2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware

Thresholded Range Aggregation in Sensor Networks

Zhifeng Lin† Man Lung Yiu‡ Nikos Mamoulis†

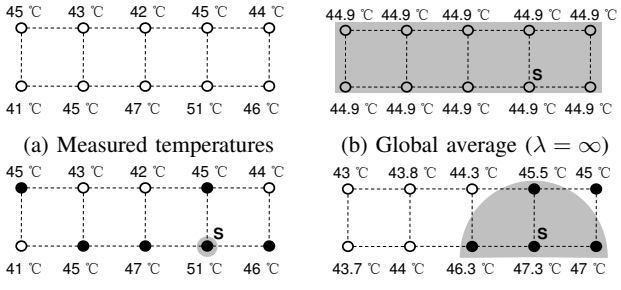
†*Department of Computer Science, University of Hong Kong*
{zflin, nikos}@cs.hku.hk

‡*Department of Computer Science, Aalborg University*
mly@cs.aau.dk

***Abstract*— The recent advances in wireless sensor technologies (e.g., Mica, Telos motes) enable the economic deployment of lightweight sensors for capturing data from their surrounding environment, serving various monitoring tasks, like forest wildfire alarming and volcano activity. We propose a novel query called *thresholded range aggregate query* (TRA), which retrieves the IDs of the sensors for which the average measurement in their neighborhood exceeds a user-given threshold. This query provides results that they are robust against individual sensor abnormality, and yet precisely summarize the sensors' status in each local region. In order to process the (snapshot) TRA query, we develop energy-efficient protocols based on appropriate operators and filters in sensor nodes. The design of these operators and filters is non-trivial, due to the fact that each sensor measurement influences the actual results of other nodes in its neighborhood region. Furthermore, we extend our protocols for continuous evaluation of the TRA query. Experimental results show that our proposed solutions indeed offer substantial energy savings for both real and synthetic sensor networks.**

I. Introduction

The recent sensor network technology (e.g., Mica, Telos motes) allows economic deployment of large number of sensors, for measuring values from their residing environment. Sensor networks are essential for monitoring applications, e.g., agricultural industry maintenance [1] and environmental monitoring [2]. Figure 1a illustrates a sensor network for measuring temperatures, in which each white spot represents a sensor (node) and the dotted edges indicate the pairs of nodes within their limited communication range (e.g., 100m). Each sensor mainly spends its energy on communication with neighborhood sensors. Due to the limited energy stored in the sensors, energy-efficient protocols [3]–[8] have been developed to reduce the power consumption of sensors, while processing aggregation queries in the network.



(a) Measured temperatures (b) Global average ($\lambda = \infty$)

(c) Individual value ($\lambda = 0$) (d) Local average ($\lambda = 1$)

Fig. 1. Aggregation query in a network ($\delta = 45°$C)

We introduce a novel query type that finds regions in the sensor network space, where the aggregate measurements in the region qualify some predicate (e.g., average temperature above 45 degrees). Given a threshold value δ and a radius λ, the Thresholded Range Aggregate (TRA) query retrieves each sensor (ID) s such that the average measured value (of sensors) within its neighborhood range (of radius λ) is above the threshold δ. In a volcano monitoring application, a TRA query can be applied to study local activities of the volcano in an effective manner; each local circular area (of $\lambda = 100$m) with average temperature above $\delta = 90°$C indicates high volcano activity. In a forest wildfire monitoring application, a local circular area (of $\lambda = 100$m) with average temperature above $\delta = 45°$C reflects a potential wildfire in that particular area. An appropriate range λ allows us to extract reliable yet localized results from the environment.

Our TRA query provides more meaningful results than alternative approaches. Suppose that the sensor network of Figure 1a has been deployed to monitor a potential area of forest wildfire (e.g., temperature above threshold $\delta = 45°$C). A typical aggregate query [9] returns the global average temperature of all sensors, e.g., $44.9°$C in Figure 1b. This result is robust against individual abnormal sensor readings, e.g., some sensors are located at wet shadows or exposed on the rock. However, it cannot show the temperatures at local regions in the network. For this purpose, one may consider retrieving each individual sensor reading above the temperature threshold ($45°$C), i.e., black spots in Figure 1c. Unfortunately, this allows abnormally high readings to be returned. Unlike the two extreme approaches discussed above, our TRA query computes the average temperature in each local area. In Figure 1d, each value next to a sensor s indicates the average temperature within its neighborhood region (i.e., within $\lambda = 1$ hop). Only the nodes in black report their result. In summary, the result set of TRA (i) is *robust* against fluctuation of individual sensor reading, and (ii) accurately reflect the overall measurement in each *local* region.

Despite being an important query, the TRA has not been studied in the literature before. We present energy-efficient protocols for processing the query, based on *in-network* evaluation strategies [10]–[12], by pushing appropriate query predicates from the base station into sensor nodes. The challenge is that existing in-network techniques for joins [5], [13] are either infeasible or inefficient for TRA. For instance, Abadi et al. [5] considered a relational join between the sensors

978-0-7695-3650-7/09 $25.00 © 2009 IEEE
DOI 10.1109/MDM.2009.27

values and a static table of predicates. In contrast, the sensors in the TRA result are dynamically influenced by the other sensors in their neighborhood region, at run time. Yiu et al. [13] proposed a generic *spatial pattern query* that returns each combination of sensor nodes that satisfy a predefined condition of sensor values and neighborhood relationship. Such a query is inherently different from TRA because each TRA result is a sensor node (as opposed to a combination of sensors nodes). Hence, there is a need for developing in-network techniques tailored to TRA.

In this paper, we not only examine snapshot TRA queries (i.e., querying once) but also study the evaluation of continuous TRA queries. For example, the continuous query "report the TRA result every 10 seconds" is being periodically evaluated at each *epoch* (10 seconds). In a typical environment with slowly changing measurements (e.g., temperatures), there are few differences between the results of consecutive epochs. To reduce power consumption, a local tolerance bound $[s.lb, s.ub]$ can be installed at each node s, such that any future measurement $s.m$ of s falling into the interval can be safely without affecting the query result. The challenges here are: (i) the correctness of the query result must be guaranteed by these bounds, and (ii) the actual bounds depend on the previous readings at the sensor nodes. We propose a novel technique for deriving these local tolerance bounds in the network, and discuss their maintenance.

In summary, our main contributions in this paper are:

- the proposal of the a novel query type (TRA), which identifies interesting spatial regions in a sensor network based on the aggregate sensor readings in them
- the development of various protocols for evaluating the snapshot TRA query
- solutions for processing the continuous TRA query

The remainder of the paper is organized as follows. Section II reviews the work related to our problem. Section III formally defines the problem and the relevant notations. In Section IV, we present several protocols for evaluating the snapshot TRA. Section V discusses the processing of the continuous version of TRA. Section VI experimentally demonstrates the efficiency of our solutions. Finally, Section VII concludes the paper.

II. Related Work

A sensor (node) takes a measurement (from its environment) and communicates with other sensors (i.e., neighbors) within its bounded communication range. Each sensor has limited energy and it mainly consumes energy on these three operations [2]: (i) sending/broadcasting a packet to neighbor sensors, (ii) receiving a packet from a neighbor sensor, and (iii) passively listening for messages from any neighbor sensor. By conceptually linking the sensors within the communication range, we obtain a *sensor network*. Query evaluation techniques for a sensor network aim at minimizing the energy consumption. DBMS prototypes for sensor networks (e.g., TinyDB [9], Cougar [14]) provide the user a convenient query interface for processing the measured values from sensors in the network.

Section II-A reviews the construction of the routing tree, which is essential for efficient aggregate operations in the network. Section II-B discusses the techniques for implementing operators and filters in network nodes for reducing communication cost.

A. Aggregation and Routing Tree

A typical aggregate query in a sensor network looks like: "compute the sum (or average) of all sensor values in the network". Usually, the query is registered at base station, which is connected to a root (sensor) node. To evaluate the query, we first need to build a *routing tree* [9], and then propagate the measured values from sensor nodes to the base station via the routing tree. We proceed to elaborate these two steps in detail.

In the first step, the base station sends the query to the root node, which then applies a breadth-first protocol to disseminate the query and define the routing tree dynamically in the sensor network. In the second step, sensors deliver their readings (and aggregate them) via the routing tree. The child nodes take measurements first and then send values to the level above, whereas the parent nodes wake up, listen/receive messages, and aggregate the received values with its own measurement. Observe that this aggregate operation can be done in sensor nodes, for distributive functions (e.g., sum) and algebraic functions (e.g., average). By synchronizing the above process in the routing tree level-by-level, the root eventually obtains the final aggregate value and then sends it to the base station.

The query described above is a *snapshot query*, which is executed only once in the network. In contrast, a *continuous query* periodically returns the result to the base station. An example query is: "report the average of all sensor values in the network, every 5 minutes", and the *epoch* value here is 5 minutes. The same routing tree can be reused by the continuous query in consecutive epochs. Then, the sensors values are aggregated and sent to the base station via the tree, at every epoch. Various monitoring techniques [15]–[17] have been developed for continuous queries with specific predicates.

B. In-network Operators and Filters

To reduce the query evaluation cost in a sensor network, an *operator* in a query plan can be "pushed down" from the base station to sensor nodes. This is done in the in-network aggregation described in the previous section. Here, we briefly review previous work which focuses on in-network operator placement. Bonfils et al. [19] study a query that correlates the sensors' measurements obtained from two pre-defined spatial regions, say R_1 and R_2. Coman et al. [20] developed a cost model to determine the most promising sensor location for performing the join with low communication cost. In a similar problem setting, Yu et al. [12] propose an in-network synopsis join strategy to prune unqualified tuples that cannot contribute to join results in the initial phase of join processing. In contrast, our problem searches for any sensor node with high average neighborhood measurement, and these nodes can appear anywhere in the network.

Filters can be placed at sensor nodes to eliminate measurements that are not useful to the query result, thus reducing communication cost. Abadi et al. [5] focus on joining the sensors' measurements with a static predicate table of predefined filter conditions. A tuple in the predicate table specifies the condition of reporting a sensor measurement. Sensor nodes have limited memory so their solution is to decompose the large predicate table into tiny tables and distribute them into different nodes. Each sensor node acts a partial filter that prevents a portion of non-qualifying readings to be sent to the base station. This strategy is inapplicable to our problem, since our result set depends solely on sensors' measurements at runtime (as opposed to static conditions).

In a sensor network, the distance join query [13], [21] retrieves each pair $\langle s, s' \rangle$ such that (i) the distance between the nodes s and s' is within given range λ, and (ii) the measurements of s and s' satisfy the query predicates P_1 and P_2 respectively. To detect such pairs efficiently, Kotidis et al. [21] suggest to utilize the nodes for maintaining a distributed routing index dynamically, which helps guiding the propagation of the messages in the network to potential nodes for producing join results. In the solution of [13], a *pruner* $pr(s)$ is formulated as a node such that its subtree contains all nodes within distance λ generated at a node s. This enables the node $pr(s)$ to discard safely a P_1-tuple from s, if it does not receive (from its subtree) any matching tuple (i.e., P_2-tuple) produced at any node within distance λ of s. As discussed in the Introduction, our proposed TRA query requires deriving the average measurement value from each node's neighborhood region, so it is more complicated than the above distance join query. Instead of only utilizing pruner nodes (as in [13]), we essentially develop various types of in-network operators and filters in the subsequent sections.

III. Problem Definition

Let SN be a set of sensors. Each sensor $s \in SN$ is associated with a spatial location $s.loc$, and produces a measurement $s.m$ (e.g., temperature). Given two sensors s and s', we use $dist(s, s')$ to denote their distance in terms of the number of hops, as in [13]. Given a threshold δ and a distance λ, the *thresholded range aggregate query* (TRA) retrieves each sensor ID $s \in SN$, such that the average measurement of sensors s' (within distance λ from s) is above the threshold δ.

In the TRQ query, the user-given parameters λ and δ are used to control the aggregation scope and the alarm threshold respectively. Depending on the specific application, a proper λ value helps retrieving robust and yet localized results. For instance, in a forest wildfire monitoring application, we can set $\lambda = 1$ hop and $\delta = 45$°C if such a scale of wildfire is regarded as easy to manage (or extinguish).

There are two types of query evaluation in sensor networks. A *snapshot query* is evaluated once. It can be expressed in the following SQL statement. In the rest of the discussion, we use $s.avg$ to represent the average value $AVG(s'.m)$ in the λ-neighborhood of sensor s.

SELECT	$s.id$	FROM	SN as s, SN as s'
WHERE	$dist(s, s') \leq \lambda$		
GROUP BY	$s.id$	HAVING	$AVG(s'.m) \geq \delta$

On the other hand, a *continuous query* is evaluated periodically at every *epoch* (e.g., 1 second), for a *lifetime* (e.g., 60 seconds). Both epoch and lifetime are parameters specified by the user. A continuous TRA query can be specified by adding the following clause to the above SQL statement.

SAMPLE PERIOD 1s FOR 60s

For the ease of exposition, we denote TRA by a tuple $\mathcal{Q} = \langle \delta, \lambda, dir, mode \rangle$ where $dir, mode \in \{0, 1\}$. To report each sensor s with average measurement $s.avg$ above δ (or below δ), we set dir to 1 (or 0). $mode$ indicates whether the query type is snapshot (1) or continuous (0). For instance, the above snapshot query and continuous query in SQL syntax are denoted by the tuples $\langle \delta, \lambda, 1, 1 \rangle$ and $\langle \delta, \lambda, 1, 0 \rangle$ respectively.

Our research objective is to develop efficient protocols for processing TRA with low communication cost.

IV. Snapshot Query Processing

In this section, we develop efficient protocols for processing the snapshot TRA query $\mathcal{Q} = \langle \delta, \lambda, 1, 1 \rangle$. We first formulate the concepts of parent-child relationship, ancestor-descendant relationship, common ancestor, and nearest common ancestor.

Definition 1: A node s is defined as the **parent** of a node s', if the routing tree has a direct link between them, and s is at an upper level than s'.

Definition 2: A node s is said to be an **ancestor** of a node s', if there exists the nodes $s_1, s_2, \cdots, s_{k+1}$, such that $s = s_1$, $s' = s_{k+1}$, and s_i is the parent of s_{i+1}, for all $i \in [1, k]$. We express this relationship as: $s \succeq s'$.

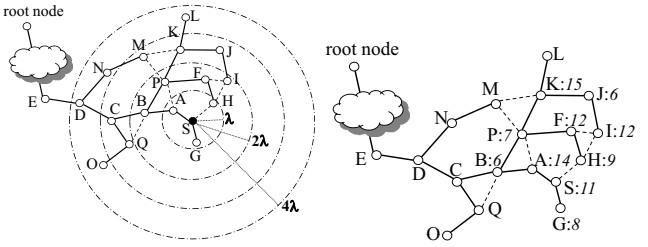
Definition 3: Given a subset $V \subseteq SN$ of nodes, a node s is said to be a **common ancestor** of V, if $s \succeq s'$ holds for each node $s' \in V$. The **nearest common ancestor** $\Psi(V)$ is the common ancestor of V whose tree level is the closest to the top tree level of the nodes in V.

Figure 2a depicts an example of a sensor network that will be used throughout the paper. Nodes within communication range are connected by edges, and the solid edges denote the parent-child links in the routing communication tree. The distance between sensors can be expressed in terms of hops. In this example, we have $B \succeq S$ and $B \succeq B$. Note that, the nodes A, F, J have their nearest common ancestor as B.

In the following, Section IV-A presents two brute-force protocols for evaluating TRA queries. Section IV-B develops effective in-network filters for pruning unqualified measurements early in the network. These filters are then applied in the advanced protocols proposed in Sections IV-C and IV-D.

A. Brute-force Protocols

Acquisitional Brute-force Protocol. We first discuss an *Acquisitional Brute-force Protocol* (ABrute), for evaluating the TRA query. This protocol requests each sensor node to send its measurement to the base station via the routing tree.



(a) Neighborhood ($\lambda = 1$ hop) (b) Measurements

Fig. 2. Sensor network example

The collection of measurements from sensor nodes is synchronized level-by-level. First, the nodes at the lowest level wake up. Each such node s takes its measurement $s.m$ and generates a message $\langle s.id, s.m\rangle$ with its id, then it sends the message to its parent node. After that, each node at the next level wakes up, receives messages from its children, then *appends* those messages to its own message as a combined message, and sends the combined message to its parent. This procedure continues with the upper levels until reaching the root node, which then forwards all the messages to the base station.

We assume that the base station has recorded the static location $s.loc$ of each node s in advance so it is able to compute $s.avg$ for each s, and check whether $s.avg$ is above the threshold δ (i.e., whether s is a result). In summary, the ABrute protocol is similar to the aggregation protocol discussed in Section II-A. However, in the ABrute protocol, the nodes cannot compute average values for their descendant nodes because their locations are not kept in other nodes. As a result, each node only merges its received messages with its own message to form a combined message.

Distributional Brute-force Protocol. This section presents a *Distributional Brute-fore Protocol* (DBrute), which consists of two steps: (i) computing the average values $s.avg$ of sensors locally in the network, and (ii) sending only the qualified node IDs to the base station.

First, DBrute employs a *multi-hop broadcasting protocol*, during which each sensor receives the measurements of all nodes within distance λ. To implement this, each node s generates a message $\langle s.id, s.m, cnt\rangle$ with $cnt = \lambda$, and then broadcasts its message. Whenever s receives a message $\langle s'.id, s'.m, cnt'\rangle$ with a positive cnt' value, it broadcasts the message $\langle s'.id, s'.m, cnt'-1\rangle$. This process takes λ processing cycles of sensor nodes in total.

Each node s eventually receives the measurements from all its λ-neighbors from, so s is able to compute its average value $s.avg$ and check whether it is a result. In the second step of DBrute, only result nodes send their IDs to the base station via the tree, using a protocol similar to ABrute (except that individual measurements are not sent).

B. Building In-network Filters

The high broadcasting cost of distributed protocols like DBrute is unavoidable, we pursue effective optimizations for ABrute. By placing suitable *filters* at appropriate sensor nodes, unqualified values can be eliminated early and the communication cost can be significantly reduced. In this section, we focus on developing such filters, so that they become applicable in some advanced protocols presented in later sections.

Formulation of decider and filter nodes. We need to answer the two questions below, for a node s in the routing tree.
(I). Which node can compute the average $s.avg$ for s?
(II). Which node can know the average $s'.avg$ for all s' that can be influenced by $s.m$?

The first question identifies the node (say, s^*) for computing $s.avg$ (and checking whether this qualifies as a result). The second question determines the node (say, s^{**}) for pruning the measurement $s.m$, since it has already been applied to calculate $s'.avg$ of all s' affected by $s.m$. Before answering these questions, we introduce the notations below.

Definition 4: Given the threshold λ, the **neighborhood sensor set** $\mathcal{L}(s)$ and the **super neighborhood sensor set** $\mathcal{SL}(s)$ of a node $s \in SN$ are defined as:

$$\mathcal{L}(s) = \{s' \in SN \mid dist(s, s') \leq \lambda\}$$
$$\mathcal{SL}(s) = \{s' \in SN \mid dist(s, s') \leq 2\lambda\}$$

Figure 2a illustrates the neighborhood sets of the sensor S. In this example ($\lambda = 1$ hop), the neighborhood set $\mathcal{L}(S)$ of the sensor S is $\mathcal{L}(S) = \{S, A, H, G\}$, which contains all sensors in the circle centered at S with radius λ. Similarly, the super neighborhood set is $\mathcal{SL}(S) = \{S, A, G, B, P, F, I, H\}$.

The following lemma reveals an interesting property between neighborhood sets and super neighborhood sets.

Lemma 1: Given a node s, it holds that, $\mathcal{L}(s') \subseteq \mathcal{SL}(s)$, for any $s' \in \mathcal{L}(s)$.

Proof: Let s' be a node in $\mathcal{L}(s)$ and s'' be a node in $\mathcal{L}(s')$. From their definitions, we have: $dist(s, s') \leq \lambda$ and $dist(s', s'') \leq \lambda$. Thus, we get $dist(s, s') + dist(s', s'') \leq 2\lambda$. By the triangular inequality, we obtain: $dist(s, s'') \leq dist(s, s') + dist(s', s'')$. As a result, we have $dist(s, s'') \leq 2\lambda$. This implies $s'' \in \mathcal{SL}(s)$, and thus $\mathcal{L}(s') \subseteq \mathcal{SL}(s)$. ■

Definition 5: Given a node s in a routing tree, its **decider** node is defined as $s.decider = \Psi(\mathcal{L}(s))$, and its **filter** node is defined as $s.filter = \Psi(\mathcal{SL}(s))$. Conversely, we call s as the **decidee** of $s.decider$, and the **filtee** of $s.filter$.

The decider node $s.decider$ of s is the nearest common ancestor of $\mathcal{L}(s)$. This node eventually receives all measurements from $\mathcal{L}(s)$ so it is able to compute $s.avg$ correctly. This answers our first question. Instead of using an arbitrary common ancestor (e.g., the root node), we pick the most efficient choice (i.e., $\Psi(\mathcal{L}(s))$) to compute whether s is a query result.

Yet, the decider node $s.decider$ cannot immediately discard the measurement $s.m$ of s. Suppose that s' is a node in $\mathcal{L}(s)$, other than s. Conversely, we know that $s \in \mathcal{L}(s')$, i.e., the measurement $s.m$ is necessary for deriving the average value $s'.avg$ of s'. Obviously, the filter node $s.filter$ of s is the nearest common ancestor of $\Psi(\mathcal{SL}(s))$. Lemma 1 shows that $\mathcal{SL}(s)$ contains $\mathcal{L}(s')$ for any $s' \in \mathcal{L}(s)$. This implies that, the average value of any $s' \in \mathcal{L}(s)$ has been computed, when the measurement $s.m$ reaches the filter node $s.filter$ of s. Thus,

the filter node always safely discards the measurement $s.m$. This answers our second question.

As an example in Figure 2a, the decider node of S is the node B, as it is the nearest common ancestor of $\mathcal{L}(S)$. The filter node of S is also node B, since its subtree contains $\mathcal{SL}(S)$. In this case, the node B has a decidee node B, and a filtee node B. Observe that each node has exactly one decider node and one filter node. On the other hand, a node can have zero, one, or multiple decidee nodes and filtee nodes.

Deriving decider and filter nodes in the network. After the routing tree has been built, we are able to derive the *decider* and *filter* nodes in the network. We first develop a procedure for determining the decider node for each sensor node s. This procedure is applied only once before evaluating the query. It consists of three steps: (i) notifying neighborhood nodes, (ii) sending the neighborhood count to the neighborhood, and (iii) finding the nearest common ancestor node.

A multi-hop broadcasting method was introduced as the initial step of the DBrute protocol, in Section IV-A. This method can be used to implement the first two steps of the procedure above. In the first step, each node s broadcasts a message $\langle s.id, cnt\rangle$, with $cnt = \lambda$. Each time the message is propagated by another node, the value of cnt in the message is decremented, until cnt reaches zero. This essentially allows each node s to derive its neighborhood size $|\mathcal{L}(s)|$. In the second step, each node s broadcasts a message $\langle s.id, |\mathcal{L}(s)|, cnt\rangle$, with $cnt = \lambda$. This enables each node $s' \in \mathcal{L}(s)$ to know the size $|\mathcal{L}(s)|$ of s. In the third step, each sensor s sends up the routing tree a table consisting of $\langle s'.id, |\mathcal{L}(s')|, \{s\}\rangle$ tuples for all $s'.id \in \mathcal{L}(s)$ plus a $\langle s.id, |\mathcal{L}(s)|, \{s\}\rangle$ for itself. The last field in the tuple indicates the list of visited neighborhood nodes. When an intermediate tree node receives multiple tuples with the same $s'.id$, e.g., $\langle s'.id, |\mathcal{L}(s')|, \mathcal{L}_1\rangle$ and $\langle s'.id, |\mathcal{L}(s')|, \mathcal{L}_2\rangle$, they are merged together into a single tuple $\langle s'.id, |\mathcal{L}(s')|, \mathcal{L}_1 \cup \mathcal{L}_2\rangle$. In case a node obtains a tuple of the form $\langle s'.id, |\mathcal{L}(s')|, \mathcal{L}(s')\rangle$, it becomes the decider node of s' and then it stops forwarding that tuple to its parent.

In the above procedure, it is necessary for the decider node of s' to know the entire $\mathcal{L}(s')$, so that it can be later applied to compute the average value of s'. On the other hand, a filter node of s' only needs to know which measurement $s'.m$ to be discarded, but not the actual $\mathcal{SL}(s')$. The above procedure can be adapted to derive the filter nodes. The only difference is that, in the third step, the last field of each tuple is replaced by a counter. Each tuple is initialized as $\langle s'.id, |\mathcal{L}(s')|, 1\rangle$. When an intermediate node receives multiple tuples with the same $s'.id$, they are aggregated together by summing their counters.

The above procedure is applied only once and used for all snapshot queries with parameter λ (i.e., the filter/decider nodes are independent of δ). Given that the used values for λ in practical queries are limited (i.e., up to a small integer of hops), determining the deciders/filters for a sensor network with stationary nodes is a one-time process for all queries having the same λ value.

In this paper, we do not consider further reorganization of the routing tree. Decider nodes and filter nodes of each node could change, if the routing tree of Figure 2a is replaced by another tree. Theoretically, there exists an optimal routing tree that maximizes pruning effectiveness. However, the effort of finding the optimal routing tree is high and it undermines the benefit it provides.

C. Acquisitional In-network Processing Protocol

In this section, we propose the *Acquisitional In-network Processing Protocol* (AIP), which utilizes the concepts of decider nodes and filter nodes for in-network execution.

First of all, the procedure discussed in Section IV-B is applied to derive the decider nodes and filter nodes in the network. After that, each sensor node explicitly stores: (i) its list of filtees, (ii) its list of decidees, and (iii) the neighborhood set $\mathcal{L}(s)$ of each decidee. The application of (i) is to discard unnecessary measurement values; whereas both (ii) and (iii) are used to compute the average values for its decidees.

Next, we run our AIP protocol in a synchronized level-wise manner, from the nodes at the lowest level to the root node. Let s^* be the current sensor node, and δ be the threshold value for checking qualified results. Two forms of tuples are used in the subsequent discussion: (i) a measurement tuple of node s, i.e., $\langle s.id, s.m, -\rangle$, or (ii) a result tuple with the average value $s.avg$ of s, i.e., $\langle s.id, -, s.avg\rangle$.

Let $\mathcal{M}_{in}$ denotes the set of tuples to be processed, and $\mathcal{M}_{out}$ denotes the set of tuples to be forwarded to the parent of s^*. The current node takes its measurement $s^*.m$ and inserts it as a tuple into $\mathcal{M}_{in}$. Then, the node receives the tuples sent from child nodes, and inserts those tuples into $\mathcal{M}_{in}$. Next, we examine each node s in the decidee list of the current node s^*. Next, we look up the neighborhood set of s from the memory of s^*, search for their corresponding measurement tuples from messages containing $\mathcal{M}_{in}$, and then compute the average value $s.avg$ of s. In case the value $s.avg$ is above the threshold δ, a result tuple $\langle s.id, -, s.avg\rangle$ needs to be inserted into $\mathcal{M}_{out}$. Then, we remove any measurement tuple produced by a node s in the filtee list of the current node s^*. The remaining tuples of $\mathcal{M}_{in}$ are useful because they are either result tuples computed by lower-level decider nodes, or measurement tuples that are used for computing other average values at high-level decider nodes. Thus, all tuples of $\mathcal{M}_{in}$ are then inserted into $\mathcal{M}_{out}$. Eventually, the current node s^* forwards the tuples of $\mathcal{M}_{out}$ to its parent node, and then sleeps again.

We illustrate the running steps of the AIP protocol with the sensor network in Figure 2b, which is the same as in Figure 2a, but the nodes are now marked with their temperature measurement (in °C). In this example, the user issues the snapshot query $\mathcal{Q} = \langle 10, 1, 1, 1\rangle$, meaning that $\lambda = 1$ hop, $\delta = 10$°C, and he wants to find each sensor s whose average neighborhood value is above δ. We only provide detailed information of the part of sensors that are used in this example. Table I lists the information of these sensors, e.g., decider node, filter node, neighborhood set $\mathcal{L}()$, and super neighborhood set $\mathcal{SL}()$. Table II shows the ids, measurements (e.g., temperatures), average values of these sensors. The last

TABLE I

RELATIONSHIPS IN THE RUNNING EXAMPLE, AT $\lambda = 1$

id	*decider*	*filter*	$\mathcal{L}(\cdot)$	$\mathcal{SL}(\cdot)$
S	B	B	A, H, G, S	S, A, G, H, I, F, P, B
A	B	D	S, B, P, A	$A, S, B, P, G, H,$ M, F, K, C, Q
H	B	B	F, I, S, H	H, I, F, S, J, P, A, G
G	S	B	S, G	G, S, A, H
I	P	B	H, J, F, I	I, J, F, H, K, P, S
F	P	D	P, H, F, I	$F, I, H, P, J,$ S, M, K, B, A
J	K	D	K, I, J	J, I, K, F, H, L, M, P

TABLE II

NETWORK DATA IN THE RUNNING EXAMPLE, AT $\delta = 10$

id	S	A	H	G	I	F	P	B	J	K
m	11	14	9	8	12	12	7	6	6	15
avg	10.5	9.5	11	9.5	9.75	10	⁄	⁄	11	⁄
ans	✓	×	✓	×	×	✓	⁄	⁄	✓	⁄

row ans indicates whether the sensor is a result (i.e., average value above 10°C).

Starting from the leaf nodes (O, G, H, I, L), all nodes send their measurements (m) up to their parents level by level. For example, after $G.m$ is sent to S, S knows that G is its $decidee$. Thus, S calculates $G.avg$ from $S.m$ and $G.m$ (to be forwarded to S's parent). S does not filter any information, since it has no $filtee$. The node A has no $decidee$ and no $filtee$, so it just forwards its received tuples to its parents. For the node B, $B.decidee = \{S, A, H\}$ and $B.filtee = \{S, H, G, I\}$, therefore after B receives messages from its children P and A, B calculates the avg for the nodes in $B.decidee$ and finds that S and H are qualified (above 10). Meanwhile B stops forwarding the measurements for all nodes in $B.filtee$. To explain the actions in B explicitly, we clarify the calculation of $S.avg$ and filtration of H in detail. B affords to calculate $S.avg$ because $\mathcal{L}(S) = \{A, H, G, S\}$. $A.m$ is filtered at its filter node (D). The measurements $H.m$, $G.m$, $S.m$ are all filtered at B. $H.m$ is only used to calculate avg of sensors in $\mathcal{L}(H) = \{F, I, S, H\}$. $F.avg$ is calculated at $F.decider$ (P) and $I.avg$, $S.avg$, $H.avg$ are calculated at P, B, B already respectively. Thus, the value of H can be filtered at B. Finally, the root node obtains the results by combining its result and the results received from its descendants.

D. Two-phase Protocols

This section illustrates an optimization of the proposed acquisitional protocols ABrute and AIP, which is suited for snapshot queries with very selective δ (i.e., very few results), which are spatially clustered. The optimization is based on the fact that for any qualified sensor node s (with $s.avg$ above δ), there must be at least one node s' in the neighborhood set $\mathcal{L}(s)$ of s, such that the measurement $s'.m$ (of s') is above δ. This is formulated in the lemma below.

Lemma 2: If a node s satisfies $s.avg \geq \delta$, then there must be a node $s' \in \mathcal{L}(s)$ such that $s'.m \geq \delta$.

Proof: For the sake of contradiction, assume that each node $s' \in \mathcal{L}(s)$ has its measurement $s'.m$ smaller than δ. Thus, the average measurement $s.avg$ (of s) becomes smaller than δ, leading to contradiction, thus proving the lemma. ■

In other words, if a node s^* has its measurement above δ, then each node $s' \in \mathcal{L}(s^*)$ has potential to become a result. Yet, the computation of $s'.avg$ requires the measurements of each node $s'' \in \mathcal{L}(s')$. Thus, we should inform all nodes in $\mathcal{SL}(s^*)$ to prepare their measurements. To illustrate this concept in Figure 2a, suppose that all the nodes have their measurements below δ, except the sensor S. Since H belongs to $\mathcal{L}(S)$, it has potential to be a result. However, the computation of $H.avg$ requires the measurements of all nodes (e.g., I) in $\mathcal{L}(H)$. Thus, in this example, S needs to notify all nodes in $\mathcal{SL}(S)$.

Based on this observation, we extend ABrute (or AIP) into a two-phase protocol as follows. In the first phase, each sensor s takes its measurement $s.m$. If $s.m \geq \delta$, then s broadcasts the tuple $\langle s.id \rangle$ to the nodes in $\mathcal{SL}(s)$. A multi-hop broadcast protocol needs to be applied (see the initial step of DBrute in Section IV-A). Next, a node s is marked as *passive* if (i) its measurement $s.m$ is less than δ, and (ii) it has not received any message in the first phase. In the second phase, all the nodes operate according to the ABrute (or AIP) protocol, except that each passive node s does not forward its own measurement $s.m$ up the tree.

V. CONTINUOUS MONITORING

This section discusses the processing of continuous TRA query $\mathcal{Q} = \langle \delta, \lambda, 1, 0 \rangle$. The straightforward solution is to evaluate the corresponding snapshot query $\mathcal{Q}' = \langle \delta, \lambda, 1, 1 \rangle$ periodically at every epoch. Yet, the processing cost of the query can be reduced dramatically, by exploiting the fact that, real-life environmental data (e.g., temperature) usually change slowly with time, and the new measurement of a sensor is likely to stay close to its previous measurement. The general idea is to provide each sensor node s with a *local tolerance*, which is essentially an interval bound $[s.lb, s.ub]$. A node s only reports its measurement $s.m$ when it falls outside the interval; otherwise, it does not report $s.m$.

Our goal is to derive a *safe* local tolerance $[s.lb, s.ub]$ for a sensor node s, such that as long as the measurement $s.m$ stays within the filter interval, it is guaranteed that s cannot lead to change of query result. This is fundamentally different from the above naive approach because we ensure the correctness guarantee of exact query result without requiring pre-defined error tolerance. Section V-A elaborates how to set up safe local tolerances at sensor nodes, in the initial epoch of evaluation. Section V-B develops solutions for updating the local tolerances and query results, in subsequent epochs.

A. Setup of Local Tolerance at Sensor Nodes

We first study how to setup a *safe* local tolerance $[s.lb, s.ub]$ for a sensor node s, such that it guarantees the correctness of query result.

Formulation of local tolerance. Let s' be a node in $\mathcal{L}(s)$. Depending on the value of $s'.avg$, there are two cases to consider. In the first case, the current $s'.avg$ of s' is at least δ, the future average value of s' will not become less than δ if the measurement of each node $s'' \in \mathcal{L}(s')$ (including s) decreases by less than $s'.avg - \delta$. Therefore, the local tolerance of s

should be within the interval $[s.m-(s'.avg-\delta),+\infty]$. In the second case, the current $s'.avg$ of s' is below δ, the future average value of s' will not be above δ if the measurement of each node $s'' \in \mathcal{L}(s')$ increases by less than $s'.avg - \delta$. Therefore, the local tolerance of s should be within the interval $[-\infty, s.m + (\delta - s'.avg)]$. As a result, the local tolerance $[s.lb, s.ub]$ of s is taken as the intersection of the interval bounds discussed earlier considering all nodes s' in $\mathcal{L}(s)$. This is expressed by the following equation.

$$\Phi(s) = [s.lb, s.ub] = \bigcap_{s' \in \mathcal{L}(s)} \begin{cases} [s.m - (s'.avg - \delta), +\infty] & \text{if } (s'.avg \geq \delta) \\ [-\infty, s.m + (\delta - s'.avg)] & \text{if } (s'.avg < \delta) \end{cases} \quad (1)$$

For the example in Figure 2b, we elaborate how to derive the local tolerance of node S, at $\delta = 10$, based on the data in Tables I and II. Observe that $\mathcal{L}(S) = \{A, H, G, S\}$. The node A has its average value (9.5) lower than δ (10), so we compute the value $11 + (10 - 9.5) = 11.5$, and obtain the interval $[-\infty, 11.5]$ for S. Regarding the nodes H, G, S, we derive the intervals $[10, +\infty]$, $[-\infty, 11.5]$, $[10.5, +\infty]$ for S respectively. By taking the intersection of these intervals, we obtain the local tolerance $[S.lb, S.ub]$ of S as $[10.5, 11.5]$. This means that if $S.m$ is within the $[S.lb, S.ub]$, S will not change in the TRA query result, even though the actual average values of $\mathcal{L}(S)$ may change.

Necessary data for computing local tolerances of all nodes. In order to compute local tolerances of all nodes in a correct manner, we need to address the following questions for each sensor node s.
(I). Which node can compute $\Phi(s)$?
(II). Which node can know $\Phi(s')$ for all s' that can be influenced by $s.avg$?

The answer to the first question is straightforward. By Equation 1, the derivation of the local tolerance $\Phi(s)$ requires the average value $s'.avg$ of each node $s' \in \mathcal{L}(s)$. Note that the computation of $s'.avg$ requires measurements of all nodes in the set $\mathcal{L}(s')$. According to Lemma 1, the set $\mathcal{SL}(s)$ contains all such sets $\mathcal{L}(s')$. Thus, the filter node $s.filter$ of s, i.e., the nearest common ancestor of $\mathcal{SL}(s)$, can compute the local tolerance $\Phi(s)$ of s.

The answer to the second question determines until which node $s.avg$ must travel up the routing tree. It turns out that the average value $s.avg$ of s cannot be immediately discarded at the filter node $s.filter$, since the average value is still required for deriving the local tolerance $\Phi(s')$ of each node $s' \in \mathcal{L}(s)$. We illustrate this observation by using the network in Figure 2b, with the parameter $\lambda = 1$. Since $S \in \mathcal{L}(A)$, the computation of the local tolerance $\Phi(A)$ of A requires obtaining the average value $S.avg$ of S. Observe that the nodes A and S have their filter nodes as D and B respectively. If the node B discards the average value $S.avg$ of S, then the value $S.avg$ cannot reach the node D and the local tolerance $\Phi(A)$ of A cannot be computed.

To answer the second question, we need to determine which node can be used to discard the average value $s.avg$ safely, without preventing the computation of any local tolerance. According to Equation 1, the average value $s.avg$ of a node s only affects the local tolerance of any $s' \in \mathcal{L}(s)$. Since the local tolerance of s' can be computed at its filter node $\Psi(\mathcal{SL}(s'))$, it suffices to find out a common ancestor node of $\Psi(\mathcal{SL}(s'))$, for all $s' \in \mathcal{L}(s)$. We first introduce the extra neighborhood sensor set $\mathcal{XL}(s)$ of a node s in Definition 6, and then prove Lemma 3 that the set $\mathcal{XL}(s)$ contains $\mathcal{SL}(s')$ for each $s' \in \mathcal{L}(s)$. In other words, the *xfilter node* $\Psi(\mathcal{XL}(s))$ of s (see Definition 6) is allowed to discard the average value $s.avg$ safely without preventing local tolerance computation of other nodes.

In the example of Figure 2a, the node S has its extra neighborhood sensor set as $\mathcal{XL}(S) = \{A, S, G, H, B, P, F, I, Q, C, M, K, J\}$ The xfilter node of S is the nearest common ancestor of $\mathcal{XL}(S)$, i.e., $S.xfilter = \Psi(\mathcal{XL}(S)) = D$. Thus, the node D can be used to discard the average value $S.avg$ of S.

Definition 6: Given a node $s \in SN$, its **extra neighborhood sensor set** $\mathcal{XL}(s)$ is defined as:

$$\mathcal{XL}(s) = \{s' \in SN \mid dist(s, s') \leq 3\lambda\}$$

In a routing tree, the **xfilter** node of s is defined as $s.xfilter = \Psi(\mathcal{XL}(s))$. We call s as the **xfiltee** of $s.xfilter$.

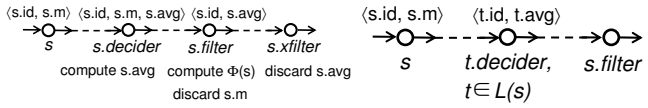
Lemma 3: Given a node s, it holds that, $\mathcal{SL}(s') \subseteq \mathcal{XL}(s)$, for any $s' \in \mathcal{L}(s)$.

Proof: Let s' be a node in $\mathcal{L}(s)$ and s'' be a node in $\mathcal{SL}(s')$. From their definitions, we have: $dist(s, s') \leq \lambda$ and $dist(s', s'') \leq 2\lambda$. Thus, we get $dist(s, s') + dist(s', s'') \leq 3\lambda$. By the triangular inequality, we obtain: $dist(s, s'') \leq dist(s, s') + dist(s', s'')$. As a result, we have $dist(s, s'') \leq 2\lambda$. This implies $s'' \in \mathcal{XL}(s)$, and thus $\mathcal{SL}(s') \subseteq \mathcal{XL}(s)$. ■

Computation of local tolerance in the network. We modify the AIP protocol follows in order to incorporate the above technique for computing the local tolerances of nodes correctly. The tuple $\langle s.id, -, s.avg \rangle$ needs to be inserted into the set $\mathcal{M}_{out}$ (and sent upwards the routing tree), regardless of whether the average value $s.avg$ is above the threshold δ or not. We examine each node s in the xfiltee list of the current node s^* and then remove a tuple of the form $\langle s.id, -, s.avg \rangle$ when $s.avg$ is below δ.

Figure 3 summarizes the tasks performed by the nodes, for evaluating the continuous query respectively. Regarding continuous query evaluation (see Figure 3a), the node $s.decider$ forwards the average value $s.avg$ up the tree, regardless of whether $s.avg$ is above δ or not. Next, the node $s.filter$ computes the local tolerance bound $\Phi(s)$ for the node s, by gathering the value $s.m$ and the average value $s'.avg$ of each node $s' \in \mathcal{L}(s)$. The node $s.filter$ then discards $s.m$. After that, the node $s.xfilter$ discards the average value $s.avg$ if it is below δ.

It remains to discuss how each filter node $s.filter$ notifies the node s of its local tolerance bound $\Phi(s)$, at the end of the first epoch round. We apply a level-wise protocol for disseminating such local tolerances from the top tree



(a) continuous query (b) update

Fig. 3. Tasks for a sensor node

level to the bottom tree level, assuming that the routing tree is static. Each filter node $s.filter$ stores its routing path $path(s.filter, s)$ to the node s. First, the node $s.filter$ generates a tuple $\langle s.id, \Phi(s), path(s.filter, s)\rangle$. After a node generated the above tuple (or received it from its parent), it forwards the tuple to its child node whose id appears in the path recorded in the tuple. Eventually, the node s will receive its local tolerance $\Phi(s)$.

B. Update of Localfilter and Query Result

In every subsequent epoch round, each sensor node s takes its new measurement $s.m$ and compares with its local tolerance bound $\Phi(s)$. The measurement is discarded if it falls inside the bound $\Phi(s)$. Otherwise, s needs to send a tuple $\langle s.id, s.m\rangle$ upwards the tree to update the query result and the local tolerance of relevant nodes. We discuss how this can be implemented in this section.

For a node s, if its new measurement $s.m$ falls outside $\Phi(s)$, then this influences the average value $s'.avg$ of each node $s' \in \mathcal{L}(s)$, and potentially alters some query results. For instance, if $s.m$ rises to a high value, then a past-unqualified $s'.avg$ may become qualified; if $s.m$ drops to a low value, then a past-qualified $s'.avg$ may become unqualified.

Since all the deciders of $\mathcal{L}(s)$ must be in the subtree of $s.filter$, the node s can send a tuple $\langle s.id, s.m\rangle$ upwards to tree, until reaching the node $s.filter$. This way, the (new) average values of all nodes in $\mathcal{L}(s)$ can be updated at $s.filter$.

In addition, the local tolerances of certain nodes need to be updated because of their updated average values. Specifically, for each node $t \in \mathcal{L}(s)$ the average value $t.avg$ has been updated at the node $t.decider$. Thus, the node $t.decider$ needs to send the value $t.avg$ upwards to the node $t.xfilter$, in order to update the local tolerances of nodes in $\mathcal{L}(t)$. Basically, the local tolerances of all nodes in $\mathcal{SL}(s)$ must be updated.

Figure 3b depicts this update procedure. When a node s has its new measurement $s.m$ outside local tolerance $\Phi(s)$, the node s sends its measurement upwards the routing tree. During the path from s to $s.filter$, the decider of each node $t \in \mathcal{L}(s)$, $t.decider$ must send the average value $t.avg$ upwards the tree until reaching the node $t.xfilter$.

VI. Experimental Study

This section studies the energy cost of our proposed protocols for TRA query processing, on a simulation platform for MICA sensor nodes. Each protocol packs multiple messages/events into one data packet, whereas the size of a packet is set to 30 bytes, as in [7]. For snapshot query processing, we compare the performance of the following protocols: (i) DBrute, (ii) ABrute, (iii) AIP, (iv) ABrute2 (Two-phase ABrute), and (v) AIP2 (Two-phase AIP). Each of them can fit 7 messages into a packet. For continuous query monitoring, we consider the direct application of the above protocols, and (vi) CAIP (Continuous Monitoring AIP) — see Section V, which can fit only 4 messages into a packet.

We measure the efficiency of a protocol as the energy cost consumed by sensor nodes in the network. According to [2], the major operations of a MICA sensor are: (i) transmitting of a packet, (ii) receiving a packet, (iii) idle listening (for 1 ms), and (iv) thermistor measurement. Their energy cost are 20, 8, 1.25, and 0.35 nAh, respectively. We evaluate the protocols experimentally by using these parameters: the threshold δ, the range λ, the number of nodes N (only for synthetic network). In each experiment, we vary one parameter value while fixing the values of other parameters.

Section VI-A describes the synthetic and real sensor networks used in the experiments. Section VI-B and Section VI-C investigate the performance of our protocols for snapshot queries and continuous queries respectively.

A. Experimental Setting

We proceed to discuss the distribution of sensor nodes and their measurements for both synthetic data and real data.

Synthetic sensor network and data. We generate a synthetic network with N sensor nodes by distributing them randomly in a square with $\sqrt{N} \times \sqrt{N}$ unit area. This ensures that the number of neighbors of each node is independent of N. The communication range (for each sensor) is set to 1.6 such that the communication links among the nodes form a connected graph. The routing tree is formed by setting the node in the center of the space as the root node. The routing tree of the default synthetic sensor network with $N = 1024$ nodes. This tree has a height of 21, and average node depth 11.2.

The domain of measurement $s.m$ is the interval [0,1]. We consider the distribution COR for the measurement values of nodes, for evaluating the snapshot TRA queries. For the location-correlated distribution COR, we randomly pick an anchor point z in the space and then generate each measurement value using the inverse Gaussian function of the distance $dist(z, s)$. In other words, nodes close to z have high measurements whereas nodes far from z have low measurements.

In order to generate measurements for continuous TRQ queries, we apply the COR distribution to generate the measurement $s.m$ of each sensor epoch-by-epoch, except that the anchor z moves along a trajectory at constant speed across the epochs. This continuous location-correlated distribution (ContCOR) models the case where extreme measurement values are caused by a moving object or phenomenon (e.g., animal, low-pressure system).

Real sensor network and data. The IntelLab dataset [22] consists of the topology of 54 sensor nodes deployed in a lab and a collection of 2.3 million measurements from those nodes. The communication range for each node is set to 6 such that the communication links among the nodes form a connected graph. The root node of the routing tree is located at a corner of the lab; the tree height 14 and the average depth

of a node is 7.42. The measurements are temperature values in the domain interval [16°C, 27°C]. The original collection of measurements contains some invalid values and missing values. Thus, we pre-processed those data by (i) discarding invalid values and (ii) utilizing known measurements of a node s to interpolate linearly its missing values at other epochs.

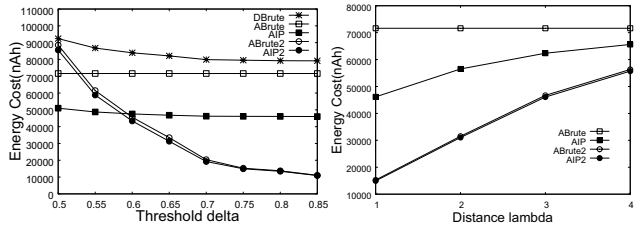
B. Performance on Snapshot Queries

In this section, we evaluate the performance of our protocols on snapshot TRA queries, by using the COR dataset discussed in Section VI-A. Note that ABrute serves as the baseline protocol for comparison.

Effect of threshold δ. We first study the effect of the threshold δ on the performance of our protocols, at $\lambda = 1$. Recall that the domain of the sensors' measurements is the interval [0,1].

Figures 4a plots the energy cost of our protocols respectively on COR data. When δ increases, fewer nodes s have measurements $s.m$ above δ and thus fewer nodes qualify as query result (i.e., $s.avg \geq \delta$). Regarding the energy cost, the cost of ABrute is independent of δ because all measurements $s.m$ are sent to the base station regardless of their values. DBrute has the overhead of broadcasting the sensors' measurements to their neighborhood nodes so it is more expensive than ABrute. AIP outperforms ABrute and DBrute because AIP employs filter nodes to discard sensors' measurements that all their influenced average values have already been computed. ABrute2 and AIP2 benefit from the correlation of data and they incur much lower cost than ABrute and AIP for large δ.

The above experiments show that DBrute is more expensive than ABrute, so we drop DBrute from subsequent experiments.



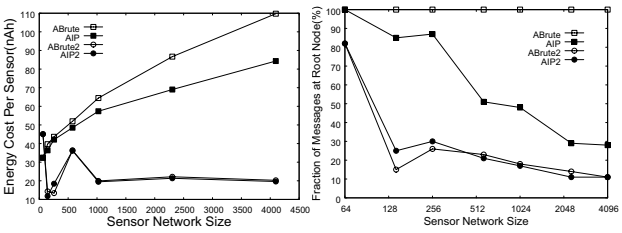
(a) varying the threshold δ (fixing $\lambda = 1$) (b) varying the range λ (fixing $\delta = 0.7$)

Fig. 4. Energy cost of protocols, COR data, at $N = 1024$

Effect of query distance λ. Figure 4b illustrates the energy cost of protocols on COR data, for various values of λ. Again, the cost of ABrute remains constant and the cost of AIP increases with λ. In contrast to IDP data, observe that the two-phase protocols (ABrute2, AIP2) have much lower cost than the other protocols. Due to the correlation of data, the broadcasting of measurements occurs only in particular region of sensor network; multiple messages can be packed in the same packet, reducing the cost of broadcasting measurements.

Effect of the number of nodes N. We proceed to investigate the scalability of the protocols, by varying the number N of nodes in the sensor network. For this purpose, we record two types of performance measures for each tested case: (i) energy cost per node, and (ii) fraction of messages (generated from all sensor nodes) received at the root node. During the process of forwarding messages to the base station, the root node usually receives much more packets than other nodes. Therefore, a desirable protocol should result in a low value of (ii).

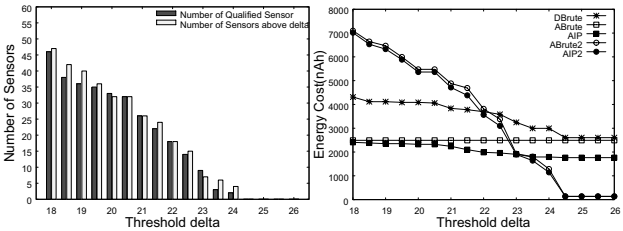
Figures 5a,b plot the energy cost per node, and the fraction of messages received at the root node, for the COR data, with respect to N. Regarding the energy cost per node, AIP achieves appreciable cost savings over ABrute, yet its cost increases slowly as the network size increases. The two-phase protocols (ABrute2, AIP2) are relatively insensitive to the the network size and they become significantly cheaper than Abrute at large networks. Regarding the fraction of messages received at the root node, both AIP and AIP2 apply filter nodes to discard sensors' measurements so the root node only receives a small fraction of messages produced from all nodes. Two-phase protocols (ABrute2, AIP2) achieve substantial cost saving for the COR data.



(a) energy cost per node (b) fraction of messages

Fig. 5. Effect of the network size N, COR data, at $\delta = 0.8$ and $\lambda = 2$

Results on real sensor network data. We proceed to use the IntelLab real data (mentioned in Section VI-A) for evaluating the performance of our protocols for snapshot queries. The sensor network of IntelLab is small (with only 54 nodes) so we set the query range to $\lambda = 1$ in subsequent experiments. Recall that the domain of measurement is the interval [16°C, 27°C]. Figures 6a,b show the result size and energy cost of the protocols as a function of δ. AIP2 outperforms the other protocols for high δ values. AIP performs better than ABrute and it has stable performance for a wide range of δ values.



(a) result size (b) energy cost

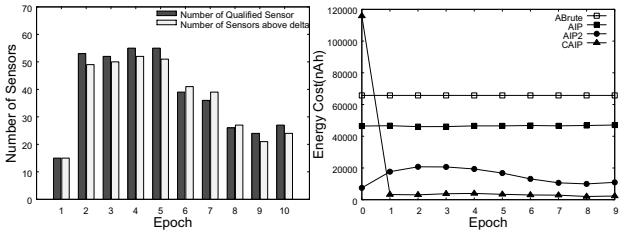
Fig. 6. Effect of the threshold δ, InterLab data, at $\lambda = 1$

C. Performance on Continuous Queries

In this section, we evaluate the performance of our protocols on continuous TRA queries, using the ContCOR dataset discussed in Section VI-A. We compare the performance of:

(i) the continuous monitoring protocol (CAIP), and (ii) the execution of snapshot protocols (ABrute, AIP, AIP2) for each epoch. Again, ABrute serves as the baseline for comparison.

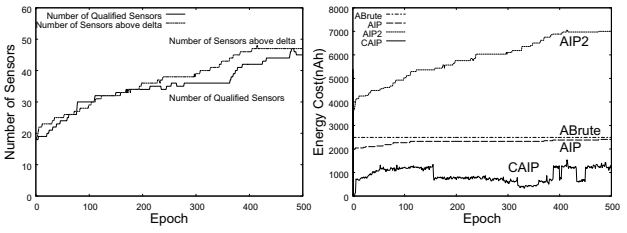
In the following experiment, we set $\delta = 0.2$ and $\lambda = 2$. Figures 7a,b show the result size and energy cost of our protocols, for the first 10 epochs. AIP performs better than the baseline (ABrute) steadily. AIP2 incurs much lower cost than AIP, and the cost of AIP2 follows the trend of the number of sensors above δ in Figure 7a. CAIP has high cost in the first epoch, where the local tolerance for all nodes is computed and disseminated. Nevertheless, CAIP has extremely low energy consumption in subsequent epochs. Therefore the total cost of CAIP for a long-running query is orders of magnitude lower than the cost of applying a snapshot protocol repetitively.



(a) result size (b) energy cost

Fig. 7. Cost of continuous evaluation, ContCOR data, at $\delta = 0.2$ and $\lambda = 2$

Results on real sensor network data. Figures 8a,b plot the result size and the energy cost of different protocols for 500 consecutive epochs, at $\delta = 22$ and $\lambda = 1$. The number of sensors above δ increases as the epoch advances, implying that the temperature was rising during that time period. Observe that the cost of AIP2 rises as the number of results. CAIP consistently outperforms all snapshot protocols (ABrute, AIP, AIP2) for all epochs.



(a) result size (b) energy cost

Fig. 8. Cost of continuous evaluation, InterLab data, at $\delta = 22$ and $\lambda = 1$

VII. Conclusions

In the paper, we introduced a novel query called the thresholded range aggregate query (TRA), which provides convenient means of summarizing the sensors' measurements in each local region, without being influenced by individual sensor abnormality. We present several protocols for snapshot and continuous evaluation of TRA queries, by developing corresponding in-network aggregation and filtering techniques.

Our experimental results suggest that our protocols achieve substantial amount of energy savings on both real and synthetic data. For snapshot queries, we recommend using the AIP and AIP2 protocols. AIP2 is scalable for a large sensor network, and performs well for queries with high threshold θ and low distance λ. AIP is more suitable for queries with low θ or high λ. For continuous queries, CAIP outperforms the repetitive application of snapshot protocols by a wide margin.

Our proposed methods assume that all sensor nodes and communication links function properly; in case of lossy communication, the routing tree could produce inaccurate result. In the future, we plan to extend our solutions by using multi-path routing structures [3], [8], [18], in order to mitigate the impact of lost messages via multiple paths.

Acknowledgement

This work was supported by grant HKU 7155/06E from Hong Kong RGC.

References

[1] M. LaPedus, "Intel Harnesses Wireless Sensors For Chip-Equipment Care," http://www.techweb.com/wire/26802594, techWeb.

[2] A. M. Mainwaring, D. E. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless Sensor Networks for Habitat Monitoring," in *WSNA*, 2002.

[3] J. Considine, F. Li, G. Kollios, and J. W. Byers, "Approximate aggregation techniques for sensor databases," in *ICDE*, 2004.

[4] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis, "Balancing Energy Efficiency and Quality of Aggregate Data in Sensor Networks," *VLDB J.*, vol. 13, no. 4, pp. 384–403, 2004.

[5] D. J. Abadi, S. Madden, and W. Lindner, "REED: Robust, Efficient Filtering and Event Detection in Sensor Networks," in *VLDB*, 2005.

[6] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

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

[8] A. Manjhi, S. Nath, and P. B. Gibbons, "Tributaries and Deltas: Efficient and Robust Aggregation in Sensor Network Streams," in *SIGMOD Conference*, 2005.

[9] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

[10] Y. Yao and J. Gehrke, "The Cougar Approach to In-Network Query Processing in Sensor Networks," *SIGMOD Record*, vol. 31, no. 3, pp. 9–18, 2002.

[11] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos, "Hierarchical In-Network Data Aggregation with Quality Guarantees," in *EDBT*, 2004.

[12] H. Yu, E.-P. Lim, and J. Zhang, "On In-network Synopsis Join Processing for Sensor Networks," in *MDM*, 2006.

[13] M. L. Yiu, N. Mamoulis, and S. Bakiras, "Retrieval of Spatial Join Pattern Instances from Sensor Networks," in *SSDBM*, 2007.

[14] W. F. Fung, D. Sun, and J. Gehrke, "COUGAR: The Network is the Database," in *SIGMOD Conference*, 2002.

[15] A. Silberstein, K. Munagala, and J. Yang, "Energy-efficient Monitoring of Extreme Values in Sensor Networks," in *SIGMOD*, 2006.

[16] X. Yang, H.-B. Lim, M. T. Özsu, and K.-L. Tan, "In-network Execution of Monitoring Queries in Sensor Networks," in *SIGMOD*, 2007.

[17] Y. Cho, J. Son, and Y. D. Chung, "POT: An Efficient Top-k Monitoring Method for Spatially Correlated Sensor Readings," in *DMSN*, 2008.

[18] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson, "Synopsis Diffusion for Robust Aggregation in Sensor Networks," *TOSN*, vol. 4, no. 2, 2008.

[19] B. J. Bonfils and P. Bonnet, "Adaptive and Decentralized Operator Placement for In-Network Query Processing," in *IPSN*, 2003.

[20] A. Coman, M. A. Nascimento, and J. Sander, "On Join Location in Sensor Networks," in *MDM*, 2007.

[21] Y. Kotidis, "Processing Proximity Queries in Sensor Networks," in *International Workshop on Data Management for Sensor Networks*, 2006.

[22] P. Bodik, W. Hong, C. Guestrin, S. Madden, M. Paskin, and R. Thibaux, "Intel Lab Data," http://db.csail.mit.edu/labdata/labdata.html.