University of Pennsylvania
ScholarlyCommons

Database Research Group (CIS) | Department of Computer & Information Science

August 2008

A Substrate for In-Network Sensor Data Integration

Svilen Mihaylov
University of Pennsylvania, svilen@seas.upenn.edu

Marie Jacob
University of Pennsylvania, majacob@cis.upenn.edu

Zachary G. Ives
University of Pennsylvania, zives@cis.upenn.edu

Sudipto Guha
University of Pennsylvania, sudipto@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/db_research

Mihaylov, Svilen; Jacob, Marie; Ives, Zachary G.; and Guha, Sudipto, "A Substrate for In-Network Sensor Data Integration" (2008). *Database Research Group (CIS)*. Paper 43.
http://repository.upenn.edu/db_research/43

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/db_research/43
For more information, please contact repository@pobox.upenn.edu.

A Substrate for In-Network Sensor Data Integration

Abstract
With the ultimate goal of extending the data integration paradigm and query processing capabilities to ad hoc wireless networks, sensors, and stream systems, we consider how to support communication between sets of nodes performing distributed joins in sensor networks. We develop a communication model that enables in-network join at a variety of locations, and which facilitates coordination among nodes in order to make optimization decisions. While we defer a discussion of the optimizer to future work, we experimentally compare a variety of strategies, including at-base and in-network joins. Results show significant performance gains versus prior work, as well as opportunities for optimization.

Keywords
Sensor networks, Routing, Joins

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/db_research/43

A Substrate for In-Network Sensor Data Integration

Svilen R. Mihaylov Marie Jacob Zachary G. Ives Sudipto Guha
University of Pennsylvania
{svilen, majacob, zives, sudipto}@cis.upenn.edu

ABSTRACT

With the ultimate goal of extending the data integration paradigm and query processing capabilities to ad hoc wireless networks, sensors, and stream systems, we consider how to support communication between sets of nodes performing *distributed joins* in sensor networks. We develop a communication model that enables *in-network join* at a variety of locations, and which facilitates coordination among nodes in order to make optimization decisions. While we defer a discussion of the optimizer to future work, we experimentally compare a variety of strategies, including at-base and in-network joins. Results show significant performance gains versus prior work, as well as opportunities for optimization.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols*

General Terms

Sensor networks, Routing, Joins

1. INTRODUCTION

As technology continues to advance, we are nearing a point where sensor networks will combine with intranets and the Internet to form integrated systems: intelligent buildings with visitor guides, surveillance and intrusion detection systems, building control systems, robot teams, etc. As this occurs, designers will need to think not simply about building sensor network systems, but about building sophisticated applications that perform **data acquisition and integration across highly distributed networks of heterogeneous device types**: streams of data may come from distributed sensors, network monitors and other "soft" sensors, server-based applications, mobile applications, and so on. This leads to a challenge in providing uniform query access to highly dynamic data from remote devices. The Aspen project focuses on this problem: extending the data integration paradigm and query processing capabilities to ad hoc wireless networks, sensors, and stream systems, in order to foster a new generation of data-intensive applications.

The primary goal of sensor network query processing is to minimize the number of messages transmitted (thus reducing battery consumption and congestion). Recent work [7, 11, 12] has shown how certain types of computation can be efficiently pushed into a sensor network built using a **tree topology**. These approaches exploit the fact that selection, projection, and aggregation can be translated to a tree topology in a natural fashion. However, in contrast to standard acquisition systems, data integration queries require *distributed join computation* over streams from different groups of sensors, perhaps using time- or space-based windows. For example, an integration query might combine smoke detector and temperature information within the same room; or combine information from distributed weather stations in a valley, detecting cold fronts by correlating temperatures at the perimeter.

A thorough understanding of how to set up join computation, so as to leverage the multi-hop nature of the networks, as well as to allow future cost-based optimization, remains an open question. Most proposals suggest a single strategy for all queries: join at the base [15], in a region near the base [6], or by routing through the base [14]. At the other end of the spectrum lies geographic hash table (GHT)-style routing [13], which can be used to support join at randomly placed nodes. A key point is that the right strategy for join *(i) establishes short, low congestion paths that are likely to be used over a long period of time, and (ii) sends the minimum number of tuples in each cycle along these paths*.

The establishment of join paths to minimize communication has not been addressed by the existing work. In the routing tree model, most data gets routed through nodes near the root of the tree, causing congestion and forcing certain nodes to expend power disproportionately. Ultimately, the maximally loaded node fails earliest (drops packets, dies, loses connectivity), fragmenting the network. The GHT model uses hashes of key values to disperse routes over the network, reducing hot spots but resulting in longer paths that increase join traffic (unless the hashed keys correlate to geography, in which case we have the same congestion issues). Establishing effective join paths, although similar to routing, is significantly different in objectives — and is the foundation of any future query optimization. The goal of this paper is to devise a substrate that allows the establishment of join paths while considering congestion (delay), power and battery life (packets forwarded). This is a key component in the Aspen project, aiming to develop an optimized query processing architecture supporting long-running data integration queries over heterogeneous multihop wireless networks.

To illustrate the issues, suppose that we have a join of source expressions, possibly with selection conditions, $(\sigma_{\theta_R}(R)) \bowtie_{\theta_{RS}} (\sigma_{\theta_S}(S))$, in a sensor network: this is a *windowed* join [3], where we take a snapshot of the tuples emitted by every R sensor satisfying θ_R in each time step, and similarly for each S sensor; then we combine the tuples of the R sensors taken within some time- (or space-) based window with the tuples of the S sensors taken within a (similarly defined) window, if they mutually satisfy predicate θ_{RS}. Each R and S tuple originates **at a different place in the network**, so each sensor node may have a different transmis-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2008 ACM 978-1-60558-284-9/08/08 ...$5.00.

sion distance. Each R and S sensor may not produce a tuple in every cycle, if θ_R or θ_S, respectively, are not satisfied. Finally, each R tuple may join with between 0 and $|S|$ tuples from S, in every sampling time step. Given these properties, it is evident (by our optimization criteria above) that the optimal join strategy is not always simply to flood large regions of the network, forward tuples to the root node, or even route *through* the root node to other destinations [14]. Instead, there are a number of different strategies that are **dependent on the query and data**. While we provide a formal description in the next section, we list some "rules of thumb":

- R and S tuples that **do not join** should be *dropped early*, reducing congestion and battery usage.
- When R and S tuples are **unlikely to join**, they should be joined *inside the network*: there are fewer join tuples than source tuples to route to the base.
- When the join condition is **not very selective** relative to R and S tuples, the join computation should be done *at the base*, as this results in fewer messages to the base.

We have sketched a problem of *join optimization*. This is quite different from query optimization in distributed databases: here we are considering settings where a *single* join involves pairs of relations that are horizontally partitioned, with each tuple at a different location, with a different number of hops to every other point in the network. Moreover, selectivities may differ for different source tuple values, and the connectivity of the network may gradually change as nodes or links fail, sensors move, batteries deplete, etc. Our contributions in this paper are an *efficient content-addressable routing substrate* for supporting distributed join computation and future optimization, which:

- **Indexes static and slow-changing attributes** (e.g. ID, position), allowing for efficient pruning of non-viable tuples
- **Supports efficient point-to-point and multicast communication based on join values** with short paths, enabling communication among sets of nodes that join.
- Enables **handling of link and node failures**.

These capabilities enable **collective cost-based decisions about where to join**: at or near one of the source nodes, or at the base. While we defer a cost-based optimization model to future work, we experimentally **validate that different join strategies have significant performance implications**, and that **in-network joins are often useful**, in terms of network messages, network congestion hot spots, and device memory usage.

Section 2 argues why a new network substrate is required and discusses prior techniques. Section 3 presents a novel routing scheme that enables efficient *join initiation*. Section 4 discusses our implementation and presents an experimental evaluation. We conclude and discuss future work in Section 5.

2. THE CASE FOR A NEW SUBSTRATE

We assume *windowed joins* [3, 9], in which the join occurs between "relations" representing state at different nodes in the network. Given a join of the form $R \bowtie_{\theta_{RS}} S$ where R and S represent two (possibly overlapping) collections of sensors and $\theta_R S$ represents a predicate over attributes of the sensors, we typically specify *time windows and sampling intervals* over R and S. As time progresses, the sensors continue to sample new readings, and they send these readings to participate in the join. In a "push"-based manner, the join buffers new tuples arriving from R and combines them with buffered tuples from S, and vice versa. The time window over each source relation defines when and how tuples are to be evicted from the buffer — and also enforces mutual consistency among the tuples in the relations. Alternatively, we also consider space-bounded windows that evict old values as they exceed capacity and new values arrive. Following a common assumption in stream joins, we assume that relations are *partitioned* into sets of independent windows based on join attributes — each join value forms its own separate window, and hence global coordination is not required across all nodes keeping state. We consider join to consist of the following steps:

- **Query dissemination**: propagates the query to all nodes; filters nodes that fail selection conditions over static attributes.
- **Join initiation**: determines where and how to send data and compute the join; prunes nodes that cannot join with others.
- **Join computation**: routes source data to *join points*, which compute join results; relays answers to the base.
- **Termination**: shuts down join computation; discards join state.

Given a select-join query of the form $(\sigma_{\theta_R}(R)) \bowtie_{\theta_{RS}} (\sigma_{\theta_S}(S))$, our goal is to find a partition of R and S nodes that minimizes overall network transmissions, such that all R and S nodes in the same partition route to some common *join point* j, where they may join and the result will be sent to the root. This cost is dependent on the *routes* taken from the R and S sensors to the join points, and from there to the base; as well as by the selectivities of the operations. Our goal is to develop techniques to help decide which strategy to use for a given query. We now survey join strategies.

2.1 Join Strategies and Related Work

When selection and join conditions are based purely on static data (IDs, coordinates of immobile devices, device capabilities, etc.), they can be evaluated *once*[1] by each node to see if they are satisfiable: if the selection fails or there are no other nodes with which to join, then this sensor cannot participate in the query. Hence we refine the predicates in our query expression: $(\sigma_{\phi_R \wedge \psi_R}(R)) \bowtie_{\phi_{RS} \wedge \psi_{RS}} (\sigma_{\phi_S \wedge \psi_S}(S))$ where each ϕ condition only considers static attributes, and ψ represents the rest of the condition θ including at least one dynamic attribute. We factor the predicates into CNF, then separate based on the presence of dynamic attributes (setting ϕ to $true$ if there are no static clauses). Most queries of interest possess such separation, and using the static selections ϕ_S, ϕ_R helps *any* strategy (hence in this paper we assume this is always done). Predicates involving non-static components, ψ, must be evaluated in every cycle. The dynamic join condition, ψ_{RS}, must be evaluated at some common point that receives potentially-joining R and S data; finally, ϕ_{RS} may be evaluated simultaneously, or during a pre-pruning step. Specifics depend on the strategy used.

2.1.1 Join paths along the routing tree

The vast majority of sensor network query processing research has focused on computation with a single routing tree.

Placing computation and data: The problem of adaptively placing in-network query operators in efficient locations is considered in [4], which assumes a directed diffusion infrastructure and periodically performs local neighbor exploration. Complementary work [8] focuses on placement of stored data in the network, collecting statistics about update frequency and query rates, then performing centralized optimization. In this paper, we focus specifically on a routing substrate that could be used to help support the above work, as well as more join-specific optimizations.

[1] Once per occasional network tree reconstruction.

Specialized joins: Several specialized join operations appear in the literature. REED [1] assumes one of the sources is a static table rather than a stream. The work of [6] assumes disjoint regions for the source nodes to be joined, computes a region "on the way to the base station" for nodes that join, then distributes a table snapshot among those nodes. It does not easily generalize to a setting with intermingled nodes, as we target. [16] returns "top-k" answers combining ranking functions over different attributes.

Naive join: Here, selection conditions are pushed to every sensor node, so that in each epoch, only tuples satisfying ψ_R or ψ_S (for R, S, resp.) are forwarded to the base over the routing tree. Both join conditions ϕ_{RS} and ψ_{RS} are then evaluated at the base station.

Base join: This approach builds upon **naive** by also pushing static condition ϕ_{RS} into the network, determining apriori which nodes have the possibility of participating in the join. If a node has such a possibility and satisfies its selection conditions, it forwards data over the routing tree to the base station, which evaluates ψ_{RS}. The push-down of ϕ_{RS} can be accomplished in at least two ways: by propagating an approximate synopsis of eligible values through the network [15], or by exploring the network for all satisfying join pairs. (We shall focus on the latter version.)

Yang+07: Recent work [14] has proposed an alternative scheme to joining at the base. The R sensors forward their data to the base, which assembles a complete snapshot of R. The base then floods this image *down* the tree to all S nodes, which perform the join and then send their results up to the base. This scheme can be arbitrarily more expensive than **base** if R has high cardinality or each R tuple joins with many S tuples. The snapshot of relation R must be split into many packets, and each needs to be flooded in the network.

Summary: These techniques have low overhead, but establish long join paths (because they are limited to tree edges) and often have high congestion and hot spots.

2.1.2 Full graph to establish paths

It is natural to consider using the full graph. However, in a fully distributed implementation (we only focus such schemes), this implies that the entire graph has to be flooded. While this achieves short paths, the setup cost of the join in flooding from many sources is prohibitive in terms of congestion, delays, and packet collisions.

2.1.3 Hash strategies to establish paths

The geographic hash table (GHT) [13] relies on positioning information (actual or virtual) at every sensor. As each sensor produces a tuple, it sends this to a destination coordinate, typically computed as a hash of that tuple key, using the Greedy Perimeter Stateless Routing protocol (GPSR). GPSR routes to the node located closest to the destination coordinate.

In-network GPSR/GHT join: Using these approaches ϕ_{RS} can be pushed into the network for equality or similar one-to-one predicates, where both R and S nodes send their data (provided ψ_R/ψ_S is satisfied) to a common node based on the hash of the static components to allow tuples to arrive at the same node. This node, which may be distant from the source nodes and the base, evaluates ψ_{RS}. (Note that for region and range predicates this common node has to be the base station, i.e., this algorithm must degrade into **base**. Otherwise, packet forwarding is done using GPSR.)

Summary: As a consequence of hashing, a GHT destination node may be at an arbitrarily distant location in the network. In addition, the strategy requires planar graphs for routing (and gives a planarization strategy) but wireless multihop networks may be deployed in a non-planar way (e.g., in a multi-story building) or positional information may simply not be available.

3. A SUBSTRATE FOR JOIN

Join initiation is the phase where the nodes in the relations R and S, after passing their respective static selection conditions, learn whether there exists at least one node with which they might join in each sampling interval (depending on dynamic conditions ψ). This is achieved by starting at each $r \in \sigma_{\phi_R}(R)$ node (the *initiator node*), then exploring where the ϕ_{RS} condition is satisfied over static attributes (while maintaining a source vector back to r). Nodes s satisfying $\phi_S(s)$ and $\phi_{RS}(r, s)$ (these are *target nodes*) can communicate directly with r by retracing the steps of the vector. Together (and possibly with other nodes that mutually join), they make an optimization decision about where to send their data during join computation. In the current presentation we omit further discussion of these optimization steps due to lack of space. For concreteness' sake, we assume that the join takes place at a mutually agreed point, say either r, s or the base station, although our implementation extends significantly beyond this simple strategy. The overall goals of the join initiation phase are (i) to have a fully distributed implementation; (ii) to find *short paths* in hops; (iii) to *avoid hot spots*; (iv) to *handle node or edge failures* (e.g., due to diminishing battery power or hardware failure); and (v) to scale to hundreds of nodes.

We observe that the benefit of using a single routing tree is that its hierarchical nature allows for summary structures [11], and allows "directed flooding" up the tree and down to any children whose summary structure includes the target join value. This requires significantly less traffic than flooding the network. But any spanning tree over a graph results in certain nodes in close proximity being numerous hops apart: consider a cycle, where the pairwise distance of a many pairs of nodes increases once the cycle is broken. The original graph has an advantage in average pairwise path length. To gain the best features of the single tree and the graph, we employ *multiple routing trees* over the same set of nodes: the initial routing tree forms the backbone of the network and is used for common tasks, whereas additional trees are formed afterward and used for directed routing and path distribution. Given multiple trees with well-chosen roots, the union of the trees together defines a *spanner*, a representation of the graph that preserves approximate pairwise distances. Multiple overlapping trees have previously been used in distributed multicast [5] in *high bandwidth* scenarios, but our goal and use of multiple trees is quite different here — sensors are typically constrained in bandwidth and power.

However, as soon as we use a graph which has cycles, *cycle detection is essential to terminate searches*. This is an advantage of hash-based schemes like GHT. We prevent cycles in our approach by restricting our algorithm to use the trees wisely (and not naively flood on all sets of edges) to balance the complexity of distributed cycle detection and the quality of the paths. More specifically, we mark packets using a single bit to indicate if the packet is traveling up a tree or down a tree, and do not allow repeated interleaving of these steps. Additionally, in order to minimize redundant transmissions, we develop a number of novel aspects in our algorithm. But we first discuss how the multiple trees are created: clearly, choosing the same edges for each tree has no benefit. The trees significantly impact the distances between nodes, and failure recovery as well, as we show later in our experimental evaluation.

3.1 Constructing Multiple Trees

We build over the existing TinyDB [11] routing tree construction algorithm, which we briefly summarize. Nodes begin in a listening/broadcast mode, where they announce their presence and their distance from the root (initially infinity, except for the root, at distance 0). Each node determines the neighbor nearest the root that it can reliably hear from. It chooses that node to be its current parent

and broadcasts its new depth; the process repeats until the network stabilizes. We collect additional information: Bloom filters on the IDs and other static attributes of all of the nodes in a subtree to facilitate equijoins; other *summary structures* — histograms or intervals for range joins on 1D data, R-Trees and quadrants for 2D data — as well as the depth of the subtree. The summary structures act like a generalized index structure [2] and support directed forwarding of messages in the tree. Depth is used in the construction of additional routing trees and in routing.

To create an additional routing tree, we query over the last-built tree for each node's *score*: at leaf nodes this score is initialized to be the sum of the node's depth from the root on every existing tree; at intermediate nodes it is recursively updated to be the maximum score of the subtree. The node with the highest final score is chosen as the root of the new tree. The new root node builds its own tree using the algorithm described above, with one variation. When two candidate parents have equal distance from the new root, we favor the one that is not a parent in a previous tree, and in the absence of such, we choose the parent with the fewest children (to minimize the number of siblings for the current node). The purpose of this step is to ensure path diversity and avoid hot spots.

3.2 Routing: Finding Paths Satisfying ϕ_{RS}

As discussed previously, join initiator nodes (say r) perform a multicast based on ϕ_{RS}, looking for nodes satisfying the condition with r. The request is forwarded in the network in a directed fashion based on summary structures. Ultimately the request may reach a target node. We encode the routing path taken by a message as a vector of node IDs: destination nodes receive path vectors to r, which can be used to send replies to r, ensuring both parties can now communicate without subsequent exploration. Since physical packets are small (under 30 bytes in TinyOS), we *fragment* logical messages (which may contain join keys, source routes, and data) into smaller physical packets. We implement a retry mechanism to ensure packet delivery in the absence of node failure.

Rather than naively flooding, our $BestRoute$ algorithm (Algorithm 1) takes a number of steps to reduce redundant transmissions and avoid cycles. The initial part of the algorithm, Lines 1 – 5, determines when the message has a recipient known to the receiver (the current node or, if the join is on ID, a known neighbor) satisfying ϕ_{RS}. We postpone a discussion of the middle portion of the algorithm to the next paragraph. The packet forwarding portion of the algorithm, Lines 25 – 40, is based on the following observation: we prefer to route from a node *down one or more subtrees* if their summaries are positive, because this allows for directed flooding. However, there is no guarantee that every node is reachable in this fashion. Hence, for correctness, every request must make its way *up* one tree to its root (also searching downwards from each node on the way). Yet there is no need to traverse up *all* of the trees in the same flood; in fact, avoiding this reduces exploration traffic and prevents a number of types of cycles. Following this logic, we start from the initiating node and begin exploring downwards (the *descending* stage), forwarding to any child node in any tree whose node matches the join condition (Lines 25 – 27). We also spawn an exploration request up the parent node on *each* tree: this is the *ascending* stage of Lines 28 – 30. Exploration requests are typically relayed by broadcast (Line 35) to all neighboring nodes.

Hence, a node that hears an exploration request must ensure that it is actually an intended recipient: the sender must be a child or parent node (Lines 9 – 13). Additionally (Lines 14 – 22), the node determines whether the traversal is in *ascending* mode (if so, we need to continue exploring up the same tree, $UpTree$) or *descending* mode (if so, we can only search downwards, and only if some summary is positive for the join condition).

Algorithm 1 BESTROUTE($CN, P, PathV, TopNode, \phi_{RS}, TAttr, SVal, UpTree$). **Input**: Receiving node CN, incoming packet P, path vector from start $PathV$, top of ascending part of path vector $TopNode$, static join condition ϕ_{RS}, target attribute $TAttr$, source value $SVal$, tree to traverse upwards $UpTree$

1: **if** $\phi_{RS}(SVal, CN.TAttr)$ **then return** and acknowledge: routing is complete
2: **if** $TAttr$ is ID and some neighbor N of CN has $\phi(SVal, N.TAttr)$ **then**
3: Forward P to N
4: **return** and acknowledge: complete
5: **end if**
6: **if** sender initiated the request and is a child of CN in tree T **then**
7: $UpTree \leftarrow T$
8: **end if**
9: Set $SourceIsChild$ **if** sender is a child of CN in tree $UpTree$
10: Set $SourceIsParent$ **if** sender is a parent in any tree
11: **if** not $SourceIsParent$ and not $SourceIsChild$ **then**
12: **return** without acknowledgment: spurious message
13: **end if**
14: **if** not $SourceIsParent$ and $TopNode$ is not $PathV.end$ **then**
15: **return** without acknowledgment: spurious message
16: **end if**
17: **if** $SourceIsChild$ and $TopNode$ is $PathV.end$ **then**
18: Set $IsAscending$
19: **end if**
20: **if** not $IsAscending$ and not exists T where P is sent by parent in T, and CN has a child N in T with a summary satisfying $\phi_{RS}(SVal, N.TAttr)$ **then**
21: **return** and acknowledge: redundant search
22: **end if**
23: $ExploreSet \leftarrow \varnothing$
24: **for all** trees T **do**
25: **if** not $IsAscending$ and some child N of CN in tree T has a summary satisfying $\phi_{RS}(SVal, N.TAttr)$ **then**
26: Add N to $ExploreSet$
27: **end if**
28: **if** $IsAscending$ and ($T = UpTree$ or CN is initiating the request) **then**
29: Add CN's parent N in T to $ExploreSet$
30: **end if**
31: **end for**
32: Append $CN.ID$ to $PathV$
33: **if** $IsAscending$ **then** $TopNode \leftarrow PathV.end$
34: **if** $|ExploreSet| > 1$ **then**
35: Broadcast updated P to every node in $ExploreSet$
36: **else if** $|ExploreSet| = 1$ **then**
37: Unicast updated P to the node in $ExploreSet$
38: **end if**
39: Wait for acknowledgment by all nodes in $ExploreSet$
40: **return** and acknowledge: complete

As a node is found to satisfy the join condition, it can use vector $PathV$ and use it to establish a path back to the sender. At this stage, we actually do not limit our traversal to the multiple routing trees, but exploit the full graph. We employ an **optimization** called *shortcutting*, which helps "shorten" the routes established by the BestRoute algorithm. Whenever a node can directly communicate (in symmetric fashion, with high stability) with another node whose ID is more than one step "ahead" on the path vector, it immediately short-circuits to that neighbor and removes the intervening node(s) from the path vector in the packet header.

4. IMPLEMENTATION AND EVALUATION

We implement and experimentally compare the most promising strategies from above, showing their performance differences. These include the **naive** strategy, an exact (as opposed to approximate) version of **base**, a version of **GPSR/GHT**, and a version of **In-net** over our multiple-tree routing substrate. We consider equality and range predicates, and both static and dynamic attributes.

We implemented our algorithms in 10,600 lines of nesC and compiled them using the toolkit for TinyOS 2.0. Our current implementation generates 71KB of code (including packet queuing, fragmenting, and reassembly) and uses a minimum of 3.5KB of memory to maintain 3 summary structures for each routing tree, state for up to 20 neighbors (including summary structures for each child subtree), and buffers for 30 simultaneous route requests. This fits easily within the 8KB RAM and 128KB program memory of the IRIS series motes. Experiments on motes are conducted using the TOSSIM [10] simulator, specifically to have results that incorporate **errors and retransmissions** due to the standard radio model. TOSSIM is perhaps the most realistic simulator of the Crossbow MICAz and IRIS hardware platforms. Experiments were run on Xeon X3220 workstations with 4GB of RAM and Fedora 7.

Methodology. To understand our algorithms under different conditions, we generated 9 runs each of synthetic network configurations with 100 nodes in the following topologies: random with an average of 6 neighbors per node ("sparse random"); random with 7, 8, and 13 average neighbors ("moderate," "medium," and "dense random"); and grid with an average of 7 neighbors ("grid"). Unless otherwise specified, experiments were conducted on 3 different sensor layouts for each topology. We also constructed 50- and 200-node networks, both having 8 neighbors on average per node. We also validated our algorithms on real sensor data and a real topology, using the Intel Berkeley Research Lab dataset [2].

Metrics. Our algorithms and measurements focus on packet transmissions, rather than actual energy, because in some settings sending packets is the predominant consumer of energy, packets are approximately the same length making the conversion simple, and we aim to generalize to a variety of sensor types with different energy consumption profiles. Given that, we are evaluating the initiation phase to be compatible with arbitrary join queries we are interested in (1) path profiles, for example, length, maximum number of paths per node (2) total and maximum routing traffic per node, when a random pair of nodes is chosen for a join.

4.1 Path Profiles and Traffic

The first task of any in-network join algorithm is to discover which pairs of nodes must join. Note that both sources and targets can detect if they are potential participants, but they must discover that there is a matching target or source, respectively.

For BestRoute we study 1, 2 and 3 trees and consider the shortest discovered path. For GPSR, we implemented the strategy outlined in Section 2.1, using a routing destination selected by a hash of the join key. Figures 1a and b show the average join path length and maximum node load, respectively, for routes connecting all pairs of nodes (100 nodes and 9900 pairs) 10 pairs at a time, across different topologies. Further experiments (not plotted due to space constraints) confirmed that the load on roughly the **top 15 most traversed nodes** followed a similar pattern to Figure 1b, in that we saw benefits in BestRoute from going from 1 to 2 to 3 trees, and that GPSR performed worse than BestRoute with 2 and 3 trees. Overall, our observations are that (a) using 3 trees provides near-optimal paths on average; (b) GPSR does not give near-optimal paths; (c) GPSR has significantly worse load distribution for the top 15 nodes. We conclude that given reasonably long-running joins, join initiation using BestRoute with 3 trees is the best strategy.

Initiation Traffic. Another critical factor is how much traffic is created during exploration related to join initiation. Figure 2 confirms that the amount of traffic generated by $BestRoute$ is significantly lower than flooding.

[2] db.csail.mit.edu/labdata/labdata.html

Node Scale-up and Number of Trees. Figure 3 shows that as we scale the number of nodes from 50 to 200, for our algorithm, per-path load on individual nodes (Figure 3b) remains quite consistent, with a slightly more significant benefit to 3 trees when we get to 200 nodes. As expected, average path length (Figure 3a) increases as the network diameter grows; again, with 200 nodes there is a noticeable benefit in using 3 trees. We conclude here that three trees is the optimal choice, because it provides short paths and reduced hot spots while only slightly increasing the state required at each node. Further increasing the number of trees leads to diminishing returns and progressively larger memory requirements.

Joining Real Data. Next, we used the Intel Research-Berkeley real-life dataset, focusing on position data (over which we constructed an R-tree summary structure) and humidity. Our query was $R \bowtie_{dist(R,S)<5m \wedge R.id<S.id \wedge abs(R.hum-S.hum)>1000} S$, which forms join pairs of nodes at most 5 meters apart, and reports results whenever they encounter a humidity difference greater than 1000. For this experiment, we performed an in-network join at the endpoint closer to the base station (breaking ties arbitrarily). Figure 7 reports the experiment in log scale. *Total traffic was over 100MB, but the congestion as well as total traffic was halved by in-network execution.* GHT performed worse than Naive/Base, and the Yang+07 algorithm [14] generated the most traffic.

Radio-on Time. A natural question is whether a scheme like ours, which supports point-to-point and multicast communication across the entire network and does not require (hence, cannot exploit) global time synchronization, inherently requires nodes to keep their radios listening for longer periods of time in each sampling epoch. We are addressing this with ongoing work. However, preliminary experiments suggest that only 5% of messages sent by our substrate would require sensor radios to be on longer than in the single-tree case. We are investigating strategies for allowing those messages to cross into the next sampling epoch, so our scheme has identical or even lower listening overhead.

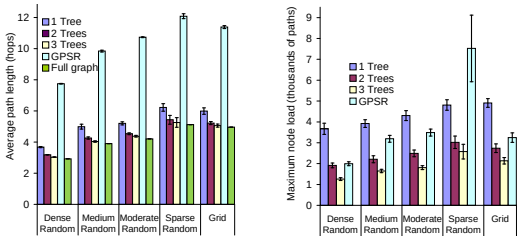
4.2 Path Distribution

We next consider the load with respect to spatial distribution of sensors, showing 3D plots where the plane represents the coordinates of each sensor node and the height of each bar represents the number of paths through that node (bars are projected against the axes). We show results for 3 queries over the synthetically generated topologies. We assigned simulated static sensor values for a static attribute x to every node in the network, sampled from an exponential spatial distribution with range $[7, 60]$, with nodes towards the center of the network biased towards higher values. Static attribute y is sampled from $[0, 10)$ random uniform distribution.

Query 1, $\sigma_{40<R.x<100} R \bowtie_{R.x=S.x} \sigma_{40<S.x<100} S$, imposes heavy communication at the core of the network. Figure 4 shows the results on the grid network topologies; other topologies behave similarly. After adding a second tree, we see a major reduction in the load placed on the interior nodes, with an evenly distributed path load in the middle of the network. Intuitively, by switching trees, our algorithm "short-circuits" across the middle of the network without traversing nodes at or near the root. (We omit the 3-tree case, which provided minor benefits beyond the second tree in this query.)

Query 2 considers joins at the fringes: $\sigma_{0<R.x<12} R \bowtie_{R.x=S.x} \sigma_{0<S.x<12} S$. Figure 5, generated with the random dense topology (but representative of the others), shows that for the 1-tree case, this workload places high load at and near the root. In contrast, 3 trees produces significantly fewer and lower hot spots. We omit the runs using 2 trees, which fell in the middle, due to lack of space.

For Query 3, the endpoints in the join are evenly distributed



(a) Average path lengths (b) Max node load

Figure 1: Path quality & congestion

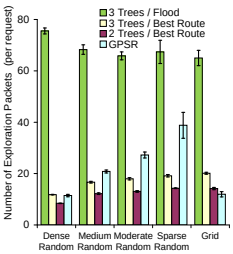
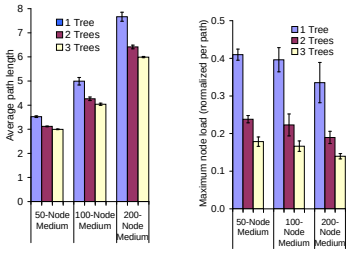


Figure 2: Traffic per exploration request



(a) Path length (b) Node load

Figure 3: Scaleup

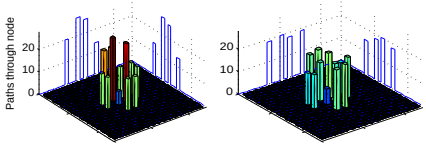


Figure 4: Query 1: 1 vs. 2 trees

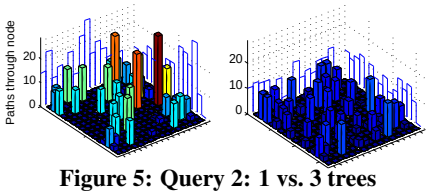


Figure 5: Query 2: 1 vs. 3 trees

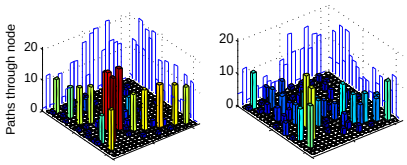


Figure 6: Query 3: 1 vs. 3 trees

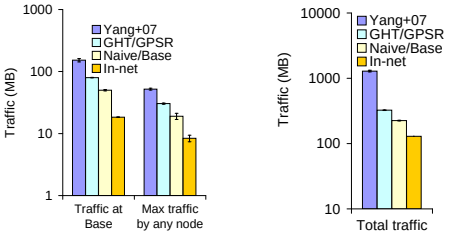
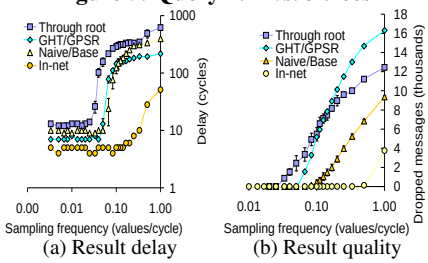
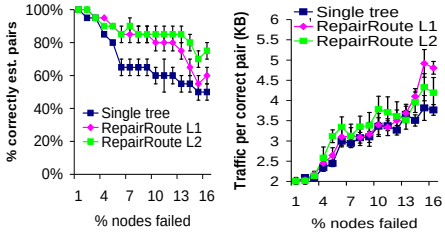


Figure 7: Intel dataset: traffic at the root node, max traffic at any node, total traffic



(a) Result delay (b) Result quality

Figure 8: Increasing sampling rate



(a) Established join pairs (b) Total traffic

Figure 9: Node failures

throughout the network: $\sigma_{R.id<25} R \bowtie_{R.x=S.y+5} \sigma_{S.id>50} S$. Figure 6 shows this for the random medium topology (similar effect is observed on all the other three topologies). Here, two trees provide some benefit but three trees provide the most gain — not only to the *maximum* load of the top nodes, but also the overall number of heavily loaded nodes is significantly reduced.

4.3 Congestion (Stress)

We apply several techniques to improve the performance of our system under high data rates. First, each node tracks free buffer memory; if it is running low on space, it notifies its neighbors through a *congestion bit* carried by data messages. Neighbors attempt to route around the node. For traffic sent through a congested node n towards the *base station*, the sender picks an alternative node to n at the same level in the first routing tree (so long as it is not also congested). For traffic sent through n towards some *other destination node* n_d, the node preceding n on the path replaces n with an alternative neighbor that can reach the n's successor on the path. This scheme does not lengthen the paths traversed by any message. Of course, in some cases congestion avoidance might be impossible: in the extreme case the removal of the congested node n might disconnect the network. In this case, n will stop acknowledging received packets and discard them until it manages to send its outgoing messages. Other nodes needing to traverse n will attempt to resend their packets later.

Multipath routing has been proposed to better reduce maximum load on nodes and congestion, **before** they occur. A similar strategy to the above can be used here: when a packet may be routed through several alternate nodes at any step in the path, the routing algorithm can randomly choose among these nodes.

In order to evaluate our system under congested settings, we took the query used for real life data and varied the sampling frequency from 0.003 values/cycle to 1 value/cycle, while executing for a fixed number of sampling cycles. We measured the freshness (propagation delays of the data values), and the number of packets dropped. In Figure 8a, for Naive (same as Base in this case), GHT, and Yang+07, the delay between the production of data values and their reception at the base station begins to grow for frequencies above 0.05 values/cycle, exceeding 100 cycles. In other words, result freshness for Naive, GHT, and Yang+07 worsens significantly under high sampling frequencies. Moreover, all methods except In-net begin to drop data values in this range, as they cannot keep up with the increased sampling frequency (Figure 8b). At very high frequencies Yang+07 has fewer drops than GHT: sending data on a single tree allows for opportunistic combining of physical packets more efficiently than GHT. To summarize, In-net incurs delays at least 4 times lower than GHT, 7 times lower than Naive, and 12 times lower than Yang+07 — with better overall result quality.

4.4 Failure Handling during Join Initiation

Failed links and nodes are a frequent occurrences in sensor networks, especially due to limited battery life. There are two failure cases in join initiation: (1) during exploration, which we do not focus on since $BestRoute$ explores redundant paths and trees at this stage; and (2) after exploration, when path vectors between join pairs have been established, and the nodes need to coordinate. Suppose node n_1 is sending to node n_d along path vector $[n_1, \ldots, n_f, \ldots, n_d]$, and n_f is unreachable from its predecessor n_{f-1}. We initiate *RepairRoute* (Algorithm 2) which unlike $BestRoute$ explores paths *sequentially* rather than in parallel, while tolerating link failures rather than node failures. $RepairRoute$ searches for any node n_g along the vector $[n_{f+1}, \ldots, n_d]$, and if successful the new path to n_g is merged with the original path vector. During this time n_{f-1} queues up messages in a repair buffer; and on success immediately forwards all of these messages along the new path. A special case is when n_d is the base station. If n_{f-1} can communicate with another node n_1 at the same level as n_f in the main tree, then we forward the request to n_1. This *single tree repair* strategy in isolation requires only one routing tree, and we use it as a baseline in our comparison with $RepairRoute$.

Algorithm 2 REPAIRROUTE($CN, P, PathV, TopNode, DestNode, PathVR, State$. **Input**: Receiving node CN, incoming packet P, path vector from start $PathV$, top of ascending part of path vector $TopNode$, $DestNode$ destination of the failed routing, path from the failed node to $DestNode$ $PathVR$, acknowledgement ACK or $NACK$ in $State$

if CN appears on $PathVR$ **then return**: ACK, routing complete
if for some neighbor N of CN: N appears on $PathVR$ **then**
  Forward P to N
  return ACK, routing complete
end if
if CN is the initiator and $State$=$NACK$ **then**
  return $NACK$: repair is unsuccessful
end if
if sender is a child of CN in tree $UpTree$ **then** Set $IsAscending$
else Set $IsDescending$
end if
if $State$=$NACK$ and ($IsAscending$ or ($IsDescending$ and P was previously seen as descending)) **then return** $NACK$
if $IsDescending$ and not exists T where P is sent by parent in T, and CN has a child N in T with a summary on ID positive for $DestNode$ **then return** $NACK$
end if
$ExploreList \leftarrow []$
for all trees T **do**
  if $IsDescending$ and some child N of CN in tree T has a summary on ID positive for $DestNode$ **then**
    Add N to $ExploreList$
  end if
  if $IsAscending$ and ($T = UpTree$ or CN is the initiator) **then**
    Add CN's parent N in T to $ExploreList$
  end if
end for
Append $CN.ID$ to $PathV$
if $IsAscending$ **then** $TopNode \leftarrow PathV.end$
if $|ExploreList| > 0$ **then**
  if $State$=ACK **then**
    Unicast to first node N in $ExploreList$
    return same result returned by N
  else if P's sender exists in $ExploreList$ at position i and $i < |ExploreList|$ **then**
    unicast P to the node N at position $i + 1$
    return same result returned by N
  end if
end if
return $NACK$

Our evaluation consists of failing up to 16 out of 100 nodes while executing $\sigma_{id<25 \wedge h(u,\sigma_r)} R \bowtie_{R.x=S.y+5 \wedge R.u=S.u} \sigma_{id>50 \wedge h(u,\sigma_s)} S$, where $h(u, \sigma_p) = (hash(u)\% \lceil 1/\sigma_p \rceil = 0)$ on the medium random topology. This query expands Query 3 with dynamic predicates: attribute u is dynamic and it is generated to conform to selectivity estimates σ_r or σ_s. As in Query 3, multiple attributes are joined, and we expect the endpoints in the join to be fairly evenly distributed throughout the network, because of the uniform y predicate, and also because by construction there is no spatial correlation between node IDs. Thus endpoints are separated by long paths, which makes failure recovery non-trivial.

We evaluate RepairRoute and single tree repair for this query. For a given number of nodes set to fail, their times of failure are scattered across the initiation run. We study the increase in the overall traffic and percentage of successfully established source-target join pairs. A given join pair is considered successfully established if both source r and target s are aware that they are joining, and both agree on a join node j, which also knows to join them. $RepairRoute L1$ and $L2$ differ in the summary structure used for routing. While $L1$ utilizes a Bloom filter of the IDs of children in a particular subtree, $L2$'s Bloom filter also holds the children's immediate neighbors, thus increasing the likelihood of success at the expense of some added exploration. As indicated by Figure 9a, by using RepairRoute we can recover completely from up to 3% node failures. We achieve up to 20% increase in the number of successfully established pairs over single tree repair. In Figure 9b we observe that for node failures up to 12% we maintain comparable traffic per successfully established pair with single tree repair.

5. CONCLUSIONS AND FUTURE WORK

This paper proposes a substrate for supporting *join initiation*: discovery of short paths among sets of nodes that mutually satisfy some join condition, and thus are eligible to join in each cycle. This is the basis of in-network join computation, which we have shown to perform better than alternative strategies in a number of cases. Our methods scale to hundreds of nodes.

We are currently focusing on further mote-specific energy-reducing techniques. Then, the next phase of our research agenda is to build upon this substrate and address the issues of query optimization. The first step is to choose a cost model appropriate for sensor networks, which considers overall message transmissions, but also hot spots. The subtlety here is that the cost model needs to take into account opportunities for shared computation: we could share a single join point across multiple source nodes, or alternatively we could keep them separate — a problem we refer to as *multi-pair* optimization. We hope to choose an initial join strategy and join point based on predicted selectivities, then *monitor* and react to actual selectivities and *move* the join point as appropriate.

6. REFERENCES

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

[2] P. M. Aoki. Generalizing "search" in generalized search trees. In *ICDE*, 1998.

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

[4] B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *IPSN*, 2003.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *SOSP*, 2003.

[6] V. Chowdhary and H. Gupta. Communication-efficient implementation of join in sensor networks. In *DASFAA*, 2005.

[7] A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The Cougar project: a work-in-progress report. *SIGMOD Record*, 32(3), 2003.

[8] T. M. Gil and S. Madden. Scoop: An adaptive indexing scheme for stored data in sensor networks. In *ICDE*, pages 1345–1349, 2007.

[9] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream window join: tracking moving objects in sensor-network databases. In *SSDBM*, 2003.

[10] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *SenSys*, 2003.

[11] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

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

[13] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensornets with GHT, a geographic hash table. *Mob. Netw. Appl.*, 8(4), 2003.

[14] X. Yang, H.-B. Lim, M. T. Özsu, and K.-L. Tan. In-network execution of monitoring queries in sensor networks. In *SIGMOD*, 2007.

[15] H. Yu, E.-P. Lim, and J. Zhang. On in-network synopsis join processing for sensor networks. In *MDM*, 2006.

[16] D. Zeinalipour-Yazti, Z. Vagena, D. Gunopulos, V. Kalogeraki, and V. Tsotras. The threshold join algorithm for top-k queries in distributed sensor networks. In *DMSN*, 2005.