Deductive Framework for Programming Sensor Networks

Himanshu Gupta, Xianjin Zhu, Xiang Xu

Department of Computer Science, Stony Brook University
Stony Brook, NY 11794
{hgupta,xjzhu,xxu@cs.sunysb.edu}

***Abstract*— Developing powerful paradigms for programming sensor networks is critical to realize the full potential of sensor networks as collaborative data processing engines. In this article, we motivate and develop a deductive framework for programming sensor networks, extending the prior vision of viewing sensor network as a distributed database. The deductive programming approach is declarative, very expressive, and amenable to automatic optimizations. Such a framework allows users to program sensor network applications at a high-level without worrying about the low-level tedious details. Our system translates a given deductive program to efficient distributed code that runs on individual nodes. To facilitate the above translation, we develop techniques for *distributed and asynchronous* evaluation of deductive programs in sensor networks. Our techniques generalize to recursive programs without negations, arbitrary non-recursive programs with negations, and in general to arbitrary "locally non-recursive" programs with function symbols. We present performance results on TOSSIM, a network simulator, and a small network testbed.**

I. 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, energy optimizations, unreliable infrastructure, and other network machineries. Thus, developing a powerful programming framework for sensor network is critical to realizing the full potential of sensor networks as collaborative monitoring systems. There has been some progress in developing operating system prototypes [13, 14] and programming abstractions [19, 41]; 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 query engines (TinyDB [21], Cougar [5], SNLog [12]) for sensor networks implement only a limited functionality. On the other hand, more expressive frameworks such as Kairos [26] are based on procedural languages and hence, are 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 be declarative (to hide low-level details from the user), be sufficiently expressive, and be amenable to automatic optimizations especially related to energy consumption.

In a recent concurrent work, a dialect of Datalog without negations was suggested for use in sensor networks [12], and a limited query processor designed. Our works extends the above, by motivating the full deductive approach for programming sensor networks, and developing optimized techniques for distributed asynchronous evaluation of deductive queries under resource constraints. Based on our query evaluation techniques, our system compiles a given deductive program into efficient distributed code that runs on individual nodes. Deductive approach is declarative, fully expressive (Turing complete), and most importantly, amenable to optimization. We believe that the collaborative (involving multiple nodes) functionality of a sensor network application can be easily represented using deductive rules, and the remaining local arithmetic computations can be embedded in built-in functions without affecting the communication efficiency of the translated code.

Our Contributions. We make the following specific contributions in this article. First, we motivate the use of the full deductive framework for programming sensor networks. Second, we develop distributed and asynchronous techniques for evaluation of recursive deductive programs without negations and non-recursive deductive programs with negations. In general, our techniques work for XY-stratified [43] and locally non-recursive deductive programs [6], which are useful in the context of sensor networks. Our system architecture justifies the feasibility of our sensor-network deductive query engine based on our query evaluation techniques. We conduct experiments on the TOSSIM simulator and a small sensor network testbed to demonstrate the robustness and efficiency of our techniques.

Article Organization. We start with a discussion of related work and giving an overview of deductive framework in the next section. The following two sections present query evaluation techniques for programs without negations and programs with negations respectively. Section V presents the system architecture and implementation details. We present our performance results in Section VI.

II. Related Work and Deductive Framework

In this section, we start with a discussion on related work, and then, give an overview of the deductive framework.

A. Related Work

Here, we give an overview of prior works on programming sensor networks and query processing in sensor networks.

NesC and Programming Abstractions. The Berkeley motes [17] platform provides the C-like, fairly low-level programming language called *nesC* [14] on top of the TinyOS [13] 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 [15, 19, 24, 25, 34, 41] for sensor networks; however, these abstractions provide only minimal programming support. Finally, authors in [4] 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 [5, 20, 21] 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 [21], Cougar [5]) implement a limited functionality of SQL, the traditional database language. In particular, they only handle single queries involving simple aggregations [32, 42] or selections [22] over single tables [33], local joins [42], or localized/centralized joins [1] 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. Recently, a dialect of Datalog without negations has been suggested for use in sensor networks [12], and a limited query processor designed. The focus of [12] 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 [26] 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 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 some examples in Section II-B, 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 II-B 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.

In-Network Query Evaluation Techniques. 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 [11, 38]; 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 [21], Cougar [5]) 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. [10] 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.

B. Deductive Programming Framework

In our 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 [39]. We illustrate the need for function symbols in Example 2. Thus, in our framework, 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. Thus, a logic rule is written as

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

H is the *head*, and G_i's are the *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. We allow use of *built-in* predicates or functions. A built-in predicate may be system defined or defined by the user in procedural code. For sake of ease in programming, we allow restricted use of negations (as discussed in Section IV) and aggregations.

Motivation. Use of deductive approach for sensor networks is motivated by the basic observation that sensor networks essentially gather sets of "facts" by sensing the physical world, and applications manipulate these facts. We believe that the collaborative functionality of a sensor network application can be easily represented using deductive rules that manipulate these facts, and the remaining local arithmetic computations can be embedded in procedural built-in functions. Most importantly, the deductive framework lends itself to automatic optimizations related to communication costs. In our context, the optimization of deductive programs is largely embedded in the efficient data storage schemes, in-network implementation of the join, join-ordering, and other query optimization techniques.

Specification and Maintenance of Sliding Windows. Sensor network data can be modeled as data streams of facts corresponding to sensing readings [20,42]. Due to limited memory resources, we store the data streams as *sliding-window* [3,8] consisting of typically the most recent tuples. In our framework, we can use temporal predicates to specify time-based windows. For instance, consider a data stream $S(a,t)$, where a is an arbitrary attribute and t is the timestamp. The following defines sliding windows R of range τ_w.

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

Above, the last subgoal is used to bound T, "_" denotes an anonymous variable, and $R(\_,\_,T)$ is a tuple in the sliding-window of time T. Time-based sliding-windows can be easily maintained in a sensor network, by independently expiring a tuple after sufficient time. 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; in-network maintenance of other types of sliding-windows is a challenge and part of our future work.

Examples. In a recent work [12], a limited form of deductive approach has been shown to be convenient for specification of many sensor network protocols and applications. Here, we present additional examples to illustrate need for negated subgoals, function symbols, and involved use of recursion with negation, for programming typical sensor network applications.

EXAMPLE 1: 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.

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

□

EXAMPLE 2: Need for Function Symbols. We now illustrate the need for function symbols in our 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). Below is a logic program to compute trajectories, and to determine pairs of parallel trajectories.

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

Here, we use R to represent (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 from 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 3: Recursion with Negation (Shortest-Path Tree). We now give a logic program for constructing a shortest-path tree (H) from a root node (A) in a network graph G. Shortest-path trees are use for data-gathering at the root from the network nodes. This example illustrates a more involved (XY-stratified) 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:

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

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 XY-stratified (see Section IV-C), and is more compact than the 20 lines of procedural code written in Kairos [26]. More importantly, it can be automatically translated into communication-efficient distributed code; we present more details in Section VI. □

Embedding Arithmetic Computations in Built-in Predicates. 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, but can be embedded in procedural built-in functions without affecting the communication efficiency of the translated distributed code. The distributed arithmetic computations can be embedded in built-in aggregates with specialized distributed implementations. For instance, in vehicle tracking [7, 36], arithmetic computations involve estimating belief states, information utilities, and future target location; the first two computations are local and can be embedded in built-in functions, while the last computation requires the *maximum* aggregate. The resulting deductive program is *much* simpler [16] than the procedural code in Kairos [26].

Extensions and Limitations of the Approach. 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 [29] (for reasoning with uncertain information). 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 3 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.

III. Queries without Negations

In this section, we discuss our techniques for in-network evaluation of deductive queries without negations. In the next section, we will extend these techniques to evaluation of queries with negations.

Bottom-up Approach. We use the bottom-up approach [40] of evaluating deductive queries, 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. Since, the join operation is at core of the bottom-up approach, we consider techniques for in-network implementation of join in the next subsection. Then, in the following subsection, we generalize the join implementation to evaluation of recursive programs without negations.

A. In-Network Implementation of Join

In this section, we address the problem of in-network implementation of join of data streams. Here, we assume only insertions into data streams; we will generalize our techniques to handle deletions in Section IV-A.

Problem Model. More formally, we wish to compute $R_1 \bowtie R_2 \bowtie \ldots R_n$, where each R_i is a data stream being generated in a distributed manner across the sensor network. The join conditions may be arbitrary; however, we give special consideration to “spatial” joins (discussed later). 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.

A naive way to compute the join is to send each generated tuple to some central server for computing the join. Such a scheme without any in-network processing may incur prohibitive communication costs [31, 32] and may result in quick failure of the nodes close to the server (rendering the central server disconnected from the network). Thus, we have designed a *Perpendicular Approach* which is communication-efficient, load-balanced, fault-tolerant, and immune to certain topology changes. 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 our other work [44]. We start with describing it for a join of two data streams, and generalize it to join of multiple data streams.

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

The above scheme can be shown [44] to incur near-optimal (within a factor of eight) communication cost in a square-grid with uniform generation rates. 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 1: (Partial Result.) Let $R_1, R_2, \ldots, R_n$ be 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

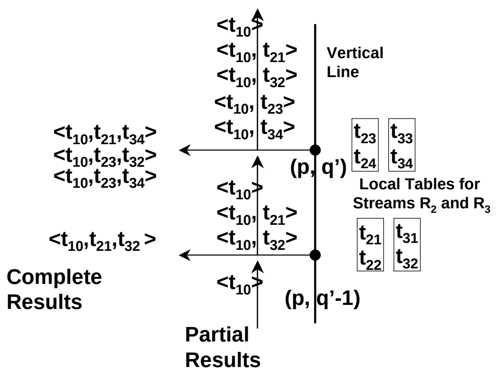

Fig. 1. One-Pass Join Computation. Here, $t_{i*} \in R_i$. We assume assume the join conditions to be such that t_{10} matches only with $t_{21}, t_{23}, t_{32}, t_{34}$, and there is no join condition between R_2 and R_3.

$k < n-1$ 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.

Multiple-Pass Scheme. The above join-computation scheme is called the *single-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.

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. We refer the reader to [44] for details of such a scheme and its performance comparison with the "Centroid Approach." Here, 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. ■

Function Symbols and Spatial Constraints. Note that PA easily generalizes to attributes with arbitrary terms involving function symbols, since the join conditions are evaluated only locally at each node. Due to inherent spatial correlation in sensor network data [9], the join predicates in sensor network queries frequently include a spatial constraint (i.e., two tuples can join only if they are generated at nodes within a certain hop/Euclidean distance). To incorporate spatial constraints in PA, 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.

Generalized Perpendicular Approach (GPA). The core idea used in the above Perpendicular Approach is that of intersecting *storage and join-computation regions*, which are the set of nodes using in the respective phases. In PA, the storage and join-computation regions were the vertical and horizontal paths. In general, such regions can be arbitrary as long as every storage region intersects with every join-computation region.[2] We refer to such a general scheme as the *Generalized Perpendicular Approach (GPA)*. The degenerate examples of GPA are (i) *Naive Broadcast Approach*, wherein the storage region is the entire network and the join-computation region is the local node, and (ii) *Local Storage Approach*, wherein the storage region is the local node, and the join-computation region is the entire network. The correctness of our query evaluation schemes in the next section hold for the Generalized Perpendicular Approach, irrespective of the specific storage and join-computation regions.

B. Evaluation of 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.

[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 IV.

[2]However, for arbitrary join-computation regions, the multiple-pass scheme may be easier to implement than the single-pass scheme.

Hashing Derived Tuples; Derived Data Streams. The Generalized Perpendicular Approach (GPA) discussed in the previous subsection generates 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 V for more discussion).

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. Note that later duplicates of t are not considered as generations.

Evaluation of Deductive Queries without Negations. Consider a predicate Q represented by multiple deductive rules (without recursion or negation) over base streams and with common head predicate Q. Evaluating the query Q is equivalent to computing the join corresponding to each deductive rule and then, taking a union of the results. The union of results is facilitated by the hashing/storage scheme described in the previous paragraph. It is easy to include recursion in the deductive rules, since a recursive subgoal can be treated just like a subgoal of another predicate. Finally, the above evaluation algorithm can also be use to evaluate an arbitrary recursive deductive program without negations, due to the generation of each derived result as a data stream. Note that the above evaluation algorithm is for insert-only base streams, as is typically the case in sensor networks where data is a stream of sensor readings.

IV. Queries With Negations

In this section, we generalize our Generalized Perpendicular Approach of Section III-A to evaluate deductive queries involving negations.

High-Level Plan. In Section III-A, we showed that the Generalized Perpendicular Approach (GPA) correctly maintains a join-result in response to simultaneous insertions. We generalize GPA to evaluate deductive programs with negations in the following steps.

- First, in Section IV-A, we generalize GPA to maintain a join-result in response to deletions to the operand streams. This generalization is fundamental to handle negations in deductive programs.
- Then, in Section IV-B, we generalize GPA to evaluate queries represented by a single deductive rule involving negated subgoals.
- Finally, in Section IV-C, we 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. Lastly, we discuss evaluation of general stratified deductive programs.

Nature of Extensions to GPA. In general, generalizing GPA for the above cases requires one or more of the following types of changes: (i) The storage phase of a deleted tuple t will involve *removing* (see below) the replicated copies of t from its storage region, (ii) The result of the join-computation phase needs to be appropriately processed to maintain the query result, (iii) The join-computation phase will need to be extended to evaluate rules with negated subgoals, and (iv) There would be an appropriate delay introduced before the start of the join-computation phase. We use the term *removal* of a tuple to signify removing the replications of a tuples from its storage region; in contrast, *deletion* refers to an actual deletion of the tuple from its table. Note that deletion of a tuple occurs only at its source node.

A. Generalizing GPA 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 must compute $t_1 \bowtie R_2 \ldots \bowtie R_n$ and "delete" it from the maintained join-query result. However, a straightforward subtraction of sets may not work, since the *set* $(R_1 - t_1) \bowtie R_2 \ldots \bowtie R_n$ may *not* be equal to the *set* $(R_1 \bowtie R_2 \ldots \bowtie R_n) - (t_1 \bowtie R_2 \ldots \bowtie R_n)$. We can solve the above problem in one of the following ways: (i) *Counting Approach [27]:* Store the query result as a bag, or keep a count of multiplicity of each result tuple, (ii) *Rederivation Approach [27]*: Partly recompute the result, or (iii) Keep the actual set of derivations (as described below) for each result tuple. The counting approach is difficult to implement accurately for a fault-tolerant technique such as GPA, due to non-deterministic duplication of result tuples. The rederivation technique will result in a lot of communication overhead. However, the technique of keeping the actual set of derivations incurs no additional communication overhead. There is a space overhead of storing the derivations, but it may be tolerable if tuples have only a few derivations.

Definition 2: (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 generated at its hashed location.

Tuple-ID is used to uniquely identify a tuple in the system. We use (I, τ) as the ID of a tuple t, where I is its source node and τ is its generation-timestamp (local time at I when t was generated).

A *derivation* of a derived tuple t is the *list* of the tuple-IDs, that join to yield t, one from each of the data streams corresponding to the *non-negated* subgoals of the (safe) deductive rule used to derive t. In general, we also include in the derivation, the ID of the rule used to derive t. $\square$

Set-of-Derivations Approach. Let $T = R_1 \bowtie R_2 \ldots \bowtie R_n$ be the current join-result. For each tuple $t \in T$, we maintain its set of derivations with it. Now, in response to an isolated deletion of tuple t_{r1} from R_1, the join-result T is maintained as follows. Isolated insertions into a data stream are similarly handled.

- First, we compute $T_{r1} = t_{r1} \bowtie R_2 \ldots \bowtie R_n)$ along with the derivation of each tuple in T_{r1}.
- Second, for each derived tuple $t \in T$, we subtract the set of derivations of t in T_{r1} from the set of derivations of t in T.
- Finally, based on the above, t is deleted from T if the resulting set of derivations of t becomes empty.

The first step (computation of T_{r1}) constitutes the join-computation phase of GPA for deletion of t_{r1}. However, preceding this first step is the storage phase of GPA for t_{r1}, wherein t_{r11} is *removed* from its storage region.

Handling Simultaneous Insertions and Deletions. The above technique correctly maintains a join-result in response to *isolated* (one at a time) insertions or deletions to operand streams. To handle simultaneous updates, we start the join-computation phase after a delay of $\tau_s + \tau_c$ from the start of the storage phase; here, 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.

B. Deductive Rule with Negated Subgoals

Let T be a query represented by a non-recursive safe[3] deductive rule with possibly negated subgoals. That is, let

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

Above, each R_i or S_j (not necessarily distinct) is a data stream. We now generalize our technique of previous subsection to maintain T. Let t_{r1} be an *isolated* insertion or deletion into the stream R_1. Maintenance of T in response to the above update consists of the following steps.

- First, we compute
$$T_{r1} \;\text{:-}\; t_{r1}, R_2, R_3, \ldots, R_n,\ NOT\ S_1, \ldots, NOT\ S_m,$$
along with the derivation of each tuple in T_{r1}. To compute T_{r1}, we modify the join-computation phase of t_{r1} as follows. We compute and propagate partial results of $t_{r1} \bowtie R_2 \bowtie \ldots \bowtie R_n$, and delete partial or complete results that match with a tuple in *some* S_j.
- Then, for each tuple in T_{r1}, we add or subtract (based on t_{r1} being an insertion or deletion) its set of derivations in T_{r1} with its original set of derivations in T.
- Similarly, to process an insertion or deletion t_{s1} from S_1, we first compute
$$T_{s1} \;\text{:-}\; R_1, \ldots, R_n, t_{s1}, NOT\ S_2, \ldots, NOT\ S_m,$$
and then, for each tuple in T_{s1}, add (for deletion t_{s1}) or subtract (for insertion t_{s1}) its set of derivations in T_{s1} from the original set of derivations in T.

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

It is easy to see that the above correctly maintains T in response to such *isolated* insertions or deletions into the operand streams.

Handling Simultaneous Updates. To maintain T in face of simultaneous updates across the network, we use the following strategy. Below, an *event*-timestamp refers to the *local* timestamp at the node where and when the event (update, generation, or deletion) occurred.

- We start the join-computation phase of any tuple after a delay of $\tau_s + \tau_c$ from the start of its storage phase; here, 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 with a update-timestamp of τ, we match/join t with only those tuples that (i) have a generation-timestamp between τ and $(\tau - \tau_w)$, and (ii) do *not* have a deletion-timestamp of less than τ. Here, τ_w is the range of the sliding-window.
- To facilitate the above step, during the storage-phase of a tuple deletion, we do *not* remove the replicated copies of the tuple from the nodes, but instead only record its deletion-timestamp. They are eventually "expired" as suggested below.

The above strategy ensures that the updates are virtually processed in the order of their local timestamps. *Tuple Expiry:* 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 maximum "relative" range of the sliding-window.

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 insertions or deletions to the given data 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 or deleted in R_1 with an 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 that: (i) have a generation-timestamp of less than τ and more than $(\tau - \tau_w)$, and (ii) do not have a deletion-timestamp of less than τ. To prove the theorem, we essentially need to show that during the join-computation phase of t_{r1}, the join-computation region used by t_{r1} contains the sliding-windows R_i $(2 \leq i \leq n)$ and S_i $(1 \leq i \leq m)$. The above will show that $T_{r1} \;\text{:-}\; t_{r1}, \ldots, R_n,\ NOT\ S_1, \ldots, NOT\ S_m$ 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 update-timestamps*, which proves the theorem.

To show that the join-computation region used by t_{r1} contains 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 phases of all tuples with a generation-timestamp of less than τ have 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. Note that maintenance of a query result represented by multiple non-recursive deductive rules with the same head, is equivalent to maintaining each rule independently, and taking a union. Recall that derivation of a tuple includes the ID of the rule used.

C. General Deductive Programs

Non-Recursive Deductive Programs. In the above subsection, we described a query evaluation scheme to maintain a query represented by multiple non-recursive deductive rules with negations. Actually, the same scheme also works for evaluation of general non-recursive deductive programs, if we use the hashing/storage scheme suggested in the previous section. Essentially, hashing and storing the derived tuples based on *some* hashing scheme yields a derived data stream for each derived predicate; these derived streams can be handled in the same manner as base streams to update "higher-level" predicates. Note that in a program with negations, a derived stream may incur deletions; this doesn't pose a problem to the correctness of the evaluation scheme since Theorem 3 holds for insertions as well as deletions into the operand streams. However, we need to wait for an appropriate time before actually "finalizing" a derived fact (since it may be retracted/deleted later), which is acceptable due to bounded size sliding-windows and implicit temporal correlation in the sensor data.

The correctness of our evaluation scheme for general non-recursive deductive programs follows from Theorem 3, and the following two observations.

- First, the set of derivations of any derived tuple is finite, since non-recursive programs (even with function symbols) have finite query results.
- Second, each derivation of a derived tuple yields a valid "proof tree" with leaves as base tuples.[4] Again, this holds because the program is non-recursive, and hence the proof tree constructed (by iteratively "unfolding" the derivations) will have no directed cycles. Thus, a set of derivations of a tuple t is nonempty if and only if there is a valid proof tree.

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 [2]. However, our evaluation scheme outlined also works for evaluation of XY*-stratified* programs [43], 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 values of one or more arguments of the derived tuples. For instance, consider the *logicH* program of Example 3. The table H be partitioned into sub-tables $H_1, H_2, \ldots$, based on the the value of the third argument, such that H_d represents the sub-table consisting of all the facts $H(\_, \_, d)$. Similarly, H'_d can be the sub-table consisting of all the facts $H'(\_, d)$. Now, the dependency graph on the above sub-tables is actually acyclic, since there exists a topological order $(H_0, H'_1, H_1, H'_2, H_3, \ldots,)$ of the sub-tables in the dependency graph. Thus, the *logicH* program is XY-stratified.[6] Similarly, the program of Example 2 is XY-stratified since the *traj* table can be 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. Note that the partitioning of tables into sub-tables and acyclicity of their dependency graph is only to observe correctness of the evaluation scheme; the evaluation scheme itself is oblivious of the partitioning.

Evaluating General Recursive Programs. Recall that the correctness of our evaluation scheme (specifically, the set-of-derivations approach) 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 correctly using our evaluation scheme, since a non-empty set of derivations of a tuple may not imply existence of a valid proof tree for the tuple.

However, our evaluation scheme will correctly evaluate programs with no cycles in the dependency graph over the derived tuples, wherein there is a directed edge from tuple t_1 to t_2, if t_2 was derived using t_1. Such programs are referred to as *locally non-recursive* programs [6].

For evaluation of general stratified programs, we need to employ some variant of the rederivation approach [27]. 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

[4]A *proof tree* [40] of a derived tuple t describes how the tuple t is constructed from the base tuples; 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.

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

[6]Our notion of XY-stratified programs is slightly more general than the original notion defined in [43].

can be derived from the existing base tuples. Execution of the second step may incur additional communication cost. Thus, efficient in-network evaluation of general stratified programs is a challenge, and will be addressed in our future work.

Built-in Predicates, Function Symbols, and Aggregations. Our query evaluation scheme can be easily generalized to handle built-in predicates, since the evaluation of join-conditions and execution of built-in predicates is done only locally. Also, incorporating function symbols in deductive rules only requires extending the evaluation of join-condition using the term-matching operator [40]. However, introduction of function symbols in deductive programs may result in non-termination of recursive programs and may make optimizations difficult; but we anticipate that function symbols will be used in limited contexts, and hence, allow their use for full expressibility. Aggregates can be represented in logic rules using the Prolog's all-solutions predicate. We can use specialized distributed techniques such as TAG [32] or fault-tolerant synopsis diffusion [23] for evaluation of incremental aggregates.

V. System Implementation

In this subsection, we give an overall architecture of our system, address resource requirements, and present details of our current implementation.

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 [40] (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

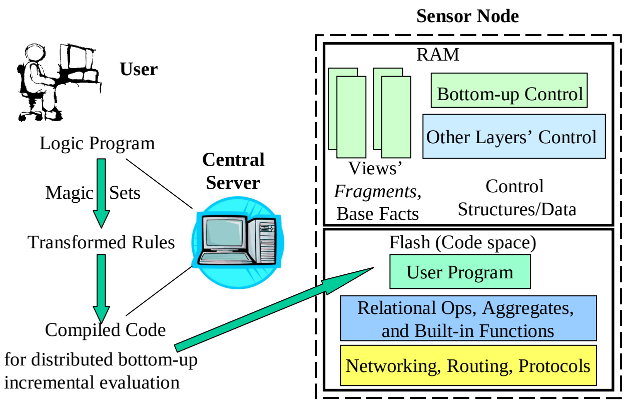

Fig. 2. System Architecture.

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

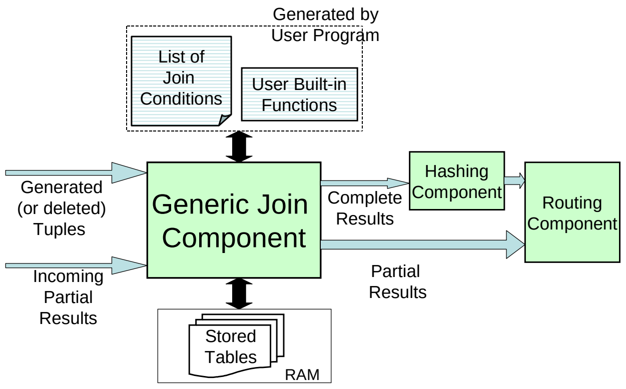

Fig. 3. The *join component* at a sensor node. Newly generated tuples are fed into the join component, which generates partial and complete results by joining with local tables. The complete results are sent to the *hashing component* for hashing, and then, forwarded to the *routing component* for storage 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).

years [18], and latest Intel mote is being designed with 64 KB RAM [28]. In our system, the user program essentially consists of the generic join interface (as described in Section VI), 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 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 program execution. 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 3, the derived results are H (or J, for the improved *logicJ* program given in Section VI) and H', and based on the storage scheme discussed in Section VI, 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.

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 [40], and hence, result in minimal processing load. The processing load due to arithmetic-intensive local built-in functions is inherent to a user program. Thus, our approach is not expected to increase the overall processing load.

Our System Implementation. The main purpose of our system is to automatically translate a given deductive program into distributed code that runs on individual sensor nodes. The generated code represents our outlined query evaluation strategy. In particular, we have developed nesC interface components for various versions (see Section III-A) of the Generalized Perpendicular Approach, viz., Naive Broadcast, Local Storage, Perpendicular Approach, and Semi-Naive Strategy (see Section VI). These generic components reside in each node. The front-end compiler parses and compiles a given deductive program into a .h file containing (i) the database schema (list of predicates and attributes), and (ii) for each rule, the list of attributes in the result and each subgoal, and the join conditions. The join-conditions are used by the generic join component to evaluate the predicates in the program. The appropriate formatted .h file is #-included in the nesC program, and read into appropriate data structures. Figure 3 shows a high-level block diagram of our implementation.

The current version of our system can handle general deductive programs with arithmetic built-in functions and predicates. Our implementation platform is the TOSSIM [30] simulator, and a small network tested of sensor motes. Later versions of our system would incorporate function symbols, aggregations, and arbitrary user-defined built-in functions. In our current implementation, the hashing scheme of the tables is provided by the user.

VI. **Performance Evaluation**

In this section, we present our simulation results for implementation of the shortest-path tree program of Example 3. The shortest-path tree program of Example 3 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, the translated code for other examples or typical applications, viz., Examples 1 to 3 in Section II-B, vehicle tracking based on belief states [16] or Darpa-Nest software [41], and multilateration [37], naturally yields communication-optimal translations (essentially, the same algorithm as the native algorithms) with appropriately chosen hashing schemes [16]. Thus, to gain more insight into our proposed approach, we choose the challenging shortest-path tree example for performance comparison.

Comparison of Program Sizes. For the above given examples, the deductive programs are much shorter and compact (very few logic rules) compared to the corresponding procedural code. Here, we ignore the size of the procedural code for user-defined built-in functions, since that is common across the deductive and procedural frameworks.

A. **Distributed Evaluation of *logicH* Program**

logicH Program (Repeated from Example 3)

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

Hashing Scheme, and In-Network Join of *logicH* Rules. Since y is the only node-ID attribute in $H'(y,d)$ tuples, each tuple $H'(y,d)$ is hashed to the y node. Also, to facilitate efficient join, we hash each tuple $H(x,y,d)$ to y. The above hashing scheme entails that all pairs of matching tuples reside within one hop of each other. Thus, we can easily use the Naive Broadcast approach for computing joins. However, for our specific context, the below described Semi-Naive Approach is most efficient.

Semi-Naive Approach. Consider a join of two tables R_1 and R_2 with the join-predicate containing a one-hop spatial constraint. We can compute the join of R_1 and R_2 using the following Generalized Perpendicular Approach (GPA). For R_1, we choose the storage and join-computation regions to be the source node itself, while, for R_2, we choose both the regions to be 1-hop neighborhoods of the source node. Thus, to compute the join, only the tuples of R_2 need to be 1-hop broadcast from the source node; this broadcast can serve the purpose of both the storage and join-computation phases for R_2 tuples.

In-Network Join of *logicH* Rules. In our *logicH* program, the third and fourth rules are essentially two-table joins with one-hop spatial constraint, since we assume each tuple $G(x,y)$ to be available at both x and y nodes. Thus, we can use the Semi-Naive Approach for them. For the third rule, we choose the subgoal $H(\_,y,d')$ as R_1, and for the fourth rule, we choose the subgoal H' as R_1.

The above scheme requires only one broadcast for each insertion or deletion into the H table, and the tuples $H'(y,\_)$ and $H(\_,y,\_)$ are derived at their hashed locations itself. Thus, the *only* communication cost incurred for the *logicH* program is the 1-hop broadcast of each H tuple.

Optimized *logicJ* Program. 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.

logicJ Program:

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

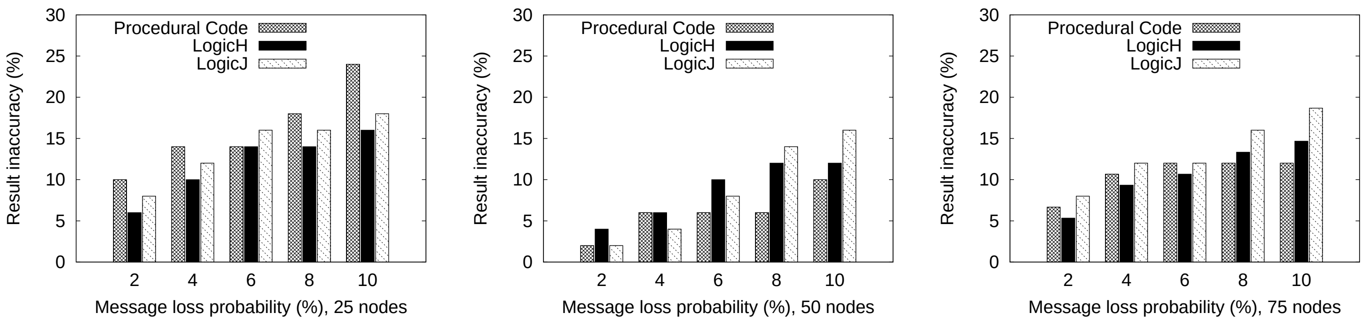

Fig. 4. Comparison of result inaccuracies of various programs for varying parameters on the TOSSIM simulator.

B. Simulation Results

We now compare the performance of our translated codes for *logicH* and *logicJ* programs with the procedural code.

Simulation Setup, Programs, and Performance Metrics. We run our simulations using the TOSSIM simulator on a randomly generated sensor network, and on a network tested of 25 motes placed in an arbitrary topology. In particular, the sensor network for TOSSIM simulations is generated by randomly placing the given number of nodes in an area of 50×50, and connected two nodes with an edge if they are within a distance of 20 (the transmission radius). For the network testbed, we use 25 Crossbow TelosB motes using TinyOS 2.0 and place them randomly in a room so as to form an arbitrary connected 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; this is in addition to the minimal message losses due to collisions. 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: (i) *procedural code*, the distributed nesC code for breadth-first search which incurs optimal communication of one message per node, (ii) *logicH*, the generated code for the *logicH* program, and (iii) *logicJ*, the generated code for the *logicJ* program. In the translated codes, we use the Semi-Naive Approach for join computation.

Comparison of Result Inaccuracies. We first compare the result inaccuracy of various programs for varying message loss probability. We conduct the experiment on three different network sizes on TOSSIM and on the 25-node network testbed. See Figure 4 and 5(a). When there are no message losses, the accuracy of the result is 100% for all the programs. For non-zero message loss probabilities, we observe the following. (i) As expected, result inaccuracy of *logicJ* is mostly higher than that of *logicH*. (ii) *Result inaccuracy of the* logicJ *and* logicH *programs is generally close to (sometimes even lower than) that of the procedural code.* The above observation shows the robustness of our techniques.

Comparison of Communication Costs. We now compare the

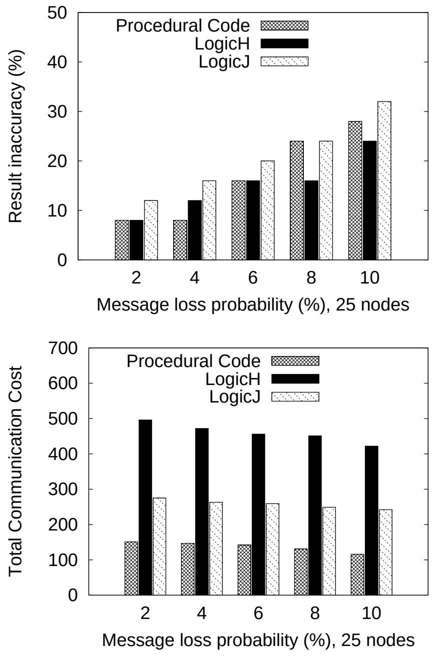

Fig. 5. Experiments results on a 25-node network testbed of Crossbow TelosB motes.

total communication cost incurred by various programs on the 25-node network testbed (Figure 5(b)) and for varying network size on TOSSIM (Figure 6). We observe that the communication cost of *logicJ* is less than twice of that of the procedural code. The ratio of the communication costs of *logicJ* and procedural code decreases with increasing network size (Figure 6), and *for larger networks the communication cost of* logicJ *is only marginally higher than that of the procedural code.* Also, the communication cost for all program is largely proportional to (i.e., linear in) the network size. The above observations show the *scalability and efficiency* of our techniques.

VII. Conclusions and Future Work

In this article, we have motivated the deductive framework for programming sensor networks and designed distributed and asynchronous techniques for evaluation of deductive queries. Our system translates a given deductive program into distributed code that runs on individual nodes. We demonstrate the robustness, efficiency, and scalability of our developed techniques using simulations and experiments. There are many challenges that need to be addressed for an optimized (in

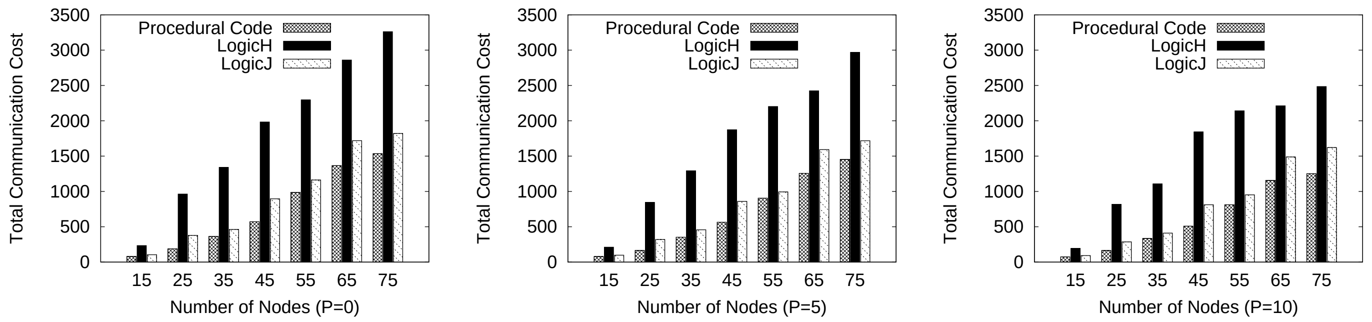

Fig. 6. Communication cost incurred by various programs for varying parameters on the TOSSIM simulator.

terms of main-memory usage and communication efficiency) deductive query engine in sensor networks. 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, and (iii) Efficient implementation of the rederivation approach of [27] which will pave the way for in-network evaluation of general locally-stratified deductive programs.

Acknowledgements. We wish to thanks various people at Stony Brook – especially, S. Das, M. Kifer, C.R. Ramakrishnan, and D. Warren, for insightful discussions. Work was supported in part by NSF grants 0713186 and 0721665.

REFERENCES

[1] D. Abadi, S. Madden, and W. Lindner. REED: Robust, efficient filtering and event detection in sensor networks. In *VLDB*, 2005.
[2] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases*. Addison-Wesley Publishing Co., Inc., 1995.
[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, 2006.
[4] A. Bakshi, J. Ou, and V. K. Prasanna. Towards automatic synthesis of a class of application-specific sensor networks. In *CASES*, 2002.
[5] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Intl. Conference on Mobile Data Management*, 2001.
[6] P. Cholak and H. A. Blair. The complexity of local stratification. *Fundamenta Informaticae*, 21(4), 1994.
[7] M. Chu, H. Haussecker, and F. Zhao. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *Intl. Journal of High Performance Computing Apps.*, 2002.
[8] L. Ding, N. Mehta, E. Rundensteiner, and G. Heineman. Joining punctuated streams. In *EDBT*, 2004.
[9] A. Deshpande et al. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, 2005.
[10] B. Loo et al. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.
[11] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.
[12] D. Chu et al. The design and implementation of a declarative sensor network system. Technical report, Univ. of California, Berkeley, 2006.
[13] D. Culler et al. TinyOS. http://www.tinyos.net, 2004.
[14] D. Gay et al. The nesC language: A holistic approach to networked embedded systems. In *PLDI*, 2003.
[15] E. Cheong et al. TinyGALS: a programming model for event-driven embedded systems. In *SAC*, 2003.
[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] J. Hill et al. System architecture directions for networked sensors. *SIGPLAN Notices*, 35(11), 2000.
[18] J. Polastre et al. The mote revolution: Low power wireless sensor network devices. In *Symposium on High Performance Chips*, 2004.
[19] K. Whitehouse et al. Hood: a neighborhood abstraction for sensor networks. In *Intl. Conf. on Mobile Systems, Apps., and Services*, 2004.
[20] R. Govindan et al. The sensor network as a database. Technical report, Univ. of Southern California, 2002.
[21] S. Madden et al. TinyDB: In-network query processing in TinyOS. `http://telegraph.cs.berkeley.edu/tinydb`.
[22] S. Madden et al. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.
[23] S. Nath et al. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys*, 2004.
[24] T. Abdelzaher et al. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *ICDCS*, 2004.
[25] T. Eicken et al. Active messages: A mechanism for integrated communication and computation. In *ISCA*, 1992.
[26] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *DCOSS*, 2005.
[27] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.
[28] Intel Research. Intel mote. http://www.intel.com/research/exploratory/motes.htm.
[29] M. Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. of Logic Programming*, 1992.
[30] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys*, 2003.
[31] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, 2002.
[32] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.
[33] S. Madden and J. M. Hellerstein. Distributing queries over low-power wireless sensor networks. In *SIGMOD*, 2002.
[34] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *DMSN*, 2004.
[35] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2), 1992.
[36] J. Reich, J. Liu, and F. Zhao. Collaborative in-network processing for target tracking. In *Euro. Assoc. for Signal, Speech and Image Proc.*, 2002.
[37] A. Savvides, C. Han, and S. Srivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *MobiCom*, 2001.
[38] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.
[39] S. A. Tarnlund. Horn clause computability. *BIT*, 17(2), 1977.
[40] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II*. W. H. Freeman & Co., 1990.
[41] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.
[42] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.
[43] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the ldl++ approach. In *DOOD*, 1993.
[44] X. Zhu, H. Gupta, and B. Tang. Join of multiple data streams in sensor networks. *TKDE*, 2009.