Deductive Approach for Programming Sensor Networks

Himanshu Gupta and Xianjin Zhu; Stony Brook University,NY

Abstract

As sensor networks become common and enable new applications, developing powerful paradigms for programming sensor networks becomes critical to realize their full potential as collaborative data processing engines. In this article, we propose a deductive framework for programming sensor networks, based on the observation that the sensor network can be viewed as a distributed database of facts gathered from the environment. We believe that the overall collaborative (distributed) functionality of a sensor network application can be easily represented using deductive (logic) rules, and the non-collaborative functionality can be embedded in built-in functions. The proposed framework is declarative, and more expressive than the previously proposed distributed database view of sensor networks. A fully-developed framework will allow the user specify with ease the high-level functionality of an application, while hiding from user the low-level details such as related to distributed computation, resource constraints, energy optimizations, etc. Our system translates a given high-level user program to energy-efficient distributed code that runs on individual network nodes. In this article, we motivate the deductive framework for programming sensor network applications, and develop query evaluation techniques for distributed evaluation of deductive queries in sensor networks. We present certain performance results that illustrate the robustness and efficiency of the generated distributed code.

1 Introduction

Programming a sensor network application remains a difficult task, since the programmer is burdened with low-level details related to distributed computing, careful management of limited resources, unreliable infrastructure, and energy optimizations. Thus, developing a powerful programming framework for sensor network is critical to realizing their full potential as collaborative monitoring systems. There has been some progress in developing operating system prototypes [14, 16] and programming abstractions [43, 44]; however, these abstractions have provided only minimal programming support. Prior work on viewing the sensor network as a distributed database provides a declarative programming framework which is amenable to optimizations. However, it lacks expressive power, and the developed database engines (TinyDB [32], Cougar [7]) for sensor networks implement only a limited functionality. On the other hand, the recently proposed Kairos [18] framework is expressive, but is based on a procedural language and hence, difficult to translate to efficient distributed code. Thus, the overall vision of a programming framework that automatically translates a high-level user specification to efficient distributed code remains far from realized. In general, a perfect programming paradigm for sensor networks must achieve the following.

- Be sufficiently expressive.
- Be declarative, i.e., provide users with a high-level abstraction of the network, while hiding all the low-level networking details.
- Be amenable to automatic optimizations (especially, related to energy consumption) without much input from user.

In this article, we motivate use of deductive approach for programming of sensor networks, and design and develop a query engine for distributed evaluation of deductive queries. In particular, our developed system facilitates automatic translation of high-level deductive queries into optimized nesC node that runs on individual sensor nodes.

Proposed Deductive Approach. We propose a programming framework based on a deductive paradigm; our proposed framework is declarative, fully expressive (Turing complete), and most importantly, amenable to automatic translation into efficient distributed code. Deductive approach has been recently used with success for declarative specification of network routing protocols [27] and overlay architectures [26]. In addition, a dialect of Datalog without negations has been suggested for use in sensor networks [10, 15], and a limited query processor designed. In the context of programming sensor networks, our deductive approach is motivated by the basic observation that sensor networks essentially gather sets of "facts" by sensing the physical world, and sensor network applications manipulate these facts. We believe that the collaborative (involving multiple nodes) functionality of a sensor network application can be easily represented using fact-manipulation deductive rules. The local arithmetic computations such as signal-processing, data fusion, etc. may be inefficient to represent

using deductive rules, and hence, are embedded in *locally-processed* built-in functions *written in procedural code*. *Embedding such local computations in locally-processed procedural functions does not affect the communication efficiency of the translated code.* The above approach facilitates easy high-level specification of an application, and is amenable to optimizations. To realize the overall vision of a powerful programming framework, we develop techniques for communication-efficient evaluation of deductive programs in resource-constrained sensor networks over streaming data. Based on the developed query processing techniques, our system automatically translates a given high-level specification of an application into efficient distributed code that runs on individual nodes.

Article Organization. The rest of the article is organized as follows. We start with a discussion on related work, and an overview of the proposed deductive approach in Section 2. We discuss distributed evaluation of logic queries and the overall system architecture in Section 3. Section 4 presents preliminary performance results from a prototype implementation on TOSSIM [24], an application used widely for simulating sensor network applications. We end with concluding remarks in Section 5.

2 Prior Approaches, and Deductive Framework for Sensor Networks

In this section, we start with an overview of prior approaches for programming sensor networks. Then, we give an overview of deductive programming, and illustrate the power of our approach through various illustrations. Finally, we propose some extensions and restrictions to the deductive framework to tailor it to programming of sensor networks.

2.1 Prior Approaches for Programming Sensor Networks

NesC and Programming Abstractions. The Berkeley motes [21] platform provides the C-like, fairly low-level programming language called *nesC* [16] on top of the TinyOS [14] operating system. However, the user is still faced with the burden of low-level programming and optimization decisions. There has been some work done on developing programming abstractions [3, 8, 34, 42–44] for sensor networks; however, these abstractions provide only minimal programming support. Finally, authors in [6] propose an interesting novel approach of expressing computations as "task graphs," but the approach has limited applicability.

Sensor Network as a Distributed Database. Recently, some works [7, 17, 32] proposed the powerful vision viewing the sensor network as a distributed database. The distributed database vision is declarative, and hence, amenable to optimizations. However, the current sensor network database engines (TinyDB [32], Cougar [7]) implement a limited functionality of SQL, the traditional database language. In particular, they only handle single queries involving simple aggregations [28, 30, 45] or selections [31] over single tables [29], local joins [45], or localized/centralized joins [2] involving a small static table. These approaches are appropriate for periodic data gathering applications. SQL is not expressive enough to represent general sensor network applications. Moreover, due to the lack of an existing SQL support for sensor networks, there is no real motivation to choose SQL. In an ongoing recent work, a dialect of Datalog without negations has been suggested for use in sensor networks [10, 15], and a limited query processor designed. The focus of the works in [10, 15] is generally on declarative representation of networking and routing protocols. Our deductive framework is essentially an expansion of the above approaches, wherein we use a more expressive language for programming high-level applications and design an efficient full-fledged in-network query engine for sensor networks.

Procedural Languages. Recently proposed Kairos [18] provides certain global abstractions and a mechanism to translate a centralized program (written in a high-level procedural language) to an in-network implementation. In particular, it provides global abstractions such as `get_available_nodes`, `get_neighbors`, and remote data access. Kairos is the first effort towards developing an automatic translator that compiles a centralized procedural program into a distributed program for sensor nodes. However, Kairos does not focus much on communication efficiency; for instance, the abstraction `get_available_nodes` gathers the entire network topology, which may be infeasible in most applications.

In some sense, our approach has the same goals as that of Kairos – to automatically translate a high-level user specification into distributed code. However, since Kairos approach is based on a procedural language, it is much harder to optimize for distributed computation. Through various examples in Section 2.3, we suggest that our proposed framework will likely yield more compact and clean programs than the procedural code written in Kairos. Moreover, the deductive programs for the examples in Section 2.3 yield efficient distributed implementations involving only localized joins.

In general, we feel that procedural languages are unlikely to be very useful in a restricted setting such as sensor networks, since they are not declarative and would be hard to distribute and optimize for communication cost.

2.2 Overview of Deductive Programming

Predicate logic is a way to represent "knowledge" and can be used as a language for manipulating tables of facts. In logic data model, each relation (table of facts) is looked upon as a predicate having a argument for each table attribute, and the predicate is true for the given argument values if and only if the corresponding fact exists in the table. For instance, consider a table *likes(drinker, beer)* wherein a fact (d, b) in the table signifies that *d* likes *b*. In the logic data model, the table *likes* is looked upon as a predicate wherein *likes(d,b)* is true iff *d* likes *b* (i.e., (d, b) is in the table *likes*).

Datalog. The simplest model of predicate logic, *Datalog*, consists of a set of declarative *logic rules*, possibly involving recursion and negations. A Datalog rule has the form "*head* :- *body*," where body is a list (implicit conjunction) of predicates over constants and variables, and the *head* defines a set of facts derived by variable assignments satisfying the body's predicates. For instance, consider the predicates *likes(drinker, beer)* and *sells(bar, beer)*. Here, *sells(br, b)* is true if the beer *b* is sold at the bar *br*. The rule $happy(d) :- likes(d, b), sells(br, b)$ defines a new predicate *happy(d)* such that a *happy(d)* is true if and only if there is *some* bar that sells a beer that *d* likes. Datalog without recursion is as expressive as the traditional database language SQL without aggregations.

Full First-Order Logic. In our proposed programming framework, we use full first-order logic which extends Datalog by allowing function symbols in the arguments of predicates,

and thus, making the framework Turing complete [40]. We illustrate the need for function symbols in Example 3 of Section 2.3. Essentially, in full first-order logic, the arguments of a predicate may be arbitrary terms, where a term is recursively defined as follows. A *term* is either a constant, variable, or $f(t_1, t_2, \ldots, t_n)$ where each t_i is a term and f is a function symbol. In this general context, a logic rule is written as

$$H \ :- \ G_1, G_2, \ldots, G_k.$$

H is called the *head*, and $G_1, \ldots, G_k$ are the *body subgoals*. The head and the subgoals are of the form $p(t_1, t_2, \ldots, t_m)$ where p is a predicate and t's are arbitrary terms. The meaning of the rule is "if $G_1, \ldots, G_k$ are true, then H is true."

Derived and Base Predicates/Tuples. We use the term *derived predicates* to refer to predicates that are defined using the deductive rules in the program. Predicates that are not derived are referred to as *base predicates*; essentially, the base predicates refer to tables that are already given (e.g., the network graph) or generated by the network (e.g., tables corresponding to sensing readings). Corresponding tuples, tables, or streams are referred to as derived or base tuples, tables or streams.

Built-In Predicates, and Added Features. Certain predicates that are given a conventional interpretation such as $X < Y$, are called *built-in* and can appear in the body subgoals. In our framework, a user may define additional built-in predicates, in which case the user provides the procedural code to evaluate the predicate. Note that built-in predicates can be easily used to specify built-in functions, and hence, we use *built-in functions* directly in the logic rules. For sake of ease in programming, we allow restricted use of negated subgoals, lists, and aggregations (as discussed in Section 3).

Specification and Maintenance of Sliding Windows. Sensor network data can be modeled as streaming sets/tables of facts corresponding to sensing readings. As in streaming databases and due to limited memory resources in sensor network, we store only a finite set of tuples (typically, most recent) called the *sliding-window* [5,13] for each data stream. Recently, [5] defined various notions of *sliding-windows* and used these notions to develop well-defined semantics for continuous queries over streaming data. Notion of sliding-windows essentially allows us to implicitly incorporate temporal correlation of data into specified queries by expiring tuples that become "irrelevant" (i.e., not join with any future tuples). In a deductive framework, we can use temporal predicates to specify time-based windows. For instance, we could use the following logic rule to define sliding-windows R of range τ_w from the given relation $S(a,t)$, where a is an arbitrary attribute and t is the timestamp of each tuple of S.

$$R(a,t,T) : -S(a,t), T - \tau_w < t < T, S(\_, \tau)$$

Above, the last subgoal is used to bound the variable T, "_" denotes an anonymous variable, and $R(\_,\_,T)$ is a tuple in the sliding-window of time T. Time-based sliding-windows in sensor networks can be easily maintained in a distributed manner, by expiring a tuple after sufficient amount of time after its generation. By default, each subgoal in a logic rules refers to an "unbounded" data stream. In this article, we restrict our discussions to time-based sliding windows; maintenance of count-based or other types of sliding-windows in an asynchronous distributed manner is a challenge and part of our future work.

Motivating Characteristics of The Deductive Approach. In short, our choice of deductive approach is motivated by its following characteristics. Firstly, a deductive programming framework is declarative and hence, amenable to optimizations. In our context, the optimization of logic programs is largely embedded in the efficient data storage schemes, in-network implementation of join, join-ordering, and query optimization techniques. Secondly, a deductive framework augmented with function symbols is fully expressive; in particular, it is more expressive than the prior distributed database approach. Extensive use of function symbols (or lists) does make optimizations difficult, but we anticipate that function symbols will be used in limited contexts and hence, allow their use for full expressibility. Thirdly, a deductive framework has strong theoretical foundations and can be easily extended to include other specialized deductive frameworks. Specialized logics that could be useful in the context of sensor networks include Probablistic LP [35] and Annotated Predicate Logic [23] (for reasoning with uncertain information).

and Annotated Predicate Logic [23] (for reasoning with uncertain information).

Prior Success of Datalog in Declarative Networking. Recently, Datalog without negations has been used for declarative specification of network routing protocols [27] and overlay architectures [26], resulting in very compact and clean specifications. The approach was shown to be efficient, secure, expressive for intended purposes, and amenable to query optimizations. This recent success of use of deductive queries for declarative networking adds to the promise of our deductive approach for programming sensor networks.

2.3 Illustrating the Power of Deductive Approach

We now illustrate the power of the deductive approach by describing how it can be used to program a few different distributed computations that have been proposed for sensor networks: vehicle tracking, trajectories, and routing tree construction. In addition, we illustrate the use of negated subgoals and function symbols in programming typical applications. We start with discussing the use of built-in functions to embed arithmetic computations.

Embedding Signal-Processing and Other Arithmetic Computations in Built-in Functions. Certain aspects of sensor network applications involve local arithmetic computations such as signal processing, data fusion, synthesis of base data, etc. Such arithmetic computations may be too inefficient to represent in a deductive framework, and hence, are embedded in locally-processed built-in functions coded in a procedural language. Such a representation does not compromise on the communication efficiency on the translated distributed code. Distributed arithmetic computations are embedded in built-in aggregates with specialized distributed implementations. For instance, in vehicle tracking [11, 37], arithmetic computations involve estimating belief states, information utilities, and estimate of the future target location; the first two computations are local, while the last computation requires the *maximum* aggregate. See Example 1 below. Finally, certain other arithmetic techniques

such as data compression may be embedded in the query engine.

EXAMPLE 1. **Vehicle Tracking.** The given program represents the algorithm for tracking vehicles described in [37]. The algorithm uses probabilistic and signal-processing techniques to maintain posterior distribution (*belief state*) of the vehicle location.

$$
\begin{array}{ll}
U(i_1,t+1,u_1) & :- P(i,t,v), G(i,i_1), Z(i,t+1,z), \\
 & \quad Z(i_1,t+1,z_1), u_1 = I(v,z,z_1) \\
P'(i_1,t+1) & :- P(i,t,v), G(i,i_1), G(i,i_2), U(i_1,t+1,u_1), \\
 & \quad U(i_2,t+1,u_2), u_1 < u_2 \\
P(i_1,t+1,F(v,z)) & :- P(i,t,v), Z(i,t+1,z), G(i,i_1), \\
 & \quad \text{NOT } P'(i_1,t+1)
\end{array}
$$

At any time instant, only one node namely the leader node is active. The leader applies a measurement of its observation and produces an updated belief state about the vehicle location. The updated belief is then passed onto one of the neighboring nodes with the highest "utility information," which becomes the new leader, and the process repeats. In the given program, we have used the same variable symbols as used in [37]. For a node i at time t, $P(i,t,v)$ signifies the belief state value v, $U(i,t,u)$ signifies the information utility value u, and $Z(i,t,z)$ signifies the sensed value z. Also, $G(x,y)$ represents the network edges, F and I are locally-processed built-in functions. The function F represents the Equation 3 of [37] which computes the updated belief state at the new leader node, and I computes the information utility of a local node. The first logic rule in the given program computes the information utility of a neighbor i_1 of the leader node i, and the third rule computes the new leader node and the new belief state. The predicate $P'(i_1,t+1)$ signifies that i_1 does *not* have the highest information utility. The given logic program is more compact than the corresponding procedural code written in Kairos (see [18]). More importantly, the given program can be automatically translated into communication-optimal distributed code, as discussed in later sections. □

EXAMPLE 2. **Need for Negated Subgoals.** Negation in deductive framework is essential (in absence of function symbols) if we need to take a difference of two sets/tables. Consider a sensor network deployed in a battlefield for tracking enemy vehicles. Here, lets assume availability of a data stream *veh(ID, type, location, time)* that signifies vehicle detection of a certain type ('friendly' or 'enemy') at a particular time and location. Now, let us say we are interested in generating an alert when there is an "uncovered" enemy vehicle, i.e., an enemy vehicle that is not within a distance of say 5 from *any* friendly vehicle. The corresponding query may be simply written as follows.

$$
\begin{array}{ll}
cov(l_1,t) & :- veh('\texttt{enemy}', l_1, t), veh('\texttt{friendly}', l_2, t), \\
 & \quad dist(l_1,l_2) \leq 5 \\
uncov(l,t) & :- NOT\ \ cov(l,t), veh('\texttt{enemy}', l, t)
\end{array}
$$

□

EXAMPLE 3. **Need for Function Symbols.** We now illustrate the need for function symbols in our programming framework. Essentially, function symbols are required when we want to create non-atomic values. For example, in case of vehicle trajectories, if we need to compute and store the actual path of the trajectory, we need to use function symbols (or lists).

$$
\begin{array}{ll}
traj([R_1,R_2]) & :- report(R_1), report(R_2), close(R_1,R_2), \\
 & \quad NOT\ \ notStartReport(R_1) \\
notStartReport(R_2) & :- report(R_1), report(R_2), close(R_1,R_2) \\
traj([X|R_1,R_2]) & :- traj([X|R_1]), report(R_2), close(R_1,R_2) \\
completeTraj([X|R]) & :- traj([X|R]), NOT\ \ notLastReport(R) \\
notLastReport(R_1) & :- report(R_1), report(R_2), close(R_1,R_2) \\
parallel(L_1,L_2) & :- completeTraj(L_1), completeTraj(L_2), \\
 & \quad isParallel(L_1,L_2)
\end{array}
$$

Here, we use R to represent the triplet (x,y,t) signifying the location (x,y) and time t of vehicle detection, and compute vehicle trajectory paths from the base data *report(R)*. For simplicity, we assume that at any instant there is only one sensor detecting the target, so the *trajectory* can be directly synthesized using a sequence of *report* tuples. For clarity, we use lists instead of function symbols; the list notation $[X|Y]$ signifies X as the head-sublist and Y as the tail-element. We use two locally-processed built-in functions: *close* checks if two reports can be consecutive points on a trajectory (i.e., close enough in the spatial and temporal domains), and *IsParallel* checks if two trajectories are parallel. □

EXAMPLE 4. **Shortest-Path Tree.** Here, we give a logic program for constructing a shortest path tree (H) with a given root node (A). in a given network graph G. This example illustrates a more involved use of recursion and negation. Note that, for general graphs with cycles, the shortest path program cannot be written using just aggregates (without negations and/or function symbols).

logicH Program:

$$
\begin{array}{ll}
H(A,A,0). & \\
H(A,x,1) & :- G(A,x) \\
H'(y,d+1) & :- H(\_,y,d'), (d+1) > d', H(\_,x,d), G(x,y) \\
H(x,y,d+1) & :- G(x,y), H(\_,x,d),\ NOT\ H'(y,d+1)
\end{array}
$$

The predicate $H(x,y,d)$ is true if there is a path of length d from A to y using the edge (x,y); essentially, $H(x,y,d)$ gives the set of edges added in the breadth-first search at d^{th} level. The predicate $H'(y,d+1)$ is true if there is already a path from A to y of length shorter than $d+1$. The first two logic rules of the above program define the base cases. The third rule defines H'; "_" is the notation for anonymous or don't care variable, and the last two terms in the rule serve the purpose of bounding d (to ensure safety). The given logic program is more compact than the 20 lines of procedural code written in Kairos [18]. More importantly, it can be automatically translated into communication-efficient distributed code; we present more details in Section 4. □

Other Examples; Limitations of the Approach. In addition to the above examples, we were also able to write compact programs for vehicle tracking algorithm based on DARPA NEST software [43, 44] and multilateration localization algorithm by Savvides et al. [38]. As with any programming framework, deductive programming has its own limitations. In particular, logic programs are sometimes non-intuitive or difficult to write; e.g., the shortest tree program of Example 4 is clean and compact, but quite non-intuitive compared to a procedural code. As such the deductive framework is targeted towards expert and trained users, for whom the relief from worrying about low-level hardware and optimization issues would far offset the burden of writing a logic program.

3 Query Evaluation in Sensor Networks

In this section, we discuss our techniques for in-network evaluation of deductive queries.

Prior Work on In-Network Query Evaluation. The traditional distributed query processing algorithms are not directly applicable to sensor networks due to their unique characteristics. There has been a lot of work done on distributed query processing for streaming data [1, 39]; however, they do not consider resource-constrained networks and hence, minimizing communication cost is not the focus of these works. As mentioned before, the current current sensor network database engines (TinyDB [32], Cougar [7]) implement a limited functionality of SQL. All of the above works are for distributed evaluation of SQL, which is less expressive than our proposed deductive framework. Recently, Loo et al. [25] presented distributed evaluation of positive (without negations) datalog programs with localized joins in a general network with no resource constraints. In contrast, for our purposes, we need to evaluate logic programs with negations involving non-localized joins in networks with limited memory and energy resources.

Our Query Evaluation Approach, and Section Organization. We use the bottom-up (instead of top-down) approach [41] of evaluating deductive queries, as in [25], since the bottom-up approach is amenable to incremental and asynchronous distributed evaluation, and has minimal run time memory requirements beyond storage of intermediate results. At a high-level, our strategy for in-network evaluation of deductive queries is as follows. Firstly, evaluation of a single deductive rule without negations is tantamount to computing a join (cartesian product followed by a selection) of tables. Thus, in Section 3.1, we discuss techniques for in-network implementation of join of multiple data streams. In Section 3.2, we generalize the implementations of join to evaluation of general deductive programs with recursion (but without negations). The generalization works by hashing derived tuples to specific network nodes, and thus, eliminating duplicates and creating derived data streams. In Section 3.3, we consider evaluation of deductive programs with negations. We observe that maintenance of general (safe) non-recursive programs with negations boils down to maintaining a join-query result in response to deletions to the operand streams. The above generalizes to XY-stratified and locally non-recursive programs, which combine recursion and negations in an involved manner, and are useful in the context of programming sensor networks. We discuss aggregations, function symbols and hashing schemes in Section 3.4, and present the system architecture in Section 3.5.

3.1 In-Network Implementation of Join

As mentioned above, the join operation is at the core of the bottom-up evaluation of deductive queries. Thus, in this section, we address the problem of in-network implementation of join of multiple data streams. In particular, we develop the *Perpendicular Approach* which is communication-efficient, load-balanced, fault-tolerant, and immune to certain topology changes. Here, we give a brief overview of our designed join implementations; more detailed discussion and analysis is presented in our concurrent work [47]. We start with a formal problem description and a definition.

Problem Description. Given n data streams $R_1, R_2, \ldots, R_n$ (not necessarily distinct) in a sensor network, we wish to compute $R_1 \bowtie R_2 \bowtie \ldots R_n$. Here, each given stream is being generated in a distributed manner across the network, and the symbol $\bowtie$ represents the database binary-join operator which essentially involves taking a cartesian product of the operand tables followed by a selection based on the join conditions. The join conditions may be arbitrary; however, we give special consideration to spatial joins (formally defined below). The join-query result tuples may be output arbitrarily across the network, since they will anyway be hashed appropriately for further use of the join-query result. As suggested before, the join operation is constrained to the join of sliding-windows of operand streams.

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 node generating the tuple, and $|x-y|$ is the distance between x and y.

Due to the inherent spatial correlation in sensor network data [12], the join predicates in sensor network queries generally involve spatial constraint (see Examples 1-4). □

Naive Broadcast Approach. The simplest way to implement a join of multiple data streams is the *Naive Broadcast Approach* wherein each generated tuple is broadcast to the entire network, and stored at each network 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. Moreover, only $n-1$ of the operand streams need to be broadcast (here, n is the total number of operand streams). The above approach can be very efficient for joins with very small spatial ranges. Otherwise, the approach is expected to be infeasible in most other cases due to severe memory constraints. Another simple approach is the *Local Storage Approach*, wherein each tuple is stored only at the generating node. However, the join-computation process requires propagation of "partial results" (as discussed later for Perpendicular Approach).

Perpendicular Approach (PA) in Grid Networks. We now describe our main approach, viz., Perpendicular Approach (PA), for in-network implementation of join. In this article, we give only the basic idea of the approach by describing how it works on 2D grid networks. The approach can be generalized to arbitrary topologies as described in [47]. We start with describing it for a join of two data streams.

PA for Two Streams in Grid Networks. Consider a *2D grid network* of size $m \times m$, which 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. Two nodes can directly communicate with each other iff they are within a unit distance of each other. Now, consider two data streams R_1 and R_2 in the above network, and a tuple t (of either data stream) generated at coordinates (p,q). PA consists of *two phases*, viz., storage and join-computation.

- *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.
- *Join-computation Phase*: In the *join-computation* phase, we route t along the p^{th} vertical line to compute

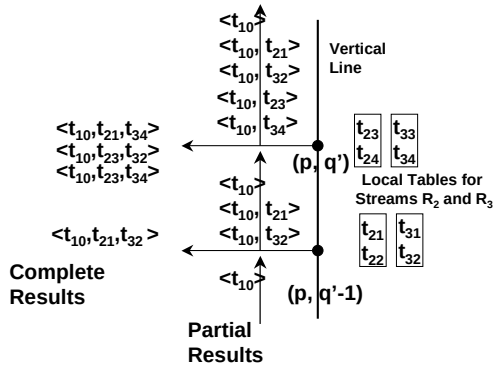

Figure 1. One-Pass Join Computation. Here, $t_{i*} \in R_i$, and we assume the join conditions to be such that t_{10} matches only with $t_{21}, t_{23}, t_{32}, t_{34}$. Also, there is no join condition between R_2 and R_3.

the result tuples due to t (i.e., $t \bowtie R_2$ or $t \bowtie R_1$ 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.

Maintenance of sliding-windows and handling of simultaneous updates is discussed below, in the more generalized context of multiple data streams.

PA for Multiple Streams in Grid Networks. We now generalize PA 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, as described below. We start with a definition.

DEFINITION 2. (Partial Result.) Let $R_1, R_2, \ldots, R_n$ 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 $T \in (t \bowtie R_{i_1} \bowtie R_{i_2} \ldots R_{i_k})$ where $k < n-1$ and $i_l \neq j$ for any l. The tuple t is also considered a partial result (for the case when $k = 0$). If $k = n-1$, then T is called a *complete result*. □

Join-computation Phase. 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, 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 1. 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.

Simultaneous Insertions and Sliding Windows. To correctly handle simultaneously generated tuples across the network, we should start the join-computation phase for a tuple only after the completion of a storage phase. Thus, we introduce a delay of τ_s between the start of two phases, where τ_s is the upper bound on the time to complete a storage phase. To maintain sliding-windows, tuples can be expired after a storage time of $(\tau_s + \tau_j + \tau_w)$, where τ_j is the upper bound on the completion time of a join-computation phase and τ_w is the sliding-window range.[1] We omit the proof of the below theorem; we prove a more general claim in Theorem 3.

THEOREM 1. *Given data streams $R_1, R_2, \ldots, R_n$ in a 2D grid sensor network, the Perpendicular Approach (PA) correctly maintains the join-query result $R_1 \bowtie R_2 \bowtie \ldots R_n$, in response to distributed (and possibly, simultaneous) insertion of tuples into the data streams. We assume bounded message delays, so as to be able to bound the completion times of storage and join-computation phases.* ■

PA in General Networks. Generalization of PA to networks with arbitrary topology requires developing an appropriate notion of vertical and horizontal paths such that each vertical path intersects with every horizontal path. Such a scheme is described in [47]. We state the below without proof.

THEOREM 2. *Given data streams $R_1, R_2, \ldots, R_n$ in a sensor network with arbitrary topology, the Perpendicular Approach (PA) correctly maintains the join-query result $R_1 \bowtie R_2 \bowtie \ldots R_n$, in response to distributed (and possibly, simultaneous) insertion of tuples into the data streams. We assume bounded message delays.* ■

To incorporate spatial constraints, we can store each tuple over only an appropriate part of the horizontal path, and similarly, traverse only an appropriate part of the vertical path. The above results in storage and communication cost savings. Finally, PA easily generalizes to attributes with arbitrary terms involving function symbols, since the join conditions are evaluated only locally at each node.

3.2 Deductive Queries without Negation

In this section, we discuss in-network evaluation of deductive queries without negations. We start with discussing storage of derived tuples, which is key to uniform treatment of derived and base streams.

Hashing Derived Tuples; Derived Data Streams. The join implementations (in particular, the Perpendicular Approach) discussed in the previous subsection generate the result tuples across the network in some arbitrary manner. However, for efficient elimination of duplicates (we store derived tables as *sets*), we need to hash and store the derived tuples across the network such that identical derived tuples are stored at same (or close-by) nodes. We can use well-known geographic hashing schemes for above (see Section 3.4). The above hashing and storage scheme facilitates transformation of each derived table into a data stream for evaluation of higher-level predicates. Essentially, a derived tuple t is considered to be *generated* (just like a base fact) at the *hashed location* at its first instance; later duplicates of t are not insertions to the derived table, and hence, not considered as generations.

Evaluation of Deductive Queries without Negations. Consider a predicate Q represented by multiple deductive rules (without recursion and negations) over base predicates and with common head predicate Q. Evaluating the predicate Q is tantamount to independent maintenance of each rule of Q and then, taking a union of the derived results (for each rule). Here, the union of the derived results is facilitated by the storage scheme described in the previous paragraph. The above is easy to generalize to include recursion, since a recursive subgoal can be treated just like a subgoal of another

[1] For simplicity, we assume the range of the sliding-window to be in terms of a global clock, or that all nodes have a synchronized clock. We relax the assumption in Section 3.3.

predicate. Finally, the above also generalizes to evaluation of multiple predicates in an arbitrary recursive deductive program without negations, due to generation of each derived result as a data stream as described in the previous paragraph. Above, we have assumed that the base data streams are insert-only (i.e., a previous generated base tuple is never deleted), as is typically the case since a base data stream generally represents a stream of sensing readings.

3.3 Deductive Queries With Negations

In this subsection, we generalize the Perpendicular Approach of Section 3.1 to evaluate deductive queries with negations. The generalizations and techniques developed here will apply to any in-network implementation of join that has independent storage and join-computation phases; e.g., the developed techniques will apply to Naive Broadcast and Local Storage Approaches.

High-Level Plan. Theorem 2 states that Perpendicular Approach (PA) correctly computes a join-query result in response to simultaneous insertions. Here, we first generalize PA to maintenance of a join-query result in response to deletions to the operand streams. Even though the base operand streams may only be insert-only streams, generalizing PA to handle deletions is fundamental to generalizing it for evaluation of deductive programs with negations. As a second step, we generalize PA to evaluate a predicate represented by a single deductive rule involving negated subgoals. As a third step, we generalize PA to general non-recursive deductive programs. We then generalize our scheme to evaluation of XY-stratified and locally non-recursive deductive programs, which incorporate a restricted form of combined recursion and negation and are useful in the context of sensor networks. Finally, we briefly discuss evaluation of general stratified deductive programs.

Generalizing PA to Handle Deletions. Consider data streams $R_1, R_2, \ldots, R_n$ in a sensor network. Let $R_1, R_2, \ldots, R_n$ also denote the *current* sliding windows of respective data streams, and let the join-query result $R_1 \bowtie R_2 \ldots \bowtie R_n$ be stored (as a set, without duplicates) in a distributed manner across the network based on some hashing scheme (as discussed in the previous subsection).

Various Possible Techniques. Let us consider deletion of a tuple t_1 from the stream R_1. For now, lets assume that there are no other insertions or deletions. To maintain the join-query result, we need to compute $t_1 \bowtie R_2 \ldots \bowtie R_n$ and "delete" it from the maintained join-query result. However, due to set semantics, $(R_1 - t_1) \bowtie R_2 \ldots \bowtie R_n$ may *not* be equal to $(R_1 \bowtie R_2 \ldots \bowtie R_n) - (t_1 \bowtie R_2 \ldots \bowtie R_n)$. We can attempt to solve the above problem using one of the following techniques: (i) Store the query result as a bag, or keep a count of multiplicity of each result tuple as suggested in [19], (ii) Keep the actual set of derivations (as described later) for each result tuple, or (iii) Use the rederivation technique of [19]. The counting technique (or bag semantics) is difficult to implement accurately for a fault-tolerant technique such as Perpendicular Approach, since fault-tolerance yields non-deterministic duplication of result tuples. The rederivation technique of [19] will require distributed computation of maintenance queries, and hence, will result in a lot of communication overhead. However, the technique of keeping the actual set of derivations (as described below) incurs no additional communication overhead and guarantees correctness. Storage of set of derivations does incur a space overhead, which may be minimal if most tuples have only a small number of derivations.

DEFINITION 3. (Source Node; Tuple ID; Derivation of a Tuple) The *source node* of a tuple is the node in the network where the tuple is generated. Note that a derived tuple is considered to be generated at its hashed location. We use $I(t)$ to denote the source node of a tuple t.

The *tuple-ID* is an identifier that uniquely identifies each tuple in a (base or derived) data stream. For our purposes, we use $(I(t), \tau)$ as the ID of a tuple t, where τ is at local timestamp at $I(t)$ when the tuple t was *inserted*.

A *derivation* of a derived tuple t is the *list* of tuple IDs, one from each of the operand streams, that match/join to yield t. Note that a tuple may have multiple different derivations. In a general deductive program, a derivation of t includes the rule-ID used to derive the tuple, but does *not* include the tuple IDs corresponding to negated subgoals. In case of recursive rules, a derivation may include a tuple-ID from the same table as t. □

Set-of-Derivations Approach. Now, to accurately maintain $T = R_1 \bowtie R_2 \ldots \bowtie R_n$ in response to deletions from an operand stream, we store (and maintain) *set* of all derivations with each tuple in T. When a tuple t_1 is deleted from R_1, we compute $T_1 = t_1 \bowtie R_2 \ldots \bowtie R_n)$ along with the derivation of each tuple in T_1. Then, for each derived tuple t in T, we subtract the set of derivations of t in T_1 from the set of derivations of t in T. Set of derivations are similarly maintained in response to insertions into operand streams. The tuple t is deleted from (inserted into) from T if the resulting set of derivations of t becomes empty (non-empty from empty). The computation of T_1 constitutes the join-computation phase for deletion of t_1. In the storage phase, the tuple t_1 is *removed* from all the nodes where it was stored in the storage phase of its insertion (i.e., from all the nodes on the horizontal path from its source node, in case of PA). We use the term *removal* of a tuple to signify removing the replications of a tuples (stored in the storage phase of a join implementation); in contrast, *deletion* of a tuple refers to an actual deletion of the tuple from its table. Note that deletion of a derived tuple occurs only at its source node (due to the hashing scheme).

Simultaneous Updates. It is easy to see that the above technique correctly maintains the join-query result T in response to updates (insertions or deletions) to operand streams, if the updates occur one at a time. To incorporate simultaneous updates, we start the join-computation phase after a delay of $\tau_s + \tau_c$ where τ_s is the upper bound on the completion of a storage phase and τ_c is the maximum difference between local clocks of two nodes. The above delay allows us to essentially process the updates in the order of their *local* timestamps. We prove the correctness of the above strategy in a more general context in Theorem 3.

Deductive Rule with Negated Subgoals. We now generalize our approach to maintain a query result T represented by a safe[2] deductive rule with negated subgoals. Let

$$T :- \ R_1, \ldots, R_n, \ NOT\ S_1, \ldots, NOT\ S_m$$

[2]In a safe rule, each variable in the rule must appear in a non-negated relational subgoal of the body.

Above, each R_i or S_j (not necessarily distinct) is a data stream in the sensor network. As mentioned in Definition 3, a derivation of a tuple contains tuple IDs from only the non-negated subgoals; in a safe rule, such a derivation uniquely determines the derived tuple. Now, to maintain T, in response to an isolated insertion or deletion t_{r1} into the stream R_1, we first compute

$$T_{r1} \text{ :- } t_{r1}, R_2, R_3, \ldots, R_n, \; NOT \; S_1, \ldots, S_m,$$

along with the derivation of each tuple in T_{r1} as follows. Essentially, in the join-computation phase, we compute and propagate partial results of $t_{r1} \bowtie R_2 \bowtie \ldots \bowtie R_n$ (join of only the non-negated subgoals), and delete partial or complete results that match with a tuple from *any* S_j. Then, we set-union (for insertion t_{r1}) or set-minus (for deletion t_{r1}) the set of derivations in T_{r1} from the original set of derivations in T. Similarly, to process an isolated insertion or deletion t_{s1} from S_1, we first compute T_{s1} :- $R_1, \ldots, R_n, t_{s1}, NOT \; S_2, \ldots, NOT \; S_m$, and then union (for deletion t_{s1}) or minus (for insertion t_{s1}) the set of derivations in T_{s1} from the original set of derivations in T. It is easy to see that the above correctly maintains T in response to isolated insertions or deletions from the operand streams. To maintain T in face of simultaneous updates across the network, we use the following strategy.

- We start the join-computation phase of any tuple after a delay of $\tau_s + \tau_c$ from its time of generation, where τ_s is the upper bound on the storage-phase time and τ_c is the maximum difference between the local clocks of any two nodes.
- During the join-computation phase of a tuple t generated at local time τ, we match/join t with only those tuples that have a local timestamp between τ and $(\tau - \tau_w)$ *and* have not been deleted before the local timestamp τ. Here, τ_w is the given range of the time-based sliding-window. The above strategy is to process the updates in the order of their local timestamps. To facilitate the above, during the storage-phase of a tuple deletion, we do *not* remove the replicated copies of the tuple from the nodes, but instead store the local timestamp of its deletion.

In conjunction with the above strategy, we can maintain sliding-windows by expiring a tuple after a *storage time* of $(\tau_s + \tau_c) + \tau_j + (\tau_w + \tau_c)$. Here, the first term of $(\tau_s + \tau_c)$ is due to the delay in starting a join-computation phase, the second term τ_j is the upper bound on the completion time of a join-computation phase, and the last term $(\tau_w + \tau_c)$ is the absolute (in terms of a global clock) range of the sliding-window.

We now prove the correctness of the above outlined strategy.

THEOREM 3. *The above described strategy correctly maintains the query result*

$$T \text{ :- } R_1, \ldots, R_n, \; NOT \; S_1, \ldots, NOT \; S_m,$$

in face of simultaneous updates (insertions or deletions) to the given operand streams, under the assumption that τ_s, τ_j, and τ_c (as defined above) are bounded and there are no message losses.

Proof: Consider a tuple t_{r1} that is inserted into or deleted from R_1 with a local timestamp of τ. Let R_i $(1 \leq i \leq n)$ or S_i $(1 \leq i \leq m)$ refer to the sliding-window of the respective data stream consisting of all (and only those) tuples with a *local* timestamp of less than τ and more than $(\tau - \tau_w)$, and not deleted locally before τ. To prove the theorem, we essentially need to show that during the join-computation phase of t_{r1}, the set of nodes encountered by t_{r1} during the join-computation phase collectively contain the sliding-windows R_i $(2 \leq i \leq n)$ and S_i $(1 \leq i \leq m)$. The above will show that T_{r1} :- $t_{r1}, \ldots, R_n, \; NOT \; S_1, \ldots, NOT \; S_m$ is correctly computed for the update of tuple t_{r1} into R_1. Updates into other streams occur in a similar manner. By the definition of the sliding-windows R_i $(1 \leq i \leq n)$ and S_i $(1 \leq i \leq m)$, the above claims will prove that simultaneous updates across the network are correctly handled in the *order of their local timestamps*, which proves the theorem.

To show that the set of nodes encountered by t_{r1} during the join-computation phase collectively contain the defined sliding windows, observe the following. Firstly, the delay of $\tau_s + \tau_c$ before the join-computation of t_{r1}, guarantees that before the join-computation of t_{r1} starts, storage phase of all tuples with a local timestamp of less than τ has been completed. Secondly, the storage time of $(\tau_s + \tau_c) + \tau_j + (\tau_w + \tau_c)$ ensures that the replications of matching tuples do not expire before the completion of the join-computation phase of t_{r1}. ∎

Multiple Rules with Same Head Predicate. In the above paragraphs, we have outlined a generalized Perpendicular Approach that maintains a query result represented by a single non-recursive deductive rule with negated subgoals, in response to simultaneous insertions or deletions to operand tables. Such a scheme can be easily generalized to maintain a query result represented by multiple non-recursive deductive rules (with negated subgoals) with the same head predicate. Essentially, we assign a unique ID to each deductive rule, and include the rule-ID in the derivation of each result tuple (as suggested in Definition 3). Then, maintenance of multiple deductive rules becomes equivalent to maintaining each rule independently.

Non-Recursive Single-Stratum Deductive Programs. Above, we have described that our query evaluation scheme can maintain a query represented by (a union of) multiple deductive rules with negations. Generalization of our evaluation scheme and the argument of its correctness is straightforward for non-recursive *single-stratum* deductive program. In a single-stratum deductive program, negation is only over the base (not derived) data streams. Essentially, each derived predicate is generated as a derived data stream, as suggested in the previous subsection, which allows each derived predicate to be handled in a similar manner as a base predicate. The correctness of the approach follows from Theorem 3, and the following two observations. First, the number of derivations of any derived tuple always remain finite (for programs without function symbols, or programs with finite number of derived tuples). Second, each derivation of a result tuple yields a valid "proof tree" with leaves as base tuples.[3] The second observation holds because the program is non-recursive, and hence, the proof tree constructed (by

[3] A *proof tree* [41] of a derived tuple t describes how the tuple t is constructed from the base tuple; an interior node in the tree corresponds to an intermediate derived tuple, and a node r's children are the tuples used to derive r using a single rule in the program.

iteratively "unfolding" the derivations) will have no directed cycles.

General Non-Recursive Logic Programs. It is interesting to note that the above scheme also works for general non-recursive programs due to the following observation. Consider the set of derived predicates $\mathcal{P}_n$ in the n^{th} stratum.[4] By the definition of strata, each negated subgoal in (a rule defining) $\mathcal{P}_n$ is over a predicate from a lower stratum. Moreover, the lower-strata predicates can be essentially looked upon as base predicates for a higher-stratum. In other words, higher-strata predicates are essentially recursive programs with negation over only lower-strata predicates which can be considered as base predicates for higher-strata predicates. Thus, higher-strata predicates can be maintained due to updates (insertions or deletions) in the lower-strata predicates exactly as outlined in previous paragraphs. Note that the change in the set of derivations for each tuple in a lower-stratum predicate is *not* required to be propagated to the higher-strata predicates; we only need to propagate actual insertions (when the set of derivations changes from empty to non-empty) and actual deletions (when the set of derivations becomes empty) to the higher-strata predicates.

The above facilitates asynchronous computation of fixpoint, i.e., we don't need to wait for the fixpoint of lower-strata predicates to be reached (which never happens, due to the streaming base data) before evaluating higher-strata predicates. However, a deduced fact in a higher-strata predicate may have to be later retracted/deleted due to updates in the lower-strata; or, we could wait for certain time before "finalizing" a fact. The latter is acceptable/reasonable due to bounded size sliding-windows and implicit temporal correlation in the sensor data. Our correctness arguments and claims essentially guarantee that the fixpoint will eventually be reached if and when the insertions to the base data streams cease.

Combining Recursion and Negation – Evaluating *XY*-stratified Programs. Evaluation of logic programs with unrestricted negation and recursion is infeasible in sensor networks, since it will require a series of distributed fixpoint checks for evaluation of well-founded semantics [4]. However, the strategy outlined in previous paragraphs for evaluation of general non-recursive programs easily generalizes to evaluation of *XY-stratified* programs [46], wherein the derived tables can be partitioned into "sub-tables" such that the dependency graph[5] on the sub-tables is acyclic. The partitioning of tables into sub-tables is generally based on the ordering imposed by built-in arithmetic functions on the argument values of the derived tuples. For instance, consider the *logicH* program of Example 4. Let the predicate/table H be partitioned into sub-tables $H_1, H_2, \ldots$, based on the the value of the third argument, i.e., let H_d represent the sub-table consisting of all the facts $H(\_, \_, d)$. Similarly, let H'_d denote the sub-table consisting of all the facts $H'(\_, d)$. Now, the dependency graph of the sub-tables is acyclic, since there exists a topological order $(H_0, H'_1, H_1, H'_2, H_3, \ldots,)$ of the sub-tables. Thus, the *logicH* program is *XY*-stratified.[6] Similarly, it is easy to see that the program of Example 1 is also *XY*-stratified; the program of Example 3 can be considered *XY*-stratified if the *traj* table is partitioned based on the path length. The concept of *XY*-stratification is particularly useful in the context of sensor network because of the ordering imposed sometimes by timestamp attribute. In-network evaluation of *XY*-stratified programs is done using the same strategy as outlined in previous paragraphs for evaluation of general non-recursive programs. Note that the concept of sub-tables and the acyclicity of their dependency graph is only to prove correctness of the evaluation scheme; the evaluation scheme is oblivious of the sub-tables and their strata.

Evaluating General Recursive Programs. Recall that the correctness of set-of-derivations approach (for maintenance of query results in response to deletions into operand streams) hinges on the fact that each remaining derivation of a derived tuple indeed yields a valid proof tree. A derivation of a tuple is *guaranteed* to yield a valid proof tree only if there are no directed cycles in the tree constructed by unfolding the derivations. Thus, general recursive programs (even with stratified negations) cannot be evaluated using our set-of-derivations approach, since a non-empty set of derivations of a tuple may not necessarily mean that there exists a valid proof tree for the tuple. However, our evaluation scheme outlined in the previous paragraphs will correctly evaluate programs as long as there are no cycles in the "derivation graph" of the derived tuples, i.e., for *locally non-recursive* programs [9].

For evaluation of general recursive programs (with stratified negation), we really need to employ some variant of the rederivation approach [19]. The rederivation approach in our context will essentially consists of two steps: First, temporarily delete a tuple if the set of derivations *reduces*, and then, check if the temporarily-deleted tuple can be derived from the existing base tuples (i.e., whether it has a proof tree). Execution of the second step requires evaluation of a maintenance query over the network, and hence, may incur additional communication overhead. In our future research, we plan to address the above challenge of efficient in-network evaluation of general stratified deductive programs.

3.4 Generalization and Hashing Schemes

Built-in Functions, Function Symbols, and Aggregations. Our query evaluation scheme can be easily generalized to handle built-in functions, since the evaluation of join-conditions and execution of built-in functions is done only locally. For the same reason, incorporating function symbols in deductive rules only requires extending the evaluation of join-condition using the term-matching operator [41]. However, introduction of function symbols in deductive programs may result in non-termination of programs and may make optimizations difficult; but we anticipate that function symbols will be used in limited contexts, and hence, their use can be allowed in the programming framework for full expressibility.

Aggregates are typically represented in logic rules by using the Prolog's all-solutions predicate to construct a list of

[4] *Stratum* of a derived predicate Q is defined recursively as one plus the maximum stratum of any predicate R such that there is a rule with Q as the head and R as a negated subgoal. Base predicates are defined to have a stratum of zero.

[5] In the dependency graph, an edge exists from a predicate P to a predicate Q if there exists a rule in the program whose head is P and body contains Q.

[6] Our defined notion of *XY*-stratified programs is slightly more general than the original notion defined in [46].

values to be aggregated, then, computing the desired aggregate. However, an efficient *implementation* should aggregate the elements iteratively (for incremental aggregates) without actually constructing the list. Thus, we can use TAG [28] or fault-tolerant synopsis diffusion [33] techniques for incremental aggregates (without actually constructing the list). For non-incremental aggregates, we need to first construct the list.

Hashing Schemes. In the previous subsections, we suggested that each derived table is hashed across the network using a geographic hashing scheme. However, choice of a hashing scheme can have a substantial effect on the incurred communication cost during the join implementation. For instance, if the derived table is involved in a join query wherein the join-predicate involves a range predicate, then we can use the join-attribute values to hash the tuples, for efficient computation of the range-join. If a derived table is part of multiple join-queries involving different join-attributes, then the table tuples may need to be hashed to multiple locations (one for each join-attribute), which may result in prohibitive usage of main-memory. In general, if a join-attribute refers to a node or its location, then it is natural to map/hash a tuple t to the corresponding node. For simplified decisions on hashing schemes, we let the user dictate the hashing scheme (i.e., the join-attribute used for hashing) of each derived predicate, as is suggested in prior works [15, 25, 27] using a concept of host-id attribute. Choice of hashing scheme (i.e., choice of attribute used for hashing a derived result) to minimize the overall communication cost is a challenging problem, and would be addressed in our future works.

3.5 System Architecture and Resource Requirements

In this subsection, we give an overall architecture of our system for in-network processing of logic queries, and address memory requirements of our system.

Overall Architecture. Figure 2 depicts our overall system architecture and high-level plan of in-network evaluation of logic queries. Basically, the user specified logic-program is first optimized using magic-set transformations [41] (used to optimize the bottom-up evaluation strategy), and then translated into appropriate code which represents distributed bottom-up incremental evaluation of the given user program. The compiled code is downloaded into each sensor node. Within each sensor node, there is a layer of in-network implementations of relational operators (such as join), aggregates, and built-in predicates/functions. The above layer is in addition to the usual layers of routing and networking layers.

Memory Requirements. Currently available sensor nodes (motes) have 4 to 10 KB of RAM and 128 KB or more of on-chip flash memory. The memory capacities have evolved over years [36], and latest Intel mote is being designed with 64 KB RAM [22]. In our system, the user program essentially consists of the generic join interface (as described in Section 4), the list of join-conditions for the deductive rules, and the (procedural) code for the system/user built-in functions. This is in addition to the other networking layers. A typical on-chip flash memory is ample to easily contain the native code of a user program and various system layers.

Overall Main Memory Usage. The strain on sensor nodes' main memory is due to (i) run-time control structures used by

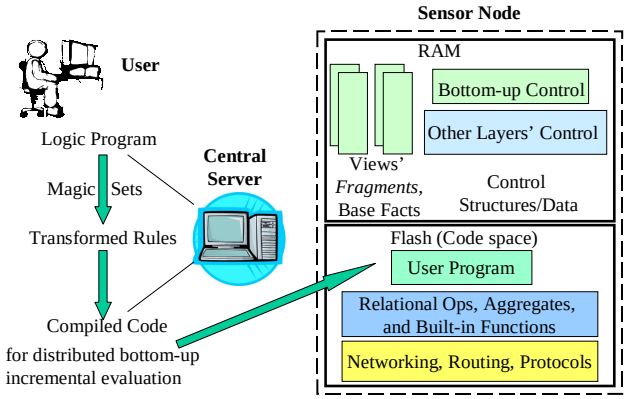

Figure 2. System Architecture.

various system layers and the bottom-up approach, and (ii) derived results, (i.e., storage of derived tuples and the corresponding set of derivations, and their replication to facilitate efficient join computation) during execution of user program. The bottom-up query evaluation approach requires minimal run-time control structures beyond those needed for setting up indexes and joins. Note that the list of join-conditions, being read-only part of the user program, can reside on the on-chip flash memory.

Derived Results. The derived/intermediate tables are stored in a distributed manner across the network. So, the *total main memory available for storing derived tables is the cumulative main memory of the entire network.* Thus, we expect the available main memory resources to be sufficient for most user programs. For instance, for the shortest-path tree program of Example 4, the derived results are H (or J, for the improved *logicJ* program given in Section 4) and H', and based on the storage scheme discussed in Section 4, each node y stores only tuples of the form $H(\_, y, \_)$ (or $J(x, \_)$) or $H'(y, \_)$ where x is a neighbor of y. Thus, the total number of tuples stored at any node is at most 2 to 3 times its degree. In a stable state, each node contains a single tuple of H.

In general the storage and replication of intermediate results (materialized views) is required for communication efficiency and is inherent to the user program, rather than the programming framework. Since tables are maintained as sliding-windows, the space required for intermediate results can be adjusted depending on the available memory and desired accuracy of results. In addition, the techniques for selecting which of the intermediate results to store (i.e., selection of views to materialize) can be used to further satisfy the given memory constraints while maintaining sufficient accuracy of results. In our future research, we plan to investigate to above space optimization techniques to further optimize the main memory usage based on given performance objectives and resource constraints.

Computation Load. Most of the processing in our system is in the form of distributed evaluation of logic rules or local built-in functions. The bottom-up evaluation of logic rules requires simple local operations such as term-matching [41], join-predicate evaluations, arithmetic comparisons, etc., and hence, result in minimal processing load. The processing load due to arithmetic-intensive local built-in functions is inherent to a user program, i.e., largely independent of the programming framework. Thus, our overall framework and approach is not expected to increase the processing load on

the network.

4 System Implementation and Performance Evaluation

In this section, we present details of our current system implementation, and present performance results that illustrate the feasibility of our proposed approach and query evaluation techniques.

Current System Implementation. The main focus of our system is to automatically translate high-level user program written in form of deductive rules to nesC code that runs on individual sensor nodes. The generated code must represent our outlined query evaluation strategy. In particular, we translate a given user program into distributed nesC code as follows. First, we developed nesC interface components for various in-network join implementations corresponding to the Naive Broadcast, Local Storage, and Perpendicular Approaches as described in Section 3.1. These components reside on each node, and are very generic, i.e., do not need to be generated for a specific user program. Any given user (deductive) program is now translated into the database schema (list of predicates and attributes) and the list of deductive rules (i.e., the list of subgoals and join conditions for each rule). The list of rules and join-conditions are used by the generic join component to evaluate the predicates in the program. See Figure 3. Our current version of the system can handle general deductive programs without function symbols. In addition, the current implementation handles simple arithmetic built-in functions and predicates such as addition, subtraction, equality, less than, etc. In the current implementation, the hashing scheme of the derived results (i.e., the choice of join-attribute to use for geographic hashing) is given by the user.

The current implementation has been tested on TOSSIM [24], and the Perpendicular Approach join implementation is based on a 2D grid topology. In the immediate future implementations, we plan to (i) incorporate the generalized version of Perpendicular Approach join-implementation for arbitrary topologies as developed in [47], and (ii) incorporate use of arbitrary user-defined built-in functions (written in procedural code), aggregations, and function symbols.

Comparison of Program Sizes. In general, the deductive programs are expected to be much shorter and compact (few logic rules) compared to the corresponding nesC code. However, logic programs are sometimes non-intuitive to write. As such deductive framework is targeted towards expert and trained users, for whom the relief from worrying about low-level hardware and optimization issues would far offset the burden of writing a logic program. The size of the generated/translated nesC code is of not much relevance to the performance comparison – since the translation is done automatically and a typical flash memory of a sensor node is ample to easily store the executable of resulting program code. In our framework, the generated code essentially includes the set of join conditions and the procedural code for user-defined built-in functions; the code for the join implementation is common to all user programs.

4.1 Performance Evaluation

In this subsection, we present our simulation results for implementation of the shortest path tree program of Example 4. The shortest path tree program of Example 4 incorporates quite a non-trivial combination of negation and recursion, and the resulting query evaluation algorithm is quite different from the native implementation. In contrast, as discussed below, the translated code for other examples naturally yields communication-optimal translations (essentially, the same algorithm as the algorithm of procedural code). Thus, to gain more insight into our proposed approach, we choose the challenging shortest path tree example, and study the performance comparison.

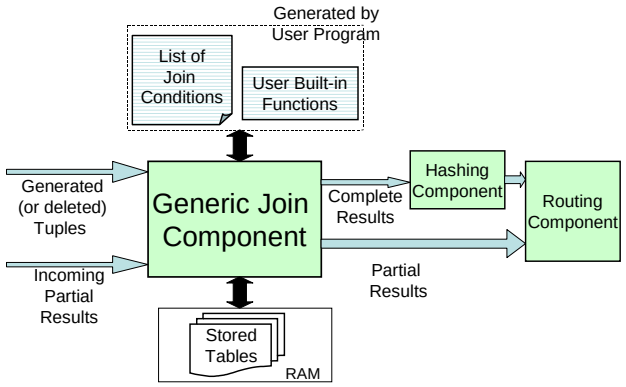

Figure 3. **The *join component* at a sensor node. Newly generated (base or derived) tuples are fed into the join component, which generates partial and/or complete results by joining with local tables. The complete results are sent to the *hashing component* for hashing, and then, forwarded to the *routing component* for storage at the hashed location. The partial results are also forwarded to the routing component to route to the next node on the join-computation path. Partial results received from other nodes are treated similarly. In addition, newly generated tuples are also routed for replication in the storage phase (not shown).**

Performance of Other Example Programs. It is easy to see that other examples of deductive programs from Section 2.3 naturally yield communication-optimal translations, with appropriate hashing schemes. For instance, for the program of Example 1, if we use a hashing scheme such that $P(i,\_,\_)$, $U(i,\_,\_)$ and $Z(i,\_,\_)$ are stored at node i, then each tuple needs to be broadcast to only its neighbors, which results in essentially the same algorithm and performance as the distributed code in a procedural language. Similar argument and analysis holds for the other programs shown in Section 2.3, and also for the programs of vehicle tracking based on DARPA NEST software and multilateration (not shown in the article).

Below, we start with discussing details of the distributed evaluation of *logicH* program, and then, present simulation results comparing the performance of the translated code with the procedural (nesC) program.

Distributed Evaluation of *logicH* Program. For convenience, we repeat the *logicH* program here; recall that *logicH* can handle general graphs with cycles.

logicH Program:

$$
\begin{array}{lll}
H(A,A,0). & & \\
H(A,x,1) & :- & G(A,x) \\
H'(y,d+1) & :- & H(\_,y,d'),(d+1)>d',H(\_,x,d),G(x,y) \\
H(x,y,d+1) & :- & G(x,y),H(\_,x,d),\ NOT\ H'(y,d+1)
\end{array}
$$

Hashing Schemes, and Join Strategy. The above *logicH* program produces a shortest-path tree rooted at node A. We assume that the fact $G(x,y)$ is available (stored) at both the

nodes x and y, which essentially means that each node is aware of its immediate neighbors. Since y is the only join-attribute in $H(x,y,d)$, a tuple $H(x,y,d)$ is hashed to the node y. Similarly, a tuple $H'(y,d)$ is hashed to the node y. Based on the above hashing scheme, all pairs of joining tuples reside in neighboring nodes.

Naive-Broadcast Approach. The Naive-Broadcast Approach of evaluating the joins in the third and four rules on *logicH* entail that one of the tables be broadcast to neighboring nodes, while the other table be stored locally at each node. Thus, we broadcast and store each tuple $H(\_,y,\_)$ at all the neighbors of y. Thus, the above approach requires only one message transmission for each update (insertion/deletion of tuple) into H tables. The above hashing scheme and broadcast strategy means that $H'(y,\_)$ and $H(\_,y,\_)$ can derived at their hashed locations itself. Thus, the *only* communication cost incurred in the entire evaluation of the logic program is the replication of each H to the neighbors of the generating node.

Perpendicular Approach. In the Perpendicular Approach, tuples are stored and propagated for join-computation along horizontal/vertical paths as discussed in Section 3.1. During the join-computation phase of a rule involving a negated subgoal, we first compute the complete result corresponding to the positive subgoals, and then, check for existence of tuples corresponding to the negated subgoals at the hashed location. For instance, in the case of the *logicH* program, $H(\_,y,d)$ is inserted after checking if there is a $H'(y,d)$ at y.

Distributed Evaluation. For the *logicH* program, initially, the node A generates the fact $H(A,A,0)$ using the first logic rule, and each neighbor x of A then generates a fact $H(A,x,1)$ using the second rule. Recall that derivation of a new fact is looked upon as generated at its *hashed* node. Thus, based on our hashing strategy, the fact $H(A,A,0)$ is considered to be generated at node A, and $H(A,x,1)$ is considered to be generated at node x. In the Naive-Broadcast approach suggested in previous paragraph, each insertion or deletion of an H tuple is broadcast to the neighbors of the generating node. Such a broadcast of an $H(\_,\_,l)$ tuple may result in new derivations of some $H'(\_,l+p)$ tuple ($p>0$; due to the third logic rule) and/or some $H(\_,\_,l+1)$ tuple (due to the fourth logic rule). If the set of derivations of a tuple t becomes non-empty from empty in the above process, then the tuple t is considered to be an insertion to the corresponding table. Insertion of a $H'(\_,l)$ tuple may result in deletions of a tuple $H(\_,\_,l)$ due to the fourth logic rule. Since the given program is XY-stratified with finite strata (bounded by the diameter of the network), the above process is guaranteed to terminate to a fixed point.

Optimization. The *logicH* program for shortest path tree can be optimized by a simple aggregation or "pushing down projection." Note that the evaluation of the third and fourth logic rules in *logicH* is independent of the value of the first argument of the subgoal predicates H. Thus, we do not need to process an insertion $H(z,x,d)$, if there already exists a tuple $H(z',x,d)$. Thus, we need to only process insertions or deletions of $J(y,d)$ where $J(y,d) :- H(x,y,d)$. We can thus rewrite the *logicH* program as follows.

Simulation Results. We now compare the performance of our translated/generated code for the *logicH* and *logicJ* programs with the optimized distributed code written in nesC.

logicJ Program:

$H(A,A,0).$
$H(A,x,1) :- G(A,x)$
$J(y,d) :- H(x,y,d)$
$H'(y,d+1) :- J(y,d'),(d+1)>d',J(x,d),G(x,y)$
$H(x,y,d+1) :- G(x,y),J(x,d),\ NOT\ H'(y,d+1)$

Simulation Setup, Programs, and Performance Metrics. We run our simulations using the TOSSIM simulator on a sensor network with a grid topology. Unless being varied, the total number of nodes is chosen to be 49 (in a 7×7 grid network). In certain simulations, we vary the message loss probability to compare the robustness of various approaches. We simulate a message loss probability of P by ignoring a message at the receiver with a probability of P.

We compare the performance of various programs using two performance metrics, viz., the *result inaccuracy* and *total communication cost*. Here, we define the *result inaccuracy* as the ratio of the number of missed shortest paths over the total number of shortest paths computed by a centralized program.

In our simulations, we compare the performance of three programs: *the procedural code* (optimized distributed nesC code), *logicH* (generated code for the *logicH* program, and *logicJ* (generated code for the *logicJ* program). In the generated codes, we use the Naive Broadcast Approach for join computation, because of the join involved involve only a 1-hop spatial constraint. Later, we will show the effectiveness of the Perpendicular Approach of computing join by simulating the programs for larger "transitivity factor" (as defined later).

Varying Message Loss Probability. In this first set of experiments, we vary the message loss probability and compare the result inaccuracy of various programs in networks with different size. First, we confirmed that when there are no message losses, the accuracy of the result is 100% for all the programs. See Figure 4. Also, we observe that the generated code for the *logicH* and *logicJ* programs compute about 80% correct shortest paths for message loss probability up to 10%. The result inaccuracy continues to increase with increase in message loss probability. However, we observe that the robustness of the translated code is close to that of the procedural code, and *logicH* code is sometimes even more robust (i.e., has a lower result inaccuracy value) than the procedural code for small values of message loss probability.

Varying Network Size. In Figure 5, we plot the total communication cost incurred by various programs for varying network size. We observe that the total communication cost incurred by all programs is largely proportional to the total number of nodes in the network. As expected, the communication cost incurred by *logicJ* is about twice as that of procedural code, due to the insertions and deletions of each tuple $J(y,d)$. Moreover, *logicJ* code performs much better than the *logicH* code. The total communication cost of all approaches decreases with the increase in the message loss probability.

Varying Transitive Factor. In this set of experiments, we modify the various programs to compute shortest-path tree in G^k (for various k), where G is the network graph and G^k is defined as the graph wherein there is an edge between any two nodes x and y that are within k-hops in the network graph G. We refer to k as the *transitive factor*. Note that

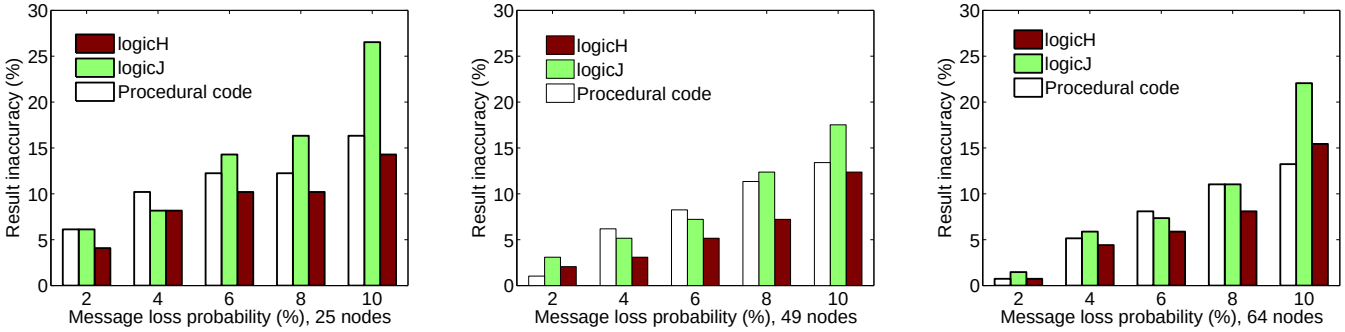

Figure 4. **Effect of message loss probability on result inaccuracy, for three different network sizes.**

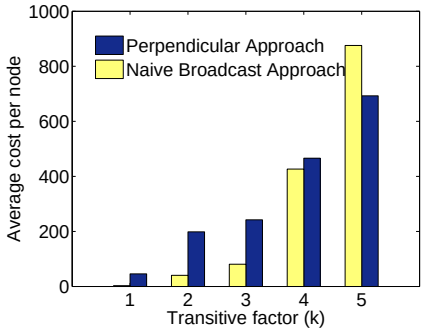

Figure 6. **Average communication cost per node incurred by the generated code using Naive Broadcast and Perpendicular Approach for varying transitive factor.**

here we are only changing the definition of what a shortest-path tree is, while keeping the network graph (transmission radii and neighborhoods of each node) the same. Our logic programs can be easily changed to reflect the above. In Figure 6, we show the effect of k on the total communication cost incurred for computing the shortest-path tree in G^k for varying k. We compare the performance of the generated codes for the *logicJ* program with two different join computation strategies: Naive Broadcast and Perpendicular Approach (PA). We notice that the communication cost of both approaches increases with increase in k. This is because with the increase in k, the number of paths with shortest length increases which results in more number of $J(y,d)$ tuples for the same y and d. However, PA increases at a lower rate than Naive Broadcast, and eventually outperforms Naive Broadcast when k becomes greater than 4, which suggests that PA is more suitable for implementation of join that involves distant tuple matching. Also, note that the memory requirements for PA is much less that of the Naive Broadcast Approach. We present more extensive simulations and performance comparison of various join implementation strategies in [47], with varying memory constraints.

Summary of Simulation Results. The above simulations illustrate the robustness and communication efficiency of the translated codes. We see that Naive Broadcast is a better choice for low-range spatial constraints, while PA is expected to outperform other approaches for joins involving matching of tuple generated/stored far away. Also, the main-memory usage at each node for the *logicH* and *logicJ* programs were minimal (8-10 tuples per node, with at most two derivations per tuple).

5 Conclusions

In this article, we have addressed the need for a high-level programming abstraction for sensor network applications. In particular, we proposed and motivated the deductive framework, and designed a full-fledge query engine for in-network evaluation of deductive queries in a sensor network. We presented implementation details of our system that compiles a given user deductive program into distributed code that runs on individual nodes. There are many challenges that need to be addressed for an optimized (in terms of main-memory usage and communication efficiency) implementation of an in-network deductive query engine. As outlined in the article, some of the challenges include: (i) Efficient (perhaps, approximate) implementation of the counting approach for incremental maintenance of join queries; such an implementation is unlikely to be fully accurate but will have minimal space overhead, (ii) Automatic determination of attributes to use for hashing derived results to minimize overall communication cost, (iii) Efficient implementation of the Rederivation approach of [19] which will pave the way of in-network evaluation of general deductive programs, with locally-stratified [9] negation, and (iv) The problem of selection of views to materialize [20] in the context of sensor networks. The above challenging issues are of great interest to us, and will be addressed in our future works.

6 References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] D. J. Abadi, S. Madden, and W. Lindner. REED: Robust, efficient filtering and event detection in sensor networks. In *VLDB*, 2005.

[3] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[4] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases*. Addison-Wesley Publishing Co., Inc., 1995.

[5] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[6] A. Bakshi, J. Ou, and V. K. Prasanna. Towards automatic synthesis of a class of application-specific sensor networks. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems (CASES)*, 2002.

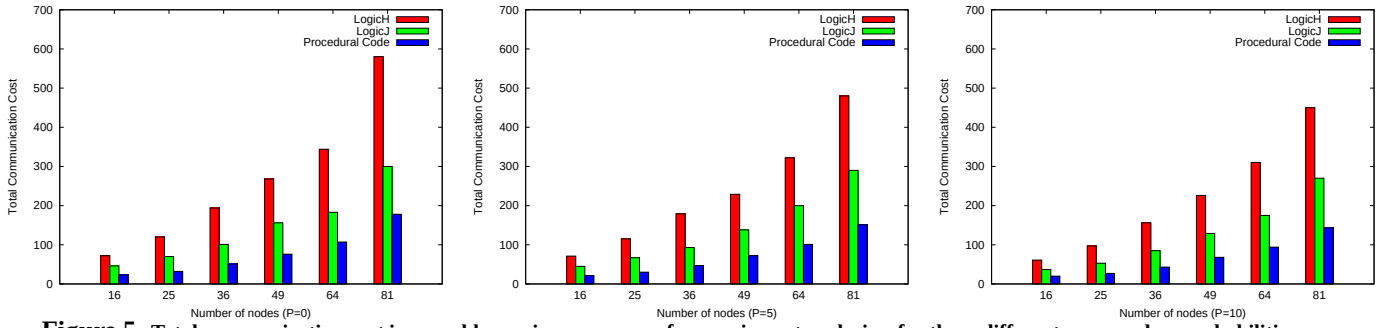

Figure 5. Total communication cost incurred by various programs for varying network size, for three different message loss probabilities.

[7] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Intl. Conference on Mobile Data Management*, 2001.

[8] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: a programming model for event-driven embedded systems. In *Symposium on Applied Computing*, pages 698–704, 2003.

[9] P. Cholak and H. A. Blair. The complexity of local stratification. *Fundamenta Informaticae*, 21(4), 1994.

[10] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein. Entirely declarative sensor network systems. In *Demo session in VLDB*, 2006.

[11] M. Chu, H. Haussecker, and F. Zhao. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *International Journal of High Performance Computing Applications*, 2002.

[12] A. Deshpande, C. Guestrin, S. Madden, and W. Hong. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, 2005.

[13] L. Ding, N. Mehta, E. Rundensteiner, and G. Heineman. Joining punctuated streams. In *EDBT*, 2004.

[14] D. C. et al. TinyOS. http://www.tinyos.net, 2004.

[15] D. C. et al. The design and implementation of a declarative sensor network system. Technical report, University of California, Berkeley, 2006.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

[17] 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.

[18] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[19] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[20] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[22] Intel Research. Intel mote. http://www.intel.com/research/exploratory/motes.htm.

[23] M. Kifer and V. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, pages 335–368, 1992.

[24] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys*, 2003.

[25] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[26] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, 2005.

[27] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, 2005.

[28] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[29] S. Madden and J. M. Hellerstein. Distributing queries over low-power wireless sensor networks. In *SIGMOD*, pages 622–622, 2002.

[30] 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.

[31] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, pages 491–502, 2003.

[32] S. R. Madden, J. M. Hellerstein, and W. Hong. TinyDB: In-network query processing in tinyos. http://telegraph.cs.berkeley.edu/tinydb, 2003.

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

[34] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *International Workshop on Data Management for Sensor Networks (DMSN)*, 2004.

[35] R. T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[36] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler. The mote revolution: Low power wireless sensor network devices. In *Proceedings of Hot Chips 16: A Symposium on High Performance Chips*, 2004.

[37] J. Reich, J. Liu, and F. Zhao. Collaborative in-network processing for target tracking. In *European Association for Signal, Speech and Image Processing*, 2002.

[38] A. Savvides, C. Han, and S. Srivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *MobiCom*, 2001.

[39] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.

[40] S. A. Tarnlund. Horn clause computability. *BIT*, 17(2), 1977.

[41] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[42] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[43] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *ACM Symposium on Networked Systems Design and Implementation*, 2004.

[44] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, pages 99–110, 2004.

[45] Y. Yao and J. Gehrke. Query processing for sensor networks. In *CIDR*, 2003.

[46] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the ldl++ approach. In *DOOD*, 1993.

[47] X. Zhu, H. Gupta, and B. Tang. Join of multiple data streams in sensor networks. Technical report, SUNY, Stony Brook, 2007.