{\sigma_c^2 * u_c}$ and root r's incoming bandwidth budget B_r^I; SMART computes a simple SUM aggregate across children inputs to obtain this information. As a simple optimization, these messages are piggy-backed on data updates.

Enforcing Outgoing Bandwidth Constraints: Note that the above load assignment assumes that the outgoing bandwidth constraint $b_c^{opt} \leq \min\{B_c^O, u_c\}$ holds for every child c. If for a node i, the above solution doesn't satisfy $b_i^{opt} \leq \min\{B_i^O, u_i\}$, then we need to set $b_i^{opt} = \min\{B_i^O, u_i\}$ to satisfy its bandwidth constraint. This situation may arise in heterogeneous environments where a subset of nodes may experience higher input loads (e.g., DDoS attacks) or may become severely resource-constrained (e.g., sensor networks with low power devices).

Setting $b_i^{opt} = \min\{B_i^O, u_i\}$ can free up part of r's incoming capacity B_r^I, which can be reassigned to other children to increase their bandwidth budget thereby improving the overall accuracy. SMART therefore applies an iterative algorithm that in each iteration determines all *saturated* children C_{sat} at each step, fixes their load budgets and error settings, and recomputes Equations (6), (7) for all the remaining children (assuming child set C_{sat} is absent). A child j is labeled saturated if $b_j^{opt} \geq B_j^O$ i.e., its available outgoing bandwidth is at most the estimated optimal setting of its outgoing bandwidth. Note that for our DHT-based aggregation trees, the fan-in for a node is typically 16 (i.e., a 4-bit correction per hop) so the iterative algorithm runs in constant time (at most 16 times).

C. Self-Tuning Bandwidth Settings

The above solution is derived assuming that σ_i and u_i are given and remain constant. In practice, σ_i and u_i may change over time depending on the workload characteristics. SMART therefore self-tunes bandwidth settings to improve accuracy.

Cost-Benefit Throttling: We apply cost-benefit throttling to dynamically adapt the bandwidth settings. Specifically, after computing the new bandwidth budgets, a node computes a *charge* metric for each attribute a which estimates the reduction in error gained by refreshing a:

$$charge_a = (T_{curr} - T_{lastSent}^a) * D_a \tag{8}$$

where T_{curr} is the current time, $T_{lastSent}^a$ is the last time an attribute a's update was sent, and D_a denotes the deviation between a's current value at that node and its last reported range $[V_{min}, V_{max}]$ to the parent. For example, if a's AI error range cached at the parent is [1, 2] and a new update with value 11 arrives at a child node, we expand the range to [1,11] at the child to include the new value setting D_a =10.

Notice that an attribute's charge will be large if (i) there is a large error imbalance (i.e., D_a is large), or (ii) there is a long-lasting imbalance (e.g., $T_{curr} - T_{lastSent}^a$ is large). Further, only using D_a may hurt precision if the attribute has a repeated behavior of quickly diverging after the last refresh. Therefore, the temporal term ($T_{curr} - T_{lastSent}^a$) prioritizes attributes who are likely to again diverge slowly after being refreshed thereby giving a long-term precision benefit.

Since redistribution also consumes bandwidth budgets, we only send messages to readjust the bandwidth settings when doing so is likely to reduce the time-averaged error for those attributes by at least a threshold τ (i.e., if $charge_a > \tau$). Further, to ensure the invariant that a node does not exceed its bandwidth while sending updates for multiple attributes, we present a simple prioritization technique in Section IV-E.

D. Multi-Level Trees

To scale to a large number of nodes, we extend our basic algorithm for a one-level tree to a distributed algorithm for a multi-level aggregation hierarchy. Note that in a multi-level tree, leaf nodes use AI error δ to filter sensor updates and internal nodes retain a local AI error, δ_{self}, to help prevent updates received from their children from being propagated further up the tree [10], [35].

In an aggregation tree, each internal node applies the self-tuning algorithm similar to the one-level tree case: the internal node is a local root for each of its immediate children, and the bandwidth targets B_p^I for parent p and B_c^O for each child c are input as constraints in Equation (5). The key difference is that for children who are internal nodes, we use their AI error δ_{self} in this optimization framework. To estimate the optimal bandwidth and AI error settings for each child, the parent node p tracks its incoming update rate i.e., the aggregate number of messages sent by all its children per time unit (u_p) and the standard deviation (σ_p) of updates received from its children. Note that u_c, σ_c reports are accumulated by child c until they can be piggy-backed on an update message sent to its parent.

Given this information, a parent estimates for each child c, the optimal outgoing bandwidth b_c^{opt} and the AI error $\delta_{c(self)}^{opt}$ that improves the total precision in the aggregate value computed across the children given bandwidth constraints[1].

E. Prioritizing Pending Updates

Finally, we describe how our implementation addresses the challenge posed by variability in bandwidth targets. In particular, we want to avoid situations where a node meets its *average* bandwidth target but may temporarily exceed *instantaneous* bandwidth target due to spikes of update load for an attribute, coincident updates for multiple attributes, or a sudden increase in available bandwidth.

Each node needs to send messages to the root to minimize the query error. Therefore, we need to prioritize sending those messages that benefit precision the most. A naive refreshing algorithm that updates attributes in a round-robin fashion could easily spend network messages that do not reduce error while consuming valuable bandwidth resource. Limiting sending of non-useful updates is a particular concern for applications like DHH that monitor a large number of attributes, only a few of which are active enough to be worth optimizing.

To address this problem, our SMART implementation provides a priority heap of all pending updates that need to be sent ordered by their priority. Using this heap, SMART prioritizes pending updates based on the impact they will have on their aggregate values and drains them to the network at a target rate. Specifically, at each time step, a node keeps removing the maximum priority attribute from the heap and sending it to the corresponding parent until the node's instantaneous target bandwidth is reached. Note that to compare priorities across different attributes that have different value ranges across different queries, we normalize an attribute a's refresh priority (Equation (8)) by dividing the deviation D_a by the standard deviation σ_a of that attribute.

[1]Note that finding a globally optimal solution is too expensive in this environment as it requires perfect knowledge of (possibly varying) (1) per node bandwidth settings, (2) network topology, and (3) update rate and variance of input workload at all times. Therefore, SMART adapts bandwidth in a best-effort, self-tuning manner to achieve high precision in hierarchical trees and our experiments show that this approach works well in practice.

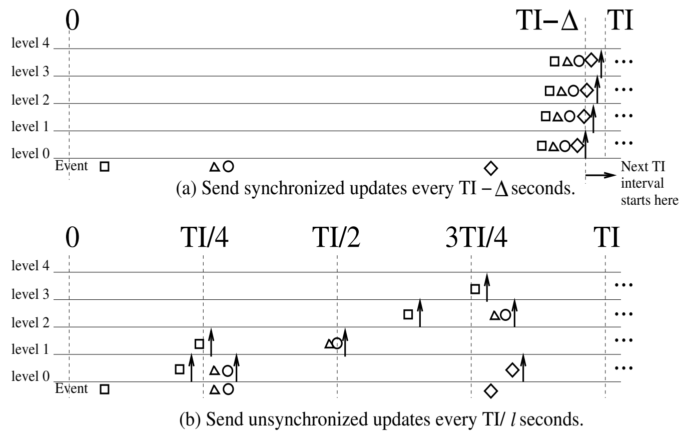

Fig. 5. For a given TI bound, pipelined delays with synchronized clocks (a) allows nodes to send less frequently than unpipelined delays without synchronized clocks (b).

V. SMART Implementation

In this section, we describe the prototype implementation of SMART. First, we present a key performance optimization of temporal batching of updates to reduce load and to quantify staleness of query results. Second, we describe how to build bandwidth-aware aggregation trees to improve accuracy in heterogeneous network environments. Then, we describe how SMART handles failures and reconfigurations. Finally, we discuss how to improve precision for different aggregates.

A. Temporal Imprecision

SMART integrates temporal imprecision with arithmetic filtering to provide staleness guarantees on query results and to reduce the monitoring load.

Temporal Imprecision (TI) bounds the delay from when an event/update occurs until it is reported [8], [9], [11], [36]. A temporal imprecision of *TI* (e.g., TI = 30 seconds) guarantees that every event that occurred *TI* or more seconds ago is reflected in the reported result; events younger than *TI* may or may not be reflected. In SMART, each attribute has a TI interval during which its updates are batched into a combined message, checked if the combined update drives the aggregate value out of the last reported AI range, and then pushed into the priority queue to be sent to the parent.

Temporal imprecision benefits monitoring applications in two ways. First, it accounts for inherent network and processing delays in the system; given a worst-case per-hop cost hop_{max} even immediate propagation provides a temporal guarantee no better than $\ell * hop_{max}$ where ℓ is the maximum number of hops from any leaf to the root of an aggregation tree. Second, explicitly exposing TI allows SMART to reduce load by using temporal batching: a set of updates at a leaf sensor are condensed into a periodic report or a set of updates that arrive at an internal node over a time interval are combined into a single message before being sent further up the tree [17]. This temporal batching improves scalability by reducing processing and network load as we show using experiments on a network monitoring application in Section VI.

SMART implements TI using a simple mechanism of having each node send updates to its parent once per TI/ℓ seconds similar to TAG [8] as shown in Figure 5(b). Further, to maximize the possibility of batching updates, when clocks are synchronized[2], SMART pipelines delays across tree levels so that each node sends once every $(TI-\Delta)$ seconds with each level's sending time staggered so that the updates from level i arrive just before level $i+1$ can send (Figure 5(a)). The term Δ accounts for the worst-case per hop delays and maximum clock skew; details are in the extended technical report [17].

B. Bandwidth-Aware Tree Construction

As described in Section III, SMART leverages DHTs [16], [29], [30], [38] to construct a forest of aggregation trees and maps different attributes to different trees [14]–[16], [39] for scalability and load balancing. SMART then uses these trees to perform in-network aggregation.

SMART constructs bounded fan-in, bandwidth-aware aggregation trees to improve result accuracy and quickly detect anomalies in heterogeneous environments. Recent studies [7], [10], [19]–[21] suggest that only a few attributes (e.g., elephant flows [19]) generate a significant fraction of the total traffic in many monitoring applications. Thus, to provide fast anomaly detection, an aggregation tree should quickly route the updates of elephant flows towards the root such that no internal node becomes a processing/communication bottleneck due to either high in-degree or low bandwidth.

DHTs provide different degrees of flexibility in choosing neighbors and next-hop paths in building aggregation trees [40]. Many DHT implementations [18], [30] use proximity (usually round-trip latency) in the underlying network topology to select neighbors in the DHT overlay. This neighbor selection in turn determines the parent-child relationships in the aggregation tree. However, in a heterogeneous environment where different nodes have different bandwidth budgets, an aggregation tree formed solely based on RTTs may degrade the quality precision of the query result for two reasons. First, a node may not have sufficient outgoing bandwidth to send updates up in the tree even though its underlying tree may be sufficiently well-provisioned. In such an environment, this node becomes a bottleneck as the updates sent by the underlying subtree go wasted. Second, a resource-limited parent may not be able to process the aggregate outgoing update rate of all its children. Thus, a practical technique for building trees would be to bound the number of children at each internal node and use both latencies and bandwidth constraints to select the best parent at each tree level.

To improve accuracy and to quickly identify anomalies, SMART builds DHT-based aggregation trees as follows:

- Bound the fan-in (i.e., number of children) at each parent node. In our implementation, a child node selects its parent such that the fan-in at the parent is at most 16 i.e., each parent has maximum up to 16 children.

[2]Algorithms in the literature can achieve clock synchronization among nodes to within one millisecond [37].

- Use both network latency and available bandwidth capacity as the proximity metric for selecting parent nodes. SMART orders DHT neighbors of a node such that they have the highest incoming bandwidth capacity and have network latency below a specified threshold. Thus, nodes close in proximity and having high bandwidth capacities are highly likely to be selected as parent nodes.

Finally, note that in some environments, it might be useful to select nodes with low bandwidth as parents e.g., if the input workload at the leaf nodes comes from an independent uniform distribution, then a node closer to the root is expected to receive very few updates since an aggregate (e.g., SUM) is likely to become more "stable" going towards the root. However, in practice, real workloads (1) are often non-uniform with few attributes generating a significant fraction of the total traffic [19] and (2) exhibit both temporal and spatial skewness with input rates unexpectedly increasing over time and across nodes. We quantify the effectiveness of constructing bounded fan-in, bandwidth-aware aggregation trees in Section VI.

C. Robustness

Failures and reconfigurations are common in distributed systems. As a result, a query might return a stale answer when nodes whose inputs are needed to compute the aggregate result become unreachable. Furthermore, in a large-scale monitoring system, such failures can interact badly with our techniques for providing scalability—hierarchy, arithmetic filtering, and temporal batching. E.g., if a subtree is silent over an interval, it is difficult to distinguish between two cases: (a) the subtree has sent no updates because the inputs have not significantly changed or (b) the inputs have significantly changed but the subtree is unable to transmit its report. As a result, reported results can deviate arbitrarily from the truth.

Addressing this problem of node failures and network disruptions in large-scale monitoring systems is beyond the scope of this paper. In a separate paper [41], we describe how a new consistency metric called Network Imprecision (NI) safeguards the accuracy of query results in the face of failures, network disruptions, and system reconfigurations.

D. Discussion

Improving precision for different aggregates: SMART focuses on the SUM aggregate since it is likely to be common in network monitoring [9], [10] and financial applications [32]; maximizing precision for the AVG aggregate is similar to SUM. For MAX, MIN aggregates, if the AI error budget is fixed, then the best error assignment is to give equal AI error budget to all the leaf nodes. However, since bandwidth budget is limited in practice, each node may set its precision differently to meet its available bandwidth. An intuitive solution to compute global MAX, MIN values is to simply broadcast them to each node, and a node sends an update only if it changes the global aggregate. However, this approach may limit scalability in large-scale data stream systems that monitor a large number of dynamic attributes. For the TOP-K aggregate, SMART achieves high accuracy by prioritizing updates based on the

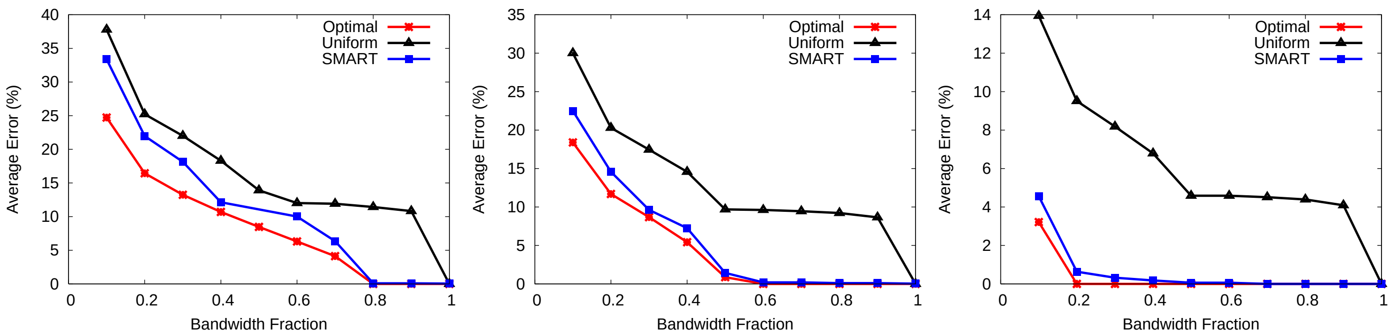

Fig. 6. SMART provides higher precision benefits as skewness in a workload increases. The figures show the average error vs. load budget for three different skewness settings (a) 20:80%, (b) 50:50%, and (c) 90:10%.

highest aggregate values and the result deviation D_a. Our experiments in Section VI show that this approach works well in practice. We plan to develop mechanisms for other aggregates such as quantiles in future work.

Bounding Query Result Divergence: SMART aims to deliver results with useful accuracy under limited bandwidth resources. Some applications, however, may require guaranteed upper bounds on the divergence of query results e.g., setting a threshold on the total traffic sent by a distributed experiment in PlanetLab [42]. To accommodate such cases, SMART allows an attribute to be tagged as a *hard precision limit* attribute, and it uses STAR [10] that adapts given AI error budgets to minimize bandwidth cost for such attributes.

VI. Experimental Evaluation

In this section, we present the precision and scalability results of an experimental study of SMART. First, we use simulations to evaluate the improvement in query result accuracy due to SMART's adaptive bandwidth settings. Second, we quantify the scalability and accuracy achieved by SMART for the DHH application in a network monitoring implementation. To perform this experiment, we have implemented a prototype of SMART using the PRISM aggregation system [17] on top of FreePastry [29]. We use Abilene [33] netflow traces and perform the evaluation on 120 node instances mapped to 30 machines in the department Condor cluster. Finally, we investigate the precision benefits of constructing bandwidth-aware aggregation trees using our prototype.

A. Simulation Experiments

In this section, we quantify the result accuracy achieved by SMART compared to uniform bandwidth allocation and an idealized optimal algorithm. First, we assess the effectiveness of adaptive bandwidth settings in improving result precision as skewness in a workload increases. Second, we analyze the effect of fluctuating bandwidth. Finally, we evaluate SMART for different workloads.

In all experiments, all active sensors are at the leaf nodes of an aggregation tree. Each sensor generates a data value every time unit (round) for two sets of synthetic workloads for 100,000 rounds: (1) a random walk pattern in which the value either increases or decreases by an amount sampled uniformally from [0.5, 1.5], and (2) a Gaussian distribution with standard deviation 1 and mean 0. We simulate $m \in \{100, 1000\}$ data sources each having $n \in \{10, 100\}$ attributes in one-level and multi-level trees, and under fixed and fluctuating bandwidth. All attributes have equal weights, messages have the same size, and each message uses one unit of bandwidth.

Evaluating Update Rate Skewness: First, we evaluate the precision benefits of SMART as skewness in a workload increases. We compare it with (1) the optimal algorithm under the idealized and unrealistic model of perfect global knowledge of each attribute's divergence at each data source and (2) a uniform allocation policy where the incoming bandwidth capacity B at a parent is allocated equally ($\frac{B}{C}$) among its child set C. Although this simple policy is correct (the total incoming load from the children is guaranteed to never exceed the incoming load of their parent), it is not generally the best policy as we show in our experiments.

We first perform a simple experiment for a one-level tree and later show the results for hierarchical topologies. Figure 6 shows the query precision achieved by SMART for m=100 and n=10 under the following skewness settings: (a) 20:80%, (b) 50:50%, and (c) 90:10%. For example, the 20:80% skewness represents that a randomly selected 20% attributes are updated with probability 0.01 while the remaining ones are updated consistently every round under the random walk model. In all subsequent graphs in this section, the x-axis denotes the bandwidth budget as a fraction of the total cost $m.n$ of refreshing all the attributes across all nodes; the y-axis shows the resulting average error.

For 20:80% skewness, since only a small fraction of attributes are stable, SMART can only reclaim up to 20% load budget from stable attributes sources and distribute it to dynamic sources to reduce their error. For small bandwidth budgets, SMART improves accuracy by up to 35% compared to uniform allocation. The optimal algorithm improves accuracy by 27% over SMART. As the load budget increases, SMART converges to the optimal solution. SMART improves error by 40% over uniform allocation under 20% load budget and by more than an order of magnitude under sufficiently large budgets. For the 50:50 case, SMART can reclaim 50% of the total load budget compared to uniform allocation and give it to unstable sources. In this case, SMART reduces error

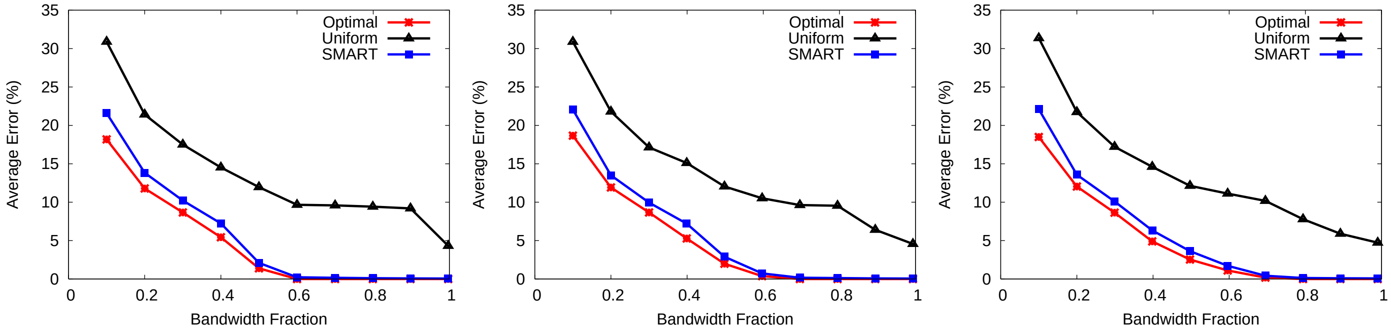

Fig. 7. SMART improves accuracy even under fluctuating bandwidth. The maximum bandwidth variation in the figure is (a) 10%, (b) 20%, and (c) 30%.

by up to 50% over uniform policy at 40% bandwidth and achieves accuracy close to the optimal solution. Finally, for 90% skewness, SMART achieves the accuracy of the optimal algorithm even under 20% fraction of the total bandwidth and improves accuracy by more than an order of magnitude over uniform allocation. We observed qualitatively similar results for other m and n settings.

Note that the advantages of SMART depend on the workload skewness. We expect that for systems monitoring a large numbers of attributes (e.g., DHH), some attributes (e.g., elephants) will have a high variability in data values and update rates so they gain only a modest advantage in accuracy from SMART, while other attributes (e.g., mice) will have large ratios and hence, a query (e.g., top-k) will gain large advantages since it only needs to provide high accuracy for the top-k flows.

Effect of Fluctuating Bandwidth: Next, we evaluate the effectiveness of SMART under fluctuating bandwidth. We vary the incoming bandwidth over time following a sine wave pattern and set the maximum rate of bandwidth change to 10%, 20%, and 30% for m=1000 nodes each having n=100 attributes. We use update rate skewness of 50% as described above. From Figure 7, we observe that under 10% variation, SMART provides 50% reduction in error over uniform allocation at 40% bandwidth fraction. As we increase the bandwidth fluctuation from 10% to 30%, SMART reduces error by about 70% under 40% bandwidth fraction, and more than an order of magnitude for larger fractions. In all cases, SMART achieves accuracy close to the optimal algorithm.

Evaluating Different Workloads: Finally, we evaluate SMART under different configurations by varying input workload, standard deviation (step sizes), and update frequency at each node. The workload data distribution is generated from a random walk pattern and Gaussian models. For standard deviation/step-size, 70% nodes have uniform parameters as previously described; the remaining 30% nodes have these parameters set proportional to *rank* (i.e., with locality) or randomly assigned (i.e., no locality) from the range [0.5, 150].

Figure 8 shows the corresponding results for different settings of input workload and standard deviation for m=100 and n=100. The update frequency is set to 0.7 skewness as described previously. We make three key observations. First,

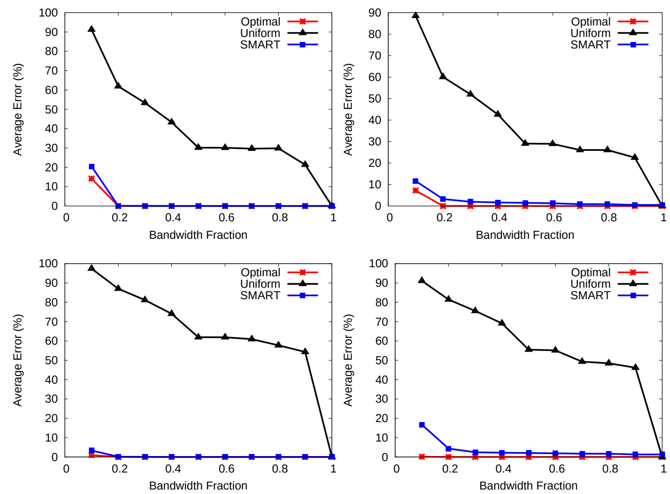

Fig. 8. Precision benefits of SMART vs. optimal algorithm and uniform allocation for different {workload, step sizes/standard deviation} configurations: (a) random walk, rank, (b) random walk, random, (c) Gaussian, rank, and (d) Gaussian, random.

SMART's error is close to the optimum algorithm under the rank based assignment as shown in Figure 8(a),(c). Second, under random assignment, SMART's accuracy benefits are reduced since updates generated from within the same subtree are likely uncorrelated. In both cases, as bandwidth increases, SMART's error approaches the optimal. Third, because step-sizes are based on node rank, SMART prioritizes attributes having higher step-sizes and update rates, and applies cost-benefit throttling to ensure that the precision benefits exceed costs. The uniform policy, however, does not make such a distinction equally favoring all attributes that need to be refreshed, thereby ineffective in improving precision. Finally, under limited bandwidth, refreshing mice attributes with small step sizes does not significantly reduce the result error for queries such as top-k heavy hitters but consumes valuable bandwidth resources. For all these configurations, SMART improves accuracy by up to an order of magnitude over uniform allocation. The optimal approach improves accuracy by 20% over SMART.

Overall, across all configurations in Section VI-A, SMART reduces inaccuracy by up to an order of magnitude compared to uniform allocation and is within 27% of the optimal algorithm under modest bandwidth fraction.

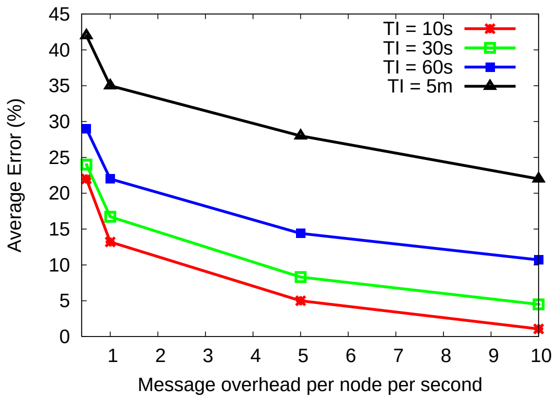

Fig. 9. SMARTS provides high precision benefits under limited bandwidth budgets for the DHH application.

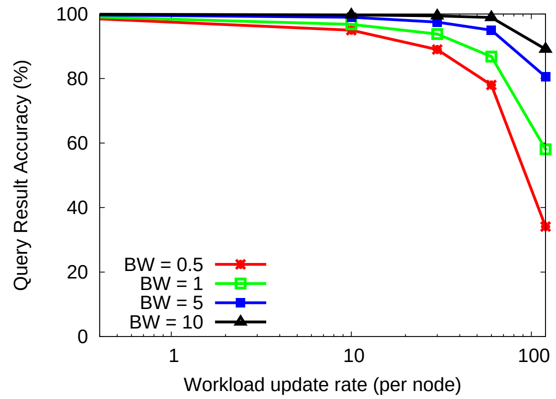

Fig. 10. SMART provides a graceful degradation in query precision as the input load increases. The x-axis is on a log-scale.

B. Testbed Experiments

Next, we quantify the improvement in query precision due to SMART's self-tuning algorithm for the DHH application.

We use multiple netflow traces obtained from the Abilene [33] backbone network. The traces were collected from 3 Abilene routers for 1 hour; each router logged per-flow data every 5 minutes, and we split these logs into 120 buckets based on the hash of source IP. As described in Section III-C, our DHH application executes a TOP-100 heavy hitter query on this dataset for tracking the top 100 flows (destination IP as key) in terms of bytes received over a 30 second moving window shifted every 10 seconds. We analyze this workload and observe that the 120 sensors track roughly 80,000 flows and send around 25 million updates in an hour [17]. Further, it shows a heavy-tailed Zipf-like distribution: 60% flows send less than 1 KB of aggregate traffic, 90% flows less than 55 KB, and 99% of the flows send less than 330 KB during the 1-hour run; the maximum aggregate flow value is about 179.4 MB. We observe a similar heavy-tail distribution for the number of updates per flow (attribute) [17].

For this experiment, we fix the outgoing bandwidth to between 0.5 and 10 messages per node per second. Since we bound the fan-in of an internal node in our DHT-based aggregation tree to 16, the maximum incoming load at any node is thus 160 messages per second which is a reasonable processing load in our environment. Figure 9 plots the outgoing load per node on the x-axis and the result precision achieved for the TOP-100 heavy hitter query on the y-axis. The different lines in the graph correspond to a temporal batching interval of 10 seconds, 30 seconds, 60 seconds, and five minutes. Each data point denotes the average result divergence for the TOP-100 heavy hitters set. It is important to note that for this query, the result sets (TOP-100 heavy hitters) under TI of 10, 30, and 60 seconds were consistent with the true result set. However, under TI of 5 minutes, the result sets differed in at most three entries from the true result set.

We observe that under TI = 10s, SMART reduces the average error from 22% at 0.5 load budget to about 1% at a load budget of 10. As expected, comparing across TI values, the average error increases with increase in the TI batching interval. In all cases, the 99-percentile CPU and memory overhead of SMART was less than 8% and 43 MB, respectively.

Figure 10 shows the query precision achieved by SMART as the input load increases given fixed bandwidth budgets (BW = 0.5, 1, 5, and 10) and TI = 10s. As load increases, SMART degrades accuracy gracefully across all bandwidth settings. Further, as bandwidth increases, SMART improves accuracy from 32% (BW = 0.5) to 87% (BW = 10) at a load of about 100 updates per second.

C. Evaluating Bandwidth-aware Tree Formation

Finally, we evaluate the benefits of SMART's bandwidth-aware tree construction. As described in Section V, SMART uses both bandwidth and latency as proximity metrics in DHT routing compared to only latency in traditional DHT implementations. For quantitative comparison, we compute a *Tree-BW* metric for a tree as the sum of $L_i * B_i$ across all nodes, where L_i is the number of leaves in the subtree rooted at node i and B_i is the bandwidth of node i. This weighted sum metric is higher for trees that select internal nodes having higher bandwidth capacities. For this experiment, we classify nodes belonging to two different classes of bandwidth budgets: 100Mbps and 1Mbps. A 0.1 skewness in bandwidth implies that 10% of the nodes have 100Mbps bandwidth and the remaining have 1Mbps bandwidth. Figure 11 compares the benefits of SMART's tree construction using bandwidth-aware DHT routing against trees constructed using bandwidth-unaware routing for a 1024-node system; the y-axis shows the normalized Tree-BW (with respect to the maximum value observed) metric for various bandwidth skewness settings (x-axis). Note that SMART's bandwidth-aware DHT routing achieves better Tree-BW metric values by up to a factor of 3.7x over a bandwidth-unaware DHT.

In summary, our results show that SMART is an effective substrate for scalable monitoring: SMART incurs low overheads, provides high result accuracy while bounding the monitoring load, continuously adapts to dynamic workloads, and achieves significant precision benefits for an important monitoring application of detecting distributed heavy hitters.

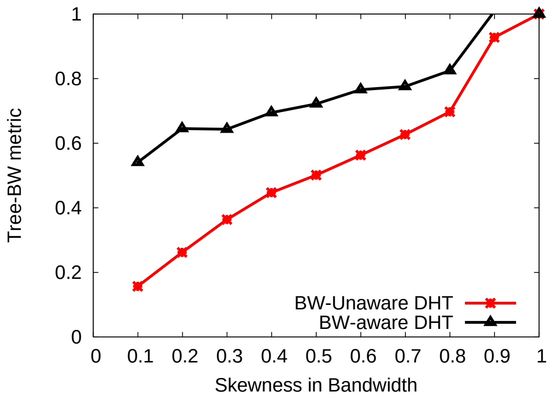

Fig. 11. Bandwidth skewness vs. normalized Tree-BW metric for bandwidth-aware and -unaware DHTs.

VII. Conclusions and Future Work

We design, implement, and evaluate SMART—a scalable, adaptive, bandwidth-aware monitoring system that performs self-tuning of bandwidth budgets to maximize precision of continuous aggregate queries over dynamic data streams. Without adapting bandwidth constraints, monitoring systems may risk overload, loss of accuracy, or both, under bursty workloads.

In future work, we plan to examine techniques for secure information aggregation across multiple administrative domains and build a broad range of monitoring applications that can benefit from SMART.

Acknowledgments

We thank Joe Hellerstein, Adam Silberstein and the anonymous reviewers for their valuable feedback. Navendu Jain is supported by an IBM Ph.D. Fellowship. This work is supported in part by NSF Awards CNS-0546720, CNS-0627020, SCI-0438314, and EIA-0303609.

References

[1] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the stream processing core," in *SIGMOD*, 2006.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Centintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005.

[3] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly-available, fault-tolerant, parallel dataflows," in *SIGMOD*, 2004.

[4] J. Kleinberg, "Bursty and hierarchical structure in streams," in *KDD*, 2002.

[5] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *ICDE*, 2004.

[6] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive Control of Extreme-Scale Stream Processing Systems," in *ICDCS*, 2006.

[7] R. Keralapura, G. Cormode, and J. Ramamirtham, "Communication-efficient distributed monitoring of thresholded counts," in *SIGMOD*, 2006.

[8] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," in *OSDI*, 2002.

[9] C. Olston, J. Jiang, and J. Widom, "Adaptive filters for continuous queries over distributed data streams," in *SIGMOD*, 2003.

[10] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang, "STAR: Self-tuning aggregation for scalable monitoring," in *VLDB*, 2007.

[11] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *TOCS*, 2002.

[12] R. van Renesse, K. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *TOCS*, 2003.

[13] R. Cheng, B. Kao, S. Prabhakar, A. Kwan, and Y. Tu, "Adaptive stream filters for entity-based queries with non-value tolerance," in *VLDB*, 2005.

[14] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," in *SIGCOMM*, 2004.

[15] A. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," in *SIGCOMM*, 2004.

[16] C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," in *SPAA*, 1997.

[17] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang, "PRISM: Precision-Integrated Scalable Monitoring (extended)," UT Austin Department of Computer Sciences, Tech. Rep. TR-07-15, 2007.

[18] "FreePastry," `http://freepastry.rice.edu`.

[19] C. Estan and G. Varghese, "New directions in traffic measurement and accounting." in *SIGCOMM*, 2002.

[20] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, "Querying the Internet with PIER," in *VLDB*, 2003.

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

[22] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing," in *VLDB*, 2007.

[23] Y. Hu, D. M. Chiu, and J. C. Lui, "Adaptive flow aggregation - a new solution for robust flow monitoring under security attacks," in *NOMS*, 2006.

[24] C. Olston and J. Widom, "Best-effort cache synchronization with source cooperation," in *SIGMOD*, 2002.

[25] J. Cho and H. Garcia-Molina, "Synchronizing a database to improve freshness," in *SIGMOD*, 2000.

[26] B. Babcock and C. Olston, "Distributed top-k monitoring," in *SIGMOD*, 2003.

[27] A. Silberstein, R. Braynard, C. Ellis, K. Munagala, and J. Yang, "A sampling-based approach to optimizing top-k queries in sensor networks," in *ICDE*, 2006.

[28] G. Cormode and M. Garofalakis, "Sketching streams through the net: distributed approximate query tracking," in *VLDB*, 2005.

[29] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems," in *Middleware*, 2001.

[30] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for Internet applications," in *SIGCOMM*, 2001.

[31] C. Olston and J. Widom, "Offering a precision-performance tradeoff for aggregation queries over replicated data," in *VLDB*, 2000.

[32] R. Gupta and K. Ramamritham, "Optimized query planning of continuous aggregation queries in dynamic data dissemination networks," in *WWW*, 2007.

[33] `http://abilene.internet2.edu/`.

[34] D. Aksoy and M. J. Franklin, "Scheduling for large-scale on-demand data broadcasting," in *INFOCOM*, 1998.

[35] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos, "Hierarchical in-network data aggregation with quality guarantees," in *EDBT*, 2004.

[36] A. Singla, U. Ramachandran, and J. Hodgins, "Temporal notions of synchronization and consistency in Beehive," in *SPAA*, 1997.

[37] D. Veitch, S. Babu, and A. Pasztor, "Robust synchronization of software clocks across the Internet," in *IMC*, 2004.

[38] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content Addressable Network," in *SIGCOMM*, 2001.

[39] M. J. Freedman and D. Mazires, "Sloppy Hashing and Self-Organizing Clusters," in *IPTPS*, 2003.

[40] K. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The Impact of DHT Routing Geometry on Resilience and Proximity," in *SIGCOMM*, 2003.

[41] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang, "Network Imprecision: A new consistency metric for scalable monitoring," in *OSDI*, 2008.

[42] `http://www.planet-lab.org`.