Dynamically Self-Organizing Sensors as Virtual In-Network Aggregators and Query Processors in Mobile Ad-Hoc Sensor Databases*

Aris Ouksel
Department of Information and Decision Science
University of Illinois at Chicago
aris@uic.edu

Lin Xiao
Department of Computer Science
University of Illinois at Chicago
lxiao5@uic.edu

Manfred Hauswirth
Digital Enterprise Research Institute (DERI)
National University of Ireland, Galway, Ireland
manfred.hauswirth@deri.org

Abstract

Efficient in-networking processing of higher-level query types such as range and aggregate queries are a major challenge in distributed, data-intensive, and sensor networks. In this paper we propose a novel data management infrastructure based on multidimensional indexing techniques to support fast aggregate and non-aggregate query processing. Our approach applies to stationary and mobile environments and is based on an overlay structure, called AGGINDEX. AGGINDEX organizes the sensors in a tree structure of virtual processors which continuously compute both precise and approximate aggregations. Our experiments show that AGGINDEX provides a significant gain in latency and message costs over gossip-based aggregation and spanning-tree based aggregation techniques as used by TAG and Cougar.

1. Introduction

Recent advances in sensor technologies enable a new generation of massive-scale, self-organizing, wireless sensor networks (WSNs), consisting of small low-cost, low-power, and easy-to-deploy sensors. These networks can be applied in a wide spectrum of commercial, environmental, health care and military application areas. As these networks are being deployed at larger scales and incorporate more complex functionalities, the volume of data sensed, stored, and queried increases significantly. These large amounts of data along with the need to provide support for global network monitoring and continuous collection of aggregated application data makes range and aggregation queries the two most frequent query types in WSNs to reduce latency and message costs. Therefore, distributed indexing mechanisms, which cluster query answers to smaller areas of the physical space by preserving data locality, support the major goals of data access efficiency and low energy consumption, which are key requirements in sensor networks.

The practical requirements and constraints under which query processing in mobile ad-hoc sensor databases has to be performed is best illustrated by a typical application scenario, for example in a modern health care facility: Here, vital parameters of patients such as pulse, blood pressure, and body temperature could be monitored through sensors attached to the bodies of patients. As patients physically move in the health care facility, sensors are mobile and form dynamically organized sensor networks. Sensors can also track the daily activities of patients and may also be deployed in rooms to collect environmental conditions at the health care facility such as humidity, noise level, and air temperature. All measured data are recorded as events, which are then stored within the network.

Sensor networks deployed in a such a scenario enable the continuous monitoring of patients without actually constraining them physically while offering the possibility of immediate reaction in case of an emergency. Health care staff may simply be interested in finding out a patient's location or a doctor may request "the medical records of all cancer patients whose body temperature is above 98 degrees today" or "the maximum and average body temperatures for each resident in the West Wing". A wireless mobile sensor network with database capabilities will transparently process such queries and return the answers regardless of pa-

*This work was partially sponsored by NSF grant IIS-0326284.

tient's location or availability. This can significantly reduce the costs in health care while being beneficial to patients as well. With such data being gathered, stored and aggregated in-network, regular reports could be sent to hospital staffs automatically. Also, alarm notifications in emergency situations, such as when a resident suffers a sudden heart attack, can be sent without delay, thus increasing his chance for survival and recovery.

From a data processing side such scenarios require efficient data management and query processing techniques which support point, range and aggregate queries. Efficiency and scalability of the query processing both in terms of number of sensors and amount of data while minimizing query latency has to be guaranteed as well. Typical queries to be expected in a health care scenario include:

- `SELECT pulse-rate FROM pulse-sensors WHERE resident-id = 998` (point query)
- `SELECT resident-id FROM complex-readings WHERE systolic > 135 and body temperature IN (98, 102)` (range query)
- `SELECT COUNT(*), AVG(pulse-rate) FROM complex-readings WHERE diastolic > 100` (aggregate query)

Such queries are also representative for many other application domains and serve as good overall test cases. Aggregation in traditional database approaches is done at the back-end at one or a few centralized servers after collecting all data from the network. This communication-intensive approach is not feasible in large-scale, data-intensive sesnor networks due to power constraints of sensors. In-network aggregation [17, 30, 29] and group-aware network configuration methods [24], which calculate partial results along intermediate nodes on query routing paths, perform much better in such settings and can decrease significantly energy consumption of sensors by reducing the transmission and computation costs.

When an aggregate query is issued in TinyDB [17] or Cougar [29], a routing tree rooted in the query issueing node will be constructed to perform in-network aggregation along the tree. According to the aggregation mechanism of TAG [17] which is implemented on top of TinyDB, this tree is constructed dynamically by flooding the network from the source node and a different tree is constructed each time a query is issued from a different source node. Clearly, these aggregation processing mechanisms suffer from constantly flooding the network when queries are issued by different nodes. Also, since each sensor can store data of any possible value, searching for records with a specific value or value range requires exhaustive search of the network, i.e., each node in the network.

The infrastructure proposed in this paper tries to avoid these drawbacks and organizes the physical sensor network by partitioning the data space into subspaces and assigning each subspace to a sensor, which then takes over the responsibility of managing data belonging to this subspace. For routing purposes a simple overlay called AGGINDEX is constructed. The construction is based on the data space partitioning among sensor nodes. An AGGINDEX tree is shared by all aggregate queries. Updates to the AGGINDEX tree and to data space partitions, which may occur due to sensors going online/offline or and mobility of sensors, are always local, i.e., only affect one to a few nodes. The AGGINDEX tree supports fast aggregation propagation and computation along paths while concentrating query searches to affected subareas in the network, thereby reducing energy consumption and latency. It also enables intermediate result caching for re-use by future queries. Additionally, AGGINDEX offers summary keeping and query caching, methods which further improve the performance of aggregate query processing. Analysis and experiments presented in this paper show that our approach significantly reduces query routing cost and latency compared to other approaches, without incurring significant overhead for constructing and maintaining the overlay. Our infrastructure also provides efficient approximate aggregate query processing solutions by having sensors estimate values stored in its local storage according to the data space partitioning without examination of each data record. This significantly reduces the latency and local processing cost for answering aggregate queries.

The paper is organized as follows: Section 2 presents related works. Section 3 covers definitions and assumptions, setting up the background for following sections. Section 4 proposes the concept of AGGINDEX tree and its construction methods in static or mobile environments. Further on, methods for both precise and approximate data aggregation using the AGGINDEX tree are presented and analyzed. Section 5 presents non-aggregate query routing and processing techniques. Section 6 covers our simulation system and performances of query processing and aggregation performance through comparison with other approaches. The paper is concluded in Section 7 on future works.

2. Related Works

Aggregation has been studied extensively in the database community [12]. Madden et al. [17] propose a generic framework, called TAG, to support aggregate in-network queries by constructing spanning trees rooted at predefined base-stations ("sinks"). Query evaluation in TAG is done in two steps: In the distribution phase, a query is flooded from the requesting node into the network and constructs an aggregation spanning tree. In the collection phase, intermediate results are propagated back along the tree by child-parent messages. A node may select more than one parent at a higher level, in which case the intermediate result is sent to each of the parents. Spanning trees are also used in Cougar [29] and similar approaches are also described in [30]. [24] introduces the TiNA framework which can exploit pre-existing aggregation schemes such as TAG, to reduce energy consumption in the context of in-network aggregation.

Kempe et al. [15] propose a gossip-style protocol to compute aggregations in P2P networks. Each node assigns a non-negative probability to its neighbors with the sum of all probabilities being equal to 1. Then in the next round it sends each neighbor its share of the results and probability according to the probability assigned. However, convergence of gossip-style protocols in general is very slow, as is shown in [6] to require at least $O(n^{1.5}logn)$ transmission even for optimized gossip protocols.

For robustness and scalability the concept of *sketches* is applied frequently [12, 10, 11, 2]. For example, [4] uses sketches to compute approximate duplicate-sensitive aggregates, such as COUNT or AVERAGE, across faulty sensors. The basic idea is to use a hash function to map the value of a data tuple into the appropriate bit of the sketch. Query computation consists of two steps: (i) flooding across the network having each node computing its level and (ii) each node constructing its own local sketch and broadcasting it to the next higher level. The steps are repeated in each iteration until the sink gets the final result. Nath and Gibbons [18] introduced synopsis diffusion, a concept similar to sketches.

Multidimensional indexing structures have been extensively studied in the past twenty years. A literature survey can be found in [7]. Recently, researchers have revisited the indexing problem in a distributed environment first in P2P networks, such as CAN [21], CHORD [25], P-Grid [1], or [20], and then in wireless sensor networks with structures such as GHT [22] , DIM [16], or our previous works on localization-integrated indexing of mobile ad-hoc sensor networks [28, 27]. In wireless sensor networks, sensors are distributed over the physical space as ad-hoc and self-organizing agents. An appropriate indexing structure must emerge from the interactions of sensors and be adaptive to the topology changes caused by sensors online/offline or being mobile.

DIFS [3] is an indexing structure that sets up a multi-rooted quad-tree when partitioning the data space. Each node has 2^l parents depending on its level l in the tree. Each leaf node stores the full data range of an attribute. Each intermediate node stores a portion of its children's range. For a node, all its parents cover exactly its data range. Meanwhile, a parent covers all its children's data histograms. Thus, a node closer to the root of the tree will have more limited data range but broader data histogram. Multi-resolution storage techniques are also exploited in DIMENSION [8] and [9], which use in-network wavelet-based summarization and progressive aging of summaries to support long-term querying in storage and communication-constrained networks. The goal is to enable highly efficient drill-down search over summaries and efficient use of network storage capacity through load-balancing and progressive aging of summaries.

[26] proposes a two-tier data storage strategy for answering precision-constraint approximate queries by keeping a high-precision version of data in the sensor and a low-precision version of data in the base station. A query with a required precision can be answered at the base station if the required precision can be met by the data in the base station. A refreshment policy keeps the base stations updated with the latest sensor data.

Also, several protocols have been proposed for monitoring the network state including node failures, computing the coverage and exposure bounds, energy supply depletion, and topology discovery [30]. Deshpande and Madden [5] describe a centralized probabilistic approach where approximations are computed with probabilistic confidence intervals at sensors. Using the summary maintained by the individual sensors, a statistical distribution is built hierarchically. Hellerstein et al. propose an online aggregation interface [13], which permits users to observe the progress of their queries and control their execution on-the-fly. These various approaches, as well as sketches described earlier, are complementary to our work and can be integrated into our overall infrastructure.

3. Data space partitioning

The basic idea of indexing is to partition a d-dimensional data space into a finite number of subspaces which cover the search space, and then assign sensors to each of the subspaces to manage data and answer queries relevant for their subspaces. The goal of indexing is to obtain efficiently optimal storage and search cost for data in the network.

Our approach described maps the data space partitions space onto the physical space of sensor locations. This approach is proven to be able to optimize search cost of multidimensional range queries by its locality preserving capability [19, 28, 16]. Alternative approaches map data space partitions onto a virtual coordinate space of sensors. For simplicity, we assume that the sensor network environment covers a bounded two-dimensional rectangular area as the extension to three dimensions is straightforward.

Suppose a relation R has attributes $R_1, R_2, \cdots, R_k$, each R_i taking its value from a bounded but not necessarily finite domain D_i. An index is to be built along attributes $R_1, \cdots, R_d$ of relation R, called *index attributes*.

Let the d-dimensional normalized data space $U^d = [0,1)^d$ be partitioned into subspaces of hyper-rectangles defined as *data regions*. Each data region is uniquely represented by an identifier *r-id*, a binary sequence with length l: $i_1, i_2, \cdots, i_l$. l is also called the *level* of a data region. A *split* along one of its dimensions divides a data region into two equal data regions, whose *r-id*s are obtained by appending 0 or 1 to the binary representation of the current r-id.

The *key* of a data tuple $t = (t_1, t_2, \cdots, t_d)$ of relation R, is calculated by interweaving the bits of t_i's binary representation $t_{i,1}t_{i,2}\cdots t_{i,max}$:

$$k = k_1 k_2 \cdots k_{d*max} = t_{1,1}t_{2,1}\cdots t_{d,1}t_{1,2}\cdots t_{d,max} \quad (1)$$

A sensor obtains an *r-id* when it joins the network. Unlike intrinsic properties such as permanently assigned MAC addresses or permanent-IDs, *r-id*s are assigned to sensors according to sensor locations and may change as a result of sensor movement. Suppose a sensor n is assigned an *r-id* I. All data tuples whose keys are a prefix of I are now mapped to sensor n. Sensor n is responsible for the storage, maintenance, and query answering of those tuples.

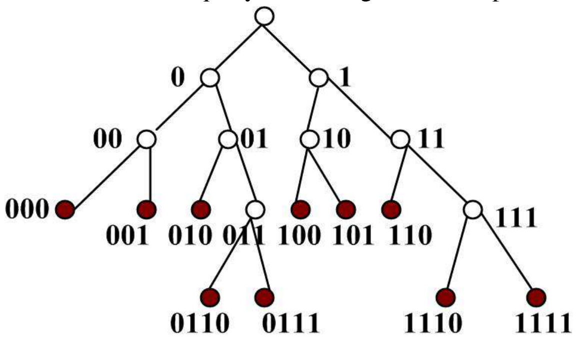


Figure 1. R-id Tree

To illustrated this strategy we give a brief example: Let us assume a 2-dimensional use case where the two domains D_1 and D_2 are pulse rate (40 beats/minute – 160 beats/minute) and systolic blood pressure (70mm Hg – 230mm Hg). A data record $(130, 190)$ will be normalized to $((130-40)/(160-40), (190-70)/(230-70)) = (90/120, 120/160) = (0.75, 0.75)$, which is represented in binary form by (0.11, 0.11). The key of t, denoted as $k(t)$ and calculated by interweaving the bits cyclically from each dimension, is 1111. In Figure 1, this data record is mapped to data region 1111.

The data space partition can be represented by a virtual tree's decomposition as shown in Figure 1, referred to as the *real identifier tree*, or the *r-id tree*. Starting from the root, a split induces two new *r-id*s as leaves, which become children of the *r-id* that has just split. One of them remains at the splitting sensor, while the other is assigned to the new node. The binary decomposition process induces a recursive partitioning of the data space into *r-id*s of data regions represented by the leaves of the tree. The partitioning of the data space stops when a data region has exactly one sensor. A sensor then acquires the *r-id* of that data region. In a fully specialized tree as shown in Figure 1 each sensor's current *r-id* is represented by a leaf node in the *r-id tree*. Intermediate nodes are not represented by a physical node (sensor).

Sensors become *neighbors* if they can communicate directly. Each node n maintains an up-to-date *r-id* list of its neighbors. A *buddy node* of identifier *i* is defined as a node which shares the longest common prefix with *i* in binary form. Before a node turns itself off, or leaves its current area, its storage will be transferred to its buddy node. If the buddy node discovers that the identifiers of two data regions only differ in the last bit, they will be merged into a new data region. For example, a merge of data regions 1110 and 1111 produces a new data region 111. Data mapped to a *hole* in the data space not covered by any sensor will also be stored at the buddy node.

More details of this approach for organizing an indexing structure can be found in [28].

Before describing our approach in the next section, we make the following model assumptions: The wireless sensor network is connected all the time regardless of sensor mobility and power shift. This is necessary for the validity of query routing. The case of a disconnected network is beyond the scope of this paper. Each sensor is able to share its identifier with its neighbors, but only when requested. In the sequel, sensor and node are used interchangeably if there is no ambiguity.

4. Aggregation using the AGGINDEX tree

The *AGGINDEX tree* is a tree overlay where *r-id*s are also represented by leaf nodes. But it differs from the *r-id* tree in that an AGGINDEX tree is a rooted tree with internal nodes as virtual identifiers assigned to sensors, while a *r-id* tree is not a real tree because its internal nodes are not mapped to sensors.

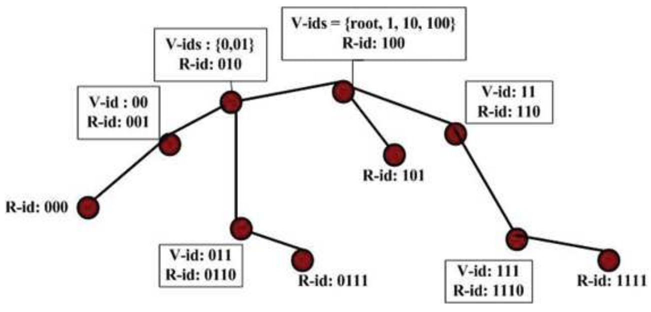


Figure 2. AGGINDEX Tree

In an AGGINDEX tree, some sensors will have multiple identifiers, one being the so-called *real identifier*, or *r-id* which defines the data partition stored at this sensor. The other possibly existing identifiers are historical identifiers which represent the node's evolution in terms of the data space partitioning process. We call these identifiers a node's *virtual identifiers* or *v-id* here. Virtual identifiers enable the assignment of responsibilities such as sub-result collection from its sub-tree for aggregate query, data summary maintenance for its sub-tree, and query result caching. As an example, a sensor node with a *r-id* 01011 may be assigned a *v-id* 010. Although this sensor only manages storage and retrieval of data mapped to data region 01011, it was given the additional responsibility of maintaining the data summary of its sub-tree for the virtual data region 010 by the assignment of *v-id* 010.

4.1. History approach to construct an AGGINDEX tree

In the static case where the network topology and node reachability are stable we can construct and maintain an AGGINDEX tree by using history information of the space partitioning process. The resulting AGGINDEX tree is a by-product of data space partitioning without incurring significant extra cost for setup and update.

The AGGINDEX tree is updated when updates to the indexing structure occur. Each sensor node stores its *r-id* and a list of *v-id*s locally. Every *id* has a *parent* in the tree. The parent of an identifier of length *l*'s is defined as a *v-id* whose binary presentation is a prefix of the *r-id* with length of *l*-1.

Two types of updates can occur: a "merge" or a "split." When splitting, the *r-id* is assigned as already described in the previous section. Additionally, the old *r-id* will be added as a *v-id* to the node's *v-id list* and the parent of the new r-id is set to the the old *r-id*. In the case of merging, the *r-id* is set to a new r-id according to the partition merging process defined in the previous section and the old *r-id* is deleted from the *v-id list*.

This can be exemplified in Figure 2. Sensors are represented by tree nodes and their identifiers are given next to them. In fact, the topology in Figure 2 is the same as in Figure 1. A tree node with *r-id* 000 is managing data region 000. It has a parent with *r-id* 001 and v-id 00. Let us a assume a new sensor node *m* joins, and node *n* with *r-id* 1111 splits its data region 1111 into a data region 11110 managed by itself and 11111 to be managed by m. The AGGINDEX tree is updated by adding n's old *r-id* 1111 to its v-id list and making n the parent of m.

The height of an AGGINDEX tree is $O(logn)$ in the average case, because the tree height is determined by the length of the longest *r-id* in the network and the average *r-id* length is $O(log_2 n)$. However, in the worst case if the node distribution is skewed, the tree height could significantly increase.

In a mobile environment where the topology is not stable, constructing an AGGINDEX tree is more complicated. For example, let us assume a node with *v-id* 110 in Figure 2 moves east and eventually loses connection with its parent node of *v-id* 100. Node 110 was responsible for the virtual data region 11 before its movement. Since partial aggregation results for virtual data region 11 may need to be propagated along the tree to the virtual data region 1 and the data region 1 is managed by the parent node of *v-id* 100, the AGGINDEX tree becomes disconnected although the network is still connected through other alternate paths. This situation requires a further maintenance strategy described in the next section.

4.2. Dynamic update approach for mobile environments

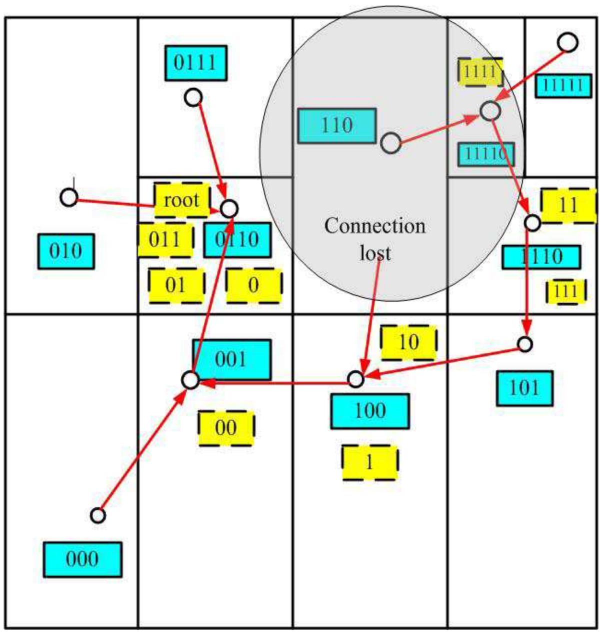


Figure 3. Dynamic Update in the AGGINDEX tree

When a sensor detects that the connection with its parent node in the index tree has been lost, it sends a *re-adjust* message with its own information including *id*s and the lost parent's *id* to its own children. Figure 3 shows an example.

In the example of Figure 3, the node with *r-id* 110 has lost its connection with its parent with *r-id* 100. Thus it initiates a re-adjust message *reAdjust*($\langle$*n.info, 110,*$\{$*11*$\}\rangle$, $\langle$*p.info, 100, 1*$\rangle$) containing its own *r-id* 110, *v-id* 11, and lost parent 100's information. Let us assume the message first arrives at a child with *r-id* 11110. Node 11110 does not have a connection to *100*, so it forwards the message to its children, i.e., node 1110 which in turn forwards it to node 101 which actually has a connection to node 100 at its new position. Since 101 is a child of 100, node 1110 has found a new connection to 100 and sends a "reconnect" message back along the routing path. Upon receiving the "reconnect" message, each node will mark the message sender as its new parent node. Nodes on the path will adjust their *v-id*s accordingly. The adjustment result is shown in Figure 3 by arrows pointing from children to parents.

Figure 4 shows the corresponding pseudo-code. Update is called when the connection to the parent is lost. Each node keeps a neighbor-list with all its neighbors' *id*s and parent *r-id*. When a parent node cannot be located within a certain TTL, the node one level up in the tree from the lost parent will temporarily serve as the parent.

aggindexDynamic()

```
if ($parentconnection == unvailable$)
{initiate message reAdjustTree(rid, vidlist, lostparentid) and send it to all children;}

While ($time < TTL$) {
Upon receiving a re-connect message from node m:
$parent = m$; Update vidlist.

Upon receiving reAdjustTree message from node m:
check if ((parentlostid appears any of its neighbors k) or (parentlostid belongs to any neighbor k's parent))
Yes: send reconnect(self) to m; $parent = k$; Update vidlist;
No: forward reAdjustTree message to its children attaching n's own information {rid, vidlist}
}
$parent \leftarrow$ oldparentid.parent;
```

Figure 4. Dynamic index tree re-adjustment

Clearly, the history approach is efficient for static environments by incurring minimal additional cost. When applied to the mobile environment, however, it performs sub-optimally. This is because frequent topology changes in mobile environments will outdate history data rapidly. The tree can become disconnected when a node moves away from its parent. In a mobile environment, the history approach must be complemented with dynamic updates to maintain the tree when a disconnection from parent node is discovered by a sensor. The dynamic update method incurs more message cost per update compared to the history method, but updates are still local thus keeping the overhead small.

4.3. Efficient in-network aggregation

In this section we will discuss how the AGGINDEX tree can be utilized to efficiently answer aggregate queries. TAG [17] lists the following common attributes when classifying aggregation in sensor networks: (1) *Duplicate sensitivity*, which specifies whether an aggregate function will return the same result when there are duplicate values. Examples are MEDIAN, AVERAGE, and COUNT. MIN, MAX, and COUNT DISTINCT are, on the contrary, insensitive; (2) *exemplary/summary*: Exemplary aggregations (such as MIN, MAX, and MEDIAN) always return a representative value while summary aggregates (such as AVERAGE and COUNT) perform some calculation over the entire data space and return the calculated value; (3) *monotonic aggregates*, allow early testing of predicates in the network, for example, MAX and MIN. Significant savings in the overall number of messages sent through the network is possible by knowing in advance the location of a node or a set of nodes that hold such values. (4) *Partial state requirements*. Aggregates such as SUM and COUNT require partial state records that are the same size as the final aggregate. The AVERAGE function requires a partial state records containing two values, i.e., both the SUM and the COUNT. Types of aggregate queries including MIN, MAX, SUM, AVERAGE, COUNT are discussed next in this paper.

For monotonic aggregate operators such as MIN and MAX, the aggregate value is mapped to the sensor with the smallest or largest *r-id* within the specified query data range in our infrastructure, because the intrinsic characteristics of the indexing structure prescribes that the smallest or largest value is stored at a sensor with the smallest or largest *r-id*. Processing these aggregations can be reduced to search in the one sensor that fits this criteria. Query routing can be treated in the same way as single point and range queries in the following section, without relying on the AGGINDEX tree. This is because monotonic aggregates do not require partial state. Significant saving of energy is possible because of the early knowledge of result-holding sensors.

The AGGINDEX tree is exploited for processing non-monotonic aggregate operators such as SUM, AVERAGE and COUNT. When an aggregate query is issued, the boundary of the data set to be examined for the aggregate computation is specified in the query key. *Query key*, denoted *qk*, is the key calculated from a query q to resolve point and range queries. When resolving a point query, *qk* is calculated by interweaving the bits of the normalized query attributes. When resolving a range query, the query key is calculated as the *common prefix* of query keys covered by the spectrum of attributes to be searched. A query q then is routed towards all sensors whose *r-id*s are prefix of $qk(q)$.

Each query key *qk* is mapped uniquely to a tree node n in the AGGINDEX tree where the binary representation of *qk* is equal to or contains one of n's *v-id*s or *r-id*. This tree node is referred to as the *aggregate root* for query q. Only the sub-tree from the aggregation root will contain relevant data queried by q.

Processing of a non-monotonic aggregate query consists of the following steps: First, the query issuer s generates a query q including *qk* and sends out q. Second, query q

is transmitted in the network by routing methods such as geographic routing toward its aggregate root. When a node *r* discovers a match between one of its *id*s and $qk(q)$ on receiving *q*, it declares itself aggregation root of *q* and adds its own information to *q*. Third, *r* sends out *q* along its sub-tree in the AGGINDEX tree. *q* will be propagated down the sub-tree until leaves are reached. Nodes will examine their local storage and perform partial result aggregation if possible. The partial result set is then propagated up along the tree to its parent node. Finally, nodes of each level in the sub-tree await information from all its children, combine the result, and sent it up to the next level. The aggregate root *r*, on receiving and the complete result, will post-process the aggregation result and route a query answer back to the query issuer using the same mechanism for query routing described earlier.

4.4. Approximate Aggregations

Madden et al. [5] propose a centralized scheme to support probabilistic query processing using approximations with probabilistic intervals of confidence. Interestingly, we observe that the AGGINDEX tree somewhat captures the goal in that non-monotonic aggregate queries such as SUM can be approximated along the tree without examining raw data stored at sensors.

In many scenarios, exact aggregate computations may not be necessary and are often too costly to be performed frequently. Approximate answers can reduce processing time and energy consumption significantly. Since a *r-id* captures the data range mapped to it, it can be used to answer approximate queries by just estimating the average value stored at the sensor or the number of records stored at the sensor.

Let the number of data items stored at a sensor *n* be *c(n)*. As discussed earlier, query *q* is directed to the aggregate root. The value of all tuples stored at each sensor can be approximated by the middle point of each sensor's data range indicated by *r-id* or through a histogram. [28] provides the mathematical details for this calculation.

A partial aggregation result is approximated with the middle point value and *c(n)*, and then propagated back along the tree. For example, with SUM, the partial result at *n* is obtained by multiplying the middle point value and *c(n)* and adding it to the partial result obtained from *n*'s sub-tree. Clearly, approximate aggregation can respond very quickly to queries. It is especially beneficial if time and energy requirements are very stringent and accuracy can be sacrificed a little bit. Experiments show high accuracy can be achieved by approximate queries in our infrastructure.

5. Non-aggregate query processing

In this section all prefix operations on keys refer to to the binary representation of the keys. Upon receiving a point query, each node checks whether its current *r-id* is a prefix of the query key. If it is the case, a query match may be found from searching the data currently stored at this node. When resolving a range query, the query key is calculated as the *common prefix* of data keys within the query's range. A query *q* is routed towards all sensors whose *r-id*s are a prefix of the query key $qk(q)$. Upon receiving a range query, a node checks if either of the following applies: (1) Its *r-id* is a prefix of the query key; (2) the query key is a prefix of the node's *r-id*. Either condition is possible because a query may cover a broad range of data and may have a query key shorter than a sensor's *r-id*. Conversely, it may also cover only a narrow range of data and has a query key longer than a sensor's *r-id*. In either case sensors satisfying this criteria can hold potential results and will be searched for tuples that match the specified query range.

Each node with a partial query result will deliver those data in a reply message back to the destination specified in query q. One simple but more costly way to assemble the final query result is to do it at the destination node after it has received all individual partial results. A more efficient strategy is to aggregate or assemble the result in-network as described earlier.

Query routing works as follows: The routing function $queryrouting$ is invoked when a node n receives a query q. $nexthop$ is invoked when n needs to determines which of n's neighbors q will be forwarded to. The query type is "multi" when q is a multidimensional range query and "single" when q is a point query. For a range query *qk* is computed as the common prefix of the enclosing values of the range query defined as (*lowerrange, upperrange)*. The algorithms for both functions are described in pseudo-code below.

Intermediate nodes are nodes whose *r-id*s do not share any prefix with the query key. Those nodes need not be searched. When forwarding a query, a node n calculates the Euclidean distance $\delta(n_i, q)$ between each neighbor n_i's *r-id* and $qk(q)$ and the node with the shortest Euclidean distance $\delta(n_i, q)$: $\delta(n_i, q) < \delta(n, q)$ is chosen as the next hop. If a node has multiple neighbors within the range of *qk*, each neighbor will be forwarded with *q*. If a node has multiple neighbors of the same distance to *qk(q)*, the query is forwarded to a randomly chosen neighbor.

If at any time, a closer node to the query key is not available among a node n's neighbors, an iterative breadth-first-search (BFS) is performed to discover a closer node to the destination. This is done as follows: Node n sends a *IterativeBFS* message which includes *q*, its own information including its *r-id*, and an iteration number to all its neighbors.

```
queryroutingn, Queryobject q)
```

```
queryrouting(n, Queryobject q)
next = $nexthop(n, q)$, forward q to next.
if (q.type = "multi")
$qk = multikey$ = common prefix of q's enclosing values: $(lowerrange, upperrange)$
else if (q.type = "single")
qk = $singlekey(q)$

if (($n.id$ is a prefix of $qk$ or qk is a prefix of $n.r - id$) and q.type = "multi")
search in $n$'s storage for tuples within (lowerrange, upperrange)
retrieve all matching tuples → result
else if ($n.id$ is a prefix of $qk$ or $qk$ is a prefix of $n.r - id$ and q.type = "single")
search in $n$'s storage for tuples matching the queried value, retrieve all matching tuples → result

return (result)
```

```
nexthop(n, Queryobject q)
For each neighbor $n_i$ of $n$,
$\delta(n_i, q) = sqrt((n_i.r - id_x - q.qk_x)^2 + (n_i.r - id_y - q.qk_y)^2)$
next={$\{n_i\} \mid \delta(n_i, q) = min(\delta(n_j, q), \delta(n, q))$}
if next != n, return(next)
else
{isCloser = false; i = 1
while (!isCloser)
{send message iterativeBFS(q,n,i) to all neighbors $n_i$ of n
if (one isCloserBFS($q, n_i, i$) messages received from $n_i$ has the value "negative") isCloser = true; i++ }
} return($n_i$)
```

Figure 5. *queryrouting* **and** *nexthop* **algorithms**

On receiving the message each neighbor decides if any of its own neighbors is closer to the query key than node n. If this is the case, the node will send a positive reply message *isCloserBFS* including its own information including *r-id* and the iteration number back to n. Otherwise the node will send a negative reply message to n. Once a closer node is discovered, the iterative-BFS process terminates and q will be forwarded to the closer node. Otherwise each node reached in the current iteration will start the next iteration by repeating the above steps. Figure 6 shows an example of a node *n*, which on receiving an incoming query *q*, starts an iterative-BFS discovery when a closer node cannot be found among its neighbors.

If no node is found matching a query key, i.e., the query is directed toward a hole in the data space not covered by any sensor, the query will return an empty result.

An alternative to iterative-BFS routing would be geographic parameter routing [14], which was introduced in GPSR. While our scheme can incorporate GPSR as the routing protocol, GPSR has some potential problems when applied to the situation of location-aware indexing where location information is likely to be imprecise [28]. GPSR uses greedy forwarding to forward a packet to a neighbor geographically closest to the destination. A neighbor to receive a query in the routing process is required to be closer than the node itself. If no such neighbor node is found, GPSR uses perimeter routing to route around the perimeter of the region until it discovers a node that is closer to the destination. However, perimeter routing works only on planar graphs, i.e., graphs with no crossing edges, generated from the current network graph by removing edges. Because of GPSR's reliance on correct planarization of the network, it is likely to suffer from many errors as the correctness of planarization is impacted by inexact position information and incorrect removal of cross edges, as shown in [23]. This has a negative impact on the robustness of the GPSR routing protocol. Also in mobile wireless sensor networks, where changes are frequent and relatively large signal errors occur, a more robust routing mechanism is necessary to compensate for the effect of imprecise or incorrect localization. Our approach provides a more robust routing strategy. It uses greedy forwarding and iterative BFS for packet forwarding when no closer node is found. Since routing is composed almost entirely of greedy forwarding [23], we argue that in denser networks iterative BFS will happen very infrequently and will return to greedy forwarding within a few iterations. In sparse networks, our routing algorithm will perform much better than GPSR in respect to robustness, and in fact will succeed as long as the network is connected. In summary, our routing structure tends to have a better success rate in both cases without significantly increasing average message cost.

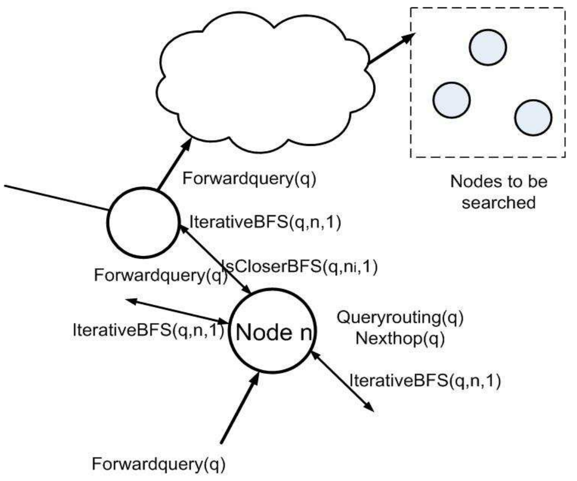


Figure 6. Routing of a query q **at node** n

The cost to resolve a query consists of: (1) Routing cost for directing a query to the vicinity of the query destination and (2) the cost for retrieving matching tuples from all destinations. For (1), the average cost in our case is $O(\sqrt{n})$ hops for dense networks. For (2), the cost depends on the number of destination nodes, which is approximately $N * qp$, plus local searching cost. Here N refers to the network size and *qp* refers to the ratio of the query range over the entire data range.

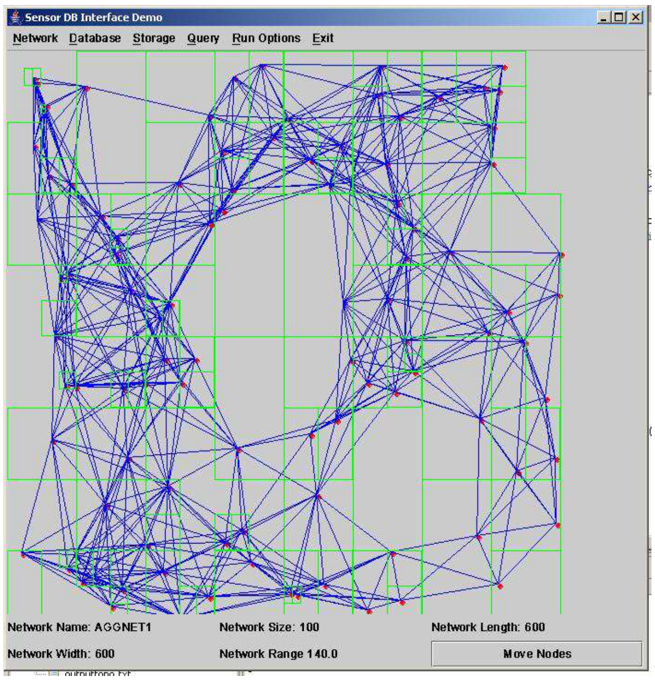
6. Experimental evaluation

To evaluate our approach we set up an extensive simulation model. We tested with network sizes varying from 25 to 1,000 with randomly distributed sensors in the network. In random networks nodes are not necessarily uniformly distributed. Some areas are denser than other areas, making the performance evaluation more realistic, unlike the strictly uniform grid topology often used. For simplicity, our simulator does not consider certain low-level properties of the network such as radio contention and the topology is maintained by regular "hello" messages every specified interval to advertise a node's existence among its neighborhood. We used both static and mobile networks in our experiments. For mobile networks, node movement is modeled in a discrete way: A random number is generated in each iteration step to determine the motion direction. A sensor has equal probability to move toward any of the four directions. This simulates a slow random walk in 2-D space. Figure 7 shows a network's topology and space partition (the rectangle enclosing a node). Our simulator can take network configurations and queries from user input and facilitates online observation of the simulation.

The period of time allotted for exchanging messages between two levels in a tree is called an *epoch*. The latency of a query result is dominated by the product of the epoch duration and the total number of epochs for a query to be answered since it was issued. By assuming the same epoch duration for different approaches, our comparison of latency is dominated by the number of epochs elapsed for a query to be answered.

We first compare our approaches with gossip-based aggregation (GOSSIP) and the class of aggregation techniques on spanning-trees exemplified by TAG and Cougar (TAG). *Query size* is defined as the percentage of query-related records over all records. In the uniform case, query size is proportional to the number of sensors that answer the query.

The benefit of using our approach over TAG and GOSSIP to answer monotonic aggregates such as MAX and MIN is obvious, because knowing in advance the nodes that hold such values significantly reduces query propagation cost. For non-monotonic aggregates such as SUM, we conducted experiments to compare different approaches with varying query size in latency and message cost spent per aggregate query, as shown Figures 8 and 9. Experimental results shown in these two figures were conducted with random networks of 100 nodes where SUM queries were issued by randomly selected nodes. TAG was implemented with the original scheme given in [17]. For GOSSIP the uniform gossip method presented in [15] was used.



Network Name: AGGNET1 Network Size: 100 Network Length: 600
Network Width: 600 Network Range 140.0

Figure 7. Network topology and partition by simulator. The solid lines represent network topology, and the rectangles show space partitioning.

Figure 8 shows the latency measured in the total number of epochs elapsed for answering a SUM query. For GOSSIP, we calculate the number of epochs elapsed until gossiping results fall within 95% of the correct result. Among these approaches, GOSSIP has the longest latency for all query sizes. This explains the slow convergence of gossiping protocols. TAG has an almost uniform latency for all query sizes. When a query covers most of the records in the network, TAG incurs shorter latency than our AGGINDEX approach. However, as query size drops, latency of AGGINDEX drops significantly and comes close to TAG when query size is 5% of all records in the network. The latency of AGGINDEX eventually drops below TAG as query size further decreases. Also, by incorporating summary keeping and query caching, the latency for AGGINDEX could be further reduced.

Figure 9 shows total number of messages sent per SUM query. Again the number of messages sent by GOSSIP is calculated as the total number of messages sent until the gossiping result is within 95% of the correct result. GOSSIP incurs exponentially more messages (as shown in the figure on the logarithmic Y-axis). This can be explained by the fact that every node in GOSSIP needs to send a message at the end of each epoch, while in TAG and AGGINDEX, for most of the epochs a node simply waits for the message from its next level without sending a message. Message cost for query sizes up to 100% of all records is similar for TAG and AGGINDEX. As the query size decreases, AGGINDEX tends to outperform TAG by requiring only $1/\sqrt{n}$ of the total messages of TAG (Figure 9 shows 1/10 since the network size is 100).

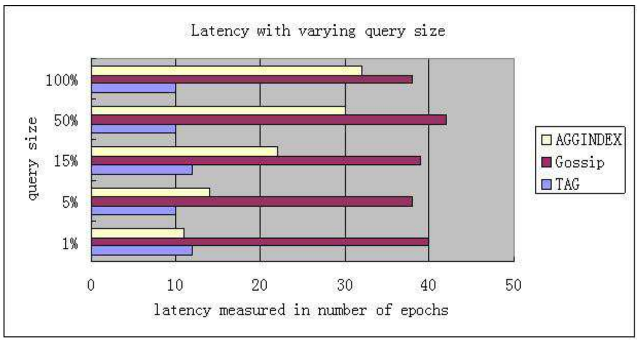


Figure 8. Latency vs. varying query size

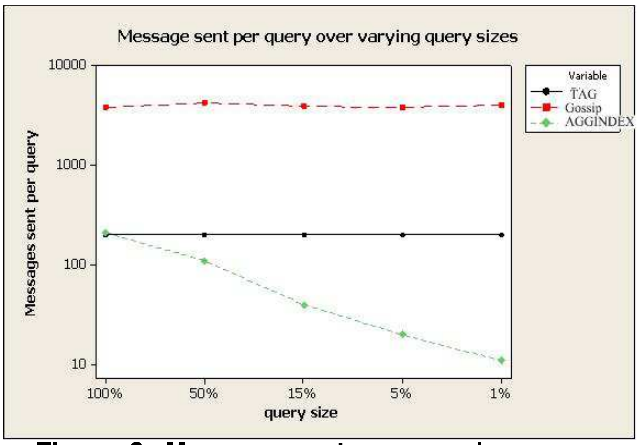


Figure 9. Message cost vs. varying query size

Next we demonstrate the accuracy of our approximated scheme for varying network sizes. Figure 10 shows the accuracy of the approximated schemed for uniform data distribution. Obviously, the approximated scheme can answer queries very accurately regardless of network sizes.

Figure 11 shows the approximated scheme applied to a SUM aggregation for an unknown skewed data distribution. The worst case is simulated by each sensor holding only upper-bound values of its data range. On average the approximated result is fluctuating within 10% around the exact result. The worst observed accuracy in the experiments is 86%, i.e., 14% below the exact result.

We would like to point out that the indexing mechanism used in our approach requires an insertion cost of $O(\sqrt{n})$ on average for each tuple, while TAG and GOSSIP are built on Direct Storage (DS) schemes and store tuples locally. [16] shows that as long as the ratio of the number of insertions to the number of queries is less than $\sqrt{n}$, our indexing scheme will out-perform flooding, which has a O(n) query cost. TAG requires flooding to construct the spanning tree for aggregation processing. Thus our scheme outperforms TAG when the ratio of the number of insertions of tuples to the number of queries is less than $\sqrt{n}$.

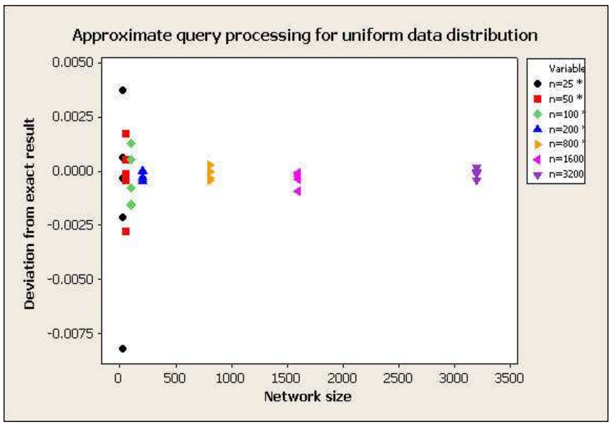


Figure 10. Approximation result for uniform data distribution

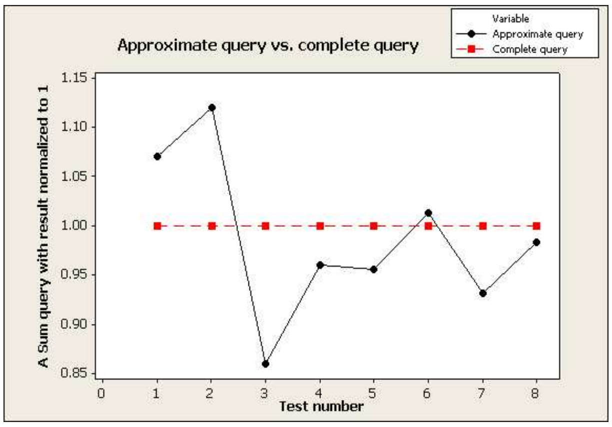


Figure 11. Approximation result for unknown skewed data distribution

7. Conclusions

Non-uniform distribution of sensors or data can adversely affect the indexing structures in performance. Where sensors are distributed unevenly in the network, the structure covered in this paper does not provide a balanced solution. The imbalance can cause some sensors to deplete their memory or power much sooner than the others. Our solution is to design a balanced AGGINDEX tree that spreads the load among sensors by alternating roles among sensors.

Another goal is to avoid the possibly long distances of data transmission from a generating sensor to its storage sensor and costly retrieval. We are developing a location-context-aware data management protocol that takes both location and context as indexing attributes. Simply treating location as other context attributes can be inefficient because vastly different attribute values generated at the same place can be stored at sensors far apart from each other. We use an identifier system for location-context-aware data management that reduces the data routing cost by mapping the storage place of an event to a larger sub-area and further decomposing the sub-place according to other attributes containing context information.

References

[1] K. Aberer, P. Cadure-Mauroux, A. Datta, Z. Desptovic, M. Hauswirth, M. Punceve, and R. Schmit. P-grid: A self-organizing structured p2p system. *SIGMOD RECORD 32(2)*, 2003.

[2] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in data stream. *6th International Workshop on Randomization and Approximation Techniques*, 2002.

[3] B.GreenStein, D.Estrin, R.Govindan, S.Ratnasamy, and S. Shenker. Difs: A distributed index for features in sensor networks. *First IEEE International Workshop on Sensor Network Protocols and Applications*, 2003.

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

[5] A. Deshpande, C. Guestrin, and S. R. Madden. Using probabilistic models for data management in acquisitional environments. *CIDR*, 2005.

[6] A. G. Dimakis, A. D. Sarwate, and M. J. Wainwright. Geographic gossip: Efficient aggregation for sensor networks. *IPSN*, 2006.

[7] V. Gaede and O. Gnther. Multidimensional access methods. *ACM Computer Surveys (CSUR)*, 1998.

[8] D. GANESAN, D. ESTRIN, and J. HEIDEMANN. Dimensions: Why do we need a new data handling architecture for sensor networks? *First Workshop on Hot Topics in Networks*, 2003.

[9] D. Ganesan, B. Greenstein, D. Estrin, J. Heidemann, and R. Govindan. Multi-resolution storage and search in sensor networks. *ACM Transactions on Storage, Volume 1, Issue 3*, 2005.

[10] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. *SIGMOD*, 2003.

[11] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. *ACM Symposium on Parallel Algorithms and Architectures*, 2001.

[12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–53.

[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD*, 1997.

[14] B. Karp and H. T. Kung. Gpsr: Greedy perimeter stateless routing for wireless networks. *Mobicom*, 2000.

[15] D. Kempe, A. Dobra, and J. Gehrkey. Gossip-based computation of aggregate information. *44th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003)*, 2003.

[16] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks, 2003.

[17] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *OSDI*, 2002.

[18] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. *SenSys*, 2004.

[19] A. M. Ouksel. The interpolation-based grid file. *PODS*, 1985.

[20] A. M. Ouksel and G. Moro. G-grid: A class of scalable and self-organizing data structures for multi-dimensional quering and content routing in p2p network. *Second International Workshop on Agents and Peer-to-Peer Computing (AP2PC'2003)*, 2003.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *SIGCOMM*, 2001.

[22] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. Ght: A geographic hash table for data-centric storage. *First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[23] K. Seada, A. Helmy, and R. Govindan. On the effect of localization errors on geographic face routing in sensor networks. *Third international symposium on Information processing in sensor networks POSTER SESSION*, 2004.

[24] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *VLDB*, 2004.

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM*, 2001.

[26] M. Wu, J. Xu, and X. Tang. Processing precision-constrained approximate queries in wireless sensor networks. *MDM*, 2006.

[27] L. Xiao and A. Ouksel. Scalable self-conguring integration of localization and indexing in wireless ad-hoc sensor networks. *MDM workshop on Mobile Location-Aware Sensor Networks*, 2006.

[28] L. Xiao and A. M. Ouksel. Tolerance of localization imprecision in efficiently managing mobile sensor databases. *MOBIDE*, 2005.

[29] Y. Yao and J. Gehrke. Query processing in sensor networks. *First Biennial Conference on Innovative Data Systems Research (CIDR).*

[30] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. *1st IEEE International Workshop on Sensor Network Protocols and Applications*, 2003.