Communication-Efficient Implementation of Range-Joins in Sensor Networks

Aditi Pandit and Himanshu Gupta

SUNY, Stony Brook, NY 11754.
apandit,hgupta@cs.sunysb.edu

Abstract. In this article, we consider energy-efficient implementation of the SQL join operation in sensor databases, when the join selection condition is a range predicate. Apart from two simple approaches, we propose distributed hash-join and index-join algorithms for implementation of range-join operations in sensor networks. Through extensive simulations, we show that hash-join as well as index-join approaches significantly outperform the simple approaches, even for moderately sized networks. Our experiments also reveal that although both approach scale well, the index-join algorithm performs better than the hash-join algorithm especially in large sensor networks.

1 Introduction

A sensor network is a multi-hop ad hoc wireless network of resource constrained sensor nodes. Each sensor node has limited computing capability and memory, and is equipped with a short-range low-power radio, a small limited battery, and various sensing devices. Sensor networks combine sensing, computing, and networking capabilities to realize high-level sensing tasks in a collaborative manner. Each sensor node in a sensor network generates a stream of data items that are readings (typically, scalar values) from its sensing devices. This motivates visualizing sensor networks as distributed database systems [2, 4, 10]. Since, message communication is the main consumer of battery energy and sensor nodes have limited battery power, it is important to implement the queries in sensor networks with minimum communication cost. Moreover, due to the limited computing and memory resources at each node, the query processing in sensor networks is necessarily distributed.

In this article, we focus on communication-efficient implementation of certain special cases of SQL join operation in sensor networks. In particular, we address in-network processing of the SQL *range-join* operation, which is a special case of the join operation when the selection condition involved is a range predicate. We propose various distributed algorithms. One of our proposed hash-join algorithm can be shown to incur optimal communication cost under certain assumptions.

2 Range Join in Sensor Networks

In this section, we start with presenting an overview of sensor network databases.

Sensor Network Databases. A sensor network consists of a large number of sensors distributed randomly in a geographical region. Each sensor has limited processing capability, is equipped with sensing devices, and has a low-range radio. Two sensor nodes can communicate with each other if the distance between them is less than the *transmission radius*. We assume that each sensor node in the sensor network has a limited storage capacity. Also, sensors have limited battery energy, which must be conserved for prolonged unattended operation. Each sensor node in a sensor network generates a streams of data tuples, and groups of sensor nodes producing tuples with the same format contribute to a single *data stream table*. In a sensor network, such data stream tables can be looked upon as partitioned horizontally across (or generated by) a set of sensors in the network. In a sensor database system, a query is typically initiated at a node called the *query source* and the results are routed to the query source for storage and/or consumption.

Problem Formulation. The SQL join ($\bowtie$) operation is a binary operation used to correlate data from multiple tables. *Range-joins* are joins wherein the join-predicate is whether two columns (*join-attributes*, usually with the same semantics), one from each operand table, have values that are within a given range of each other. *Equi-joins* are a further specialization of range-joins wherein the join-predicate is an equality of two columns, one from each operant table. In this article, we consider the problem of efficient in-network implementation of range-joins in sensor networks. In particular, we consider a join operation, initiated by a query source node Q, involving two data streams R and S distributed across some geographic regions $\mathcal{R}$ and $\mathcal{S}$ in the network. The main performance criteria for our distributed implementation is minimum communication cost, which is defined as the total data transfer between neighboring sensor nodes.

Related work. The vision of sensor network as a database has been proposed by many works [2, 4, 10, 14]. However, prior research has only addressed limited SQL functionality – single queries involving simple aggregations [6, 8, 15] and/or selections [9] over single tables [7], or local joins [15]. So far, it has been considered that correlations such as median computation or joins should be computed on a single node [1, 9, 15]. The problem of distributed and communication-efficient implementation for general join operation has not been addressed in the context of sensor networks, except for our recent work [3] described in the next paragraph.

Chowdhary and Gupta [3] address the problem of communication-efficient distributed implementation of the join operation in the context of sensor networks. The paper presents a provably optimal algorithm for join operation that incurs provably minimum communication cost under reasonable assumptions, and a suboptimal heuristic that performs empirically close to optimal. However, they consider the general join operation that requires matching each tuple of one operand with each tuple of the other operand. In contrast, we consider implementation of range-join operations in sensor networks, for which we develop more efficient algorithms by using hashing and indexing techniques.

3 Implementation of Range-Join in Sensor Networks

In this section, we develop various algorithms for communication-efficient implementation of range-joins in sensor networks. As described in the previous section, we consider a join operation, initiated by a query source node Q, involving two data streams R and S being generated by two geographic regions $\mathcal{R}$ and $\mathcal{S}$ in the network. We first start with describing our general approach of implementing a range-join operation in sensor networks.

General Approach. Traditional database join algorithms such as nested-loop join or merge-join are unsuitable for direct implementation in sensor networks because they are "blocking" and sensor nodes have limited memory resources. To perform the join operation in a non-blocking manner, we determine the sliding windows W_r and W_s of the data streams R and S respectively and store them at some appropriately chosen regions in the network. We use the generation time of tuples to determine their membership in sliding windows. The size, shape, and location of the regions storing the windows depends on the memory capacity of each node, maximum size of each window, and the location of the regions $\mathcal{R}$ and $\mathcal{S}$ that are generating the respective data streams.

After the sliding windows W_r and W_s have been stored in the network, we perform the following high-level operations whenever a tuple r of table R (and vice-versa for a tuple of S)[1] arrives.

1. Find tuples of the window W_s that match with the new tuple r.
2. Join the matching pairs of tuples, and route the resulting tuples to the query source Q.
3. Insert the tuple r in the region storing W_r.

It is easy to see that performing the above operations for every arriving tuple of data streams R and S will correctly compute the join of R and S. The various approaches proposed in this paper differ in the manner in how and where the sliding windows are stored and how the above three operations are performed.

Naive Algorithm. The Naive algorithm uses the simplest way of storing the sliding windows. In particular, the Naive approach stores the windows W_r and W_s around the center of the regions $\mathcal{R}$ and $\mathcal{S}$ that are generating the respective data streams. Let the regions storing the windows W_r and W_s be $\mathcal{W}_r$ and $\mathcal{W}_s$ respectively. Now, when a new tuple r of the data stream R arrives, we need to broadcast r in the $\mathcal{W}_s$ region to find matching tuples of W_s.

Centroid Algorithm In the Centroid Algorithm, both the windows W_r and W_s are stored within a region around some point C in the network region. When a new tuple r of the data stream table R arrives, it is routed to the point C, and then, broadcast within the appropriate region around C to find matching tuples from the window W_s. The resulting joined tuples are routed to the query source Q. Finally, the tuple r is stored at a nearby node around C with available space.

[1] Throughout this article, we discuss the tasks performed on arrival of an R tuple. The same discussion applies to arrival of S tuples.

The total communication cost incurred in the above described approach consists of the cost of routing r to C, broadcasting r in the region around C, and routing the resulting joined tuples to the query source Q. It is easy to show that the total communication cost is minimized when C is the weighted centroid of $\triangle \mathcal{RS}Q$ formed by the centers of the regions $\mathcal{R}$ and $\mathcal{S}$, and the query source Q, where the centroid is weighted by the sizes of R, S, and $R \bowtie S$ (at Q) respectively.

3.1 Hash-Join Algorithm

The Naive and Centroid algorithms involve a broadcast of every newly arriving tuple in an appropriate region. In this subsection, we present a distributed Hash-Join Algorithm that exploits the fact that the join-predicate is a range predicate.

Basic Idea. The main idea of our distributed Hash-join algorithm is to "bucketize" (partition and store) each arriving tuple into certain buckets based on its join-attribute value. In particular, for each arriving tuple r or R, we hash its join-attribute value onto geographic coordinates and insert the tuple r at a node closest to the hashed geographic coordinates (as in GHT [11, 12]). To minimize communication cost, we wish to execute the "find W_s tuples" and "insert r in W_r" operations in the same region. Thus, use the same hash function for both operand data streams, and hence, the sliding windows W_r and W_s get stored in the same common region. For each new tuple r, the node closest to the hashed geographic coordinates is delegated with the responsibility of storing r, and performing the join with the stored sliding window W_s.

Hash-Join Algorithm Steps. We now outline the sequence of steps undertaken for each arriving tuple. For simplicity of presentation, we right now restrict ourselves to equi-join operations (and assume that there is sufficient available memory at each node I to store all hashed tuples (i.e., there is no overflow). We relax both the assumptions in later paragraphs. Now, for each arriving tuple r of a data stream R, the following operations are performed.

1. Hash the join-attribute value of the tuple r to geographic coordinates.
2. Route r to the node I that is closest to the hashed geographic coordinates. We use the standard location-aided routing mechanism such as GPSR [5] to route to I.
3. Insert r at the node I.
4. Join of r with matching tuples of W_s can be computed at I, since the matching tuples (having the same join-attribute value as that of r) of W_s must be available at I.
5. Route the resulting join tuples to the query source Q.

We note here that the above described distributed Hash-join approach is similar to the symmetric hash-join [13] algorithm proposed for evaluation of equi-joins in streaming database systems. We omit the proof of the following theorem for lack of space.

Theorem 1 *Let C be the weighted centroid of the centers of the regions $\mathcal{R}$ and $\mathcal{S}$, and Q, where the weights correspond to the sizes of the tables R, S, and $R \bowtie S$ respectively. Consider the hash-function that hashes the join-attribute values uniformly around C.*

The Hash-join algorithm using the above hash-function incurs optimal communication cost for implementation of an equi-join operation if each sensor node has sufficient memory to store all the hashed tuples. □

Hash-Join for Range-Joins. In order to extend the Hash-join algorithm to perform range-join operations, we need to only modify the fourth step of finding the matching tuples of W_s. More specifically, in case of a range-join operation, the tuples of W_s that may match with r need not have the same attribute value as that of r, but would be within a range of r's join-attribute value. If we use a *locality preserving* hash function, i.e., a hash function that maps close attribute values to close geographic coordinates, then the fourth step of our distributed hash-join algorithm can be modified to the following.

- The tuples of W_s that match with r must be available at nearby nodes *around* I. Thus, the tuple r should be broadcast in a region around I to find the matching tuples. The size of the broadcast region depends on the range of the join-predicate and the locality of the hash function, assuming there are no overflows.

Hash function for Range-Joins. To enable communication-efficient processing of range-joins, we use a hash function that maps a join-attribute value to radii coordinates (d, θ) with respect to the centroid C. In particular, we use the lower-order bits of the join-attribute value to obtain d, and the higher-order bits to obtain θ. Thus, a small range of join-attribute values would get mapped from (d_1, θ) to (d_2, θ) with respect to the centroid C for some values of d_1, d_2, and θ. Then, the set of tuples of W_s for a given range of of join-attribute values will lie on a radial straight line away from the centroid (see Figure 1 (a)), which can be efficiently targeted using location-aided routing such as GPSR [5].

Managing Overflows. Due to memory limitations, a sensor node I may not be able to store all the W_r and W_s tuples hashed to it. There are many ways to solve such an overflow problem. Our technique to handle overflows at individual nodes is to store the overflow tuples in nodes close (as close as possible) to the originally hashed node I. The node I keeps track of the maximum distance of the node that stores the overflow tuples, using overflow radii variables O_r^I and O_s^I for R and S data streams respectively. The overflow radius variables are kept updated.

The third step of inserting the tuple r in W_r and the fourth step of finding the matching tuples in W_s of the Hash-join algorithm need to be modified to incorporate our overflow technique. For the third step, if the node I doesn't

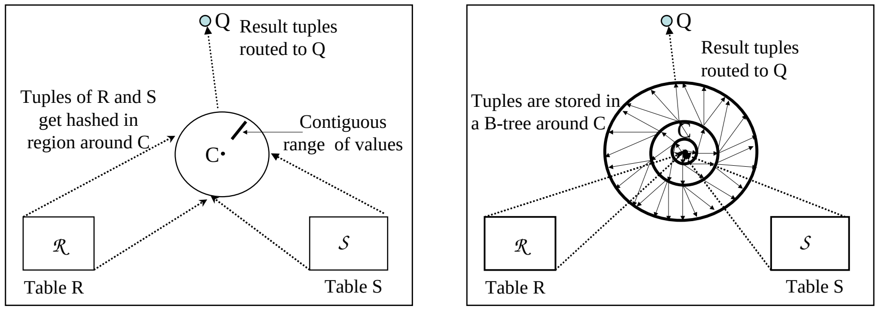

Fig. 1. (a) Hash-join algorithm, and (b) Index-join algorithm

have available memory to store the tuple r, it needs to find the closest node with available memory around it and possibly, update the O_r^I value. For the fourth step, to find matching tuples in W_s, the newly arrived tuple is broadcast in a region of radius O_s^I around I. In practice, the extent of overflow reduces the efficiency of the Hash-join algorithm.

To handle node failures and mobility, we can replicate tuples of a node I at nearby nodes.

3.2 Index-Join Algorithm

In this subsection, we propose an algorithm based on a distributed index data structure to achieve efficient searching of matching tuples for every newly arrived tuple. Essentially, the proposed Index-join algorithm uses a distributed index structure embedded within the sensor network to efficiently route the newly arrived tuple to the sensor nodes storing the matching tuples. In particular, we choose to build the classical B-tree index structure in a distributed manner in the sensor network. To avoid the cost of routing to two different regions, we use a single index structure to store both W_r and W_s windows.

B-Tree in Sensor Networks. To build a distributed B-Tree index structure in a sensor network, we need to first determine the location of the B-tree root and number of children/keys at each node (which in turn determines the height of the tree). Using similar arguments as in Theorem 1, we can show that to optimize the overall communication cost, the root of the B-tree index structure should be located at the weighted centroid C of $\triangle\mathcal{RS}Q$. The number of children (degree) at each node is determined by the memory available at each node for join processing and the number of communication-neighbors of a node in the network. Once the degree of the B-tree has been determined, we can determine the join-attribute key values to be used at each node in the B-tree starting from the root. At each node in the B-tree, the children nodes are distributed at uniform angles around the parent node. Due to limitations in the number of direct communication neighbors available, a child may not necessarily be a direct communication neighbor of its parent. In fact, the communication distance of a child from its parent may increase with the increase in the node's depth from the root.

To start building the index, the chosen root node determines its children, sets its child-pointers to its children, and sends a message to the chosen children with information about the range of join-attribute values each child is responsible for. Note that in traditional database systems, B-tree nodes use memory addresses as pointers to point to their children. However, in sensor networks, we can use geographic coordinates as pointers and use location-aided routing mechanism to reach children that are multiple hops away. The above process of creating more B-tree levels terminates when the remaining data range at each sensor node is small enough that the corresponding set of tuples of W_r and W_s can be stored at a single node. Finally, we need to set sibling pointers at the leaves, which can be done easily. To alleviate the problem of maintenance of the B-tree structure in response of insertions and deletions, we keep additional empty space in each sensor node to accommodate future insertions and do not reclaim space of expired/deleted tuples (since the overall rate of insertions is same as the overall rate of deletions).

Index-Join Algorithm. For every arriving tuple r of the data stream R, we essentially search for matching tuples in W_s using the constructed B-tree index structure, and then insert the tuple r in the index structure.

More specifically, we search for tuples in W_s with join-attribute value a, which is the lowest join-attribute value that could possible match with the join-attribute value of the tuple r. The root node finds the range in which the value a lies, and transmits the tuple to the geographic coordinates corresponding to the appropriate child. Eventually, a leaf node is reached and the sibling pointers are followed to access all the nodes storing tuples of W_s having join-attribute values from a to the maximum join-attribute value that could possibly match with the join-attribute value of r. The resulting joined tuples are finally routed to the query source.

Insertion of the tuple r happens similarly. In particular, we search for the leaf node that stores tuples of W_r with join-attribute value equal to that of r, and try to insert the tuple r at that node. Typically, the node should have enough space to store the new tuple because of the expiry of older tuples and the additional space available to accommodate insertions. In case of inavailability of empty space, we use the standard technique of insertions into B-trees. To make the distributed B-tree structure more load balanced, we replicate the higher-level nodes (ones closer to the root) into multiple nodes in a region around them.

4 Performance Evaluation

In this section, we present our simulation results which compare the performance of various range-join algorithms viz., Naive, Centroid, Hash-join, and Index-join algorithms, proposed in our article. Since incurred communication cost is

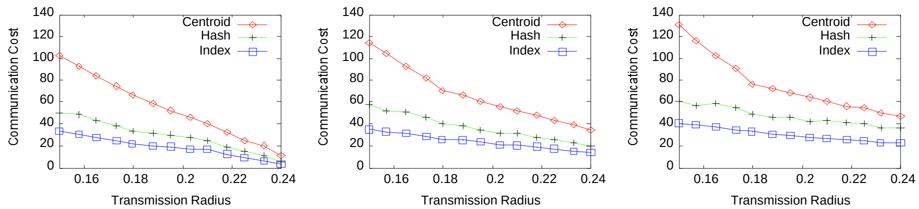

Fig. 2. Varying transmission radius for three different predicate ranges (10, 30, and 50).

the dominant consumer of limited battery power in the sensor nodes and the computation performed by all algorithms is minimal, we present only the total communication cost (in number of hops) incurred by various algorithms. Below, we present a discussion on our simulation results.

Experiment Setup. In our simulations, we generate a sensor network by randomly placing 10,000 nodes in an area of 10×10 units. Each sensor has a uniform transmission radius and two sensors can communicate with each other if they are located within each other's transmission radius. Varying the number of sensors is equivalent to varying the transmission radius, and hence, we fix the number of sensors and measure performance of our algorithms for different transmission radii. Each sensor node stores tuples in a local table of fixed size (5 tuples/node) occupying 300 bytes of memory. For the distributed Index-join algorithm, we use the same memory to also store the index structure entries, so as to be fair across various algorithms in terms of memory usage at individual nodes. Data tuples are generated at a uniform rate of 600 tuples/second by sensor nodes in the regions $\mathcal{R}$ and $\mathcal{S}$, and the (default) sliding window size consists of tuples that are at most 0.5 seconds old resulting in a sliding window size of about 300 tuples for each data stream. We perform simulations demonstrating the effect of varying various parameters such as transmission range, range of the join-predicate, size and shape of $\triangle\mathcal{RS}Q$, and the size of the sliding window.

Varying Transmission Radius for Different Predicate Ranges. In this set of experiments, we fix the locations of the regions $\mathcal{R}$ and $\mathcal{S}$ and the query source Q, and analyze the effect of increasing transmission radius on the total communication cost incurred for different values of the predicate range. The regions $\mathcal{R}$ and $\mathcal{S}$ are centered around the coordinates (1,1) and (9,1) which are the far-left and far-right corners at the bottom of the network, while the query source Q is located at (5,9) towards the top of the network. We vary the transmission radius from 0.15 to 0.24. Lower transmission radii left the sensor network disconnected, while higher transmission radius resulting in very low communication cost. We chose three different ranges of the join-predicate, viz., 10, 30, and 50. Note that range of the join-predicate signifies join-selectivity factor, and hence, determines the size of the join result.

The simulation results are shown in Figure 2. In all the figures of this section, we have not shown the plot for Naive approach, since it performed much worse

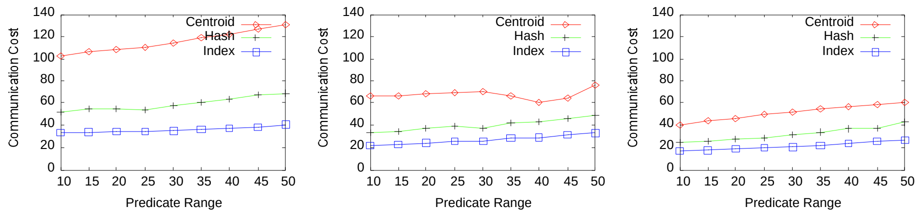

Fig. 3. Varying predicate range for three different transmission ranges viz., 0.15, 0.18, and 0.21.

(incurred twice the communication cost incurred by Centroid) than all other approaches. In Figure 2, we can see that the Hash-join and Index-join algorithms significantly outperform the Centroid approach in all three graphs. Also, the Index-join consistently outperforms the Hash-join algorithm. Note that the better performance of Index-join with respect to Hash-join does not contradict Theorem 1 due to the underlying assumptions made therein. With the increase in the transmission radius, the reduction in the number of hops leads to decrease in the overall communication cost incurred. All the three predicate ranges depict the above behavior, with the higher predicate ranges resulting in higher communication cost.

Varying Predicate Range for Different Transmission Radii. In this set of experiments, we fix the locations of the regions $\mathcal{R}$, $\mathcal{S}$, and Q as before, and analyze the effect of increasing the join-predicate range for different values of transmission radius. We vary the join-predicate range from 10 to 50, for three different transmission radii viz., 0.15, 0.18, and 0.21. The simulation results are shown in Figure 3. Here also, we observe the similar trend as in the first set of experiments, i.e., Index-join and Hash-join algorithms significantly outperform the Centroid approach, Index-join slightly outperforms the Hash-join, and increase in the transmission radius or predicate ranges causes the communication cost to decrease or increase respectively.

Varying $\triangle\mathcal{RS}Q$ for Different Predicate Ranges. In this set of experiments, we study the effect of different shapes and sizes of $\triangle\mathcal{RS}Q$ on the total communication cost, for three different predicate ranges (10, 30, and 50). Here, we fix the transmission radius to be 0.18. To vary the size and shape of the $\triangle\mathcal{RS}Q$, we fix the centers of the regions $\mathcal{R}$ and $\mathcal{S}$, and change the position of the query source Q. We plot the graphs in Figure 4, where on the x-axis we represent the various instances of $\triangle\mathcal{RS}Q$ in the order of the area of the triangle. Again, we see that the Hash-join and Index-join algorithms perform significantly better than the Centroid, with Index-join consistently performing much better than the Hash-join algorithm. We note that increase in the area of the triangle for a fixed predicate range causes increase in the total communication cost incurred, since increase in the area of the triangle results in increase in the distance to the centroid.

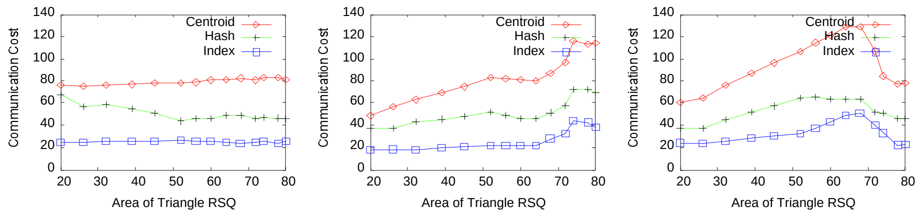

Fig. 4. Various $\triangle \mathcal{RSQ}$ for three different predicate ranges viz., 10, 30, and 50. Here, the transmission radius is 0.18.

5 Conclusion

In this article, we have proposed techniques for communication-efficient implementation of range-joins in sensor networks. We designed various approaches viz., Naive, Centroid, Hash-join, and Index-join, and evaluate their relative performance in random sensor networks. Our simulations indicate that the Hash-join and Index-join approaches perform much better than the other two simple approaches. Our designed algorithms could be incorporated in the sensor network query engines such as TinyDB. Some of the promising future directions include generalizing our technique for join for more than two tables, determining efficient join ordering, approximate evaluation of joins, and multiple query optimization involving join queries.

References

1. B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proceedings of the International Workshop on Information Processing in Sensor Networks (IPSN)*, 2003.
2. P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceeding of the International Conference on Mobile Data Management*, 2001.
3. V. Chowdhary and H. Gupta. Communication-efficient implementation of join in sensor networks. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2005.
4. R. Govindan, J. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker. The sensor network as a database. Technical report, University of Southern California, Computer Science Department, 2002.
5. B. Karp and H. Kung. Gpsr: greedy perimeter stateless routing for wireless networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*, 2000.
6. S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
7. S. Madden and J. M. Hellerstein. Distributing queries over low-power wireless sensor networks. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 622–622, 2002.
8. S. Madden, R. Szewczyk, M. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.
9. S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 491–502, 2003.
10. S. R. Madden, J. M. Hellerstein, and W. Hong. TinyDB: In-network query processing in tinyos. http://telegraph.cs.berkeley.edu/tinydb, Sept. 2003.
11. S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensornets with GHT, a geographic hash table. *Mobile Networks and Applications*, 8(4):427–442, 2003.
12. S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage. In *Proceedings of ACM Intl. Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.
13. A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.
14. Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. In *SIGMOD Record*, 2002.
15. Y. Yao and J. Gehrke. Query processing for sensor networks. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2003.