Join of Multiple Data Streams in Sensor Networks

Xianjin Zhu Himanshu Gupta Bin Tang

Department of Computer Science, Stony Brook University. {xjzhu, hgupta, bintang}@cs.sunysb.edu

***Abstract*— Sensor networks are multi-hop wireless networks of resource-constrained sensor nodes used to realize high-level collaborative sensing tasks. To query or access data generated by the sensor nodes, the sensor network can be viewed as a distributed database. In this article, we develop algorithms for communication-efficient implementation of join of multiple (two or more) data streams in a sensor network. The distributed implementation of join in sensor networks is particularly challenging due to unique characteristics of the sensor networks such as limited memory and battery energy on individual nodes, arbitrary and dynamic network topology, multihop communication, and unreliable infrastructure. One of our proposed approaches, viz., the Perpendicular Approach (PA), is load-balanced, and in fact, incurs near-optimal communication cost for the special case of binary joins in grid networks under the assumption of uniform generation of tuples across the network. We compare the performance of our designed approaches through extensive simulations on the *ns2* simulator, and show that PA results in substantially prolonging the network lifetime compared to other approaches, especially for joins involving spatial constraints.**

I. Introduction

Sensor networks are multihop wireless networks formed by a large number of resource-constrained sensor nodes. Each sensor node typically generates a stream of data items that are readings obtained from the sensing devices on the node. This motivates modeling the data in a sensor network as relational data streams, and visualizing sensor networks as distributed databases systems [5, 18]. More recently, recursive deductive approach has been suggested as a framework for programming sensor networks [9]. Like a database, the sensor network can be queried, and efficient in-network (distributed) implementation of database queries is of great importance. Join is an important database operator, and as shown in our concurrent work [16], can form a basis of a deductive query engine for sensor networks. In particular, join operator can be used to represent complex events in sensor networks [1, 29]. Thus, efficient implementation of join in sensor networks is of great significance; the challenge comes from limited network resources.

Motivated by the above, we develop efficient distributed implementations for join of multiple data streams in sensor networks. Since each sensor node has limited battery energy and message communication is the main consumer of energy, distributed implementation of join must minimize the communication cost. In particular, we are interested in in-network implementation strategies since routing all sensor data to a central server would incur prohibitive communication costs. In addition, load-balanced implementation strategies are highly desirable, because unbalanced strategies are likely to result in a much shorter network lifetime. Design of communication-efficient and load-balanced in-network implementations of join in sensor networks is particularly challenging due to limited memory available at each node and arbitrary network topologies.

The main contribution of our work is the design of various distributed implementations for join in sensor networks. In particular, we propose the Perpendicular Approach (PA) which is communication-efficient and load-balanced, and in fact, incurs near-optimal (within a constant factor) communication cost for binary joins in grid networks under the assumption of uniform generation of tuples across the network. PA works by using appropriately defined horizontal and vertical paths for tuple storage and join-computation respectively. The approach is able to efficiently incorporate joins with spatial constraints, and can be generalized to sensor networks without location information. To the best of our knowledge, our is the first work to address distributed implementation of multi-table join in sensor networks with memory constraints. We analyze the communication cost of our approaches, and compare their performance through extensive simulations on *ns2* simulator. We observe that use of PA results in a substantially prolonged network lifetime compared to other approaches. The performance gap is much larger for the more realistic scenario of joins involving spatial constraints.

Our proposed approach for in-network implementation of join generalizes to evaluation of recursive deductive rules. In addition, our approach can form a basis for in-network evaluation of general deductive programs [16]. Facilitating development of a full-fledged deductive query engine for sensor networks is a significant aspect of our work.

Paper Organization. We start with describing and motivating the addressed problem in Section II, and discuss related work. In the following sections, we present our proposed approaches. We analyze the communication cost of our approaches and address the join ordering problem in Section V, and present simulation results in Section VI.

II. Problem Description, Motivation, and Related Work

We start with presenting an overview of sensor network databases.

Sensor Network Databases. A sensor network consists of a large number of sensor nodes distributed (typically, randomly) in a geographical region. Each sensor node is equipped with sensing devices, a short-range radio, and a limited battery. Two sensor nodes can either communicate with each other directly (if within each other's *transmission radius*) or indirectly using intermediate nodes. The data generated in a sensor network is simply the readings of the sensing devices on the nodes, and can be modeled as relational data streams [3, 33]. Thus, researchers have proposed visualizing sensor network as a distributed database system [5, 18, 21] of data streams. As mentioned before, the sensor network data corresponding to readings of sensing devices can be modeled as relational data streams. In addition, there may be other data streams in the network corresponding to derived views (such as detected events). Each of the data streams may

be generated by an arbitrary set of nodes (perhaps, the entire network), and a node may generate tuples for multiple streams. The node that generates a particular tuple is referred as its *source node*. Due to limited memory resources, we store only a finite set of tuples (typically, most recent) called the *sliding window* [2, 13] for each data stream, and constrain the join operation to the join of sliding windows of operand streams.

Problem Description. Given data streams $R_1, R_2, \ldots, R_n$ (not necessarily distinct) in a sensor network, we wish to compute $R_1 \bowtie R_2 \bowtie \ldots R_n$ in a communication-efficient and load-balanced manner. We do not make any assumptions about the join conditions. However, the above join operation is taken over the sliding windows of the data streams; i.e., we implicitly assume that two tuples match only if they were generated within a certain interval (size of the time-based sliding windows [2]). Moreover, since the sensor data is highly correlated in the spatial domain [12, 23], we give special consideration to spatial joins (formally defined below) and modify our techniques to efficiently implement them. The join-query result is output as a data stream across the network, and can then be hashed across the network to facilitate evaluation of higher-level queries.

Definition 1: (Spatial Join) A join between two data streams R_i and R_j is said to be a *spatial join* of range s if the join condition is a *conjunction* of ($|R_i.nodeLocation - R_j.nodeLocation| \leq s$) and other arbitrary predicates. Here, *nodeLocation* is the attribute for the location of the tuple's source node, and $|x - y|$ is the distance between x and y. □

Performance Criteria. Our main performance criteria of a join implementation is the resulting *network lifetime*. In general, network lifetime is defined as the time after which the network is rendered "useless" (ineffective or inoperable) or disconnected, due to failure of enough nodes. However, the precise definition of the network lifetime depends on the specific objective of an application. In either case, the network lifetime is prolonged by conserving overall battery energy and uniform depletion of battery resources across the network. The former is achieved by minimizing communication cost and the latter by a load-balanced implementation. Thus, we focus on design of *communication-efficient* **and load-balanced** implementations. In our simulations, we define network lifetime in terms of the approximation ratio of the obtained join result (see Section VI).

Motivation. One of the strong motivations for distributed implementation of join in sensor networks is that the join operation can form a basis of a distributed database query engine for sensor networks. Recently, deductive approach has been suggested [9] as a vehicle for programming sensor networks; e.g., [9, 16] show that typical sensor network applications such as shortest path tree, vehicle trajectories, localization, etc. can be expressed as simple deductive programs. In our concurrent work [16], we show how our techniques developed here can form a basis for distributed evaluation of deductive rules.

Another specific motivation for join implementation is event detection, one of the most prominent applications of sensor networks. An *event* indicates a point in time of interest based on certain conditions over the sensor data. In certain cases, events may simply depend on the local value of a sensor reading. Higher-level events or complex events may be specified using composition operators over the primitive events. In particular, the complex events may be represented as a join, involving spatial and temporal constraints, as illustrated below.

Motivating Example 1. Consider a sensor network deployed in an underground mine to detect explosions. Let us assume that the event of an explosion is characterized by interaction between three phenomena/events viz., sound, light, and temperature, and each phenomenon is detected by respective sensors. A temperature event is said to occur when the temperature sensed at any sensor node reaches (or increases by) a certain threshold. Light and sound events are similarly defined. Each of these events is detected locally, and stored in the respective tables along with the locally computed *duration* of the event.

The *explosion event* may be defined to occur when the following conditions are satisfied [29]. (i) The light, sound, and temperature events occur within 10 meters of each other, (ii) the ratio of the durations of sound and light events is at least c (some constant depending on the speeds of sound and light), and (iii) the duration of the temperature event is at least 60 seconds. The query that can be run in the network to detect the above explosion event is as follows.

SELECT *, event as "EXPLOSION"
FROM Sound, Light, Temperature
WHERE |Sound.location-Light.location| < 10
AND |Light.location-Temperature.location| < 10
AND |Sound.location-Temperature.location| < 10
AND Sound.duration > 60
AND Sound.duration/Light.duration $> c$
AND Temperature.duration > 60

The above query may result in an `Explosion` event stream being generated in the network. Note that above every pair of streams has a spatial join of range 10, and the temporal correlation is implicit in the maintenance of sliding windows.

Motivating Example 2. Consider a sensor network deployed for tracking moving vehicles. Each sensor has some means (possibly, vibration or magnet sensors) of detecting presence of a vehicle in the proximity. Consider the event: A vehicle surrounded (from all four directions) by four other vehicles [22]. If detection of a vehicle by a node results in generation of a corresponding record in a global table T, then detection of the above event requires a 5-way self-join of the table T using an appropriately defined *surrounded* predicate over five `nodeLocation` arguments.

Motivating Example 3. Consider a more complex event defined over the `Explosion` event stream of Example 1: A vehicle surrounded (from all four directions) by four explosion events within a certain time window. Here, the location of an explosion event can be defined as the centroid of the sound, temperature, and light event locations. This example illustrates the use of a derived view stream in defining a more complex event query.

Related Work. The vision of sensor network as a database has been proposed by many works [5, 18], and simple query engines such as TinyDB [35] have been built for sensor networks. A simple implementation of an SQL query engine for sensor networks involving shipping all sensor nodes' data to an external server is proposed in [33]. However, such an implementation would incur high communication costs and congestion-related bottlenecks. Thus, in-network implementation of database queries is considered fundamental to conserving energy in sensor networks [25, 51]. Prior research on in-network implementation of database queries has mostly addressed only simple cases such as simple aggregations [34, 52] or selections [35] over single tables [36], local joins [4, 35, 52], etc. In particular, [4, 35, 52] assume that join is computed on a single node. Recently, Abadi et al. [1] consider in-network implementation of join of two tables,

wherein one of the tables is static and small enough to reside in any node's memory. In their approach, they broadcast the small table to all network nodes. To the best of our knowledge, the problem of distributed (non-local) in-network implementation of join in sensor networks has been addressed only in our prior work [8, 38]. Both our prior works restricted their focus to in-network implementation of join ([38] considers range joins) of two tables with a fixed query source. In this article, we focus on designing communication-efficient *and* load-balanced in-network implementation of join of two or more data streams.

There has also been a lot of recent work done on evaluating window joins over data streams, in the context of data stream processing systems [6, 14, 17, 37]. In particular, [13, 27] investigate algorithms for window joins over pairs of data streams, [10, 46] considers the problem of approximating the window joins, and [20, 24] designs algorithms for joining multiple streams constrained by sliding windows. Each of the above works assumes a centralized system. Recents works [45, 48] (which consider operator placement problem) in the context of distributed stream processing systems [15] have worked with the assumption that each operator (including join) is executed locally and fully on a single node. Moreover, minimizing communication cost has not been the focus of research in distributed stream systems.

Traditional query processing techniques from distributed database systems are not directly applicable to sensor networks due to the unique characteristics (severe resource limitations, multihop communication cost model, and larger number of nodes) of sensor networks. Moreover, most of the research done [31, 41, 44, 47] on efficient computation of join in distributed databases has been restricted to equi-joins, join of two tables, minimizing computation time, static relational tables, and/or complete or regular topology.

The idea of using perpendicular sets of nodes (as in the Perpendicular Approach of our work) has been used previously in other contexts such as transaction processing [7], information retrieval in complete networks [49] and uniformly dense networks [42]. In a closely resembling recent work, [30] uses a slightly different concept (of combs and needles) for information retrieval in random-grid networks. However, ***none of the above approaches is easily extendible to our problem of multi-table joins in arbitrary topology networks with memory constraints.***

III. **Simple Approaches**

In this section, we present a few simple approaches, viz., Centralized, Naive Broadcast, and Centroid (CA), for distributed implementation of $R_1 \bowtie R_2 \bowtie \ldots R_n$ where each R_i is a data stream in the network. Another simple approach (called *Local Storage*) is described in Section VI. Note that hash-based approaches [38] are limited to joins with range predicates, and not easy to extend to multi-table joins.

Centralized Approach. One way to compute a join of data streams is the centralized approach, wherein each node sends the generated tuples to a central server (or the *query* source). Such a scheme is unable to give special consideration to spatial joins, and in general, schemes without any in-network processing may incur prohibitive communication costs [33, 34]. Moreover, in homogeneous networks, such an approach is viable only in a "star" network wherein each node is connected directly to the server. In a non-star network, the set of nodes that are directly connected to the server will fail sooner than others (since all tuples in the network will flow through them) – at which point, the central server would be disconnected from the network. Finally, the query source needs to have enough query resources to compute the entire join. The remaining approaches discussed in our article are in-network implementations. In particular, the Centroid Approach is essentially an optimized in-network version of the centralized approach.

Naive Broadcast Approach. The simplest way to implement join is to broadcast each generated tuple to the entire network, and store all the sliding windows at each node. Then, the join can be computed locally at any network node. In case of spatial joins, a tuple of R_i needs to be broadcast only within a region of radius $\max_j s_{ij}$, where s_{ij} is the range of the spatial join between R_i and R_j. Note that a non-spatial join is a spatial join of infinite range. The above Naive Broadcast approach is infeasible in most cases due to severe memory constraints in sensor networks.

Centroid Approach (CA). CA works by first choosing appropriate *storage regions* $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ in the network for storing the sliding windows for the streams $R_1, R_2, \ldots, R_n$ respectively. To facilitate efficient computation of join, the regions $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ (not necessarily different) are all located close to each other. Each generated tuple t of each stream R_i is first routed from its source node to its storage region $\mathcal{C}_i$, where it is stored at some node with available memory (see below for details). Thereafter, the tuple t and the resulting intermediate tuples are routed through the regions $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{i-1}, \mathcal{C}_{i+1}, \ldots, \mathcal{C}_n$ (in some order) to compute the join result.

Storage Regions, Routing, and Storage. Let ρ_i be the rate of generation of tuples of data stream R_i, and let $\mathcal{R}_i$ be the set of nodes (possibly, the entire network) generating the tuples of R_i. Let us define the centroid $\mathcal{C}$ as the location in the network that minimizes the value $\sum_{i=1}^{n} \rho_i \bar{d}(\mathcal{R}_i, \mathcal{C})$, where $\bar{d}(\mathcal{R}_i, \mathcal{C})$ is the average number of hops between a node in $\mathcal{R}_i$ and $\mathcal{C}$. Now, it can be shown (we skip the simple proof here) that choosing the storage regions closely around the centroid $\mathcal{C}$ minimizes the total communication cost (number of hops traversed) of CA for dense networks. In either case, nodes around the storage region would see more traffic, and hence, would fail (due to battery depletion) sooner than other nodes. When sufficient nodes in the storage region fail, a new storage region is selected and all nodes informed.

The location of the storage regions can be either broadcast to the entire network initially or maintained at the node closest to the network center. The above allows each generated tuple to be routed to the required storage regions using location-based routing [28]. In sensor networks without location information, we need to construct and maintain routing paths from each node to the storage regions to route tuples to required storage regions. In either case, when a tuple t of stream R_i reaches the node I closest to the center of $\mathcal{C}_i$, the node I searches for a close-by node in $\mathcal{C}_i$ with available memory. Such a node can be found by broadcasting an appropriate request message to nodes in $\mathcal{C}_i$, gathering responses from nodes that have available memory, and picking the closest node among them. After storage, the tuple t (along with other intermediate results) is routed to other storage regions (in some order) for computation of join.

IV. **Perpendicular Approach (PA)**

In this section, we describe the Perpendicular Approach (PA), which is load-balanced and communication-efficient. In partic-

ular, it provably incurs near-optimal (within a constant factor) communication cost for binary joins in grid networks under uniform-generation assumption. We start by describing PA in grid networks, and then, generalize it to general topologies.

A. PA in Grid Networks

In this subsection, we first describe the PA for join of two data streams, and then, generalize it to multiple data streams. We start with formally defining 2D grid networks.

Definition 2: (2D Grid Network.) A two-dimensional (2D) *grid network* of size $m \times m$ is formed by placing a node of unit transmission radius at each location (p, q) ($1 \leq p \leq m$ and $1 \leq q \leq m$) in a 2D coordinate system. □

Join of Two Streams in Grid Networks. In a 2D grid, every horizontal line (i.e., a line parallel to x-axis) intersects every vertical line (i.e., a line parallel to y-axis). Thus, if each generated tuple is stored at all nodes of some horizontal line, then the set of nodes on any vertical line will collectively contain all sliding windows. In this article, we arbitrarily choose horizontal lines for storage and vertical lines for join-computation; however, their roles can be easily swapped.

Based on the above observation, PA consists of *two phases*, viz., storage and join-computation. Consider a grid network with data streams R_1 and R_2, and a tuple t (of either data stream) generated at coordinates (p, q).

- *Storage Phase*: In the storage phase, the tuple t is stored (replicated) along the q^{th} *horizontal line*, i.e., at all nodes whose y-coordinate is q. This ensures that set of nodes on *each* vertical line collectively contain the entire sliding windows for R_1 and R_2. See Figure 1.
- *Join-computation Phase*: In the join-computation phase, we route t along the p^{th} vertical line to compute the result tuples due to t (i.e., $t \ltimes R_2$ or $t \ltimes R_1$ depending of whether t is in R_1 or R_2). The result tuples are computed by locally joining t with matching tuples of R_1 or R_2 stored at nodes on the p^{th} vertical line.

To maintain time-based sliding windows [2], each stored tuple t is kept in the local memory of node I until $\tau_w + \tau_s + \tau_j$ time after its arrival at I, where τ_w is the interval of the sliding window, and τ_s and τ_j are the upper bounds on the time to complete the storage and join-computation phases respectively. To correctly handle simultaneously generated tuples across the network, we start the join-computation phase for a tuple only after the completion of its storage phase. Thus, we introduce a delay of τ_s between the start of two phases. The correctness of above approach follows from the more general claim in Theorem 2.

Spatial Joins. If $R_1 \ltimes R_2$ is a spatial join of range s, then a tuple t (of R_1 or R_2) generated at (p, q) is stored at only those nodes on the q^{th} horizontal line that are within a range of s from (p, q). Similarly, in the join-computation phase, the tuple t is routed only to nodes within a range of s from (p, q) on the p^{th} vertical line.

Near-Optimality of Communication Cost. We now show that PA incurs near-optimal communication cost (in addition to being perfectly load-balanced) for uniformly generated data streams in a grid network.

Theorem 1: Consider an $m \times n$ grid network with the data streams R_1 and R_2 being similarly and uniformly generated over the entire network. The total communication cost (total number of tuple-hops traversed) incurred by PA to compute the join of R_1 and R_2 is at most eight times the minimum communication cost needed.

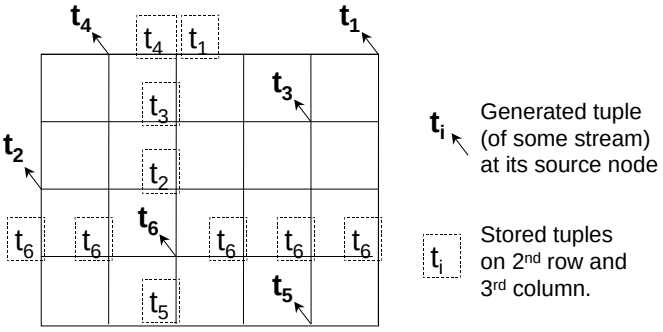


Fig. 1. Storage/Replication of PA in 2D Grid Networks.

Proof. For simplicity, we assume a general (non-spatial) join; however, the proof easily generalizes to spatial joins. Now, it is easy to see that the total communication cost incurred by PA is at most $(m + n)$ units for each generated tuple.

Note that in an $m \times n$ grid network, there are at least $mn/2$ disjoint pairs (I_1, I_2) of nodes such that the distance (in hops) between I_1 and I_2 is $m/2$. The above is true since there are at least $m/2$ such disjoint pairs of nodes on each vertical line (of length m), and there are n disjoint vertical lines. Now, consider a pair of nodes (I_1, I_2) in the above set of disjoint pairs. Each tuple t of R_1 generated by I_1 must join with each tuple t' of R_2 generated by node I_2. Since the distance between I_1 and I_2 is $m/2$, the communication cost incurred in joining t and t' is at least $m/2$. Thus, if each network node generates one tuple each for R_1 and R_2, then the minimum communication cost incurred in computing the join of these $2mn$ generated tuples is at least $(m/2)(m/2)n = nm^2/2$. If the generation of operand tuples is uniform, then the minimum communication cost required is $m/4$ per generated tuple, which is at least one-eighth of that incurred by PA if we assume $m > n$ (which is without loss of generality). ■

Multiple Streams in Grid Networks. Consider a grid network with data streams $R_1, R_2, \ldots, R_n$. PA can be generalized to handle more than two data streams as follows. First, the storage strategy remains the same as before, i.e., each tuple t generated at (p, q) is still stored along the q^{th} horizontal line. However, in the join-computation phase, we need to traverse the vertical line in a more involved manner. Below we describe two schemes, viz., one-pass and multiple-pass, for traversing the vertical line in the join-computation phase. We start with a definition.

Definition 3: (Partial Result.) Let $R_1, \ldots, R_n$ be given data streams and let t be a tuple of R_j. A tuple T is called a *partial result* for t if T is formed by joining t with less than $n - 1$ given data streams (other than R_j). More formally, T is a partial result for t if $T \in (t \ltimes R_{i_1} \ltimes R_{i_2} \ldots R_{i_k})$ where $k < n-1$ and $i_l \neq j$ for any l. The tuple t is also considered a partial result (for the case $k = 0$). If $k = n - 1$, then T is called a *complete result*. □

One-Pass Join Computation Scheme. Consider a tuple t (of some data stream) generated at a node (p, q). In the one-pass scheme, the tuple t is first unicast to one end (i.e., $(p, 0)$), and then,

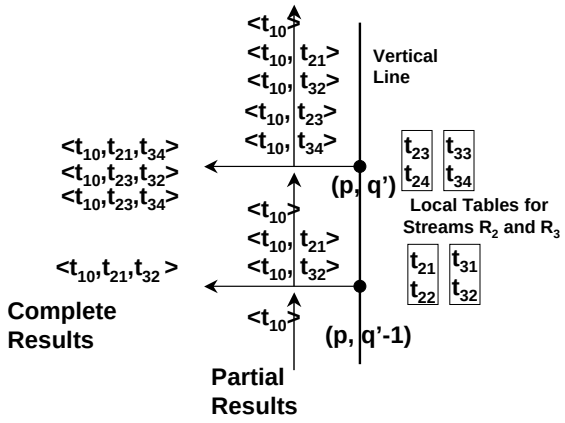


Fig. 2. One-Pass Join Computation. Here, $t_{i*} \in R_i$, and we assume the join conditions to be such that t_{10} matches only with $t_{21}, t_{23}, t_{32}, t_{34}$. Also, there is no join condition between R_2 and R_3.

is propagated through all the nodes on the p^{th} vertical line by routing it to the other end. At each intermediate node (p, q'), certain partial and complete results (as defined above) are created by joining the incoming partial results from $(p, q'-1)$ with the operand tuples stored at (p, q'). The computed partial results along with the incoming partial results are all forwarded to the next node $(p, q'+1)$. See Figure 2. Certain incoming tuples may join with the operand tuples stored at (p, q') to yield complete results, which are then output and not forwarded. The partial results generated at the last node (other end) are discarded.

Theorem 2: Given data streams $R_1, R_2, \ldots, R_n$ in a sensor network, the Perpendicular Approach (with one-pass join-computation scheme) correctly computes $R_1 \bowtie R_2 \bowtie \ldots R_n$, in response to distributed (and possibly, simultaneous) generation of tuples. We assume bounded τ_s and τ_j, the time for completion of storage and join-computation phases respectively, and no message losses.

Proof. By description of the scheme and the definition of complete results, the one-pass scheme outputs only those tuples that belong to the final join result. To show that every tuple of final join result is eventually output, consider an arbitrary tuple T in the final result. Let T be the result of matching of $\{t_1, t_2, \ldots, t_n\}$, a set of n tuples $t_i \in R_i$ one from each data stream. Let t_l (for some $l \leq n$) be the tuple among t_i's whose storage phase was completed the last. Now, we claim that the tuple T must be output during the one-pass join-computation phase of t_l. Let t_l be generated at node (p, q). When the join-computation phase of t_l starts, the storage phase of each t_i $(1 \leq i \leq n)$ has been completed by definition of l and the fact that the join-computation phase of t_l starts after the completion of its storage phase. Thus, during the join-computation phase of t_l, each of the tuples t_i is available (and *not-expired*[1]) at some node on the p^{th} vertical line, and the tuple t_l encounters (in some arbitrary order) each one of these t_i tuples. Thus, the tuple T is eventually output. ∎

Multiple-Pass Scheme. In the multiple-pass scheme, the join-computation phase takes place in a certain order of data streams. Each iteration of the multiple-pass scheme is essentially a one-pass scheme involving join of a data stream with partial results generated in the previous iteration. More formally, let the pre-determined join-ordering of data streams be $R_{i_1}, R_{i_2}, \ldots, R_{i_{n-1}}$ (not including the stream of the new tuple t). In the first iteration, the tuple t is propagated through the vertical line (from one end to another) to join with R_{i_1}. In general, in the k^{th} iteration, the partial results obtained from the previous $(k-1)^{th}$ iteration are propagated through the vertical line to join with R_{i_k}. Thus, the partial results generated in the k^{th} iteration constitute $t \bowtie R_{i_1} \bowtie \ldots R_{i_k}$.

B. General Topology Sensor Networks

We now generalize our PA to general network topologies.

Challenges in General Networks. The main challenge in generalizing PA is to define appropriate notions of horizontal and vertical paths such that each horizontal path intersects each vertical path. Due to network topology being dynamic and limited resources at each node, we do not wish to *construct* and maintain generic horizontal and vertical paths for each network node. Ideally, we would like the horizontal and vertical paths to be the set of nodes encountered during routing to source-node-specific destinations. Moreover, we would like the communication and data replication costs to be near-perfectly balanced across the network. The presence of topological holes and arbitrary network boundaries make the above particularly challenging. Keeping the above challenges in mind, we define the vertical and horizontal paths as below. First, we start with presenting a brief background on location-based routing.

GPSR – Location-Based Routing. In sensor networks, nodes are typically referred to by their geographic locations (instead of IDs), and each node is aware of its location (using GPS or localization techniques [43]). Thus, in this section, we consider sensor networks with location information, and use location-based routing (described below) for routing between nodes/locations. We generalize our techniques to networks without location information in Section IV-C.

In *location-based routing* protocols, the destination is specified by its geographic location. Due to severe memory constraints, the location-based routing protocols in sensor networks are reactive (on-demand), and determine the next hop on the fly. One simple location-based routing protocol is the greedy approach [28] wherein each node forwards the packet to the neighbor closest to the destination. However, greedy approach can get *stuck* at nodes that have no neighbor closer to the destination than itself. In contrast, face-routing [28] protocol routes the packet through a sequence of faces (in a planar subgraph of the network) that intersect the line segment connecting the source and the destination. For efficiency, face-routing is combined with the greedy approach – yielding the well-known GPSR [28] protocol. We use GPSR as the base routing protocol throughout the article; *however, our techniques generalize to any location-based routing.*

Vertical Paths in General Networks. An intuitive way to define the vertical path for a node at (p, q) could be: set of nodes traversed when a packet is routed using GPSR from (p, q) to $(p, +\infty)$ and from (p, q) to $(p, -\infty)$. However, for such a

[1]Non-expiry of t_i at a node follows from the fact that t_i and t_l matched to form T (and hence, must have been generated within τ_w time of each other), and tuple t_i expires at node I only after $\tau_w + \tau_s + \tau_j$ time of its arrival at I. Here, τ_w is the interval of the time-based sliding window.

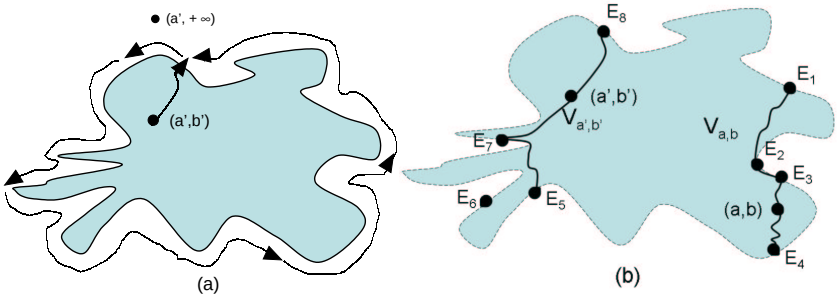


Fig. 3. Defining vertical paths in an general networks. (a) The path taken by GPSR for source node (a', b') and destination $(a', Y_{max}+1)$. Since the destination is outside the network field, the path traverses the entire external boundary. (b) Vertical paths for nodes (a, b) and (a', b'). Here, the markings on the given boundary nodes is as follows: E_1 and E_8 are marked `Highest`, E_4 and E_5 are marked `Lowest`, and the rest are marked `Middle`. So the vertical path $V_{a',b'}$ stops at nodes E_8 and E_5.

definition, the boundary nodes (external face) would be part of every vertical path, and hence, overloaded. See Figure 3(a). The above problem of overloading of boundary nodes can be alleviated by modifying the GPSR protocol as follows. When routing a packet from (p, q) to $(p, +\infty)$ (for defining vertical paths), the packet stops on reaching a boundary node marked `Highest`. A boundary node v is marked `Highest` if there is no other node higher than v along the same vertical line. On reaching any other boundary node, the packet is forwarded to a neighboring node that is closer to some boundary node marked `Highest`. Routing of a packet from (p, q) to $(p, -\infty)$ can be similarly defined. The above informal description forms the basis of our definition of vertical paths in general sensor networks.

Below, we formalize the above notion of vertical paths. First, we formally define boundary node markings, and then formalize the above described behavior of GPSR.

Definition 4: (Markings on Boundary Nodes.) A node (p, q) on the boundary is marked `Highest (Lowest)` if there is no node I such that I has an edge intersecting the line $x = p$ and has a y-coordinate greater (less) than q. Otherwise, the node (p, q) is marked `Middle`. See Figure 3(b). □

Modifying GPSR for Vertical Paths. For a packet being routed upwards (e.g., towards $(p, +\infty)$) the GPSR protocol is modified as follows. When GPSR reaches a boundary node I: If I is marked `Highest`, then GPSR stops and the vertical path is completed; else GPSR is directed to (i) a non-boundary neighbor of I with a higher y-coordinate than I, or (ii) (if no such non-boundary neighbor exists) the left or right boundary neighbor of I whichever is on the shorter path to some node marked `Highest`. For instance, in Figure 3 (b), at E_2 GPSR is directed to a non-boundary neighbor, while at E_3 GPSR is directed to the left boundary neighbor. GPSR is similarly modified for packets being routed downwards. In both cases, the behavior of GPSR on non-boundary nodes is not modified (i.e., remains the same as the original GPSR).

The markings and information (left or right boundary neighbor for directing GPSR) at boundary nodes can be easily computed by routing a packet along the boundary. Moreover, as suggested by Theorem 4, we need to compute these markings and information only once (for static sensor networks).

Definition 5: (Vertical Path.) Let Y_{max} and Y_{min} be the largest and smallest y-coordinate values in the entire network. Vertical path $V_{p,q}$ for a node at a location (p, q) is defined as the concatenation of the two paths when a packet is routed (i) upwards from (p, q) to $(p, Y_{max} + 1)$, and (ii) downwards from (p, q) to $(p, Y_{min} - 1)$,[2] using the modified GPSR protocol. Essentially, each vertical path connects *some* boundary node marked `Highest` to *some* boundary node marked `Lowest`, if the network is connected. □

The above definition of vertical paths ensures the following: (i) If the network is connected, each vertical path $V_{p,q}$ is a continuous path connecting *some* `Highest` node to *some* `Lowest` node (this ensures intersection with every horizontal path as defined later); (ii) A vertical path includes a few number of nodes and is different for different (p, q) (this ensures efficiency and load-balance); and (iii) A vertical path for a node at location (p, q) does not deviate much from the line $x = p$ (this is due to routing towards $(p, Y_{min} - 1)$ and $(p, Y_{max} + 1)$ instead of (p, ∞) and $(p, -Y_{max} + 1)$); this allows efficient implementation of spatial joins.

Horizontal Paths in General Networks. Defining horizontal paths precisely in the same manner as vertical paths will not ensure intersection of a horizontal path with each vertical path. For instance, see H_{bad} and $V_{a,b}$ in Figure 4. Hence, we define horizontal paths as follows.

Definition 6: (Horizontal Path.) Let I_{left} and I_{right} be the leftmost (i.e., the node with the smallest x-coordinate) and rightmost nodes respectively in the entire network. We denote the *horizontal path* for a node at location (p, q) as $H_{p,q}$ and define it as the concatenation of paths traversed by the GPSR protocol when a packet is routed from (p, q) to I_{right} and from (p, q) to I_{left}. If the network is connected, each horizontal path connects I_{right} to I_{left}. □

The following theorem proves pairwise intersection of such defined vertical and horizontal paths.

Theorem 3: Consider two arbitrary nodes (p, q) and (r, s) in a connected sensor network. The paths $V_{p,q}$ and $H_{r,s}$ intersect, i.e, the paths contain a common node or a pair of edges (one from each path) that cross each other.

Proof. (sketch) By definition, $V_{p,q}$ is a continuous path connecting a pair of nodes $H = (h_x, h_y)$ to $L = (l_x, l_y)$ such that H is marked `Highest` and L is marked `Lowest`. It can be shown that H and L lie on different sides of $H_{r,s}$(or one of them lies *on* $H_{r,s}$), since $H_{r,s}$ is a continuous path from the leftmost node I_{left} to the rightmost node I_{right} in the network. Thus, $V_{p,q}$ intersects $H_{r,s}$. ■

Overall Approach. Using the above notions of horizontal and vertical paths, overall PA works as follows. Each tuple t (of

[2]We use Y_{max} and Y_{min} instead of $+\infty$ and $-\infty$ to reduce the deviation of $V_{p,q}$ from the vertical line $x = p$, to ensures load-balance and efficient implementation of spatial joins (as discussed later).

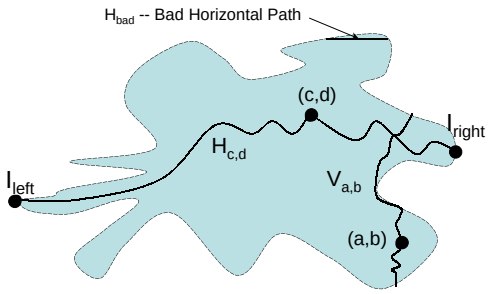


Fig. 4. Horizontal path for node (c, d). Here, I_{left} and I_{right} are the leftmost and rightmost nodes in the network. The path H_{bad} shows that defining horizontal paths in a similar way as vertical paths does not ensure intersection of vertical and horizontal paths.

any data stream) generated at a node (p, q) is first stored on every node of $H_{p,q}$. In the join-computation phase, the tuple t is routed through *and broadcast by* all the nodes on $V_{p,q}$. Due to the wireless multicast advantage, the tuple t is automatically received by at least one node of each edge that crosses $V_{p,q}$.[3] Thus, by Theorem 3, the tuple t is received by at least one node of each horizontal path. As in grid networks, we introduce an appropriate delay between the storage and join-computation phase to handle simultaneous generation of tuples across the network. We can use either one-pass or multiple-pass scheme for the join-computation phase. The correctness of the overall approach follows by Theorem 2. Finally, we note that PA requires minimal memory resources beyond the storage of data stream tuples, and has minimal processing needs beyond local computation of join conditions.

Correctness in Dynamic Topologies. We now discuss immunity of PA to changes in network topology due to node additions and deletions. First, the below theorem shows that node deletions (due to node failures, battery depletion, etc.) do not affect the correctness of PA.

Theorem 4: Consider a connected sensor network with markings on the boundary nodes (as in Definition 4). Let I_{right} and I_{left} be the rightmost and leftmost nodes in the network. If some nodes fail/die without disconnecting the network, then intersection is still guaranteed for each pair of vertical and horizontal path (based on *old* boundary nodes, markings, and I_{right} and I_{left} values).

Proof. (sketch) The proof is based on the following facts. (i) Boundary nodes with `Highest` (or `Lowest`) marking still remain "highest" (or "lowest") even when some nodes fail; (ii) PA resorts to original (unmodified) GPSR at new or unmarked boundary nodes; (iii) Thus, each vertical path still connects some `Highest` node to some `Lowest` node (assuming network remains connected) or contains the entire external boundary; (iv) Each horizontal path still connects I_{right} to I_{left} node (if they haven't failed), or contain the entire (current) external boundary. ∎

In a sensor network, some nodes will deplete battery energy earlier than others. Thus, the above result is quite important. In particular, the above result says that, to ensure correctness, we do not need to *recompute* boundary nodes, their markings, or I_{right} or I_{left} values, as some nodes fail (due to battery depletion or otherwise). However, in general, node additions (due to mobility of nodes or otherwise) may require recomputation of boundary nodes, boundary markings, I_{right} and I_{left} nodes, to ensure overall correctness.

Fault Tolerance; Storage vs. Communication Cost Tradeoff. PA is inherently fault-tolerant to node/link failures, since in an irregular topology a vertical path $V_{p,q}$ is likely to intersect a horizontal path $H_{r,s}$ at multiple nodes/edges. Thus, the join result is likely to contain duplicate tuples – making the approach fault-tolerant.

To reduce data replication (at the expense of more communication cost), we could store a tuple on every k^{th} node on the horizontal path for an appropriately chosen parameter k. However, to ensure correctness, we need to broadcast the tuple t to the ($\lfloor k/2 \rfloor$+1)-hop neighborhood of every node on the vertical path during the join-computation phase. See Figure 5. The above strategy gives us a way to trade replication cost for additional communication cost.

Traffic Congestion. The horizontal paths may result in traffic congestion in the region around the I_{left} and I_{right} nodes.[4] We can solve the above congestion problem by excluding the "ears" (elongated left and/or right sides) of the network. In particular, we periodically determine two values X_{left} and X_{right} such that (i) the number of nodes between X_{right} and X_{left} is large, (ii) the number of nodes having an incident edge that intersects with the line $x = X_{right}$ is large, and (iii) the number of nodes having an incident edge that intersects with the line $x = X_{left}$ is large. Now, we define the paths for a node (p, q) as follows. Lets assume that (p, q) is *not* in the ear; for a node in the discarded ear, we use an arbitrary node not in the ear for definition of paths. The horizontal path $H_{p,q}$ is defined as the concatenation of the following paths: (i) Path traversed when routing from (p, q) to (X_{left}, q) until $x = X_{left}$ is reached, and (ii) Path traversed when routing from (p, q) to (X_{right}, q) until $x = X_{right}$ is reached. Vertical path $V_{p,q}$ is defined as in Definition 5, except that now the lines $x = X_{left}$ and $x = X_{right}$ must be treated as part of the network boundary. However, intersection of paths (Theorem 3) cannot be guaranteed for the above notion of horizontal/vertical paths, especially for sparse networks with internal holes. Nevertheless, in our simulations over random dense networks, we observed that as per the above definitions, each horizontal path still intersected each vertical path.

Incorporating Spatial Joins. For spatial joins, we need to store and propagate each tuple along only parts of the paths. For a data stream R_i, let $s_i = \max_j s_{ij}$ where s_{ij} is the range of the spatial join between R_i and R_j. Note that a non-spatial join is a spatial join of infinite range. Now, let d_x be such that the x-coordinate of any node on any vertical path $V_{p,q}$ is most $(p+d_x)$ and at least $(p - d_x)$. In other words, d_x is the maximum *deviation* of any vertical path from its vertical. Similarly, let d_y be the maximum deviation of any horizontal path from its horizontal line. Then, for spatial joins, the vertical path $V_{p,q}$ used for a tuple of R_i at (p, q) is the concatenation of paths traversed when routing from (p, q) to $(p, q + s_i + d_y)$ and from (p, q) to $(p, q - s_i - d_y)$.

[3] This is because in a unit-disk graph if two edges cross, then at least one of the four nodes is connected to both nodes of the other edge.

[4] It is for this reason that we defined vertical paths differently than horizontal paths. Otherwise, defining vertical paths in a manner similar to horizontal paths does ensure correctness (intersection).

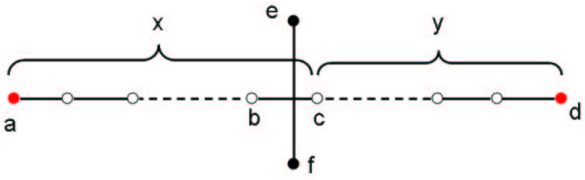


Fig. 5. Reducing replication by storing each tuple on every k^{th} node of the horizontal path. Above, tuples are stored only at the solid nodes (and not at the hollow nodes). Here, either x or y must be at most $\lfloor k/2 \rfloor$, and either e or f must be connected to b or c.

Similarly, the horizontal path $H_{p,q}$ used is the concatenation of paths traversed when routing from (p,q) to $(p+s_i+d_x,q)$ and (p,q) to $(p-s_i-d_x,q)$.

Note that the value d_x (d_y) used for $H_{p,q}$ ($V_{p,q}$) above should only be such that the deviation of a vertical (horizontal) path of any node within a range of s_i from (p,q) is at most d_x (d_y). Thus, for $H_{p,q}$ or $V_{p,q}$, we need to care about path-deviations of only those nodes that are within a range of s_i from (p,q). Thus, the values d_x and d_y depend only on the local network density, and hence, can be gathered periodically from nodes within a range of s_i. Such gathering of deviations can be achieved by each node broadcasting its path-deviations to nodes within a range of s_i.

Incorporating Range Joins. A join between two data streams R_i and R_j is said to be a *range join* of range s if the join condition is a *conjunction* of ($|R_i.A - R_j.A| \leq s$) and other arbitrary predicates, for some attribute A. A range join can be "converted" into a spatial join by mapping the range-attribute value to a geographical location using a locality-sensitive [11] hash function. Let a geographic hash function $h : A \rightarrow \mathcal{R} \times \mathcal{R}$ be called c-sensitive if Euclidean distance between locations $h(a_1)$ and $h(a_2)$ is at most $c|a_1 - a_2|$ for some constant c. Then, we can convert a range join of range s to a spatial join of range cs by mapping tuples to geographic locations based on a c-sensitive geographic hash function. Space-filling curves [39] can be used as c-sensitive geographic hash functions; however, efficient distributed implementation of such hash functions is a challenge and part of our future work.

Consumption of Output Tuples. In our proposed schemes, the output tuples are generated across the network in some arbitrary manner. If the join query was issued by a specific network node (*query source*), then the result tuples are routed to the query source. In our model, the query source is not required to be fixed or predetermined. If the join query was issued as part of an overall query processing scheme, then the result tuples can be hashed and stored across the network [40], to facilitate evaluation of higher-level queries. The above hashing/storage scheme is actually imperative for efficient elimination of duplicate result tuples or efficient execution of selection queries later.

C. Networks Without Location Information

Our PA is built on top of location-based routing, and hence, assumes that each node is aware of its location. However, in certain applications, location information is either not accurate enough or not even available. For such networks, we define perpendicular *regions*, viz., connected k-dominating set (defined below) and k-hop neighborhood for some carefully chosen k, for each node, and use them for storage and join-computation. Since we need to traverse the k-dominating set, we use connected k-dominating sets instead.

Definition 7: (k-Dominating Set; Clusterheads; Connected k-Dominating Set.) Given a graph G, a subset of vertices S is called a *k-dominating set (k-DS)* if every vertex in G is within k hops of some node in S. We refer to each node in the set S as a *clusterhead*. A subset of vertices C is called a *connected k-dominating set (k-CDS)* if C is a k-DS and the subgraph induced by C in G is connected. A k-CDS C can be thought of as composed of a k-DS S of clusterheads and a set $C-S$ of *gateways* used to connect S. □

Constructing Connected k-Dominating Sets (k-CDS). As mentioned above, we use k-CDS and k-hop neighborhoods as the perpendicular regions for sensor networks without location information. For construction of k-CDS, we use the distributed algorithm of [50] augmented with reliable messaging (using messages acknowledgments and retransmissions). To achieve load-balance, we construct multiple such k-CDS so that different nodes can use different k-CDS; we construct such multiple k-CDS using different random IDs for each node. To use the above, each node I, we associate an arbitrary (preferably, the closest) k-CDS, and maintain a path connecting I to the associated k-CDS.

Tuple Storage and Join-Computation Phases. Note that each k-DS intersects (i.e., has a common node) with the k-hop neighborhood of any node. Thus, the k-DS and k-hop neighborhoods can be looked upon as "perpendicular" to each other. We connected the k-DS to allow easy propagation of tuples (for storage or join computation) over the clusterheads. We arbitrarily use either the k-DS or k-hop neighborhood in the storage phase, and the other in the join-computation phase. In particular, each new tuple generated at I is routed over the k-CDS associated with I and stored at the clusterheads. For join-computation, the tuple is joined (using a multiple pass scheme) with the tuples stored in the k-hop neighborhood $N_k(I)$ of I. Note that $N_k(I)$ is guaranteed to contain complete sliding windows of each data stream, since it intersects with every k-DS.

Choice of Parameters. The parameter k needs to be carefully chosen to optimize performance. In particular, if we use the k-DS for storage, then larger k entails lesser degree of replication and larger communication cost and delay. Opposite is the case when k-hop neighborhoods are used for storage. Also, k should be chosen such that a k-hop neighborhood is large enough to store all the sliding windows. After having chosen k, we construct multiple number of such k-CDS to ensure that each node is a clusterhead in at least one of the k-CDS.

Dynamic Topology. The constructed k-CDS are preprocessed data structures, and hence, need to be maintained in response of changes in topologies (node failures or additions).

In our discussion, we assume that a failing node informs its neighbors about its impending failure. Such an assumption is reasonable for failures due to battery depletion. The above assumption can be easily relaxed by requiring each node to send periodic beacons. Since, each k-CDS can be maintained independently, we consider maintenance of a single k-CDS C.

Node Additions. When a new node I joins the network, it gathers $(k+1)$-hop neighborhood information. If there is no clusterhead in the k-hop neighborhood, then I selects itself as a new clusterhead. In either case, I connects itself to C (using new gateway nodes) using the gathered $(k+1)$-hop information. Here, we have

assumed that the node I is connected to at least one network node.

Gateway Node Failures. Before a gateway node I_g dies, it gathers l-hop neighborhood information (for some l) and constructs a Steiner tree (in a centralized manner) connecting the neighbors of I_g that are in C. The nodes in the constructed Steiner tree are added to the set C, and notified of their membership in the k-CDS C.

Clusterhead Failure. The situation is more complicated when a clusterhead I_h fails. Here, we select some of the neighbors of I_h as new clusterheads, connect the selected new clusterheads with new gateways (if needed), and add all of these new nodes to C. Let B be the set of neighbors of I_h that are in the k-CDS C, and $\overline{B}$ be the set of neighbors of I_h that are not in C. We start off by selecting each element of B as a clusterhead, and designate each element of $\overline{B}$ as a *temporary clusterhead*. Next, we determine which nodes in $\overline{B}$ should be selected as clusterheads. First, all neighbors of I_h (i.e., $B \cup \overline{B}$) broadcast a `probe` message in their k-hop neighborhoods. Consider a node I that had only I_h as its clusterhead. If I does *not* receives a `probe` message from any node in B, then it sends a `make-permanent` message to the lowest-ID temporary clusterhead that I received a `probe` message from. A temporary clusterhead selects itself as a clusterhead (i.e., adds itself to C) on receiving a `make-permanent` message from any node. Finally, we add additional gateway nodes to connect the newly added clusterheads, by computing a Steiner tree (in a centralized manner) connecting them. Such a Steiner tree can be constructed by the failing clusterhead I_h (before failure) by gathering certain neighborhood information.

V. Communication Cost Analysis and Efficient Join Ordering

In this section, we analyze the communication cost incurred by various approaches discussed in the previous sections. *The purpose of the below analysis is to derive formulae for communication cost incurred by one-pass and multiple-pass PA schemes. These derived formulae can be used to determine a good join-ordering within the multiple-pass scheme and to choose between one-pass and multiple-pass scheme, for given parameters.* Here, we define the *communication cost* incurred as the sum of the total number of hops traversed by each operand tuple.

Definition 8: (Selectivity Factor.) The *selectivity factor* σ_P of a join condition P between two data streams R_i and R_j is defined as the fraction of tuple pairs (one each from R_i and R_j) that satisfy the join condition P. More formally, $\sigma_P = |R_i \ltimes_P R_j|/(|R_i||R_j|)$. □

Communication Cost in One-pass PA. In PA, the total communication cost is due to storage and join-computation. For a newly generated tuple, the communication cost incurred in PA (in either one-pass or multiple pass) for storage is just the hop-length of the horizontal path. The communication cost incurred during the join-computation phase of the one-pass PA can be computed as follows. Consider a vertical path of L nodes. Let us assume that the tuples of each sliding window are uniformly distributed along the vertical path. Consider a newly generated tuple t_1 of data stream R_1 (we choose R_1 for simplicity of presentation). In the one-pass scheme, t_1 traverses along the vertical path from one end (first node) to another end (L^{th} node). Consider the l^{th} node, i.e., the node that is l hops away from the first node on the vertical path. Below, we derive an expression for $N_l^2(\bar{n})$, the number of *new* partial results generated at the l^{th} node due to t_1, t_2 (some tuple of R_2; we choose R_2 for simplicity), and a set of $\bar{n}-2$ data streams other than R_1 and R_2. Here, $\bar{n} < n$ since we are counting only partial results. Let R_i' denote the part of the sliding window for R_i stored between the first and the l^{th} nodes. Since we assume uniform distribution, $|R_i'| = |R_i|l/L$. Now, the expression for $N_l^2(\bar{n})$ can be written as:

$$N_l^2(\bar{n}) = \sum_{S \subset_{\bar{n}-2}\{3,\ldots,n\}} |t_1 \ltimes t_2 \ltimes R'_{i_1} \ltimes R'_{i_2} \ldots R'_{i_{\bar{n}-2}}|,$$

where the summation is taken over all subsets of size $\bar{n}-2$ from $\{3,\ldots,n\}$ and $S = \{i_1, i_2, \ldots, i_{\bar{n}-2}\}$ is an instance of such a subset. Now, let σ_{uv} denote the selectivity factor of the join condition between R_u and R_v. Then, we get

$$N_l^2(\bar{n}) = \sum_{S \subset_{\bar{n}-2}\{3,\ldots,n\}} \Big(\prod_{u,v \in (S \cup \{1,2\})} \sigma_{uv} \Big) \prod_{u \in S} (|R_u|l/L).$$

Now, the total number of partial results generated at l^{th} node is $\sum_{\bar{n}=2}^{n-1} \sum_{i=2}^{n} N_l^i(\bar{n})|R_i|/l$. Recall that each of the partial results of size $\bar{n}$ traverses the remaining part of the vertical line, and hence, incurs a communication cost of $\bar{n}(L-l)$. If L_h is the hop-length of the horizontal path (and hence, the communication cost incurred for storage), the total communication cost (OP_PA_Cost) incurred by PA one-pass scheme due to a tuple t_1 of R_1 can be given by:

$$\text{OP\_PA\_Cost} = L_h + \sum_{l=1}^{L} \sum_{\bar{n}=2}^{n-1} \sum_{i=2}^{n} \bar{n}(L-l) N_l^i(\bar{n})|R_i|/l. \quad (\mathbf{1})$$

The above equation also applies to sensor networks without location information with L_h being the size of the region (k-CDS or k-hop neighborhood) used for storage and L being the size of the other region used for join-computation.

Communication Cost in Multiple-Pass PA. Communication cost for multiple-pass PA depends on the join ordering. For simplicity, let us assume that the order of the join is $R_2, R_3, \ldots, R_n$. Consider a newly generated tuple t_1 of R_1. In the i^{th} iteration of the multiple-pass scheme, the partial results corresponding to the tuples in $t_1 \ltimes R_2 \ltimes R_3 \ldots R_i$ traverse the entire vertical path to find matches from R_{i+1}, the next data stream in the join order. Before that, each of these generated partial results also traverses $L/2$ hops to get to the first node (to get ready for the next iteration). Since the first iteration incurs a communication cost of L (hop-length of the vertical path), the total communication cost (MP_PA_Cost) incurred by PA multiple-pass scheme for the join ordering $R_2, R_3, \ldots, R_n$ in response to a generated tuple t_1 of R_1 is:

$$\text{MP\_PA\_Cost} = 1.5L\Big(\sum_{i=2}^{n-1} i(|R_2||R_3|\ldots|R_i|) \prod_{1 \le i_1, i_2 \le i} \sigma_{i_1 i_2}\Big) + L_h + L. \quad (\mathbf{2})$$

Recall that L_h (length of the horizontal path) is the cost of the storage phase. As before, the above equation also applies to networks without location information.

Communication Cost in Centroid Approach (CA). The above analysis for multiple-pass scheme can also be used to compute the communication cost incurred by CA. If r is the memory available at each node, then $|R_i|/r$ is the number of nodes in

the storage region $\mathcal{C}_i$ storing the sliding window for R_i. Let D_i is the average distance (in hops) between a network node and the storage region $\mathcal{C}_i$, and d be the average distance (in hops) between two storage regions. Then, the total communication cost (CA_Cost) incurred by CA in the join-computation phase for the join-ordering $R_2, R_3, \ldots, R_n$ in response to a generated tuple t_1 of R_1 is:

$$\text{CA\_Cost} = (\sum_{i=2}^{n-1}(d + |R_{i+1}|/r)(i)(|R_2|\ldots|R_i|) \prod_{1 \le i_1, i_2 \le i} \sigma_{i_1 i_2}) + D_1 + 2|R_1|/r + (d + |R_2|/r). \quad (\mathbf{3})$$

Above, $2|R_1|/r$ is the communication cost incurred in searching for a node with available memory in $\mathcal{C}_1$, and $(d + |R_2|/r)$ is the cost of routing and broadcasting t_1 in $\mathcal{C}_2$.

Join Ordering Problem. The *join-ordering* problem of finding an ordering of data streams that minimizes the communication cost (Equation 2 for multiple-pass PA or Equation 3 for CA) can be shown to be NP-hard using a reduction from maximum clique [26]. Note that the above join-ordering problem is equivalent to finding the optimal left-deep tree for evaluation of the given join query, and that the communication cost incurred in the multiple-pass PA and CA is proportional to the sum of the sizes of the intermediate results. Thus, we could directly use the techniques developed in [26] to determine an efficient join-ordering of data streams. In particular, we use the greedy heuristic of [26] which works by first selecting the data stream that minimizes the communication cost incurred in the last iteration of the join-computation phase, as the last stream in the ordering. After picking the last data stream, the best choice for last-but-one data stream in the ordering is selected, and so on. The above greedy heuristic essentially works on the premise that the communication cost incurred in the last iteration dominates the overall cost. As typical sensor network queries are long running, we assume that all the catalogue information needed (estimated sizes, locations of the operand relations and selectivity factors) can be gathered by initial sampling of the operand tables.

Summary: One-Pass Versus Multi-Pass Scheme. In the one-pass scheme, all possible partial results are computed and propagated. In contrast, in the multi-pass scheme, only "ordered" (determined by the join-order) partial results are computed. Thus, the multi-pass scheme in conjunction with an efficient join-ordering is generally expected to outperform one-pass scheme. However, for joins with high selectivity factors, the one-pass scheme may be more efficient than the multi-pass scheme, since in the one-pass scheme the partial result traverse through less number of nodes.

VI. Performance Evaluation

In this section, we present our simulation results that compare the performance of various approaches. We simulate our algorithms on *ns2* [19], a general purpose network simulator capable of simulating wireless ad hoc networking protocols. Since our techniques are targeted for large sensor networks (hundreds of nodes), it was infeasible to simulate our techniques on real sensor networks or real sensor data (since the largest available data we could find online is for 30-40 nodes).

Parameter Values and Settings. We generated random sensor networks by randomly placing 1000 nodes in an area of 3000×3000 meters. We fix the transmission radius of each node to be 250 meters to ensure a connected network graph, and consider the case of join of 4 data streams. Each stream is generated uniformly across the network at the rate of 150 tuples per unit time. We compute the join based on a sliding window of size 150 tuples (or one unit time) for each data stream. The default memory capacity at each node is 30 tuples, but we vary it in one set of experiments. Note that *the absolute value of the sliding window size or memory capacity per node is immaterial for purposes of performance comparison*; thus, we only vary the ratio of sliding window size to the memory capacity by varying the latter. We set the battery energy, transmission power, receiving power of each node to 120J, 0.28W, and 0.14W respectively [32]. By default, we store a tuple on every *other* node (of the horizontal path) in the storage phase, and do a one-hop broadcast from each node on the vertical path; we consider different replication factors in one set of experiments. We use a uniform selectivity factor (1/2 for spatial joins and 1/10 for non-spatial join) for all pairs of streams. For the given selectivity factors and sizes of sliding windows, the communication-cost Equations 1 and 2 suggest that the multiple-pass will be more efficient than the one-pass scheme. Thus, we use multiple-pass scheme for PA. Note that beyond the choice of one-pass vs. multiple-pass, *the absolute value of selectivity factors does not have any effect on the relative performance of the approaches*. In one set of experiments, we use non-uniform selectivity factors, and compare the performance of one-pass versus multiple-pass with different join-orderings.

Performance Metrics. We use the following performance metrics (over time) to measure the performance of our approaches: total battery energy dissipated (i.e., total communication cost), number of battery-depleted nodes, and "approximation ratio" of the output results. The ***approximation ratio*** of an approach at a given time is defined as the ratio of (i) the number of result tuples output by the approach, to (ii) the total number of tuples in the actual join result (computed independently in a centralized way), due to the input tuples generated in the last T units of time. In our graph plots, we choose T to be 10 units. The approximation ratio metric signifies the *current* state of the network (based on last 10 units of time) and incorporates almost all aspects of the performance of an approach. Thus, we use approximation ratio as our main performance criterion. *Network lifetime* can be defined as the amount of time for which the approximation ratio remains above a certain threshold; we consider 80% threshold in our discussions. Low approximation ratio could be due to node failures, message collisions, non-availability of sliding window tuples due to limited memory and/or network partitioning. We expect load-balanced and efficient PA to have a much higher network lifetime than CA.

Approaches. For the given parameter values, the Naive Broadcast approach is infeasible; e.g., for default values each node can only store 57.3% (= $(30 * 3000^2/(600 * \pi(500)^2)$) of the entire sliding windows and thus, the approximation ratio can be at most 57.3%. Thus, we implement the *Local Storage (LS)* approach, which stores each tuple only locally (at its source node) and uses multiple passes (as in the multiple-pass PA scheme) for join computation. In each pass, each partial result is broadcast upto the range of the spatial join or to the entire network for the case of non-spatial join. The LS approach is actually a special case (for very large k) of the extended PA (Section IV-C) without location information. LS approach is load-balanced, but incurs more communication cost than PA. Below, we compare CA,

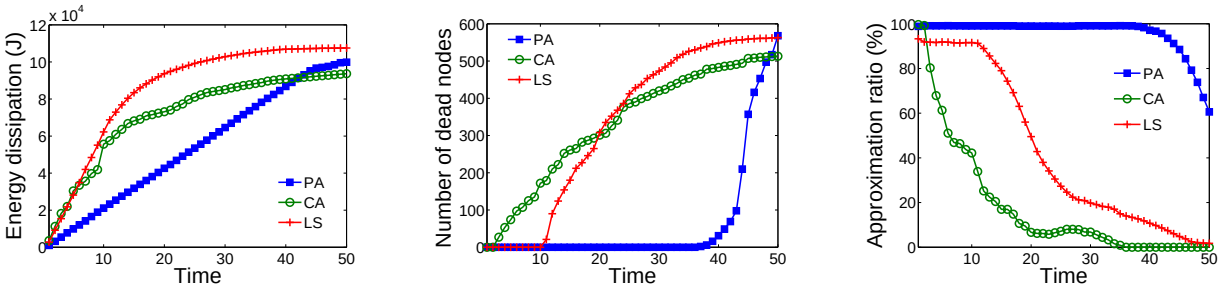
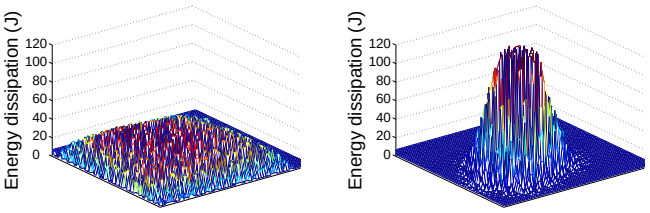


Fig. 6. Performance of various approaches for a spatial join of range 500 meters with memory capacity of 30 tuples/node. (a) Total energy dissipated, (b) Number of node failures, and (c) Approximation ratio.

PA, and LS approach for various parameter values and settings. For the PA scheme, we include all the overhead cost, except for the minimal (two messages per boundary node) one-time cost of computing the boundary nodes and markings. Also, duplicates are an inherent fault-tolerant feature of PA, and are not eliminated. Finally, as discussed in Section IV, the result tuples are output across the network. Collecting results at a central *node* will have similar problems as in the Centralized Approach discussed in Section III; hashing of result tuples or shipping them to a central server connected to all nodes will have similar cost for all schemes and is thus ignored.



(a) Energy dissipation in PA. (b) Energy dissipation in CA.

Fig. 7. Energy dissipation in CA an PA, for a spatial join of range 500 meters.

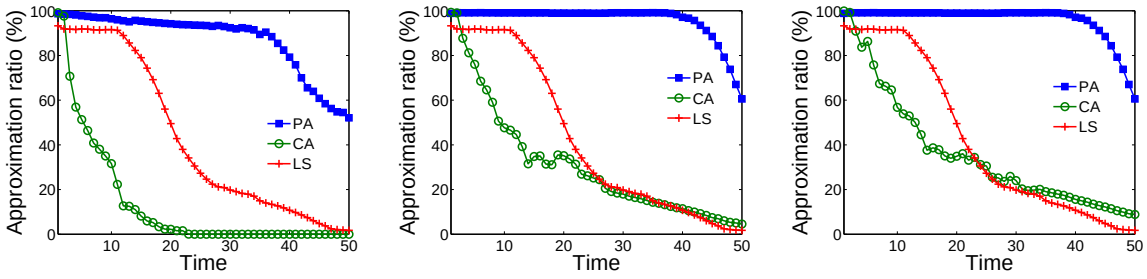
Spatial Join of Range 500 Meters. We start with considering performance of various approaches for the case of a spatial join of range 500 meters. As mentioned before, we use an *additional* (beyond the selectivity due to the spatial constraint) selectivity factor of 1/2 for all pairs of streams. We plot our simulation results in Figure 6. We see that the rate of energy dissipation in PA is less than in CA or LS. The rate of energy dissipation tapers off in each approach after some time, due to decrease in the number of active nodes. We notice that in PA the nodes start failing much later than in CA or LS, due to the communication-efficient and load-balanced operation of PA. Finally, we can see in Figure 6(c), that the approximation ratio of PA stays close to 100% for a long time. Essentially, when all nodes are alive, the fault-tolerance of the approach makes up for the few lost messages. The message collisions were observed to be rare due to "non-convergent" communication pattern and low rate of tuple generation. If we use the approximation ratio threshold (for network lifetime) of 80%, then the network lifetime of PA is about 3 times longer than that of LS and about 12 to 15 times longer than that of CA. In Figure 7, we show the distribution of energy dissipation at some snapshot of time for PA and CA. The distribution for LS is similar to that for PA, except that the peaks are much higher (by a factor of about 3).

Varying Memory Capacities. In Figure 8, we compare performance of various approaches for different values of memory capacities (10, 60, and 90 tuples per node). We observe that PA continues to outperform both CA and LS by a large factor (3 to 10) in terms of the network lifetime. Note that LS approach doesn't change with change in memory capacity. We observe that the performance of PA is same for memory capacities of 30 or more, and the performance of CA improves with increase in memory capacity.

Different Spatial-Join Ranges. In Figure 9(a)-(b), we consider other ranges of spatial join, viz., 750 and 1000 meters. Since the transmission radius is 250 meters, considering lower range value is too perfect for PA, and a value of 1500 or higher will almost cover the entire network (and hence, equivalent to a non-spatial join). Here, we plot only the approximation ratio over time, since it incorporates all the performance metrics. For the range of 750 meters, the network lifetime of PA is about 3 times longer than LS and about 20 times longer than CA. For the range of 1000 meters, both CA and LS have an effective network lifetime of zero, while that of PA is about 10. The low approximation ratios of CA or LS (even in the initial phases) is due to a large number of message collisions in the join computation phase, which requires repeated broadcast (within the storage region for CA or entire network for LS) for each computed partial result. Note that even though CA does not incorporate spatial joins, performance of CA worsens with increase in spatial-join range due to the increase in the overall selectivity factor.

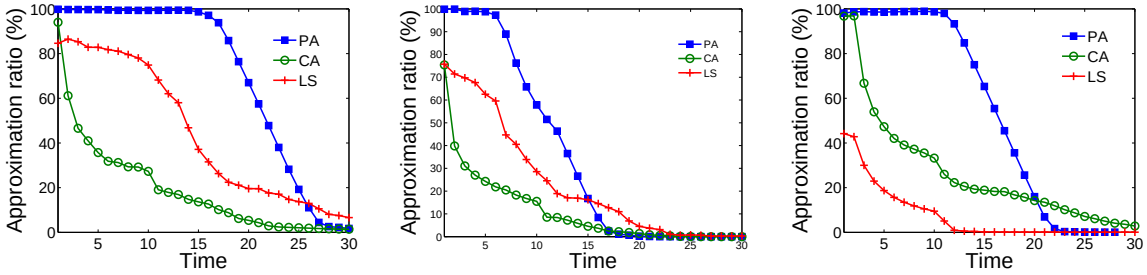
Non-Spatial Join. As mentioned before, for non-spatial join, we use a selectivity-factor of 1/10 for each pair of streams. Since the overall selectivity-factor for the non-spatial join is perhaps (they aren't easily comparable) less than the spatial join of range 1000, we see that CA and PA perform better for the latter. See Figure 9 (c). Moreover, we see that LS performs very poorly; it has an approximation ratio of at most 50%.

Varying Replication Factor (k) and Memory Capacity. In this set of experiments, we vary the replication factor k, which signifies how often we store each tuple on a horizontal path. As mentioned in Section IV, if we store a tuple at every k^{th} node on the horizontal path, then we need to do a $\lfloor k/2 \rfloor + 1$-hop broadcast from each node on the vertical path. However, in most cases, the last hop broadcast is not required due to the inherent fault-tolerance of the approach and the random network topology. Thus, we use 1-hop broadcast for $k = 2$, and for $k = 3$ or 4 we



(a) 10 tuples per node. (b) 60 tuples per node. (c) 90 tuples per node.

Fig. 8. Approximation ratios over time for the spatial join of range 500 meters with different memory capacities.



(a) Spatial-join range of 750 m. (b) Spatial-join range of 1000 m. (c) Non-spatial join.

Fig. 9. Approximation ratios over time for different spatial-join ranges, and non-spatial join.

do the last-hop broadcast with a 20% probability. Figure 10(a) plots the approximation ratio of PA for varying memory capacity. Here, we plot the approximation ratio during one unit of initial time, when all nodes are alive. As expected, we see that the approximation ratio decreases with decrease in memory capacity or replication factor. In Figure 10(b), we plot the communication cost for varying k and memory capacities per node. Increase in k should result in more energy dissipation. However, we notice that $k = 1$ incurs a much higher communication cost than $k = 2$ or 3, due to a large number of duplicate partial results generated when $k = 1$. From the given plots in Figure 10, we can conclude that to achieve an approximation ratio of at least 80%, we should use $k = 2$ (with one-hop broadcast) for memory capacity of 20 or higher. For lower memory capacities, higher values of k are needed. Note that $k = 1$ is never a good choice.

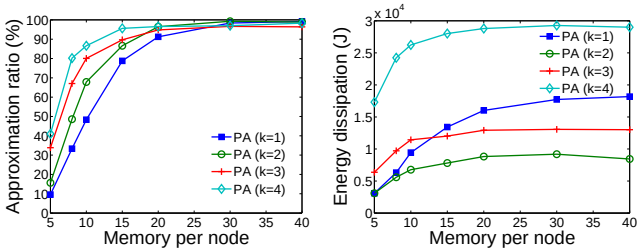


Fig. 10. Varying replication factor k and memory capacity for non-spatial join. (a) Approximation ratio and (b) Energy dissipation, in one (initial) time unit.

Effect of Join Ordering. We now depict the effect of join-ordering on the performance of PA. In this set of experiments, we choose non-uniform selectivity factors as shown in Figure 11(a). We compare the performance of one-pass PA and multiple-pass PA.

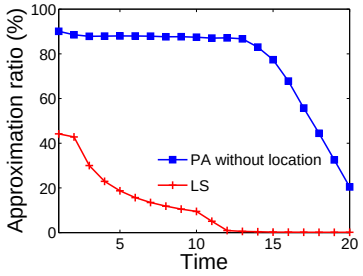
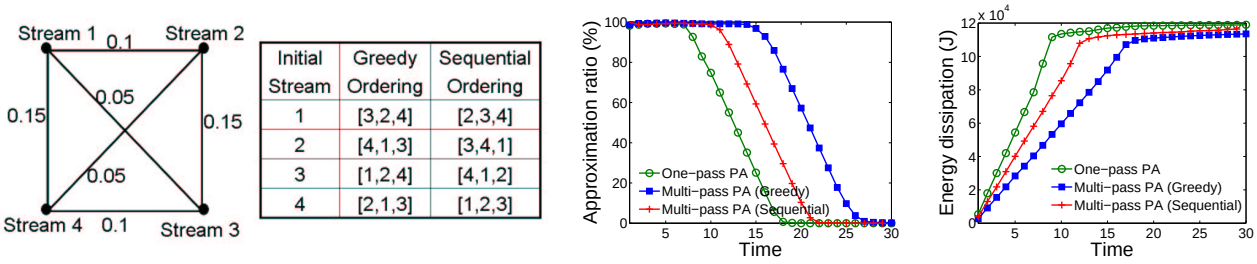


Fig. 12. Approximation ratios for PA and LS in a network without location information for non-spatial join.

For the multiple-pass PA, we use two different join orderings, viz., greedy (as described in Section V) and sequential (where each new tuple iteratively picks the next data stream in sequence). See Figure 11(b). We see in Figure 11 (c)-(d) that the multiple-pass PA with greedy join ordering performs the best, followed by the multiple-pass PA with sequential join ordering.

Networks Without Location Information. Figure 12 shows the performance of extended PA (Section IV-C) in networks without location information for non-spatial join; we used 2-hop neighborhoods and 2-CDS for join computation and storage respectively, and incorporated all cost pertaining to the maintenance of the 2-CDS trees. We observe that PA performs much better than the LS approach – the only other feasible approach for networks without location information.

Summary of Simulation Results. In our simulations, we have compared PA with other approaches for a wide range of network parameters. In general, we observed that PA resulted in a much longer network lifetime for computation of join, due to its communication-efficiency and load-balance. For spatial joins, LS



Initial Stream	Greedy Ordering	Sequential Ordering
1	[3,2,4]	[2,3,4]
2	[4,1,3]	[3,4,1]
3	[1,2,4]	[4,1,2]
4	[2,1,3]	[1,2,3]

Fig. 11. Non-uniform selectivity factors and join ordering for non-spatial join. (a) The join graph depicting various selectivity factors, (b) Greedy and sequential join orders, (c) Approximation ratio, and (d) Energy dissipation.

outperformed CA, while for non-spatial joins LS performed very poorly. In general, the performance of CA and PA improve with increase in memory capacity. Finally, we evaluated the tradeoff between storage and communication cost (by varying the replication factor) and observed the efficacy of the greedy heuristic for join-ordering.

VII. Conclusion and Future Work

Communication-efficient implementation of sensor network programs remains a challenging research direction. More specifically, when sensor networks are viewed as distributed databases, most of the functionality of sensor network applications can be specified in terms of database queries. Due to the large amount of data generated in the network, efficient implementation of queries can have a great impact on prolonging the network lifetime. In this article, we have addressed communication-efficient implementation of join in sensor networks, and designed various approaches. One of our designed approaches, viz. Perpendicular Approach (PA), is quite load-balanced and significantly outperforms (in terms of network lifetime) other approaches over a wide range of parameter values. Our future work is focussed on design of an in-network deductive query engine, and developing techniques for multiple query optimization and selection of views to materialize in the context of sensor networks.

References

[1] D. Abadi, S. Madden, and W. Lindner. REED: Robust, efficient filtering and event detection in sensor networks. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2005.

[2] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15, 2006.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2002.

[4] 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.

[5] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceeding of the International Conference on Mobile Data Management (MDM)*, 2001.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, 2000.

[7] S. Cheung, M. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. In *Proceedings of the International Conference on Database Engineering (ICDE)*, 1990.

[8] V. Chowdhary and H. Gupta. Communication-efficient implementation of join operation in sensor networks. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, 2005.

[9] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[10] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proceedings of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, 2003.

[11] M. Datar, P. Indyk, N. Immorlica, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the ACM Symposium on Computation Geometry (SoCG)*, 2004.

[12] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-based approximation querying in sensor networks. *VLDB Journal*, 2005.

[13] L. Ding, N. Mehta, E. Rundensteiner, and G. Heineman. Joining punctuated streams. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2004.

[14] D. Abadi et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12, 2003.

[15] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2005.

[16] H. Gupta et al. Deductive approach for programming sensor networks. Technical report, Stony Brook University, 2007. `http://www.cs.sunysb.edu/~hgupta/ps/logicSN.pdf`.

[17] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, 2003.

[18] S. Madden et al. TinyDB: In-network query processing in TinyOS. `http://telegraph.cs.berkeley.edu/tinydb`.

[19] K. Fall and K. Varadhan (Eds.). The *ns* manual. `http://www-mash.cs.berkeley.edu/ns`.

[20] L. Golab and M. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceed-*

ings of the International Conference on Very Large Data Bases (VLDB), 2003.

[21] 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, 2002.

[22] L. Guibas. Sensing, tracking, and reasoning with relations. *IEEE Signal Processing Magazine*, 19(2), 2002.

[23] H. Gupta, V. Navda, S. Das, and V. Chowdhary. Energy-efficient gathering of correlated data in sensor networks. In *Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2005.

[24] M. Hammad, W. Aref, and A. Elmagarmid. Stream window join: tracking moving objects in sensor-network databases. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2003.

[25] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2001.

[26] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 1984.

[27] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the International Conference on Database Engineering (ICDE)*, 2003.

[28] B. Karp and H. T. Kung. Greedy perimeter stateless routing for wireless networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[29] S. Li, Y. Lin, S. Son, J. Stankovic, and Y. Wei. Event detection using data service middleware in distributed sensor networks. *Wireless Sensor Networks of Telecomm. Systems*, 2004.

[30] X. Liu, Q. Huang, and Y. Zhang. Combs, needles, haystacks: balancing push and pull for discovery in sensor networks. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[31] H. Lu, M. Shan, and K. Tan. Optimization of multi-way join queries for parallel execution. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1991.

[32] M. Srivastava. Power Considerations for Sensor Networks. `http://ipsn.acm.org/2001/slides/Srivastava.pdf`.

[33] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the International Conference on Database Engineering (ICDE)*, 2002.

[34] 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.

[35] S. Madden, M. Franklin, J. 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 (SIGMOD)*, 2003.

[36] 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 (SIGMOD)*, 2002.

[37] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2003.

[38] A. Pandit and H. Gupta. Efficient implementation of range-joins in sensor networks. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, 2006.

[39] S. Patil, S. Das, and A. Nasipuri. Serial data fusion using space-filling curves in wireless sensor networks. In *Proceedings of the International Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, 2004.

[40] 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), 2003.

[41] J. Richardson, H. Lu, and K. Mikkilineni. Design and evaluation of parallel pipelined join algorithms. In *Proceedings of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, 1987.

[42] R. Sarkar, X. Zhu, and J. Gao. Double rulings for information brokerage in sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*, 2006.

[43] A. Savvides, M. Srivastava, L. Girod, and D. Estrin. *Chapter: Localization in sensor networks*. Kluwer Academic Publishers, 2004.

[44] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the ACM SIGMOD Conference on Management of Data (SIGMOD)*, 1989.

[45] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2005.

[46] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[47] K. Tan and H. Lu. Processing multi-join query in parallel systems. In *Symposium on Applied Computing*, 1992.

[48] Y. Xing, J. Hwang, U. Cetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2006.

[49] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Transactions on Database Systems (TODS)*, 24, 1999.

[50] S. Yang, J. Wu, and J. Cao. Connected k-hop clustering in ad hoc networks. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2005.

[51] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3), 2002.

[52] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2003.