Energy-Efficient Monitoring of Extreme Values in Sensor Networks*

Adam Silberstein Kamesh Munagala Jun Yang
Department of Computer Science, Duke University, Durham, NC 27708, USA
{adam,kamesh,junyang}@cs.duke.edu

ABSTRACT

Monitoring extreme values (MAX or MIN) is a fundamental problem in wireless sensor networks (and in general, complex dynamic systems). This problem presents very different algorithmic challenges from aggregate and selection queries, in the sense that an individual node cannot by itself determine its inclusion in the query result. We present novel query processing algorithms for this problem, with the goal of minimizing message traffic in the network. These algorithms employ a hierarchy of local constraints, or thresholds, to leverage network topology such that message-passing is localized. We evaluate all algorithms using simulated and real-world data to study various trade-offs.

1 Introduction

The nodes in a wireless sensor network generate vast amounts of data that must be communicated to the network root (also known as the base station) using radio transmission. Nodes are battery-powered, and radio usage dominates their energy consumption. Therefore, while any query can be answered by continuously streaming all data to the root and processing it there, we can greatly extend the lifetime of the network by developing query-specific plans that limit the amount of data transmitted by the nodes.

One of the queries that greatly benefits from such optimization is the continuous MAX (or MIN) query, which returns the node with the current maximum (or minimum) value along with the value itself. This query is a typical example of a more general *exemplary* aggregate. An exemplary aggregate [9] is one where the solution consists of one or more representative values from the network, as opposed to a *summary*, where the solution is computed over all the values. We focus on *exact* query answering in this paper. The techniques we develop also extend to approximate answers, where the precision is traded off with the communication cost. We focus on the MAX problem; the discussion for MIN is identical. We also show that the one-sided quantile query (e.g., one that finds the top 10% values) is a straightforward extension of MAX, and can be supported with the same techniques.

*The authors are supported by NSF CAREER award IIS-0238386, NSF grant CNS-0540347, and an IBM Faculty Award.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

Our aim in focusing on the MAX query is twofold. First, in developing a solution to MAX, we hope to gain insight into optimizing this general class of monitoring queries. Secondly, in practice, continuous MAX is useful for detecting abnormal or extreme behavior. This type of behavior is not captured by summary aggregates such as mean or sum, establishing a fundamental difference between the applications of these two query types.

The continuous MAX query is useful in several scenarios like maintaining maximum temperature in a factory, or tracking the locations and amounts of highest rainfall across geographic regions. In both cases, part of the challenge in answering the query is that it is not known *a priori* what constitutes high values.

Our basic insight into MAX optimization is to use the history of values recorded, combined with suitable constraint settings, to prevent nodes unlikely to have the maximum value from transmitting. Though this intuition is straightforward, designing algorithms with good performance guarantees based on this premise is tricky—even quantifying "good performance" is challenging in itself.

It is important to note the distinction between MAX/quantiles and selection queries: Unlike selection queries, in MAX/quantile queries, nodes cannot decide their inclusion in the query result themselves. Making this decision correctly naturally implies additional communication. Our challenge is to build plans that minimize this communication.

We illustrate the challenges of the continuous MAX query through a study of two basic approaches and their shortcomings.

Prior Approaches

Temporal Suppression A first strategy (which is standard in continuous query processing) is to apply a *temporal suppression* policy at all nodes. A node transmits its value if it has changed since the last transmission. This policy keeps nodes from repeatedly sending identical data and has great benefit in a mostly unchanging environment. Nevertheless, *it does not differentiate between important and unimportant nodes*. Suppose in one timestep we have a maximum value of 100, and a large number of nodes with value around 10. Even if those nodes all double in value to 20 in the next timestep, they are not in contention for the maximum value. Under temporal suppression, however, they would report their new value.

Range Caching The problem of differentiating important and unimportant nodes is similar to the problem of balancing the precision of cached values [11]. In their case, the root caches a range for each value at a remote node. The remote node synchronously maintains the same range stored at the root. The node only reports its value if it violates the range. The length of the range naturally provides a mechanism for controlling the trade-off between accuracy of the response to a query on the value, and the amount of communication between the root and the remote node.

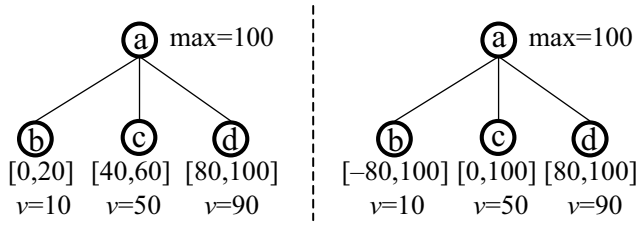

Figure 1: Range caching.

In our setting, a natural extension of the range caching framework assigns ranges with similar lengths, but such an approach would just be a minor improvement over temporal suppression. Making the ranges unequal in length can yield much better results. Returning to our example where the max value is 100, a node with a value of 90 should have a smaller range than a node with a value of 10 for us to infer the correct max without additional remote accesses. In Figure 1, we compare tight equal-length ranges against varied-length ranges, where the lower values are allowed greater slack. Later in Section 3.1.2, we will see that a generic, adaptive range adjustment policy can automatically tune range lengths to match the perceived importance of values. This approach, however, still has a number of shortcomings:

- The basic range adjustment policy makes independent decisions for each monitored value according to accesses to this value by the overall continuous query. While this approach is very general, it is also oblivious to potential optimization opportunities that arise from knowing the semantics of the overall query. For selection and summary queries, this generic approach works extremely well, but the MAX problem has special characteristics that make this approach less effective. For example, unlike selection and summary, a node cannot independently determine its contribution to the MAX solution. Also, the semantics of MAX imply we need not maintain lower bounds for values that are currently not max, because only rising values can affect max.
- The range caching approach assumes separate, independent connections between each node and the root. In a sensor network, nodes are often organized into a tree for communication efficiency. We need to exploit this hierarchy, especially since values may be spatially correlated. If a group of nearby nodes all become max candidates due to other values falling, we should not have to pay to access each of these nodes independently.

Contributions and Outline

In this paper we thoroughly investigate applying existing approaches to the MAX query, illustrating their pitfalls. We then introduce three novel algorithms, the most advanced of which is HAT. We identify and employ the use of *constraint localization* as a fundamental method for reducing message traffic and, therefore, energy consumption. For MAX, this technique involves setting threshold values at nodes to suppress their reporting. We cover these contributions in detail in Section 3. Section 4 analyzes various policies for setting the threshold values. We present policies with worst-case performance guarantees, as well as efficient practical policies. Section 5 provides additional enhancements and extensions of our approach. Finally, in Section 6, we compare the performance of various algorithms and policies on both simulated and real data.

2 Related Work

Considine et al. [4] study the problem of accurately calculating duplicate-sensitive aggregate queries such as sum and mean when multiple copies of sensor readings are sent along separate paths to the root to improve robustness. In contrast, MAX is not affected by duplicates. Shrivastava et al. introduce the *q-digest* structure [14] to approximately answer queries such as histograms using compression techniques. MAX, being a single value, does not benefit from compression.

Madden et al. [10] suggest strategies for running ad hoc queries. Even while operating in a "one-shot" setting they recognize the advantages of pushing threshold-based filters into the network. For example, while running a pipelined MIN query, they suggest aborting the query after some amount of time, determining the minimum value so far, and running a new query requiring nodes to only respond if they have an even smaller value.

As mentioned in Section 1, Olston et al. [11] present the problem of caching value ranges of remote data sources to support queries at a server. Their techniques can be applied to the sensor setting, which we do for MAX in Section 3.

Deligiannakis et al. [6] address the problem introduced by nodes being organized in a network hierarchy. Their algorithm is a direct adaptation of the algorithm in [11]. It is designed for summary aggregates such as sum, where nodes cache ranges covering the subsolutions for their subtrees. Because MAX is exemplary, it requires a different translation of [11].

The algorithm in [8] can be adapted for continuous MAX monitoring. The basic strategy, similar to [11], is to install a value range at each node, and store that range at the root. Nodes send data to the root only if their values leave their ranges. When an ad hoc MAX query is received, the root sorts nodes by range upper bounds. The root queries each node in order one-by-one, while maintaining a running maximum value. If enough nodes have been searched such that all unsearched nodes have upper bounds below the running maximum, the running maximum is the solution. This one-by-one approach is costly since it ignores network topology. Consider two nodes adjacent in sort order (and partially overlapping in ranges) that are far away from the root and share a common parent in the network. We should clearly query both nodes simultaneously so the transmission cost of the queries and replies can be shared. While this example is extreme, most realistic situations would also benefit from more carefully designed data acquisition plans.

Cheng et al. [2] apply adaptive threshold setting to the processing of entity-based queries in a stream in a distributed environment. MAX can be seen an example of an entity-based query. They install identical thresholds at every node to detect when its value either enters or leaves a specified range. If the set of nodes within the range shrinks too much, their approach is forced to potentially query the entire network to build a new range. Our algorithm for setting thresholds is more powerful because we permit *non-identical* thresholds at nodes that can adapt locally. As we will show, this flexibility brings superior performance.

As discussed in [12] and [13], there is a trade-off between sending correlated data promptly to the root via shortest paths or to a common meeting point where such data can be compressed. This issue arises in our context as well. There is a benefit in assigning nodes with similar values to the same parent, and we try exploit this benefit when possible by allowing nodes to change parent-child relationships dynamically and opportunistically to improve performance (Section 5).

Finally, our work on monitoring extreme values complements earlier work we have done on processing ad hoc top-k queries in sensor networks [15]. The continuous monitoring aspect we now address requires we constantly and thoroughly leverage previous state to avoid querying the entire network.

3 Algorithms for MAX

Our network consists of n fixed-location nodes, $u_1 \ldots u_n$. Each node, u_i, measures some feature, such as temperature, obtaining a value, v_i. The network is rooted at a base station node with full computing capabilities. All nodes either contact the root directly or indirectly through a routing protocol installed in the network.

Our goal is to minimize energy consumed by the network. We primarily focus on radio communication, the dominant consumer of node energy. We aim to minimize the number and size of messages sent and received in the network. The MAX problem is to maintains, continuously, the $(node\_id, value)$ pair for the node with the maximum value in the network. We assume the query is processed repeatedly over a series of *rounds*, where each node generates a value in each round. Each round is long enough for all necessary messaging to occur in order to complete the query. This generally consists of no more than a few sets of exchanges between the root and nodes. We use rounds for two reasons. First, it simplifies the notion of a continuous query by avoiding synchrony issues. Second, although we ideally want the query to be continuous, we realistically need to discretize the nodes' measurement activity since truly continuous sampling of the environment is either impossible or too expensive to implement.

The routing tree evolves over time to cope with failure and optimize performance. A node, u_c, must be registered at all times with some parent node, u_p, although that node can change. Any messages at u_c headed for the root are sent through u_p. Messages travel throughout the network stored in packets. We define the following packet types for use in our algorithms, and explore their uses in detail shortly.

Type	Description
Boot	Initial application installation and query
Trigger	Node sending own value
Query	Root initiating fetching of values
Reply	Response to *Query*
ThresholdUpdate	Update to node constraints
MaxDesignate	Designate node as current max
MaxOff	Notify node that it is no longer max

Of the five algorithms discussed in this section, the latter four are illustrated comparatively for the same scenario in Figure 2. The accompanying text is at the end of this section. These may prove helpful for tracking the algorithms during discussion of each.

3.1 Topology-Oblivious Algorithms

We begin by ignoring network topology and pretend all nodes are connected directly to the root in a one-level tree (for the purpose of algorithm design).[1] With this simplification, we present three algorithms. Two of these were introduced in Section 1. The goal of this subsection is to progress from generic policies to a query-specific policy for reducing messaging (while still confined to the simplification that each node's communication with the root is optimized in isolation).

3.1.1 Temporal Suppression (TS)

We introduced this policy in Section 1; we formally define it here. The root disseminates a *Boot* packet to all nodes. In response, each node u_i initially sends its value, v_i, to the root, and locally stores v_i in the field v_i^{old} as its last value sent. The root stores all received values. In each subsequent round, if $v_i \neq v_i^{old}$, u_i transmits v_i in a *Trigger* packet to the root and updates v_i^{old} to v_i. The root overwrites old saved values with newly received ones. It then calculates the maximum value simply by finding the maximum among all saved values.

[1] In practice, it can still take multiple hops for a message to the root. Furthermore, as a routing optimization, messages converging at the same intermediate node can still be combined into one bigger message in order to share overhead cost.

3.1.2 Range Caching (RC)

This policy, also mentioned in Section 1), is a direct application of [11] to MAX and the sensor network setting. After receiving *Boot* packets from the root, each node u_i initially transmits to the root the value v_i and a range $[lb_i, ub_i]$ around v_i (in fact, the range alone is sufficient). The range is set such that $v_i = \frac{(lb_i + ub_i)}{2}$ (setting of the range length will be discussed shortly). The root maintains the maximum value $v_{\max}$ and the corresponding node $u_{\max}$.

In subsequent rounds, if v_i falls out of the range $[lb_i, ub_i]$, u_i transmits a *Trigger* packet listing v_i along with a new range to the root. Once the root receives all reports in a round, it determines if it already has the maximum value. First it compares the highest reported value, v^*, against the highest upper bound of unreported values, ub^*. If $v^* > ub^*$, $v_{\max}$ is set to v^*. On the other hand, if $ub^* > v^*$, or if no nodes report in that round, the root identifies the highest lower bound of unreported values, lb^*. For each unreported node, u_i, with $ub_i > lb^*$, the root sends a *Query* packet requesting v_i. Each of these sends back a *Reply* packet. The solution is the maximum of v^* and all values received in *Reply* packets.

For setting lengths of the ranges, we again directly apply the adaptive scheme described in [11]. Whenever u_i does a value-based transmission (when v_i falls outside its current range), it expands the length of the range $(ub_i - lb_i)$ by a factor $\alpha > 1$. The new range is again set to be centered at v_i. Whenever u_i does a query-based transmission (in response to a *Query* packet requesting v_i), with a 50% probability, it contracts the range length by a factor of α. In both cases, u_i transmits lb_i and ub_i to synchronize the root. For the node $u_{\max}$ with the maximum value, since its value is needed at all times, range length is always set to 0.

This adaptive scheme has the effect of tuning cache ranges to match a node's importance. When a node's value decreases, it expands its bounds to lessen the chance of having to transmit in the future. The nodes competing for the max, however, contract their bounds as they are queried, to lessen the chance of having to be queried in the future. In effect, the nodes competing for the max end up with tighter ranges than nodes not competing for the max.

Note that both TS and RC are *query-oblivious*. Nodes aside from the root are not aware they are supporting the MAX query. In RC, however, the root's actions can attune the nodes' ranges to the MAX query to some extent.

3.1.3 SLAT

Our first new algorithm, SLAT, stands for *single-level adaptive thresholds*. It disseminates query-specific instructions, making it non-oblivious, unlike TS and RC.

Initialization Each node, u_i, is assigned a threshold, τ_i, known to both u_i and the root. Upon receiving a *Boot* packet, each u_i sends v_i to the root, and sets $\tau_i = v_i$. The root determines the highest returned value, v^*. In the first round, when all nodes reply, the originator of the value v^* is the node with the maximum value, $u_{\max}$. The root sets $v_{\max} = v^*$.

The root then sends a *MaxDesignate* packet to $u_{\max}$, instructing it to do temporal monitoring. Note that all thresholds, except for the node with the maximum value $v_{\max}$, are below $v_{\max}$ in this round. We formally state this property as an invariant that holds throughout all subsequent rounds.

INVARIANT 1. *In a particular round thresholds are set such that for each node u_i, $\tau_i \leq v_{\max}$.*

Behavior in a Round Each subsequent round proceeds in three stages:

- *Node-initiated reporting.* In the first stage, if a node u_i is designated $u_{\max}$ and $v_i^{old} \neq v_i$, u_i transmits a *Trigger* packet listing v_i to the root. If not designated $u_{\max}$, u_i transmits a *Trigger* to the root only if $v_i > \tau_i$.
- *Root-initiated querying.* Once all nodes have reported, the root determines v^* from the set of all returned values and, if $u_{\max}$ did not report, the stored value of $u_{\max}$. Let u^* be the node with value v^*. If $\forall i : v^* \geq \tau_i$, we set $v_{max} = v^*$, and $u_{\max} = u^*$. Otherwise, a *Query* packet is sent to each u_i for which $\tau_i > v^*$. The *Query* contains v^*. Each u_i receiving a *Query* sends a *Reply* with its value, v_i, only if $v_i > v^*$. At the root, $v_{\max}$ is then set to the maximum value in all *Reply* packets, and $u_{\max}$ is set accordingly. If no nodes reply, $v_{\max} = v^*$ and $u_{\max} = u^*$. If $u_{\max}$ designation changes from the previous round, the root sends a *MaxOff* packet to the old $u_{\max}$ and a *MaxDesignate* packet to the new $u_{\max}$.
  Note that the queries in this second stage only occurs when the value at the designated $u_{\max}$ falls. If the value stays the same or rises, but one or more nodes overtake it in value, that change will be detected in the first stage. Because of Invariant 1, any node overtaking the designated $u_{\max}$ in value must have had a threshold below $v_{\max}$ from the previous round. Therefore, a node cannot overtake a non-dropping $v_{\max}$ without breaking its own threshold and sending a *Trigger* packet. On the other hand, a node can become the new $u_{\max}$ without breaking its threshold, if the old maximum falls below it. In this case, it is possible that the new maximum value can only be discovered through querying in the second stage.
- *Threshold setting.* To maintain Invariant 1, each τ_i must be updated and consistently stored at u_i and the root. Whenever u_i breaks its threshold and sends a *Trigger* to the root (in the first stage), it awaits a threshold update. Once the root determines the $v_{\max}$ for the round (in the second stage), it transmits that $v_{\max}$ to all nodes awaiting updates in *ThresholdUpdate* packets. Each such node u_i updates its threshold τ_i to be halfway between its own value v_i and $v_{\max}$. The root carries out the exact same update to its own copy of τ_i using $v_{\max}$ and v_i, both of which are known at the root.
  Whenever a node u_i is queried in a second stage, there are two cases: if v_i exceeds the *Query* value, then u_i has already replied with v_i in the second stage, and τ_i is set to v_i. Otherwise u_i sends a *Reply* with a new τ_i, lowered to be between v_i and the *Query* value. If doing so will not lower τ_i much below the query value, u_i has the option of setting τ_i to the *Query* value and not replying. The root implicitly assumes this behavior unless it hears from u_i. This optimization saves messaging when the benefit of further lowering τ_i is small.
  This adjustment policy raises thresholds when node values break them and lowers thresholds when the root queries nodes, thereby maintaining Invariant 1. We also see the impact of the choice of thresholds. The lower a node's threshold, the more likely the node breaks it and sends its value to the root. The higher the threshold, the more likely the root has to query the node during the second stage. We explore threshold setting more in Section 4.

We explore the relative performances of TS, RC, and SLAT in Section 6. There are various trade-offs that benefit or penalize these algorithms depending on conditions. While without perfect future knowledge it is difficult to decisively choose one, we continue our discussion of algorithms building on the use of thresholds, which we believe can be advanced most naturally and effectively to exploit topology.

3.2 Topology-Aware Algorithms

We next introduce two novel algorithms that build off of SLAT. The key difference is we now acknowledge nodes communicate to the root through each other and share common routes to the root. We now leverage these characteristics.

3.2.1 SLAT-A

Our next algorithm SLAT-A stands for *single-level adaptive thresholds with aggregation*. It considers topology by making an incremental improvement over the basic SLAT algorithm. SLAT transmits all *Trigger* and *Reply* packets to the root. It is easy to see that this approach can be improved. Consider nodes u_i and u_j with values v_i and v_j. Both send *Trigger* packets that converge to the same ancestor node in the network, where we have the chance to compare their values. If $v_j > v_i$, it is impossible for v_i to be the max. In that case, the root need not know about v_i. Instead of forwarding both *Trigger* packets, we drop v_i and only pass on v_j. In general, we enforce the policy that whenever more than one *Trigger* (or *Reply*) packet headed for the root meet, we only pass the packet with the highest value.

The aggregation of messages complicates threshold setting, because the root cannot tell from a single *Trigger* message from its child what other nodes in the subtree may have broken their threshold, and by how much. Effectively, what the root observes at a child is the behavior of the maximum value in the subtree rooted at this child. To ensure correctness, SLAT-A does the following: any node that previously received a *Trigger* packet from a child remembers the child sent it, even if this packet was dropped in favor of a higher value. Then, when the root sends a *ThresholdUpdate* packet with $v_{\max}$ to a child, this packet is propagated recursively down the tree to all descendants who previously sent a *Trigger*. Note in this case the root no longer knows the exact threshold value at every node, but it is still guaranteed that every threshold value is set below $v_{\max}$, thereby ensuring correctness of the algorithm.

A more severe consequence of aggregation occurs when the maximum value falls. Since the root does not know the individual threshold values for most nodes, it is now forced to query most of the network in the second stage, when the current $u_{\max}$ falls in value, incurring significant cost in *Query* traffic.

Nevertheless, the immense savings in reducing *Trigger* and *Reply* traffic trade off well with the increased *Query* traffic. Consider the scenario where values at all nodes rise. SLAT is no better than temporal suppression, where all nodes send *Trigger*. SLAT-A, however, is able to aggregate messages significantly. Specifically, if u_i has f children, it sends a message listing only one value, rather than f.

3.2.2 HAT

We finally present HAT, our most sophisticated algorithm, which fully leverages network topology. HAT stands for *hierarchical adaptive thresholds*. Its fundamental advance is to empower nodes within the network to make local decisions, rather than simply act as conduits. We call this technique *constraint localization*, where the nodes can be seen as supporting a network of continuously monitored and dynamically adjusted constraints in support of a query. The main improvements of HAT over the other algorithms are that values are only propagated upward until they reach an ancestor with threshold higher than it, and that queries do not propagate as far

downward, stopping at nodes with threshold below the fallen max value. We introduce a stronger invariant to enable constraint localization.

INVARIANT 2. *For each node u having threshold τ, with parent u_p having threshold τ_p, $\tau \leq \tau_p$.*

With this invariant, HAT captures the advantage of aggregation in SLAT-A while avoiding its massive querying in the second stage. A perhaps more subtle point is that HAT can exploit aggregation along the time dimension as well to further reduce upward message traffic. We will substantiate these points later in this section.

HAT is built from SLAT with some key changes. HAT performs hierarchical maintenance of thresholds. Each node tracks the thresholds of each of its child nodes. The root, unlike in SLAT, maintains just the thresholds of its immediate children. We organize HAT into two stages, node-initiated reporting and root-initiated querying, with discussion of threshold setting for each.

Node-Initiated Reporting As before, whenever a node u_i receives a *Trigger* packet from a child node, u_c, it sets a flag indicating u_c requires a threshold update. If u_i either breaks its threshold τ_i or receives from a child a *Trigger* packet with value v_{Tr}, such that $\max\{v_i, v_{Tr}\} > \tau_i$, it sends a *Trigger* packet to its parent, u_p. If u_i receives several *Trigger* packets, it only passes on the highest value among them and v_i itself.

Compared with SLAT-A, which also aggregates *Trigger* packets, HAT performs additional local filtering based on τ_i. Specifically, if u_i receives a *Trigger* packet from u_c such that $v_{Tr} < \tau_i$, u_i does not forward the packet upward to u_p. Instead, it sends a *ThresholdUpdate* packet listing its own value, v_i, back to u_c. It also updates its threshold setting for u_c (details to be given later).

When a node u_i receives a *ThresholdUpdate* packet listing value v_{TU}, it first modifies its threshold value τ_i to within the range $[max\{v_i, v_{Tr}\}, v_{TU}]$, i.e. somewhere below the *ThresholdUpdate* value and above the value that previously caused it to send a *Trigger* (we defer the exact choice of threshold within this range to Section 4). Then, u_i sends *ThresholdUpdate* packet on to all child nodes it has flagged as needing a threshold update. Notice this scheme implies that all nodes along the path to $u_{\max}$ will have thresholds equal to $v_{\max}$.

Root-Initiated Querying The root first determines v^* from all values received in *Trigger* packets and, if $u_{\max}$ did not report, the stored value of $u_{\max}$. The root only sends *Query* packets with v^* to those of its child nodes with threshold values greater than v^*. In turn, each node receiving a *Query* packet only forwards it to its children with thresholds greater than v^*. Any nodes with values greater than v^* send the their values to the root in *Reply* packets, as before. If a *Reply* is sent, all nodes along the path from the source of the *Reply* to the root set their thresholds to the *Reply* value. Like *Trigger*, *Reply* packets are aggregated and only the maximum of them is transmitted.

A node has freedom to lower its threshold if no queried nodes in its subtree exceed v^*. Due to Invariant 2, however, node u_i cannot lower τ_i below any of its child thresholds. Therefore, τ_i must be set in the range $[\max\{v_i, \tau_{c_1}, \tau_{c_2}, \ldots, \tau_{c_f}\}, v^*]$, i.e., between the highest among the node's value and all child thresholds, and the *Query* value (again, we defer the exact choice of threshold within this range to Section 4). The requirement that the new threshold does not exceed v^* ensures Invariant 1, because v^* is a lower bound on the current maximum value.

Discussions The invariants maintained by HAT together establish a hierarchy of thresholds in the network and essentially disperse information about node values that empowers nodes to make decisions to reduce network traffic. A major advantage of HAT is its ability to "short-circuit" messages headed to the root or leaves, which was not possible with the other algorithms (with the exception of SLAT-A in the case of aggregation of messages to the root). To illustrate, first consider messages passed towards the root. If a node generates a value that breaks its own threshold, it may only be forwarded a couple of hops before meeting an ancestor node with a threshold higher than its value. Meeting such a threshold guarantees the value is not the maximum, so we need not pass it to the root. Next, consider messages passed downward from the root. While sending *Query* packets, a node avoids forwarding it to any child with threshold lower than the *Query* value. This optimization works because no node in that child subtree can be higher than the *Query* value; the child threshold is always no less than the highest value in the subtree.

The second advantage of HAT is more subtle. Both SLAT-A and HAT are able to drop values at a node in favor of the highest value received in a round. This represents sharing of work between multiple nodes, i.e., if many nodes rise in value, the root need not know about all of them. In the case where nodes are rising in value, but with only a few rising each round, however, SLAT-A cannot do much sharing, while HAT still does. Consider two nodes, u_i and u_j, with the same parent, u_p, where u_i rises considerably in one round, and u_j in the next. Suppose u_i sends a *Trigger* that may reach the root before meeting a threshold higher than it. All nodes along that path, including u_p, then receive a *ThresholdUpdate*, raising their thresholds considerably. In the next round, u_j rises in value such that $v_j > \tau_j$. It sends a *Trigger* to u_p, which already has a higher threshold such that $\tau_p > v_j$ (because of the *ThresholdUpdate* in the previous round). Hence, the *Trigger* packet from u_j is not passed on (in contrast, it would be passed on in SLAT-A); instead, u_p sends a *ThresholdUpdate* back to u_j. The savings in the second round is proportional to the distance from u_p to the root.

Therefore, HAT is able to exploit aggregation along the time dimension, in that *nodes benefit from work done on behalf of other nodes across rounds*. Contrast this with SLAT-A where nodes only share benefit *within* single rounds. Moreover, we recall the use of rounds is an approximation for purposes of simplifying discussion on continuous queries. In some cases, we probably cannot always depend on all nodes synchronously taking measurements and sending packets. With HAT, this possibility is not as big a concern, because *Trigger* packets do not need to act synchronously and meet up to realize the benefit of aggregation.

Comparative Example We now illustrate the behavior of the algorithms with a concrete example. Figure 2 is divided into four parts to demonstrate how each of RC, SLAT, SLAT-A, and HAT function in a common scenario. Each node is labeled with its current measured value and state: in the case of RC, the cached ranges, and in the other cases, thresholds. Each edge is labeled with the message sent across it, if any. RC, in Figure 2(a), is distinguished from the others by node d. It is the only algorithm that may transmit a value when it falls. SLAT, in Figure 2(b), and SLAT-A, in Figure 2(c) are distinguished by the longer message SLAT sends up from node b. SLAT, recall, appends all *Trigger* messages together, while SLAT-A aggregates by dropping all but the highest. HAT, in Figure 2(d), is distinguished from the others because it leverages b's high threshold to short-circuit all messages originating from under b, and sends nothing to node a.

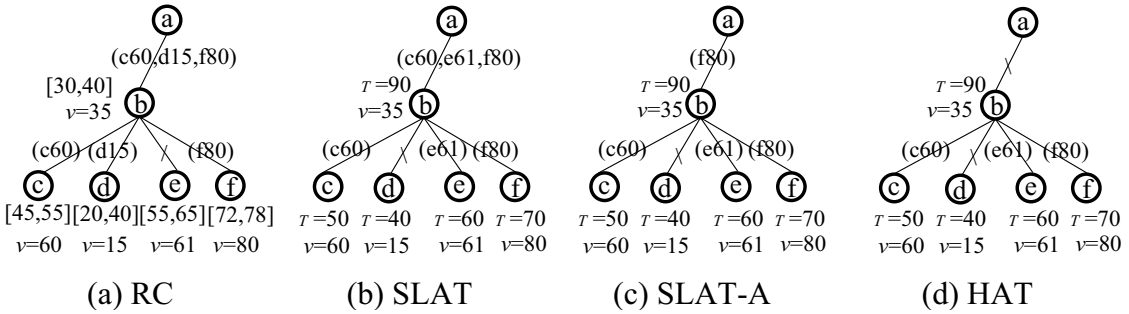

Figure 2: Algorithm comparison.

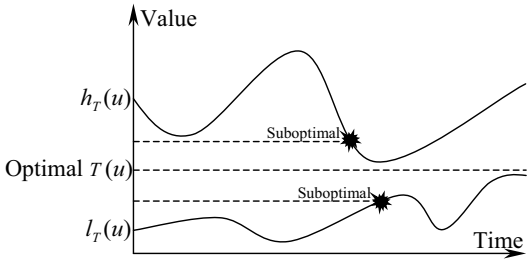

Figure 3: Goal of threshold setting.

4 Policies for Setting Thresholds

Our description of HAT leaves open questions about setting thresholds. The necessity of adjusting thresholds arises in both the upward direction toward the root, when thresholds must be raised, and the downward direction toward the leaves, when thresholds must be lowered. A node u's threshold $\tau(u)$ can be legally set anywhere below the parent threshold, which we denote $h_\tau(u)$, and above the maximum of the node's own value and all of its child thresholds, which we denote $l_\tau(u)$.

We term processes for setting thresholds *policies*, as opposed to *algorithms*, simply to distinguish them from the processes in Section 3. The policies do not compete with the algorithms, but complement them. We can employ a number of different threshold-setting policies. Some simple examples include setting the threshold of node u equal to $h_\tau(u)$, or setting it to the average of $h_\tau(u)$ and $l_\tau(u)$.

Intuitively, the goal when setting threshold $\tau(u)$ is to maximize the time until the next update to this value. This threshold needs to be updated if $h_\tau(u)$ falls below $\tau(u)$, or $l_\tau(u)$ rises above $\tau(u)$. Both events cause energy to be spent on communication to adjust $\tau(u)$. Therefore, maximizing the time until the next threshold adjustment maximizes energy savings. This goal is shown in Figure 3. Notice in this example $h_\tau(u)$ and $l_\tau(u)$ are always relatively far apart over time, but the gap between their extremes leaves only a small space to set a threshold that will not require adjustment.

Theoretical Hurdles We now consider various threshold adjustment policies. The chief difficulty in a clean theoretical analysis is the observation that the threshold settings processes at various nodes are not independent. The reason is that a reasonable threshold setting scheme must set monotonically non-decreasing thresholds on any path from a leaf to the root; without this invariant, we cannot guarantee correctness of the MAX-monitoring algorithm unless all nodes are probed. The monotonicity invariant implies that adjusting the threshold at a node might require adjusting the threshold at many other nodes in order to ensure correctness. This issue leads to the question of whether to *proactively* drop the threshold when $l_\tau(u)$ falls, or *reactively* drop it only when $h_\tau(u)$ drops. The cost of each approach depends on the pattern of future values.

We illustrate this difficulty with a concrete example. Consider a subtree T in which all nodes had high values of about v sometime in the recent past. The thresholds of all nodes in this tree are close to v. Suppose these values now drop, but there is a node u in a different subtree still at value v. If the threshold setting is reactive, when u drops, the thresholds for nodes in T, in response to a series of *Query* packets (with dropping values), gradually drop. Suppose u drops slowly. The following would occur incrementally in response to the *Query* packets. First, the threshold at the root of T drops. Then, when its value reaches the thresholds of its children, they start dropping and so on. This incurs huge communication cost. The alternative is for the nodes in T to proactively drop their threshold when the values in T drop. This proactive approach would result in lower communication costs when u does start dropping. Choosing the better alternative, however, requires predicting whether u drops before the values in T rise again—had the values in T risen before, being reactive in dropping thresholds would be better. The subtree would not have been queried in this case. No thresholds would be lowered, so being reactive would cost nothing.

4.1 Policy and Analysis for the Worst Case

We first design a threshold-setting policy in the framework of competitive analysis. In this framework, there is no model of future values; they are chosen by an adversary and are therefore worst case. The performance of the policy is measured against the cost of the best admissible policy with knowledge of the entire future. Any admissible policy should observe the following restrictions:

1. It is a threshold-setting policy, meaning there is a threshold value at every node which is also known to the parent of the node. The thresholds are monotonically non-decreasing on any path from a leaf to the root.
2. The threshold at a node is at least the current largest value in the subtree rooted at that node.
3. The threshold at a node is at most the current largest value in the entire tree.
4. Whenever the threshold at a node changes, the node incurs one unit of communication cost.

One-Level Tree Assume each communication between the root and a node incurs unit cost. Further assume the node values are drawn from the range of integers $\{1, 2, \ldots, K\}$. In general, the values can come from any finite, ordered domain, as long as they can be discretized and mapped to K distinct values. To facilitate analysis of the continuous query, we partition the course of its execution into *phases*, each containing a number of consecutive rounds.

For $u_{\max}$, which is the node currently responsible for the maxi-

mum value $v_{\max}$, the root is notified whenever this value changes; this cost in unavoidable by any policy that ensures correctness of the algorithm. From this point on, we focus on a node u that is not currently $u_{\max}$ and consider its behavior in each phase.

In each phase, u maintains two values $L(u)$ and $H(u)$, which are, respectively, the largest value at u since the beginning of the phase, and (u's belief of) the smallest value of $v_{\max}$ since the beginning of the phase. Note u can accurately track $L(u)$ at no additional cost. The node also maintains a threshold value $\tau(u)$ which is always set to $\frac{H(u)+L(u)}{2}$. The root always knows the correct value of $\tau(u)$ for all u.

Our one-level threshold-setting policy designed for the worse case can be summarized as follows.

1. If the maximum value $v_{\max}$ (known at the root) falls below $\tau(u)$, the root contacts u. The node sets $H(u) = v_{\max}$ and updates $\tau(u)$ accordingly.
2. If $L(u)$ becomes equal to $\tau(u)$, the node contacts the root and gets the current value of $v_{\max}$. It sets $H(u) = \min\{H(u), v_{\max}\}$ and updates $\tau(u)$ accordingly.
3. The phase ends if $H(u) = L(u)$. In this case, $H(u)$ is reset to the current value of $v_{\max}$, the overall maximum value at the root, and $L(u)$ is reset to the current value at the node u. The threshold $\tau(u)$ is set accordingly.

The analysis below compares this policy against the optimal policy that "magically" knows the future sequence of value changes.

THEOREM 1. *The cost incurred by the threshold-setting policy is at most* $\log K$ *times the cost of the optimal policy for the same sequence of node values.*

PROOF. Under our policy, in each phase, the maximum number of times a node u contacts the root is $\log K$. The reason is that every time a contact is made, the gap $H(u) - L(u)$ shrinks by a factor of 2, and the initial gap is at most K. On the other hand, we claim that the optimal policy must have incurred one unit of communication cost as well. The reason is that when the gap $H(u) - L(u)$ shrinks to zero, the smallest value of $v_{\max}$ during that phase becomes at most the highest maximum value at the node during the same phase. For any fixed setting of threshold at u in that phase, either $v_{\max}$ would fall below the threshold, or the node's value would rise above the threshold. Therefore, any setting of the threshold at the beginning of the phase has to be updated during the phase, incurring at least one unit of communication cost. The proof is complete. □

Multi-Level Trees We present a simple policy for multi-level trees based on the one-level policy discussed above. Without loss of generality, we assume only the leaf nodes record values. Each leaf runs the one-level policy, while each intermediate node simply sets its threshold to be the maximum of its child thresholds. This maximum can be maintained at an intermediate node with no additional cost because every message between the root and one of its descendants passes through this node.

We assume that a message incurs a unit of cost for each edge on which it travels. The performance of the above policy for multi-level trees is compared with the best possible policy, which is required to incur at least one unit of communication when the threshold at any node is updated.

THEOREM 2. *The cost incurred by the threshold-setting policy for a multi-level tree of depth* D *is at most* $2D \log K$ *times the cost of the optimal policy for the same sequence of node values.*

PROOF. For a one-level tree, the policy incurs at most $\log K$ times the communication cost of the best possible policy. For a multi-level tree, every communication between the root and a leaf node now translates into D units of communication down the tree, and D units up the tree. For each leaf node, the minimum $v_{\max}$ during each phase is at most the maximum value at that leaf node during the same phase. Thus, any setting of the threshold value at that node at the beginning of the phase has to be updated during the phase, costing one unit of communication. For the same node, the multi-level policy incurs at most $2D \log K$ units of communication during this phase. Therefore, the overall cost is also within a factor $2D \log K$ of the optimal. □

We believe this worst-case bound is the best possible. In other words, for more sophisticated threshold-setting policies, there exist input instances on which the performance against the optimal threshold-setting policy for that sequence will be no better than the bound indicated in the theorem above. Note these bounds are "worst-case" in the sense that the input sequence is chosen in an adversarial fashion.

In practice, the input sequence may have stable characteristics that can be modeled and exploited by a threshold-setting policy. We next consider such policies.

4.2 Model-Based Policies

A model that predicts the future behavior of node values can be exploited in setting thresholds. However, rigorous modeling of the inputs to a threshold-setting policy at node u is fairly complicated. Recall from the beginning of this section that the lower bound $l_\tau(u)$ is roughly based on the maximum value in u's subtree in each round, while the upper bound $h_\tau(u)$ from the parent node is roughly based on the maximum value in other parts of the network. It is not at all obvious how to model these quantities, let alone gather all information to construct models and then deliver them to each node. Instead, we take the simplifying step of modeling each sequence of $l_\tau(u)$ and $h_\tau(u)$ values as a random walk. This simple model is easy to analyze and maintain, and can lead to practical policies.

For a node u, we assume that $l_\tau(u)$ (or $h_\tau(u)$) follows a random walk W_l (W_h) where each step is a drawn from a normal distribution with mean μ_l (μ_h) and variance σ_l (σ_h).

Setting the Threshold Using a Random-Walk Model The problem for setting the optimal threshold is to maximize the expected time until either W_l or W_h generates a value crossing it, i.e., the *hitting time* to the threshold value. To simplify calculation (and to a good approximation), we characterize W_l (and similarly W_h) by an envelope, so the value produced by W_l at time t (assuming the walk starts at value 0) is bounded by the range $\mu_l t + [-\sigma_l\sqrt{t}, \sigma_l\sqrt{t}]$ with high probability. The two envelopes are illustrated in Figure 4. To set the threshold, we find the value τ^* that maximizes the time until either envelope contains τ^*. Note that τ^* approximates the hitting time if the two random walks are drifting in opposite directions (e.g., $\mu_l \geq 0$ and $\mu_h \leq 0$). The approximation may not work if the walks are drifting in the same direction; even if that is the case, however, τ^* is a valid threshold setting.

The calculation for τ^* requires solving for the time when the two envelopes first intersect. We omit the equation here, but it can be derived with standard techniques, given μ and σ values, and the initial gap between the two walks.

Learning the Random-Walk Model To compute the optimal threshold, we need the values of random-walk parameters σ_l, σ_h, μ_l, and μ_h for each node. A simple method for learning the parameters of random walk W_l (and similarly for W_h) is to use linear regression to fit a straight line on the set of observed $l_\tau(u)$ values. Drift μ_l

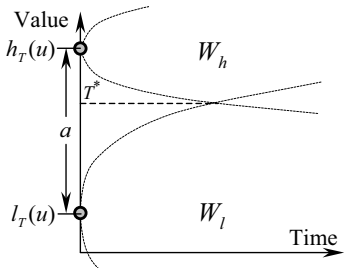

Figure 4: Finding threshold with envelopes of random walks.

is the slope of this line, calculated as the correlation coefficient between time and values. Variance σ_l is the root mean square error of the fit.

Our use of linear regression assumes that random walks of $l_\tau(u)$ and $h_\tau(u)$ are long-running in their behavior. For example, regression will be effective in identifying outliers only after enough data is collected. If node memory is severely constrained, we can use techniques from [1] to maintain statistics.

Discussions We note that other models can also be plugged into this framework. For example, temperature is likely to increase in the first half of a day and decrease in the second half. Nodes can maintain two models, one for each case. At threshold calculation time, the node can attempt to fit its recent data to both models, and use whichever fits better. While a simple example, this approach has favorable characteristics. Model fitting is done with simple calculations, and no input from the root is needed to tell nodes when conditions have changed, like that the hottest part of the day is over.

Finally, while model-based policies enable well-informed threshold settings, recall from Section 3 that in HAT a parent node must maintain its children's thresholds. When a node adjusts its threshold using a locally maintained model, it must use an additional message to transmit the new threshold to its parent. As future work, we plan to investigate using the model-based policies while avoiding this overhead. In the next subsection, we investigate simpler policies that do not carry this overhead by design.

4.3 Adaptive Pressure-Based Policies

Recall in HAT, a node u's threshold must always be set between $h_\tau(u)$ (from its parent) and $l_\tau(u)$ (from its subtree). The threshold can be broken by either $h_\tau(u)$ dropping or $l_\tau(u)$ rising. For adaptive pressure-based policies, we make a simplifying assumption about the inertia of change: If $h_\tau(u)$ falls and breaks the threshold, it will likely continue to fall; likewise, if $l_\tau(u)$ breaks the threshold, it will likely continue to rise. Later in this subsection we will discuss how to handle transient behaviors that temporarily violate this assumption.

A pressure-based policy sets the threshold to try to avoid the impending pressure from a particular direction. In its basic form, such a policy sets the threshold $\tau(u)$ of node u to $\alpha h_\tau(u) + (1 - \alpha) l_\tau(u)$, where α is an adjustment factor between 0 and 1. The parent of u tracks not only the current $\tau(u)$, but also α (and if α can change, the policy by which it does). The following discussion, which supplements the description of HAT in Section 3.2.2, shows how HAT accomplishes updating of $\tau(u)$ at both u and its parent efficiently.

- In the case that $\tau(u)$ is broken by a rising $l_\tau(u)$ at u, u sends up a *Trigger* packet listing the new $l_\tau(u)$ value to its parent, which in turn responds with a *ThresholdUpdate* packet, with the $h_\tau(u)$ vale for u. At this point, both u and the parent have all information needed to synchronously update $\tau(u)$—$l_\tau(u)$ and $h_\tau(u)$ values—without additional communication.
- In the case that $\tau(u)$ is broken by a dropping $h_\tau(u)$ in a *Query* packet sent down by u's parent, assuming u does not send back a *Reply*, the parent does not know the current value of $l_\tau(u)$, and therefore cannot mirror the calculation of $\tau(u)$ at u. Hence, in response to the *Query* packet, u must either respond with the updated $\tau(u)$, or simply lower $\tau(u)$ to the new $h_\tau(u)$ in the *Query* packet. The latter setting saves a response message to the parent; the parent, which sent and knows $h_\tau(u)$, assumes this setting if it does not receive a response.

Before presenting the adaptive pressure-based policies, we first describe pressure a few policies with fixed α values: *ceiling*, *middle* and *floor*. These fix α at, respectively, 1, 0.5, and 0. *Middle* is similar in spirit to the policy designed for the worst case in Section 4.1. It is something of a misnomer to call these adaptive, since they actually ignore pressure, but they are useful for comparison. *Ceiling* is clearly the optimal policy when node values can only rise, while *floor* is optimal when values only fall. *Middle* is suboptimal in either of these cases, and motivates dynamically adjusted α values.

We now introduce two adaptive policies: *adaptive-linear* and *adaptive-exponential*. Both initialize α to 0.5. *Adaptive-linear* also employs a constant, c, between 0 and 1. If the threshold is broken from below by $l_\tau(u)$, we increment α by c, whereas if the threshold is broken from above by $h_\tau(u)$, we decrement α by c (while observing the constraint $0 \leq \alpha \leq 1$). For *adaptive-exponential*, if the threshold is broken from below, we set α to $(\alpha+1)/2$; if the threshold is broken from above, we set α to $\alpha/2$. In scenarios such as node values always rising or always falling, both adaptive policies reach the optimal state, with *adaptive-exponential* reaching sooner. In less extreme scenarios, though, *adaptive-exponential*'s more aggressive adjustments arguably make α flip-flop between values and encourage thresholds to be alternatively broken in opposite directions when a better threshold choice lies somewhere in between. *Adaptive-linear*, with a small enough c value, has a better chance of avoiding this problem.

Discounting Short-Term Behavior One disadvantage of the previous threshold setting approaches is that a very short-term change in a node's value can modify a significant number of threshold settings (e.g., those on the path from the node to the root). These threshold settings would not reflect the steady-state values of the subtrees after the aberration ceases. One approach for rectifying this problem is to use a simple local model (such as a random walk from Section 4.2) at each node, not for predicting optimal threshold values, but for identifying short-term aberrations. These aberrations are sent toward the root (so that the root still can compute max with these), but not in *Trigger* packets. Therefore, they have no effect on threshold settings. The node transmits its value every round until the value conforms to the model. Obviously, persisting aberrations suggests the model should be changed.

5 Enhancements and Extensions

Failure Tolerance Our discussion of HAT so far has assumed fixed assignments of parent-child relationships. HAT can be made flexible to adjust assignments in response to failures. Suppose node u_i currently has parent u_p. From the perspective of u_i, failure occurs when the connection between u_i and u_p breaks, because either u_p or the communication link between u_i and u_p becomes nonfunctional. Using a reliable communication protocol where nodes receive acknowledgment that their messages are delivered, u_i can quickly detect such a failure. If HAT were to do nothing in response to this failure, we would lose all data generated in the subtree rooted at u_i. Therefore, u_i must be able to recover from this failure.

To this end, u_i broadcasts a probe message listing its current threshold $\tau(u_i)$ to try to identify a new parent among its neighbors. Each node u_j checks if it is a descendant of u_i in the current hierarchy. This investigative procedure starts at u_j itself and progresses towards the root, until it reaches a node whose threshold is higher than $\tau(u_i)$ (in which case the answer is negative), or the node u_i itself (in which the case the answer is affirmative). This procedure works because of Invariant 2. A node u_j replies to u_i's probe with $\tau(u_j)$ only if u_j has confirmed it is not a descendant of u_i. The nodes who replied constitute candidates for u_i's new parent. Assuming all choices have equal link quality, u_i chooses the candidate with the highest current threshold as its new parent, u_p^{new}. To maintain Invariant 2, any ancestor of u_p^{new} (as well as u_p^{new} itself) with threshold lower than $\tau(u_i)$ raises its threshold to $\tau(u_i)$.

As a heuristic, u_i chooses the candidate with the highest threshold. The intuition behind this heuristic is as follows. In the case that t_p^{new} is already higher than $\tau(u_i)$, choosing the highest threshold allows $\tau(u_i)$ to rise in the future (in response to rising values in its subtree) to as high a value possible before breaking t_p^{new}'s threshold; in other words, more *Trigger* packets from u_i can be stopped by a parent with a higher threshold. In the case that $\tau(u_p^{new})$ must be raised to accommodate u_i, raising the highest threshold makes the increase as small as possible. The bigger this increase, the more vulnerable it is for the parent to be queried in rounds when the $\max$ value falls.

Normally we prohibit u_i from choosing a descendant as its new parent, but in the case that there are no other candidates, this option can be invoked; the former descendant must in turn chooses a new parent. In the worst case, as long as u_i can still find a path to the root, we can form a new hierarchy of thresholds that avoids the failure. We omit the details here. Finally, it is possible that u_i cannot find a path to the root, meaning that the failure has partitioned the network. This case might happen when failure occurs an area with low network coverage. In this case, whether running HAT or any other application, nothing can be done to collect data from the network partition separated from the root. This case argues for having an adequate node density for a sensor network.

Dynamic Adjustments for Performance While a node must find a new parent if failure occurs, a node can also opportunistically change parents for performance reasons. Other factors being equal, a node is best assigned to the parent with highest threshold. Switching from a low-threshold parent to a high-threshold one means the node's value has to rise higher before *Trigger* packets can travel above the parent. Also, grouping subtrees whose maximum values behave in similar ways is beneficial. Intuitively, imbalance in child thresholds causes the parent's threshold to be set unnecessarily high, resulting in extra *Query* packets being sent down to the parent. Therefore, a parent may ask a child to switch to another parent if this child's threshold is far higher than others. Of course, as with any dynamic adjustment, we must ensure that the thresholds are relatively show-changing. Otherwise, having paid overhead cost of adjustment, the new state may quickly be no better or worse than the old.

As hinted, quality of communication links can also play a role in parent selection. If a child-parent link has poor (but still functioning) connectivity, the extra transmissions required to establish communication may easily outweigh the benefit from choosing the highest threshold parent possible. HAT must compare these relative benefits in changing parents.

One-Sided Quantiles Continuous MAX is a specific case of the *continuous one-sided quantile query*, which returns the nodes ranking at and above the p-th percentile. We can easily extend HAT to this more general problem. In the first round, the root determines the nodes in the result set and makes note of the lowest value in this set, v_l, from node u_l. All result nodes are designated as such with *ResultDesignate* (analogous to *MaxDesignate*) messages and instructed to temporally monitor themselves. If a result node falls below v_l, the network is queried to find a potential replacement or else lower v_l to this fallen value. Nodes can also join the result set, as before, by breaking thresholds and sending *Trigger* messages. If a *Trigger* reaches the root with value greater than v_l, the originating node will displace u_l from the result set.

This translation from MAX to one-sided quantile can be further extended with some energy-saving compromises. First, the user may not insist on the result set containing exactly $(1-p)n$ nodes. Hence, we can allow its size to vary within $(1-p)n \pm \epsilon$. If a node in the result set falls below v_l, and removing it does not violate the error margin, it is evicted and not replaced. Not replacing the result node saves the potentially huge cost of querying the entire network (particularly if the falling value dropped a large amount). With luck, another node will soon join the result set by exceeding v_l, balancing the effect of nodes leaving the result set. Whenever the error margin is violated, the result is restored to $(1-p)n$ by querying the network.

A second compromise is to not maintain the exact values of each node in the result set. For example, a user may be interested in a tight range $[v_l, v_h]$ for the values in the query quantile. In that case, temporally monitoring the exact values in the result set is overkill. Instead, we can apply a specialized version of RC to result set nodes. Each such node is assigned an upper threshold and lower threshold. One option is to set all upper and lower thresholds of result nodes (except the two producing v_h and v_l) to v_h and v_l respectively. The problem, however, is if the root receives a *Trigger* from a node not currently in the result and this node forces eviction of the lowest-valued result node, the root will have very little information to narrow down which of the remaining result nodes may have the current lowest value. The same problem occurs for determining the new maximum value when the current maximum drops. The solution is to assign each result node its own thresholds, such that the upper threshold is no greater than v_h and the lower threshold is no less than v_l. These thresholds should be set wide enough apart so that nodes do not repeatedly break them, but narrow enough so that not all nodes need be queried to find new v_h or v_l.

6 Experimental Results

We perform experimental analysis using our own network simulator. Nodes are modeled as Crossbow Mica2 motes [5]. We use a generic MAC-layer protocol and only account for the energy cost of communication. The simulated network area is a rectangular grid. Each grid point represents a square meter, and is assigned some value at the beginning of each round. Nodes are randomly placed at grid points and monitor the values assigned there. Node radio range is fixed at 50 meters. Because we are interested in continuous queries, we assume a long-term benefit for installing a query plan in the network. Because the cost of installing a query plan into the network is amortized over many rounds, we ignore this initial cost and instead evaluate performance over the subsequent rounds.

The goal of our experiments is a thorough comparison of all presented MAX algorithms: temporal suppression (TS), range caching (RC), SLAT, SLAT-A, and HAT. We also test HAT using different threshold setting policies to see their impact. We run a variety of experiments. Some use very specific conditions to expose particu-

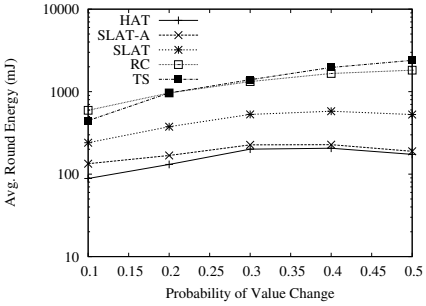

Figure 5: Random behavior.

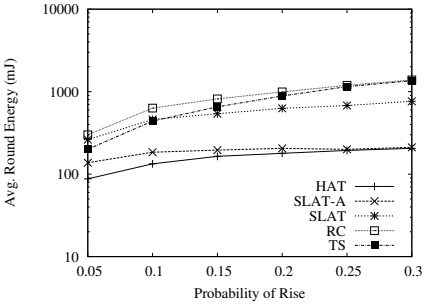

Figure 6: Randomly rising values.

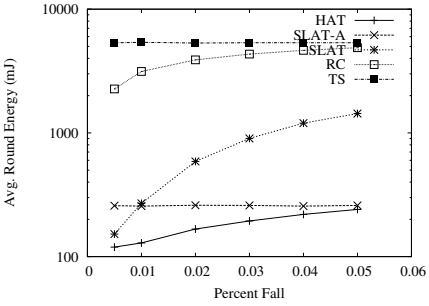

Figure 7: Uniformly falling values.

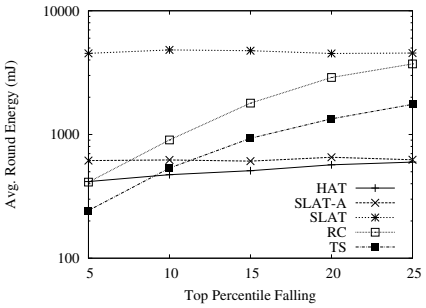

Figure 8: Highest values dropping.

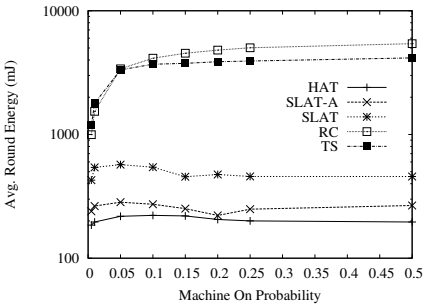

Figure 9: Factory setting.

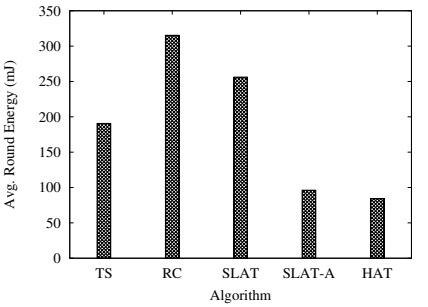

Figure 10: Intel Berkeley Lab data.

lar algorithm strengths and weaknesses. Others are realistic scenarios, using either simulated or real-world data.

All experiments, unless otherwise noted, use the same basic parameters. The network consists of 200 nodes in a 400 by 400 meter area. Each test is run for 21 rounds, with the first-round results discarded. Notice we use log-scales on the graphs' y-axes due to significant performance disparities between algorithms.

Random Behavior We begin with a simple experiment to gauge the relative performance of the algorithms. All nodes start with value equal to one another, and they each have equal chance of becoming the maximum. With some fixed probability, varied across runs, each node changes value in each round, or else remains unchanged. In the case of a change, a node's value v is adjusted randomly to a value uniformly chosen from $[v - 100, v + 100]$. This behavior corresponds to a random walk whose step is governed by a bounded uniform distribution. The results are shown in Figure 5. The three query-aware algorithms, SLAT, SLAT-A, and HAT, outperform TS and RC at all change probabilities. We see that energy spent by TS increases proportionally with change percentage, which is expected since each changing node triggers a report. RC also increases quickly at first, but then plateaus. The reason is, as change probability increases, each node changes more often over the course of the experiment, widening the cached ranges for most of them (except those competing for the maximum), to the point where changes in low values do not necessarily trigger reports.

The three query-aware algorithms considerably outperform the other two, with SLAT-A and HAT outperforming SLAT. SLAT-A and HAT perform relatively similar in this experiment (we differentiate them next). Like RC, these three algorithms plateau. The more nodes change value, the higher the expected overall maximum after some time (the variance in values increases over the course of the random walk). When nodes then break their thresholds they receive higher new thresholds, and can suppress most of their subsequent changes.

Differentiating HAT and SLAT-A The next two experiments are designed primarily to highlight HAT's two major advantages over SLAT-A (although we also show results of other algorithms in figures for completeness). These advantages, again, are that HAT's hierarchical thresholds allow the benefit of aggregation to be realized across query rounds, and that HAT is not forced to query all nodes when the maximum value drops.

- **Randomly Rising Values** The first advantage is shown with a scenario similar to the previous experiment, except that nodes can only increase in value. We vary the probability that a node rises across runs. The results are shown in Figure 6. When the probability is low, HAT significantly outperforms SLAT-A, while as probability rises, the gap closes. At low probability, only a few nodes rise and possibly break their thresholds. In this case, SLAT-A must transmit these values all the way to root, to be followed by *ThresholdUpdate* messages all the way back down. Because the amount of transmissions is small, there are not many opportunities for aggregation, so transmission costs are mostly unshared between nodes. In contrast, HAT can still achieve sharing across rounds. If a particular node breaks its threshold and all ancestor thresholds, it will cause all these thresholds to rise. At this point, the cost is the same as for SLAT-A. If in the next round, however, another node breaks its threshold, but then encounters a previously raised ancestor threshold higher than its value, the *Trigger* stops there. With SLAT-A, it does reach the root. When probability of rise is higher, the amount of traffic increases faster for HAT, lessening its advantage over SLAT-A. SLAT-A, even at low probabilities, already has high traffic at the upper level edges, so its increase is relatively milder. For HAT, the increased probability causes more nodes to rise, and therefore increases the probability that a rising node becomes the new maximum, or at least the maximum within a large subtree. This effect creates traffic at the upper level edges, bringing it up to the levels in SLAT-A.

- **Uniformly Falling Values** The second advantage of HAT over SLAT-A is demonstrated in a scenario where all nodes start at random values and drop each round. We vary the percentage by which nodes drop in value over a series of runs. Within each, however, all nodes drop by the same percentage. Therefore, whatever node starts as maximum will remain the maximum over the entire run. The results are plotted in Figure 7. Performance for SLAT-A is constant since, regardless of percentage drop, all nodes are queried every round using the lowered maximum node's value. Since all nodes are still below this value, no nodes reply to the queries. HAT, on the other hand, exploits the case where the max value drops only slightly. When the root sends a *Query* packet with the lowered maximum value, it may still be much higher than many nodes' thresholds and, therefore, any nodes in their subtrees. The *Query* packet is then dropped at these points, saving the cost of propagating it further down. As the percentage drop increases, more nodes must be queried. In the worse case, if the maximum value drops below the previous minimum value, all nodes must be queried, raising HAT's cost to that of SLAT-A.

Differentiating RC and SLAT In our initial experiment to establish a relative ordering of algorithm performance, RC performs poorly, only beating TS. Compared to query-aware algorithms, however, RC may have some better sense of second-place (and lower ranked) values. If the maximum value drops, RC can use its list of cached ranges to compile a possibly short list of potential new maximum nodes. If the maximum value drops enough, the query-aware algorithms must query the entire network. We pose a scenario to demonstrate that RC outperforms SLAT when the second-place information is useful. Nodes start with random values. In each round, a fixed top percentile of nodes have their values cut in half. We vary this percentile over the series of runs. The results are shown in Figure 8. SLAT performs consistently throughout; no matter how many values are cut, the maximum one always drops enough to require extensive querying to find the new maximum. RC outperforms SLAT throughout, but is itself best when fewer values are cut. The reason is simply that fewer values changing means fewer cache ranges are violated, so fewer reports are sent. In all cases, the new maximum value is found by querying similarly sized sets of candidate nodes. On the other hand, we see SLAT-A and HAT still perform well compared with RC, because the benefit of aggregation dominates. Interestingly, like RC, HAT also performs better when the percentile is lower. We believe this phenomenon can be explained by the fact that HAT's hierarchy of thresholds retains more information about lower ranked values than the single-level scheme of SLAT and SLAT-A.

Factory Setting The next experiment simulates the behavior of heat dispersion on a factory floor. Machines are spaced evenly throughout the grid space, while sensors are placed randomly as before. All machines are initially off, so all nodes initially have value 0. At each time, a machine is powered on with some probability. If powered on, a machine generates some large amount of heat, raising the local temperature to a ceiling amount at that point. Subsequently, heat is dispersed from each grid point to all neighboring grid points. In each time increment, $T(i,j)$, the temperature at grid point (i,j), is updated using the previous values according to the following formula from [3]:

$$T(i,j) \leftarrow T(i,j) + a\Big(T(i+1,j) + T(i-1,j) + T(i,j+1) + T(i,j-1) - 4T(i,j)\Big).$$

Here, $a \leq 0.25$ is a dispersion factor. We allows 10 time increments between two consecutive query rounds to ensure more than a trivial amount of change happens in between. Note MAX would be trivial if sensors were overlaid with machines. Our random placement of sensors and densities of sensors and machines ($1/800m^2$ and $1/1600m^2$, respectively) make this occurrence unlikely. Even then, the machine would need to be on or have been on very recently to give the sensor a high chance to be the maximum. From the simulation traces, we do not find single nodes dominating the result, confirming MAX is non-trivial in this setting. This setting combines a number of features that influence algorithm performance. Some nodes are much more likely candidates for maximum; these are the ones closest to machines. Other nodes can afford to set lower thresholds. Finally, a node that becomes the maximum can easily drop in the following round when its heat disperses.

Figure 9 shows the results obtained by varying the probability that machines are on across runs. As expected, the higher this probability, the greater the number of nodes exhibiting change. TS and RC start out performing worse than the other algorithms, and expend even more energy as probability increases. Among the others, HAT outperforms SLAT and SLAT-A. SLAT-A is, of course, a closer competitor to HAT than in Figures 6 and 7. We also see that SLAT is considerably more expensive than SLAT-A and HAT. Its handicap of larger network messages simply does not overcome its advantage over SLAT-A of limited querying when the maximum falls. The machine-on probability does not seem to have a major impact on performance among SLAT, SLAT-A and HAT. It does appear, however, at the higher probabilities, their performances improve. The reason is that the set of nodes with significant chance of becoming the maximum shrinks to those very close to machines. At lower probabilities, many or all of these "best candidate nodes" may have had their respective nearby machines off for several rounds, allowing opportunities for nodes more distant from machines (but machines which have been on recently) to become max. At higher probabilities, it is very unlikely that all of the machines contributing heat to the best candidates have been repeatedly off.

Intel Berkeley Lab Data Our next experiment evaluates the algorithms on sensor data collected by the Intel Berkeley Research Lab [7]. The data consists of environmental readings regularly collected from 54 nodes spread around their lab. We extracted the temperature readings as test data on which to run MAX. We observed some missing readings for various nodes at various time epochs and have filled these in with the average of the readings for the particular node from the previous and subsequent epochs. Finally, the network area is not large enough to provide an interesting hierarchy when node radio range is 50 meters. We have reduced that to 6 meters, the minimum distance that still fully connects the network. Epochs in this dataset are close together in time and data readings do not change much from epoch to epoch; therefore, we make each of our round 100 epochs in length. The results are shown in Figure 10. The office temperatures are still fairly stable, explaining the improved relative performance of TS and RC compared to previous experiments. This stability allows TS to beat SLAT. TS sends a limited number of *Trigger* packets such that its total cost is less than SLAT's for sending a variety of packet types needed to support thresholds. SLAT-A and HAT, however, still considerably outperform TS. Of these, HAT is still the more efficient.

Threshold Setting We now examine the choice of HAT's threshold-setting policy. We compare three adaptive pressure-based policies: *ceiling*, *floor*, and *adaptive-linear*, with α adjuster c set to 0.1. We use a scenario where all node values change in each round. The

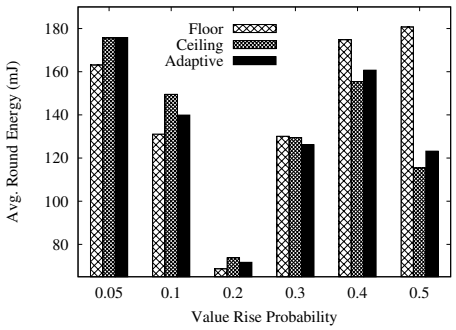

Figure 11: HAT threshold adjustment policies.

probability that a value rises is equal to one minus the probability that it falls. The magnitude of change initially varies in the range $[0, 100]$, but shrinks linearly with each round, so values eventually stabilize. By varying probability of rise across runs, we control how often the maximum value rises and falls. The results are shown in Figure 11. When probability of rise is low, the maximum value generally falls, so *floor* is the best policy. At the opposite end, *ceiling* is the best policy. In each case, *adaptive-linear* has cost almost as low as the best policy. If either *floor* or *ceiling* is best suited to a particular scenario, *adaptive-linear* only incurs a small penalty before adjusting to the optimal policy. We believe any reasonable adaptive policy that does not bog down the algorithm in worst-case behavior permanently (such as *floor* when values are only rising) will allow HAT to outperform other algorithms.

7 Conclusion and Future Work

MAX is a fundamental exemplary aggregate query. Unlike in selection queries, nodes cannot decide for themselves if they are in the solution. Despite this challenge, for continuous monitoring, it is crucial for energy conservation that nodes do not send their data to the root in every round. We have developed techniques based on constraint localization for efficiently supporting continuous MAX, which also provide general insight for developing sensor query processing algorithms. We have applied existing algorithms to the problem and introduced novel ones. This latter group consists of SLAT, SLAT-A, and HAT. These algorithms introduce localized constraints into the network in the form of thresholds. In producing each of these algorithms, we make significant improvements that culminate in HAT. HAT enforces that thresholds monotonically increase from leaves toward the network root. Therefore, a subtree root threshold serves as both an upper bound for values in its subtree, and as a lower bound for the network maximum value. Using subtree thresholds as upper bounds may let us prune those subtrees from querying when the maximum value falls. Using subtree thresholds as lower bounds may let us drop messages from nodes within those subtrees that have broken their local thresholds. We have performed analysis of threshold setting to better understand this problem and proposed simple, practical policies. Finally, we have done experiments to expose the reactions of the algorithms to certain scenarios, and evaluate their relative performances. We have also briefly described how to extend our techniques to the one-sided quantile problem. We believe our idea of constraint localization is fundamental and can be applied to other problems such as general quantile. In general, as networks grow larger, regardless of the query in discussion, it will become more important to apply constraint localization to avoid sending messages originating on the outskirts of the network all the way to the root.

8 References

[1] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining Variance and k-Medians over Data Stream Windows. In *Proc. of the 2003 ACM Symp. on Principles of Database Systems*, San Diego, California, USA, June 2003.

[2] R. Cheng, B. Kao, S. Prabhakar, A. Kwan, and Y. Tu. Adaptive Stream Filters for Entity-based Queries with Non-Value Tolerance. In *Proc. of the 2005 Intl. Conf. on Very Large Data Bases*, Trondheim, Norway, Aug. 2005.

[3] Chuck Conner. Modeling Heat Transfer in Parallel. `http://www.cas.usf.edu/~cconnor/parallel/2dheat/2dheat.html`.

[4] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate Aggregation Techniques for Sensor Databases. In *Proc. of the 2004 Intl. Conf. on Data Engineering*, Boston, Massachusetts, USA, Mar. 2004.

[5] Crossbow Inc. *MPR-Mote Processor Radio Board User's Manual*.

[6] A. Deligannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical In-Network Data Aggregation with Quality Guarantees. In *Proc. of the 2004 Intl. Conf. on Extending Database Technology*, Heraklion, Crete, Mar. 2004.

[7] Intel Berkeley Research Lab. `http://berkeley.intel-research.net/labdata/`.

[8] Z. Liu, K. Sia, and J. Cho. Cost-Efficient Processing of Min/Max Queries over Distributed Sensors with Uncertainty. In *Proc. of the 2004 ACM Symp. on Applied Computing*, Santa Fe, New Mexico, USA, Mar. 2005.

[9] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. of the 2002 USENIX Symp. on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, Dec. 2002.

[10] S. Madden, R. Szewczyk, M. Franklin, and D. Culler. Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks. In *Proc. of the 2002 IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, New York, USA, June 2002.

[11] C. Olston, B. Loo, and J. Widom. Adaptive Precision Setting for Cached Approximate Values. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, Santa Barbara, California, USA, May 2001.

[12] S. Pattem, B. Krishnamachari, and R. Govindan. The Impact of Spatial Correlation on Routing with Compression in Wireless Sensor Networks. In *Proc. of the 2004 Intl. Conf. on Information Processing in Sensor Networks*, Berkeley, California, USA, Apr. 2004.

[13] D. Petrovic, R. Shah, K. Ramchandran, and J. Rabaey. Data Funneling: Routing with Aggregation and Compression for Wireless Sensor Networks. In *Proc. of the 2003 IEEE Sensor Network Protocols and Applications*, Anchorage, Alaska, USA, May 2003.

[14] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and Beyond: New Aggregation Techniques for Sensor Networks. In *Proc. of the 2004 ACM Conf. on Embedded Networked Sensor Systems*, Baltimore, Maryland, USA, Nov. 2004.

[15] A. Silberstein, R. Braynard, C. Ellis, K. Munagala, and J. Yang. A Sampling-Based Approach to Optimizing Top-k Queries in Sensor Networks. In *Proc. of the 2006 Intl. Conf. on Data Engineering*, Atlanta, Georgia, USA, Apr. 2006.