STAR: Self-Tuning Aggregation for Scalable Monitoring*

Navendu Jain, Dmitry Kit, Prince Mahajan, Praveen Yalagandula†, Mike Dahlin, and Yin Zhang
Department of Computer Sciences, University of Texas at Austin
†Hewlett-Packard Labs, Palo Alto, CA

ABSTRACT

We present STAR, a self-tuning algorithm that adaptively sets numeric precision constraints to accurately and efficiently answer continuous aggregate queries over distributed data streams. Adaptivity and approximation are essential for both robustness to varying workload characteristics and for scalability to large systems. In contrast to previous studies, we treat the problem as a *workload-aware* optimization problem whose goal is to minimize the total communication load for a multi-level aggregation tree under a fixed error budget. STAR's hierarchical algorithm takes into account the update rate and variance in the input data distribution in a principled manner to compute an optimal error distribution, and it performs cost-benefit throttling to direct error slack to where it yields the largest benefits. Our prototype implementation of STAR in a large-scale monitoring system provides (1) a new *distribution mechanism* that enables self-tuning error distribution and (2) an optimization to reduce communication overhead in a practical setting by carefully distributing the initial, default error budgets. Through extensive simulations and experiments on a real network monitoring implementation, we show that STAR achieves significant performance benefits compared to existing approaches while still providing high accuracy and incurring low overheads.

1. INTRODUCTION

This paper describes STAR, a *self-tuning, adaptive* algorithm for setting numeric precision constraints that processes continuous aggregate queries in a large-scale monitoring system.

Scalable system monitoring is a fundamental abstraction for large-scale networked systems. It serves as a basic building block for applications such as network monitoring and management [8, 18, 42], financial applications [3], resource scheduling [20,41], efficient multicast [38], sensor networks [20, 41], resource management [41], and bandwidth provisioning [13]. To provide a real-time view of global system state for these monitoring and control applications, the central challenge for a monitoring system is scalability to process queries over multiple, continuous, rapid, time-varying data streams that generate updates for thousands or millions of dynamic attributes (e.g., per-flow or per-object state) spanning tens of thousands of nodes.

Recent studies [27, 29, 35, 38, 43] suggest that real-world applications often can tolerate some inaccuracy as long as the maximum error is bounded and further indicate that small amounts of approximation error can provide substantial bandwidth reductions. However, setting a static error budget *a priori* is both difficult when workloads are not known in advance and inefficient when workload characteristics change unpredictably over time. Therefore, for many applications that require processing of long-running or continuous queries over data streams, it is important to consider adaptive approaches for query processing [2, 5, 7, 29].

A fundamental observation behind this work is that an adaptive algorithm for setting error budgets should embody three key design principles:

- **Workload-Driven Approach:** First, to provide a general and flexible framework for adaptive error distribution for different workloads, we need a solution that does not depend on *a priori* knowledge about the input data distribution. Rather, a self-tuning algorithm should be based on first principles and use the workload itself to guide the process of dynamically adjusting the error budgets.
- **Cost-Benefit Throttling:** Second, rather than continuously redistributing error budgets in pursuit of a perfect distribution, our algorithm explicitly determines when the current distribution is close enough to optimal that sending messages to redistribute allocations will likely cost more than it will ultimately save given the measured variability of the workload. For example, in a network monitoring service for detecting elephant flows (i.e., attributes with high frequency [13]), we track bandwidth for tens of thousands of flows, but the vast majority of these flows are mice that produce so few updates that further redistribution of the error budgets is not useful. Avoiding fruitless optimization in such cases significantly improves scalability in systems with tens of thousands of attributes.
- **Aggregation Hierarchy:** Third, STAR distributes error budgets hierarchically among both the internal nodes and the leaves in an aggregation tree. Our primary goal is to provide a global view of the system by processing and aggregating data from distributed data streams in real-time. In such an environment, a hierarchical decentralized query processing infrastructure provides an attractive solution to minimize communication and computation costs for both scalability and load balancing. Further, in a hierarchical aggregation tree, the internal nodes can not only split the error budget among their children but may also

*This work is supported in part by the NSF grants CNS-0546720, CNS-0627020, CNS-0615104, and EIA-0303609. Navendu Jain is supported by an IBM Ph.D. Fellowship.

VLDB '07, September 23-28, 2007, Vienna, Austria.

retain some local error budget to prevent updates received from children from being propagated further up the tree e.g., when the net effect of aggregating two or more updates is to cancel each other out.

Unfortunately, existing approaches do not satisfy these requirements. On one hand, protocols such as adaptive filters [4, 29] and adaptive thresholded counts [25] can effectively reduce communication overhead given a fixed error budget for flat (2-tier) topologies, but they offer neither scalability to a large number of nodes and attributes nor the benefits of in-network aggregation. On the other hand, existing hierarchical protocols either assign a static error budget [28] that cannot adapt to changing workloads, or periodically shrink error thresholds [11] at each node to create redistribution error budget, thus incurring a high load when monitoring a large number of attributes. Finally, although the previous solutions have an intuitive appeal, their problem formulation does not explicitly account for varying workloads (e.g., variance, update rate.) In contrast, STAR's self-tuning solution addresses the global optimization problem of minimizing the total communication overhead under dynamic workloads.

To address these challenges, STAR's self-tuning, hierarchical approach yields three key properties:

- **High Performance:** To compute optimal error assignments, STAR formulates an optimization problem whose goal is to minimize the global communication load in an aggregation tree given a fixed total error budget. This model provides a closed-form, optimal solution for adaptive setting of error budgets using only local and aggregated information at each node in the tree. Given the optimal error budgets, STAR performs cost-benefit throttling to balance the tradeoff between the cost for redistributing the error budgets and the expected benefits. Our experimental results show that self-tuning distribution of error budgets can reduce monitoring costs by up to a factor of five over previous approaches.
- **Scalability:** STAR builds on recent work that uses distributed hash tables (DHTs) to construct scalable, load-balanced forests of self-organizing aggregation trees [6, 14, 31, 41]. 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, STAR's self-tuning algorithm directs error slack to where it is most needed.
- **Convergence:** STAR computes optimal error budgets and performs cost-benefit analysis to continually adjust to dynamic workloads. But since STAR's self-tuning algorithm adapts its solution as the input workload changes, it is difficult to qualify its convergence properties. Nonetheless, STAR guarantees convergence under stable workloads and shows good convergence empirically for dynamic workloads. Further, STAR balances the speed of adaptivity and robustness to workload fluctuations.

We study the performance of our algorithm through both simulations and measurements of a prototype implementation on our SDIMS aggregation system [41] built on top of FreePastry [15]. Experience with a Distributed Heavy Hitter detection (DHH) application built on STAR illustrates how explicitly managing numeric imprecision can qualitatively enhance a monitoring service. Our experimental results show the improved performance and scalability benefits: for the DHH application, small amounts of numeric imprecision drastically reduce monitoring load. For example, given a 10% error budget, STAR reduces network load by an order of magnitude compared to the uniform allocation policy. Further, for 90:10 skewness in attribute load (i.e., 10% heavy hitters), STAR achieves more than an order of magnitude better performance than both uniform error allocation and approaches that do not perform cost-benefit throttling.

This paper makes three key contributions. First, we present STAR, the first self-tuning algorithm for scalable aggregation that computes optimal error distribution and performs cost-benefit throttling for large-scale system monitoring. Second, we provide a scalable implementation of STAR in our SDIMS monitoring system. Our implementation provides a new *distribution abstraction*, a dual mechanism to the traditional bottom-up tree based aggregation, that enables *self-tuning* error distribution top-down along the aggregation tree to reduce communication load. Further, it provides an important optimization that reduces communication overhead in a practical setting by carefully distributing the initial, default error budgets. Third, our evaluation demonstrates that adaptive error distribution is vital for enabling scalable aggregation: a system that performs self-tuning of error budgets can significantly reduce communication overheads.

The rest of this paper is organized as follows. Section 2 provides background description of SDIMS [41], a scalable DHT-based aggregation system at the core of STAR. Section 3 describes the mechanism and the STAR self-tuning algorithm for adaptively setting numeric imprecision for reducing monitoring overhead. Section 4 presents the implementation of STAR in our SDIMS aggregation system, techniques for maintaining precision of query results under failures, and policies for initializing the error budgets. Section 5 presents the experimental evaluation of STAR. Finally, Section 6 discusses related work, and Section 7 provides conclusions.

2. BACKGROUND

STAR extends SDIMS [41] which embodies two key abstractions for scalable monitoring: aggregation and DHT-based aggregation.

2.1 Aggregation

Aggregation is a fundamental abstraction for scalable monitoring [6, 14, 20, 31, 38, 41] because it allows applications to access summary views of global information and detailed views of rare events and nearby information.

SDIMS's aggregation abstraction defines a tree spanning all nodes in the system. As Figure 1 illustrates, each physical node in the system is a leaf and each subtree represents a logical group of nodes. Note that logical groups can correspond to administrative domains (e.g., department or university) or groups of nodes within a domain (e.g., a /28 subnet with 14 hosts on a LAN in the CS department) [17, 41]. An internal non-leaf node, which we call a *virtual node*, is simulated by one or more physical nodes at the leaves of the subtree rooted at the virtual node.

SDIMS's tree-based aggregation is defined in terms of an aggregation function installed at all the nodes in the tree. Each leaf node (physical sensor) inserts or modifies its local value for an *attribute* defined as an {attribute type, attribute name} pair which is recursively aggregated up the tree. For

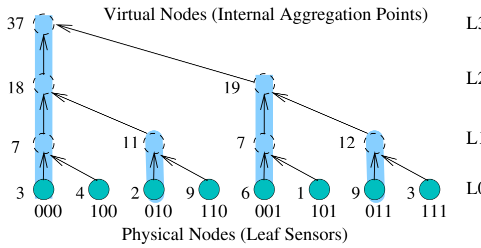


Figure 1: The aggregation tree for key 000 in an eight node system. Also shown are the aggregate values for a simple SUM() aggregation function.

each level-i subtree T_i in an aggregation tree, SDIMS defines an *aggregate value* $V_{i,attr}$ for each attribute: for a (physical) leaf node T_0 at level 0, $V_{0,attr}$ is the locally stored value for the attribute or NULL if no matching tuple exists. The aggregate value for a level-i subtree T_i is the result returned by the aggregation function computed across the aggregate values of T_i's children. Figure 1, for example, illustrates the computation of a simple SUM aggregate.

2.2 DHT-Based Aggregation

SDIMS leverages DHTs [31–33,37,44] to construct a forest of aggregation trees and maps different attributes to different trees [6, 14, 31, 34, 41] for scalability and load balancing. DHT systems assign a long (e.g., 160 bits), random ID to each node and define a routing algorithm to send a request for key k to a node $root_k$ such that the union of paths from all nodes forms a tree $DHTtree_k$ rooted at the node $root_k$. By aggregating an attribute with key k = hash(attribute) along the aggregation tree corresponding to $DHTtree_k$, different attributes are load balanced across different trees. Studies suggest that this approach can provide aggregation that scales to large numbers of nodes and attributes [6, 14, 31, 34, 41].

2.3 Example Application

Aggregation is a building block for many distributed applications such as network management [42], service placement [16], sensor monitoring and control [27], multicast tree construction [38], and naming and request routing [9]. In this paper, we focus on a case-study example: a distributed heavy hitter detection service. We study several other examples elsewhere [23].

Our case-study application is identifying heavy hitters[1] in a distributed system—for example, the top 10 IPs that account for a significant fraction of total incoming traffic in the last 10 minutes [13]. The key challenge for this distributed query is scalability for aggregating per-flow statistics for tens of thousands to millions of concurrent flows in real-time. For example, a subset of the Abilene [1] traces used in our experiments include 80 thousand flows that send about 25 million updates per hour.

[1]Note that the standard definition of a heavy hitter is an entity that accounts for at least a specified proportion of the total activity measured in terms of number of packets, bytes, connections, etc [13]. We use a slightly different definition of "heavy hitters" to denote flows whose bandwidth is greater than a specified fraction threshold of the maximum flow value.

To scalably compute the global heavy hitters list, we chain two aggregations where the results from the first feed into the second. First, SDIMS 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, 128.82.121.7), 700 KB) at the root of the aggregation tree T_H indicates that a total of 700 KB of data was received for 128.82.121.7 across all vantage points during the last time window. In the second step, we feed these aggregated total bandwidths for each destination IP into a SELECT-TOP-10 aggregation with key hash(HH-Step2, TOP-10) to identify the TOP-10 heavy hitters among all flows.

Although there exist other centralized monitoring services, in Section 5 we show that using our STAR self-tuning algorithm in the SDIMS aggregation system, we can monitor a larger number of attributes at much finer time scales while incurring significantly lower network costs.

3. STAR DESIGN

Arithmetic imprecision (AI) deterministically bounds the numeric difference between a reported value of an attribute and its true value [23, 30, 43]. For example, a 10% AI bound ensures that the reported value either underestimates or overestimates the true value by at most 10%.

When applications do not need exact answers and data values do not fluctuate wildly, arithmetic imprecision can greatly reduce the monitoring load by allowing caching to filter small changes in aggregated values. Furthermore, for applications like distributed heavy hitter monitoring, arithmetic imprecision can completely filter out updates for most "mice" flows.

We first describe the basic mechanism for enforcing AI for each aggregation subtree in the system. Then we describe how our system uses a self-tuning algorithm to address the policy question of distributing an AI budget across subtrees to minimize system load.

3.1 Mechanism

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. Here we present the AI mechanism for the SUM aggregate since it is likely to be common in network monitoring and financial applications; other standard aggregation functions (e.g., MAX, MIN, AVG, etc.) are similar and defined precisely in an extended technical report [24].

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, 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

$$\begin{array}{lcl} V_{min} & \leq & \sum_{c \in children} V_{min}^c \\ V_{max} & \geq & \sum_{c \in children} V_{max}^c \end{array} \tag{1}$$

where V_{min}^c and V_{max}^c denote the most recent update received from child c.

Note the trade-off in splitting δ_T between δ_{self} and δ_c. A large δ_c allows a child to filter updates before they reach its parent. Conversely, by setting $\delta_{self} > 0$, a node can set $V_{min} < \sum V^c_{min}$, set $V_{max} > \sum V^c_{max}$, or both to avoid further propagating some updates it receives from its children.

SDIMS maintains per-attribute δ values so that different attributes with different error requirements and different update patterns can use different δ budgets in different subtrees. SDIMS implements this mechanism by defining a *distribution function*; just as an attribute type's aggregation function specifies how aggregate values are aggregated from children, an attribute type's distribution function specifies how δ budgets are distributed (partitioned) among the children and δ_{self}.

3.2 Policy Decisions

Given these mechanisms, there is considerable flexibility to (i) set δ_{root} to an appropriate value for each attribute (ii) compute V_{min} and V_{max} when updating a parent, and (iii) divide δ_T among δ_{self} and δ_c for each child c (Section 3.3.)

Setting δ_{root}: Aggregation queries can set the root error budget δ_{root} to any non-negative value. For some applications, an absolute constant value may be known a priori (e.g., count the number of connections per second ± 10 at port 1433.) For other applications, it may be appropriate to set the tolerance based on measured behavior of the aggregate in question (e.g., set δ_{root} for an attribute to be at most 10% of the maximum value observed) or the measurements of a set of aggregates (e.g., in our heavy hitter application, we set δ_{root} for each flow to be at most 1% of the bandwidth of the largest flow measured in the system.) Our mechanisms support all of these approaches by allowing new absolute δ_{root} values to be introduced at any time and then distributed down the tree via a distribution function. We have prototyped systems that use each of these three policies.

Computing $[V_{min}, V_{max}]$: When either $\sum_c V^c_{min}$ or $\sum_c V^c_{max}$ goes outside of the last $[V_{min}, V_{max}]$ that was reported to the parent, a node needs to report a new range. Given a δ_{self} budget at an internal node, we have some flexibility on how to center the $[V_{min}, V_{max}]$ range. Our approach is to adopt a per-aggregation-function range policy that reports $V_{min} = (\sum_c V^c_{min}) - bias * \delta_{self}$ and $V_{max} = (\sum_c V^c_{max}) + (1 - bias) * \delta_{self}$ to the parent. For example, we can set the *bias* ($\in [0, 1]$) parameter as follows:

- *bias* ≈ 0.5 if inputs are expected to be stationary
- *bias* ≈ 0 if inputs are expected to be increasing
- *bias* ≈ 1 if inputs are expected to be decreasing

For example, suppose a node with total δ_T of 10 and δ_{self} of 3 has two children reporting ($[V^c_{min}, V^c_{max}]$) of [1, 2] and [2, 8], respectively, and it reports [0, 10] to its parent. Then, suppose the first child reports a new range [10, 11], so the node must report to its parent a range that includes [12, 19]. If $bias = 0.5$, then the node reports [10.5, 20.5] to its parent to filter out small deviations around the current position. Conversely, if $bias = 0$, the node reports [12, 22] to filter out the maximal number of updates of increasing values.

3.3 Self-Tuning Error Budgets

The key AI policy question is how to divide a given error budget δ_{root} across the nodes in an aggregation tree.

A simple approach is a static policy that divides the error budget uniformly among all the children. E.g., a node with budget δ_T could set $\delta_{self} = 0.1\delta_T$ and then divide the remaining $0.9\delta_T$ evenly among its children. Although this approach is simple, it is likely to be inefficient because different aggregation subtrees may experience different loads.

To make cost/accuracy tradeoffs *self-tuning*, we provide an adaptive algorithm. The high-level idea is simple: increase δ for nodes with high load and large standard deviation but low δ (relative to other nodes); decrease δ for nodes with low load and small standard deviation but high δ. Next, we address the problem of optimal distribution of error budgets for a 2-tier (one-level) tree and later extend it as a general approach for a hierarchical aggregation tree.

3.3.1 One-Level Tree

Quantify AI Filtering Gain: To estimate the optimal distribution of error budgets among different nodes, we need a simple way of quantifying the amount of load reduction that can be achieved when a given error budget is used for AI filtering.

Intuitively, the AI filtering gain depends on the size of the error budget relative to the inherent variability in the underlying data distribution. Specifically, as illustrated in Figure 2, if the allocated error budget δ_i at node i is much smaller than the standard deviation σ_i of the underlying data distribution, δ_i is unlikely to filter many data updates. Meanwhile, if δ_i is above σ_i, we would expect the load to decrease quickly as δ_i increases until the point where a large fraction of updates are filtered.

To quantify the tradeoff between load and error budget, one possibility is to compute the entire tradeoff curve as shown in Figure 2. However, doing so imposes several difficulties. First, it is in general difficult to compute the tradeoff curve without a priori knowledge about the underlying data distribution. Second, maintaining the entire tradeoff curve becomes expensive when there are a large number of attributes and nodes. Finally, it is not easy to optimize the distribution of error budgets among different nodes based on the tradeoff curves.

To overcome these difficulties, we develop a simple metric in STAR to capture the tradeoff between load and error budget. Our metric utilizes Chebyshev's inequality in probability theory, which gives a bound on the probability of deviation of a given random variable from its mathematical expectation in terms of its variance. Let X be a random variable with finite mathematical 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:

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

Intuitively, this equation implies that if $\delta_i \leq \sigma_i$ i.e., the error budget is smaller than the standard deviation (implying $k \leq 1$), then δ_i is unlikely to filter many data updates (Figure 2.)

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 upper bounded by 1. However, if $\delta_i \geq k\sigma_i$ for any $k \geq 1$, the fraction of unfiltered

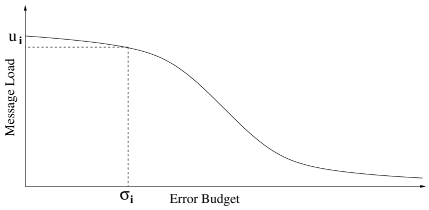


Figure 2: Expected message load vs. AI error budget.

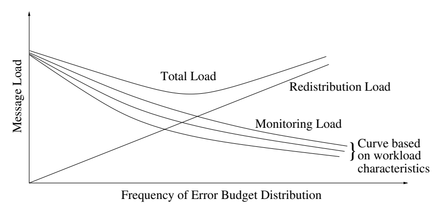


Figure 3: Cost-benefit analysis.

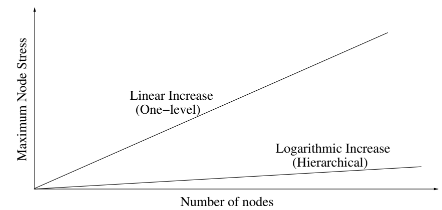


Figure 4: Maximum node stress: a one-level tree vs. a hierarchical tree.

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:

$$M_i = \text{MIN}\left(1, \frac{\sigma_i^2}{\delta_i^2}\right) * u_i \tag{4}$$

Compute Optimal Error Budgets: To estimate the optimal error distribution at each node, we can formulate an optimization problem of minimizing the total incoming network load at root R under a fixed total AI budget δ_T i.e.,

$$\begin{aligned} &\text{MIN} \sum_{i \in child(R)} \frac{\sigma_i^2 * u_i}{(\delta_i^{opt})^2} \\ &\text{s.t.} \sum_{i \in child(R)} \delta_i^{opt} = \delta_T \end{aligned} \tag{5}$$

Using Lagrange multipliers yields a closed-form and computationally inexpensive optimal solution [24]:

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

The above optimal error assignment assumes that for all the nodes, the expected cost per update is equal to $\frac{\sigma_i^2}{\delta_i^2}$ based on Equation 3 i.e., $\sigma_i \leq \delta_i$. However, for nodes with high σ_i relative to the error budget δ_i, it is highly likely that an update will be sent to the root for each incoming message. These *volatile* nodes [11] may not reap a significant benefit in spite of being allocated a large fraction of the error budget.

To account for volatile nodes, we apply an iterative algorithm that determines the largest volatile node j at each step and recomputes Equation 6 for all the remaining children assuming j is absent. A node j is labeled volatile if (1) $\frac{\sigma_j}{\delta_j^{opt}} \geq 1$ i.e., the standard deviation is larger than the optimal error budget (under fixed total budget) corresponding to Equation 3 and (2) the ratio $\frac{\sigma_j}{\delta_j^{opt}}$ is maximal among all the remaining children. If no such j exists, the procedure terminates giving the optimal AI budgets for each node; all volatile nodes get zero budget since any non-zero budget will not effectively filter their updates. 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.)

Relaxation: A self-tuning algorithm that adapts too rapidly may react inappropriately to transient situations. Therefore, we next apply exponential smoothing to compute the new error budget δ_i^{new} for each node as the weighted average of the new error budget (δ_i^{opt}) and the previous budget (δ_i):

$$\delta_i^{new} = \alpha \delta_i^{opt} + (1 - \alpha)\delta_i \tag{7}$$

where $\alpha = 0.05$.

Cost-Benefit Throttling: Finally, root R needs to send messages to its children to rebalance the error budget. Therefore, there is a tradeoff between the error budget redistribution overhead and the AI filtering gain (as illustrated in Figure 3.) A naive rebalancing algorithm that ignores such a tradeoff could easily spend more network messages redistributing δs than it saves by filtering updates. Limiting redistribution overhead 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 challenge, after computing the new error budgets, the root node computes a *charge* metric for each child c, which estimates the number of extra messages sent by c due to sub-optimal δ:

$$Charge_c = (T_{curr} - T_{adjust}) * (M_c - M_c^{new})$$

where $M_c = \frac{\sigma_c^2 * u_c}{\delta_c^2}$, $M_c^{new} = \frac{\sigma_c^2 * u_c}{(\delta_c^{new})^2}$, T_{curr} is the current time, and T_{adjust} is the last time δ was adjusted at R for child c. Notice that a subtree's charge will be large if (a) there is a large load imbalance (e.g., $M_c - M_c^{new}$ is large) or (b) there is a long-lasting imbalance (e.g., $T_{curr} - T_{adjust}$ is large.)

We only send messages to redistribute deltas when doing so is likely to save at least k messages (i.e., if $charge_c > k$). To ensure the invariant that $\delta_T \geq \delta_{self} + \sum_c \delta_c$, we make this adjustment in two steps. First, we replenish δ_{self} from the child whose δ_c is the farthest above δ_c^{new} by ordering c to reduce δ_c by Min(0.1 δ_c, δ_c - δ_c^{new}). Second, we loan some of the δ_{self} budget to the node c that has accumulated the largest charge by incrementing c's budget by Min(0.1δ_c, $\delta_c^{new} - \delta_c$, max(0.1$\delta_{self}$, δ_{self} - δ_{self}^{new})).

3.3.2 Multi-Level Trees

For large-scale multi-level trees, we extend our basic algorithm for a one-level tree to a distributed algorithm for a multi-level aggregation hierarchy. To reduce the maximum node stress (Figure 4) and the communication load, the internal nodes not only split δ_c among their children c but may also retain δ_{self} to help prevent updates received from their children from being propagated further up the tree.

At any internal node in the aggregation tree, the self-tuning algorithm works as follows:

1. *Estimate optimal distribution of δ_T across δ_{self} and δ_c.* Each 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 re-

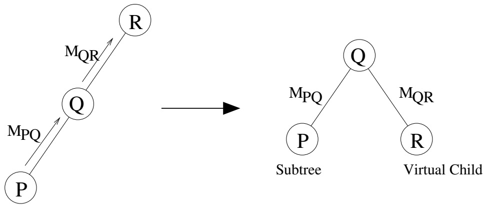


Figure 5: Q's local self-tuning view considers both incoming (M_{PQ}) and outgoing bandwidth (M_{QR}.)

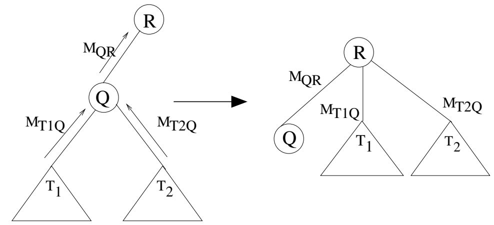


Figure 6: Global self-tuning view collapses a multi-level hierarchy to a one-level tree.

ceived 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 to its parent.

Given this information, each parent node n computes the optimal values δ_v^{opt} for each child v's underlying subtree that minimizes the total system load in the entire subtree rooted at n. We apply Equation 6 by viewing each child v as representing two individual nodes: (1) v itself (as a data source) with update rate u_v and standard deviation σ_v and (2) a node representing the subtree rooted at v. Figure 5 illustrates this local self-tuning view: when any internal node computes optimal budgets, it aims to minimize both incoming messages received as well as outgoing messages to parent (by showing parent-link as a virtual child) i.e., minimizing global communication load from a local perspective.

Given this model, we define *LoadFactor* for a node v as $\sqrt[3]{\sigma_v^2 * u_v}$. Recursively, we can define *AccLoadFactor* for a subtree rooted at node v as:

$$AccLoadFactor_v = \begin{cases} LoadFactor_v & \text{(v is a leaf node)} \\ LoadFactor_v + \\ \sum_{j \in child(v)} AccLoadFactor_j & \text{(otherwise)} \end{cases}$$

Next, we estimate the optimal error budget for v's subtree ($v \in child(n)$) as follows:

$$\delta_v^{opt} = \delta_T * \frac{AccLoadFactor_v}{AccLoadFactor_n} \tag{8}$$

Equation 8 is globally optimal since it virtually maps a multi-level hierarchy into a one-level tree (as illustrated in Figure 6) and in this transformed view, estimates the optimal error budget for each node.

To account for volatile nodes, we apply a similar iterative algorithm as in the one-level tree case to determine the largest volatile node i ($i \in child(n)$) at each step, recompute Equation 8 for all the remaining children, and so on. The condition to check whether node i is volatile becomes: $\frac{\sigma_i}{\delta_{i(self)}^{opt}} \geq 1$ and $\frac{\sigma_i}{\delta_{i(self)}^{opt}} \geq \frac{\sigma_j}{\delta_{j(self)}^{opt}} \forall j \in child(n)$ (remaining children) where $\delta_{i(self)}^{opt}$ is computed as:

$$\delta_{i(self)}^{opt} = \delta_T * \frac{LoadFactor_i}{AccLoadFactor_n} \tag{9}$$

2. *Relaxation: Adaptive adjustment of delta budgets.*

$$\delta_v^{new} = \alpha \delta_v^{opt} + (1 - \alpha)\delta_v$$

where $\alpha = 0.05$.

3. *Redistribute deltas iff the expected benefit exceeds the redistribution overhead.*

To do cost-benefit throttling, we recursively apply the formula for a one-level tree to compute the cumulative charge for a subtree rooted at node v:

$$AccCharge_v = \begin{cases} Charge_v & \text{(v is a leaf node)} \\ Charge_v + \\ \sum_{j \in child(v)} AccCharge_j & \text{(otherwise)} \end{cases}$$

Note that the terms $AccLoadFactor_v$ for computing Equation 8 and $AccCharge_v$ can be computed using a SUM aggregation function, and are piggybacked on updates sent by children to their parents in our implementation.

4. STAR IMPLEMENTATION

In this section, we describe three important design issues for implementing STAR in our SDIMS prototype. First, we present a new *distribution abstraction* to provide the functionality of distributing AI error budgets in an aggregation tree. Second, we discuss different techniques to maintain AI error precision in query results under failures. Finally, we describe a practical optimization in our prototype implementation to reduce communication load for large-scale system monitoring.

4.1 Distribution Abstraction

A distribution function is a dual mechanism to the traditional bottom-up tree based aggregation that enables an update to be *distributed* top-down along an aggregation tree.

The basic mechanism of a distribution function at a node is as follows: given inputs of (1) a list of children for an attribute and (2) a (possibly null) distribution message received from its parent, return a set of messages destined for a subset of its children and itself. For self-tuning AI, each node's distribution function implements the STAR algorithm that takes an AI error budget from the parent and distributes it to its children.

In the SDIMS framework, each node implements an Aggregation Management Layer (AML) that maintains attribute tuples, performs aggregations, stores and propagates aggregate values [41]. On receiving a distribution message, the AML layer invokes the distribution function for the given attribute and level in the aggregation tree. The logic of how to process these messages is implemented by the monitoring application. For example, for self-tuning AI, the STAR protocol generates distribution messages to either allocate more error budget to a child subtree or decrease the error budget of a child subtree.

The AML layer provides communication between an aggregation function and a distribution function by passing local messages. Conceptually, a distribution function extends an aggregation function by allowing the flexibility of defining multiple distribution functions (policies) for a given aggregation function.

In SDIMS, a store of (attribute type, attribute name, value) tuples is termed the Management Information Base (MIB.) For hierarchical aggregation of a given attribute, each node stores *child MIBs* received from children and a *reduction MIB* containing locally aggregated values across the child MIBs. To provide the distribution abstraction, we implemented a *distribution MIB* that stores as value an object used for maintaining application-defined state for adaptive error budget distribution. In STAR, a distribution MIB also holds a local copy of load information received from child MIBs.

Given this abstraction, it is simple to implement STAR's self-tuning adaptation of error budgets in an aggregation tree. For example, in STAR

(a) A parent can increase δ_{self} by sending a distribution message to a child c to reduce its subtree budget, δ_c

(b) A parent can increase δ_c for a subtree child c by reducing its own δ_{self}, and

(c) Filter small changes: a parent only changes a δ assignment if doing so is likely to save more messages than the costs for rebalancing the δs costs.

The policy decision of when the self-tuning function gets called can be based on either a periodic timer, processing a threshold number of messages, or simply on demand.

4.2 Robustness

To handle node failures and disconnections, our STAR implementation currently provides a simple policy to maintain the AI error precision in query results under churn. On each invocation of the distribution function, it receives the current child MIB set from the AML layer. If the new child MIB set is inconsistent with the previous child MIB snapshot maintained by the distribution function, it takes a corrective action as follows:

- On a *new child* event: insert a new entry in the distribution MIB for the new child, assign an error budget to that child subtree, and send it to that child for distribution in its subtree.
- On a *dead child* event: garbage collect the child state and reclaim all AI error budget previously assigned to that child subtree.
- On a *new parent* event: reset the AI error budget to zero as the new parent will allocate a new error budget.

Assuming that the error bounds were in a state where all precision constraints are satisfied prior to a failure, the temporary lost error due to the failure of a child only improves precision, thus no precision constraints can become violated. For a new child, it receives a fraction of the error budget allocation so the correctness still holds. Finally, on a new parent event, the error budget at its children will be reset.

Though this policy is simple and conservative, it always provides correctness that the reported AI error precision in query results is satisfied. Under this policy, however, a single failure might incur high communication overhead e.g., if a whole subtree moves to a new parent, the entire AI budget is lost resulting in every leaf update being propagated up in the tree until the new parent sends distribution messages to reassign the AI error budgets in the affected subtree.

Alternatively, policies based on exploiting tradeoffs between performance and meeting the error precision bounds can be used. One such policy is when a child gets connected to a new parent, it can keep reporting aggregate values to the new parent with the precision error assigned by the last parent; the self-tuning algorithm then continually adjusts the previous subtree AI budget to converge with the new allocation over time.

Another policy would be to relax the invariant that a subtree meets the AI bound during periods of reconfiguration. Instead, a new parent can send down a target budget and aggregate up the actual AI error bound reported by child's subtree. This policy is simple for both this failure case and the common case of adaptively redistributing error budgets. We leave the empirical comparison of different policies that trade strict consistency for performance as future work.

4.3 Optimizing for Scalability

In this section, we present an optimization for setting initial, default error budgets that complements STAR's adaptive settings of error budgets to further reduce the communication overhead for large-scale system monitoring.

Our self-tuning STAR algorithm eventually converges to a good distribution of the error budgets among the tree nodes to yield large benefits. However, there is still a degree of freedom in setting the initial error budgets for nodes at different levels of a tree. Further, this initialization choice can significantly affect both the cost and the time to converge to the final error distribution state.

At one extreme, we can keep the entire budget at the root and perform on demand error distribution the first time an update reaches the root. However, this policy is expensive as the initial message cost in an N-leaf tree would be $O(\log N)$ for an update to reach the root and $O(N)$ for error distribution among all the nodes. For mice flows that only send a few updates, this cost will be significantly higher than the benefits of filtering. At the other extreme, we can assign root share of zero and uniformly divide the entire budget among the leaf nodes. This policy will cull majority of the mice flows at the leaves given sufficient error budgets. For elephant flows, however, each update is likely to incur $O(\log N)$ messages as it gets propagated to the root, and that might dominate the total cost. Similarly, the time to converge will depend on the difference between the initial and the final error distribution state. Note that some initial error budget could also be allocated to the internal nodes which we do not explore in this work.

Our approach is to define a continuum of policies to divide the total budget δ_T among the root and the leaf nodes. These set of policies can be expressed as:

$$AI_{Root} = Root_{share} * \delta_T$$
$$AI_{Leaf} = \frac{\delta_T - AI_{Root}}{N}$$

For example, if $Root_{share} = 0.5$, then the root gets $\frac{\delta_T}{2}$ and each leaf gets $\frac{\delta_T}{2*N}$. To implement this policy, on receiving an update, a leaf node checks if it already has a local AI error budget. If not, it performs a lookup for the default AI_{Leaf} to filter the update. If the local budget is insufficient, the leaf sends the update to its parent which in turn applies AI error filtering. If that update reaches the root, the root initiates error distribution of its AI_{Root} error budget across its tree. Given this approach, we can cull a large fraction of the mice flows at the leaves, thus preventing their updates from reaching the root. We show the performance benefit of using this optimization in Section 5.

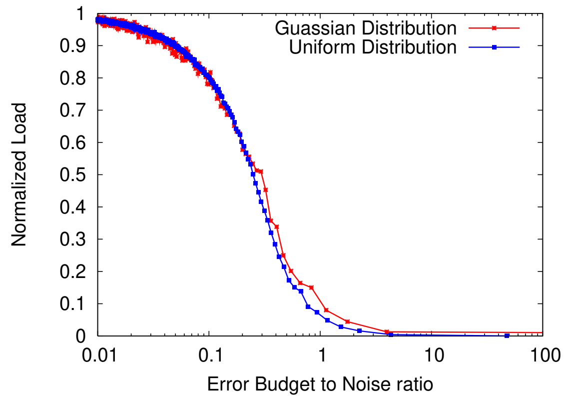


Figure 7: Normalized load vs. error budget to noise ratio for two synthetic workloads under a fixed AI error budget. If noise < AI, a majority of updates get filtered. The x-axis is on a log scale.

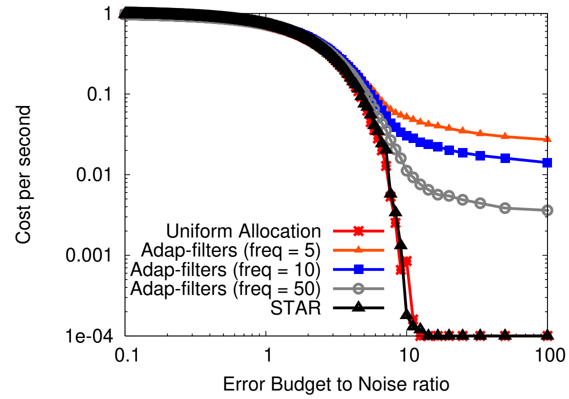


Figure 8: Performance benefits due to cost-benefit throttling. Load vs. error budget to noise ratio for a 10 node 1-level tree, random walk data. The graph is on a log-log scale.

5. EXPERIMENTAL EVALUATION

Our experiments characterize the performance and scalability of the self-tuning AI for the distributed heavy hitters application. First, we quantify the reduction in monitoring overheads due to self-tuning AI using simulations. Second, we investigate the reduction in communication load achieved by STAR for the DHH application in a real world monitoring implementation. For this evaluation, we have implemented a prototype of STAR in our SDIMS monitoring framework [41] on top of FreePastry [33]. We used two real networks: 120 node instances mapped on 30 physical machines in the department Condor cluster and the same 120-node setup on 30 physical machines in the Emulab [40] testbed. Finally, we evaluate the performance benefits of our optimization of carefully distributing the initial, default error budgets using our prototype implementation. In summary, our experimental results show that STAR is an effective substrate for scalable monitoring: introducing small amounts of AI error and adaptivity using self-tuning AI significantly reduces monitoring load.

5.1 Simulation Experiments

First, to characterize the trade-off between AI error budget and monitoring load, we determine the conditions under which is the AI error budget effective. Second, we analyze the effect of cost-benefit throttling on reducing load. Finally, we compare the performance of STAR, Adaptive-filters [29], and the uniform allocation strategy for different workloads.

In all experiments, all active sensor are at the leaf nodes of the aggregation tree. Each sensor generates a data value every time unit (round) for two sets of synthetic workloads for 100,000 rounds: (1) a Gaussian distribution with standard deviation 1 and mean 0, and (2) a random walk pattern in which the value either increases or decreases by an amount sampled uniformally from [0.5, 1.5].

Effectiveness of AI Filtering: We first investigate under what conditions is AI error budget effective. Figure 7 shows the simulation results for a 4-level degree-6 aggregation tree with 1296 leaf nodes for the two workloads under uniform static error distribution. The x-axis denotes the ratio of the total AI budget to the total noise induced by the leaf sensors and the y-axis shows the total message load normalized with respect to zero AI error budget. We observe that when noise is small compared to the error budget, there is about an order of magnitude load reduction as the majority of updates are filtered. But, as expected, when noise is large compared to the error budget, the load asymptotically approaches the unfiltered load with AI = 0. The random walk pattern allows almost perfect culling of updates for small amounts of noise whereas for the Gaussian distribution, there is a small yet a finite probability for data values to deviate arbitrarily from their previously reported range.

Cost-Benefit Throttling: Next, we quantify the cost of the periodic bound shrinking used in previous approaches [11, 29] compared with STAR's cost-benefit throttling. To motivate the importance of cost-benefit analysis, we perform a simple experiment here for a one-level tree, and later show the results for general hierarchical topologies. In our experiments, the following configuration gave the best results for Adaptive-filters: shrink percentage = 5%, high self-tuning frequency, and distributing error budgets to a small set (e.g., 10-15%) of nodes with the highest burden scores, where burden is the ratio of load to error budget. These observations are consistent with previous work [11].

Figure 8 shows the performance results of uniform allocation, Adaptive-filters, and STAR for a 10 node 1-level tree using a random walk pattern with the same step size at each node. In this case, the uniform error allocation would be close to the optimal setting. We observe that when error budget exceeds noise, Adaptive-filters incurs a constant error redistribution cost per tree (mapped to one or more attributes) that is proportional to the frequency of error redistribution. Thus, for large-scale monitoring services that require tracking tens of thousands to millions of attributes, approaches that keep sending messages periodically to adjust error budgets such as Adaptive-filters and potential gains adjustment [11] would incur a high overhead. STAR, however, performs cost-benefit throttling and does not redistribute error when the corresponding gain is negligible.

Evaluating Different Workloads: We now characterize the performance benefits of STAR compared to other ap-

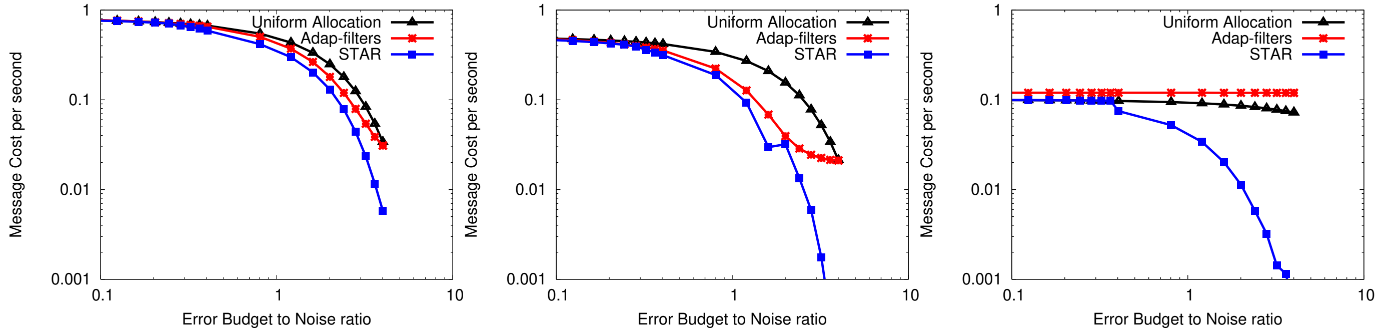


Figure 9: STAR provides higher performance benefits as skewness in a workload increases. The three figures show load vs. error budget to noise ratio for different skewness settings (a) 20:80% (b) 50:50% (c) 90:10%.

proaches as skewness in a workload increases. In Figure 9, the three graphs show the communication load vs. error budget to noise ratio for the following skewness settings: (a) 20:80% (b) 50:50% (c) 90:10%. For example, the 20:80% skewness represents that only 20% nodes have zero noise and the remaining 80% nodes have a large noise. In this case, since only a small fraction of the nodes are stable, both STAR and Adaptive-filters can only reclaim 20% total error budget from the zero-noise sources and distribute it to noisy sources to cull their updates. STAR reduces monitoring load by up to 5x compared to Adaptive-filters. Further, the latter algorithm's approach of periodic shrinking of error bounds at each node only yields small benefits compared to uniform allocation. For the 50:50 case, both the self-tuning algorithms can claim 50% of the total budget compared to uniform allocation and give it to noisy sources. However, even when the optimal configuration (error budget large compared to noise) is reached, Adaptive-filters keep readjusting the budgets due to periodic shrinking of error bounds. Finally, when 90% nodes are stable, STAR gives more than an order of magnitude reduction in load compared to both Adaptive-filters and uniform allocation.

Note that overall, for Figures 8 and 9, the advantage of STAR's cost-benefit throttling varies with the budget to noise ratio. We expect that for systems monitoring a large numbers of attributes (e.g., DHH) some attributes (e.g., the elephants) will have a low error budget to noise ratios and gain a modest advantage from STAR, while other attributes (e.g., the mice) will have large ratios and gain large advantages. We typically expect many more mice attributes than elephant attributes for common monitoring applications.

Next, we compare the performance of STAR, Adaptive-filters, and uniform allocation under different configurations by varying input data distribution, standard deviation (step sizes), and update frequency at each node. For data distribution, the workload is either generated from a random-walk pattern or Gaussian. For standard deviation/step-size, 70% of the nodes have uniform parameters as previously described; the remaining 30% nodes have these parameters proportional to *rank* (i.e., with locality) or randomly assigned (i.e., no locality) from the range [0.5, 150].

Figure 10 shows the corresponding results for different settings of data distribution and standard deviation for a 4-level degree-4 tree with fixed update frequency of 1 update per node per round. We make the following observations from Figure 10(a). First, when error budget is smaller

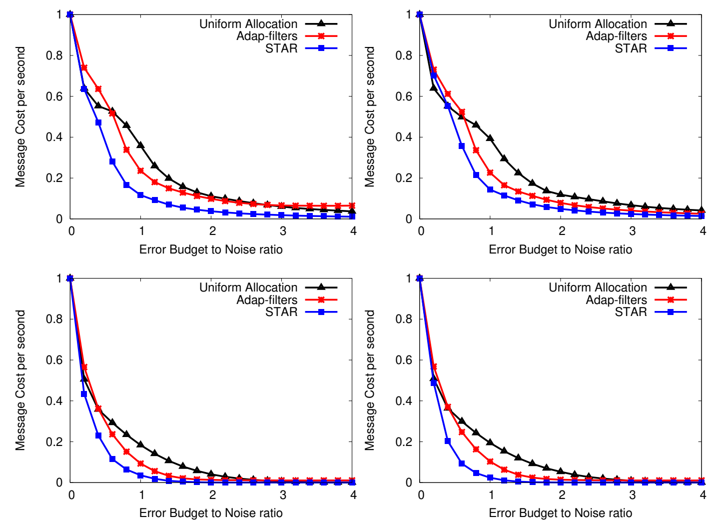


Figure 10: Performance comparison of STAR vs. Adaptive-filters and uniform allocation for different {workload, step sizes/standard deviation} configurations (a) random walk, rank (b) random walk, random (c) Gaussian, rank, and (d) Gaussian, random.

than noise, no algorithm in any configuration achieves better performance than uniform allocation. Adaptive-filters, however, incurs a slightly higher overhead due to self-tuning even though it does not benefit. In comparison, STAR avoids self-tuning costs via cost-benefit throttling. Second, Adaptive-filters and uniform error allocation reach a cross-over point having a similar performance. This cross-over implies that for Adaptive-filters, the cost of self-tuning is equal to the benefits. Third, as error budget increases, STAR achieves better performance than Adaptive-filters. Because step-sizes are based on node rank, STAR's outlier detection avoids allocating budget to the nodes having the largest step-sizes. Adaptive-filters, however, does not make such a distinction and computes burden scores based on load thereby favoring nodes with relatively large step sizes. Thus, since the total budget is limited, reducing error budget at nodes with small step sizes increases their load but does not benefit outliers since the additional slack is still insufficient to filter their updates. Finally, as expected, when error budget is higher than noise, all algorithms achieve good performance. In this configuration, STAR reduces monitoring load by 3x-5x

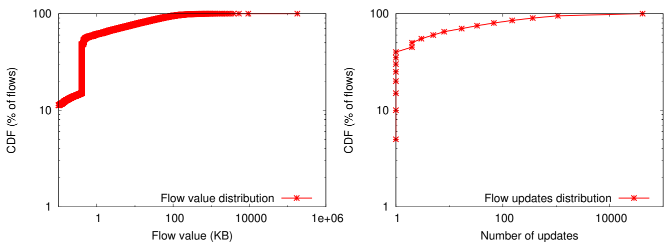


Figure 11: CDF of percentage of flows vs. (a) number of updates and (b) flow-values for the Abilene dataset. The graph is on a log-log scale.

compared to uniform allocation and by 2x-3x compared to Adaptive-filters.

Under random distribution of step-sizes as described above, STAR reduces load by up to 2x compared to Adaptive-filters and up to 3x against uniform allocation (Figure 10(b).) Comparing across configurations, all algorithms perform better under input distribution of Gaussian compared to the random-walk model. Overall, across all configurations in Section 5.1, STAR reduces monitoring load by up to an order of magnitude compared to uniform allocation and by up to 5x compared to Adaptive-filters.

5.2 Testbed Experiments

In this section, we quantify the reduction in monitoring load due to self-tuning AI and the query precision of reported results for the DHH monitoring application.

We use multiple netflow traces obtained from the Abilene [1] Internet2 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 2.3, our DHH application executes a Top-100 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.

Figure 11 shows the cumulative distribution function (CDF) of the percentage of network flows versus the number of bytes (KB) sent by each flow. We observe that about 60% flows send less than 1 KB of aggregate traffic, 80% flows send less than 12 KB, 90% flows less than 55 KB, and 99% of the flows send less than 330 KB during the 1-hour run. Note that the distribution is heavy-tailed and maximum aggregate flow value is about 179.4 MB. Figure 11 shows the corresponding CDF graph of the percentage of network flows versus the number of updates. We observe that 40% flows send only a single update (a 28 byte IP/UDP packet.) Further, 80% flows send less than 70 updates, 90% flows less than 360 updates, and 99% flows less than 2000 updates. Note that the number of update distribution is also heavy-tailed with the maximum number of updates sent by a flow is about 42,000.

Overall, the 120 sensors track roughly 80,000 flows and send around 25 million updates. Thus, the monitoring load for zero AI error budget would be about 58.6 messages per node per second for each of the 120 nodes. Therefore, a centralized scheme would incur a prohibitive cost of about 7,000 updates per second for processing this workload.

To address this scalability challenge, we apply our self-tuning STAR algorithm to reduce the monitoring load. Figure 12(a) shows the bandwidth cost per node incurred by STAR under global AI error budgets of 5%, 10%, 15%, and 20% of the maximum flow value per aggregation tree, and different settings of the $Root_{share}$ parameter: 0%, 50%, 90%, and 100%. We observe that with a $Root_{share}$ of 90% and AI of 5%, we incur an overhead of about 7 messages per node per second which is roughly three orders of magnitude less compared to 7,000 messages per second at the root in the centralized scheme. By increasing the AI error budget to 20%, we can reduce this cost by almost a factor of three to about 2.5 messages per node per second. Thus, self-tuning AI even under modest error budgets can provide a significant reduction in the communication overhead.

Further, by carefully initializing the error budgets to cull mice updates, we can gain nearly another order of magnitude load reduction. Comparing different settings of $Root_{share}$ (AI error budget of 20%) in Figure 12(a), we observe that compared to $Root_{share}$ of 90%, $Root_{share}$ of 50% reduces the load by almost a factor of five to about 0.5 messages per node per second, and $Root_{share}$ of 0% (i.e., all error budget initialized to the leaf nodes) further provides another factor of two reduction leading to an overall order of magnitude load reduction to about 0.27 messages per node per second.

However, setting $Root_{share}$ of 100% incurs a large overhead of about 30 messages per node per second since a large fraction of mice updates reach the root of their trees, and these root nodes then initiate an error distribution in their respective trees. As Figure 12(b) shows, for this $Root_{share}$ setting, the redistribution overhead of sending these error budgets in each aggregation tree dominates constituting about 70% of the total communication cost. Thus, a good default setting of the error budgets should initialize some error budget at the leaf nodes to cull a large fraction of the mice flows.

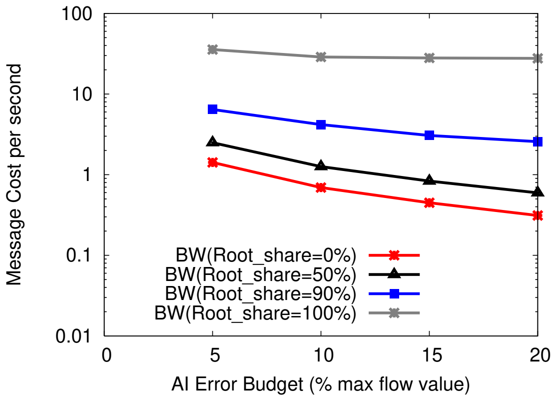
Overall, the self-tuning algorithm eventually sets the error budgets at different nodes in an aggregation tree in a way that yields the largest benefits. Yet, we can gain by initializing AI error budgets to some sensible state so that majority flows get filtered as early as possible. Finally, we want to set the error budgets optimally such that (1) the mice flows never generate any updates and (2) the elephant flows get filtered to the maximum extent possible. STAR aims to attain this optimal error setting for a majority of flows as cheaply as possible by (1) filtering mice flows at lower levels of the aggregation tree and (2) assigning δ_{self} at the internal nodes to filter modest variations in aggregate values for elephant flows e.g., even if the child values have deviated significantly from their previous reported values as to bypass their own AI error range, the net effect of merging all children updates may still be close to zero.

In summary, our evaluation shows that adaptive setting of modest AI budgets can provide large bandwidth savings to enable scalable monitoring.

6. RELATED WORK

Our STAR algorithm for self-tuning AI error budgets is part of a larger system building effort, PRISM, to enforce imprecision bounds and quantify the consistency guarantees of query results in a large-scale monitoring system [23].

Olston et al. [29] proposed Adaptive-filters (AF), a self-tuning algorithm for a *one-level* tree: increase δ for nodes with high load and low previous δ and decrease δ for nodes with low load and high previous δ. Our STAR algorithm differs from AF in three fundamental ways driven by our



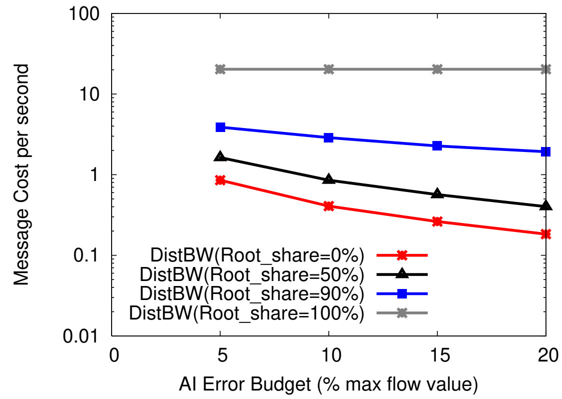


Figure 12: Performance comparison of the self-tuning algorithm under different settings of initial error budgets for the DHH application. The left figure shows the average bandwidth cost per node (BW) and the right figure shows the average redistribution overhead per node (DistBW.)

focus on scaling to a large number of nodes and attributes: (1) STAR is hierarchical and uses a distributed algorithm to divide error budget across internal and leaf nodes while AF uses a centralized coordinator to distribute error budget to leaf sensors only. (2) STAR's mathematical formulation—using the workload (e.g., update rate, variance) itself to derive the optimal error budget distribution—provides useful insights and practical benefits. (3) STAR's cost-benefit throttling is crucial for systems with (a) stable workloads where oblivious periodic rebalancing doesn't benefit or (b) large numbers of attributes where rebalancing error budgets for mice is expensive and not helpful.

For hierarchical topologies, Manjhi et al. [28] determine an optimal but *static* distribution of slack to the internal and leaf nodes of a tree for finding frequent items in data streams. IrisNet [12] filters sensors at leaves and caches timestamped results in a hierarchy with queries that specify the maximum staleness they will accept and that trigger re-transmission if needed. Deligiannakis et al. [11] propose an adaptive precision setting technique for hierarchical aggregation, with a focus on sensor networks. However, similar to Olston's approach, their technique also periodically shrinks the error budgets for each tree which limits scalability for tracking a large number of attributes. Further, since their approach uses only two *local* anchor points around the current error budget to derive the precision-performance tradeoff, it cannot infer the complete correlation shown in Figure 2 making it susceptible to dynamic workload variations. None of the previous studies to our knowledge have used the variance and the update rate in the data distribution in a principled manner. In comparison, STAR provides an efficient and practical algorithm that uses a mathematically sound model to estimate globally optimal budgets and performs cost-benefit throttling to adaptively set precision constraints in a general communication hierarchy.

Some recent studies [19, 22, 25] have proposed monitoring systems with distributed triggers that fire when an aggregate of remote-site behavior exceeds an a priori global threshold. Their solution is based on a centralized architecture. STAR may enhance such efforts by providing a scalable way to track top-k and other significant events.

Other studies have proposed prediction-based techniques for data stream filtering e.g., using Kalman filters [21], neural networks [26], etc. There has also been a considerable interest in the database and sensor network communities on approximate data management techniques; Skordylis et al. [36] provide a good survey.

There are ongoing efforts similar to ours in the P2P and databases community to build global monitoring services. PIER is a DHT-based relational query engine [20] targeted at querying real-time data from many vantage-points on the Internet. Sophia [39] is a distributed monitoring system designed with a declarative logic programming model. Gigascope [10] provides a stream database functionality for network monitoring applications.

Traditionally, DHT-based aggregation is event-driven and best-effort, i.e., each update event triggers re-aggregation for affected portions of the aggregation tree. Further, systems often only provide eventual consistency guarantees on its data [38, 41], i.e., updates by a live node will eventually be visible to probes by connected nodes.

7. CONCLUSIONS AND FUTURE WORK

Without adaptive setting of precision constraints, large scale network monitoring systems may be too expensive to implement even under considerable error budgets because too many events flow through the system. STAR provides self-tuning arithmetic imprecision to adaptively bound the numerical accuracy in query results, and it provides optimizations to enable scalable monitoring of a large number of stream events in a distributed system.

While STAR focuses on minimizing communication load in an aggregation hierarchy under fixed data precision constraints, it might be useful for some applications and environments to investigate the dual problem of maximizing data precision subject to constraints on availability of global computation and communication resources. Another interesting topic of future work is to consider self-tuning error distribution for general graph topologies e.g., DAGs, rings, etc., that are more robust to node failures than tree networks. Finally, reducing monitoring load in real-world systems would also require understanding other dimensions of imprecision such as temporal where queries can tolerate bounded staleness and topological when only a subset of nodes are needed to answer a query.

Acknowledgments

We thank Rezaul Chowdhury for many useful discussions, and Joe Hellerstein, Chris Olston, and the anonymous reviewers for their valuable feedback.

8. REFERENCES

[1] Abilene internet2 network. `http://abilene.internet2.edu/`.

[2] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[4] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, June 2003.

[5] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.

[6] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, Portland, OR, August 2004.

[7] S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *SIGMOD*, 2001.

[8] D. D. Clark, C. Partridge, J. C. Ramming, and J. Wroclawski. A knowledge plane for the internet. In *SIGCOMM*, 2003.

[9] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *IPTPS*, 2002.

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

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

[12] A. Deshpande, S. Nath, P. Gibbons, and S. Seshan. Cache-and-query for wide area sensor databases. In *Proc. SIGMOD*, 2003.

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

[14] M. J. Freedman and D. Mazires. Sloppy Hashing and Self-Organizing Clusters. In *IPTPS*, 2003.

[15] FreePastry. `http://freepastry.rice.edu`.

[16] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proc. SOSP*, Oct. 2003.

[17] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USITS*, March 2003.

[18] J. M. Hellerstein, V. Paxson, L. L. Peterson, T. Roscoe, S. Shenker, and D. Wetherall. The network oracle. *IEEE Data Eng. Bull.*, 28(1):3–10, 2005.

[19] L. Huang, M. Garofalakis, A. D. Joseph, and N. Taft. Communication-efficient tracking of distributed cumulative triggers. In *ICDCS*, 2007.

[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] A. Jain, E. Y. Chang, and Y.-F. Wang. Adaptive stream resource management using kalman filters. In *SIGMOD*, 2004.

[22] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *HotNets*, San Diego, CA, November 2004.

[23] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. PRISM: Precision-Integrated Scalable Monitoring (extended). Technical Report TR-06-22, UT Austin Department of Computer Sciences, 2006.

[24] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. STAR: Self-Tuning Aggregation for Scalable Monitoring (extended). Technical Report TR-07-15, UT Austin Department of Computer Sciences, 2007.

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

[26] V. Kumar, B. F. Cooper, and S. B. Navathe. Predictive filtering: a learning-based approach to data stream filtering. In *DMSN*, 2004.

[27] 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.

[28] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *ICDE*, 2005.

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

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

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

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

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

[34] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An infrastructure for connecting sensor networks and applications. Technical Report TR-21-04, Harvard Technical Report, 2004.

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

[36] A. Skordylis, N. Trigoni, and A. Guitton. A study of approximate data management techniques for sensor networks. In *WISES, Fourth Workshop on Intelligent Solutions in Embedded Systems*, 2006.

[37] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[38] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *TOCS*, 21(2):164–206, 2003.

[39] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *HotNets-II*, 2003.

[40] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Boston, MA, Dec. 2002.

[41] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc SIGCOMM*, Aug. 2004.

[42] P. Yalagandula, P. Sharma, S. Banerjee, S.-J. Lee, and S. Basu. S^3: A Scalable Sensing Service for Monitoring Large Networked Systems. In *Proceedings of the SIGCOMM Workshop on Internet Network Management*, 2006.

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

[44] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.